Implementing Distributed COM in Samba

Jelmer Vernooij *jelmer@samba.org*
Samba Team Member

February 2, 2005

Abstract

COM, standing for "Component Object Model" is one of the key features of the Microsoft Windows platform. COM allows developers to use and create interfaces that can have several implementations and can be called by different programs, written in different languages. The only thing the involved parties have to agree upon is the interface.

Most widely known is COM as the core of other technologies such as OLE, ActiveX and Automation. Even the .NET platform supports COM.

DCOM is the distributed version of COM. It is implemented on top of DCE/RPC and was documented in a internet RFC. This document describes the way DCOM works and how it will be implemented in Samba-4.

THIS DOCUMENT IS A DRAFT AND IS BY NO MEANS FINAL YET

Contents

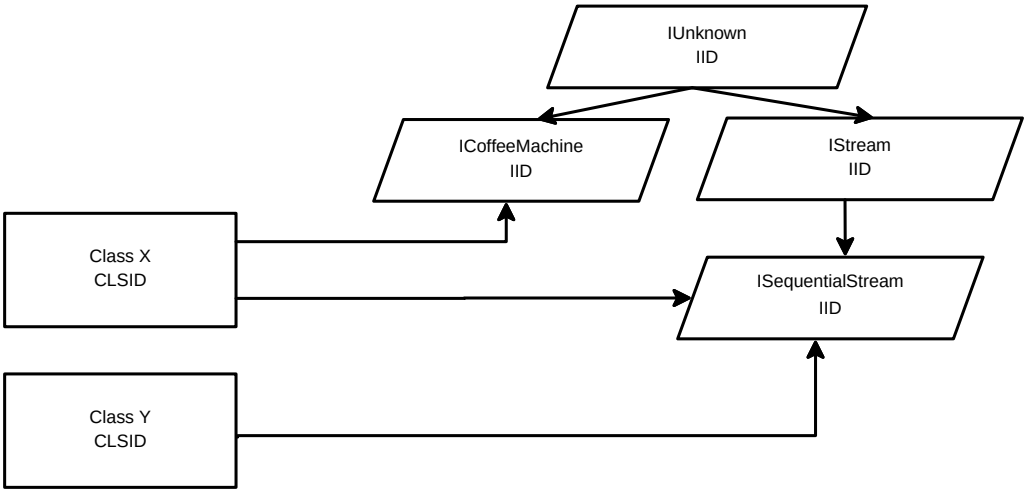

Figure 1: An example of DCOM's distinction between interface and implementation

1 Introduction to 'plain' COM

The terminology used in the COM world can be quite confusing. Especially all the various ID's [1] and the different meanings of the word class tend to confuse people. A broader introduction to COM is available on the web at [6] and [1].

Basically, there are three entities that are important in COM:

- Interface
- Implementation
- Instance

COM has a very clear distinction between interface and implementation. Calls are always made by interface, while the specific implementation is only mentioned when a new object is created.

[1] so far I have seen: MID, OXID, OID, IPID, CID, IID, CLSID, LCID and DISPID.

1.1 Interfaces

Interfaces are the basic elements in COM. An interface is nothing more then a definition of a list of methods that can be run on an object.

Interfaces support inheritance, so if interface A is based upon B it contains all of B's methods and it's own. All interfaces are based upon the "IUnknown" interface.

Interfaces are identifier by Interface ID's (IID's). IID's are globally unique (they are GUID's).

Windows hosts keep a list of interface in their registry, at $HKEY\_CLASSES\_ROOT\backslash interface$.

Note that $HKEY\_CLASSES\_ROOT$ is a "dynamic" registry key created by merging all keys in $HKEY\_CURRENT\_USER\backslash Software\backslash Classes$ and $HKEY\_LOCAL\_MACHINE\backslash Software\backslash Classes$.

1.2 Implementations (Classes)

A COM Class (usually abbreviated to "coclass") is an implementation of one or more interfaces.

Classes are identified by Class ID's (CLSID's), which are also supposed to be globally unique. Some of them also have a "Prog ID", which is a human-readable unique string (e.g. "MyCompany.MyClass"). Those familiar with Java, Python or C#, might recognize their similarity to packages in those languages.

Windows hosts keep a list of classes they know in the registry at $HKEY\_CLASSES\_ROOT\backslash CLSID$.

1.3 Instances (Objects)

Objects are instances of classes and are identified by object ID's (OID's).

Funny enough, classes by themselves are objects as well (sometimes referred to as "class objects"). The methods on a class object are somewhat equivalent to the *static* methods that can be found in languages like C++, Java and C#.

The most important method on a class object is the $CreateInstance$ method that returns a new instance of the class. The $CreateInstance$ method is part of the $IClassFactory$ interface, which is implemented by pretty much all class objects.

1.4 Interfaces Pointers

When making calls to object, the object is never referenced itself — only the specific interface on a certain instance. Such an "instance of an interface" is called an *Interface Pointer*. For example, to do a read on a stream, one does (in simplified C++):

```
char buffer[20];
IUnknown *instance;
com_create_simple_object(Foo_CLSID, &instance);
((IStream *)instance)->Read(20, buffer);
```

1.5 Calling methods

The methods in a class are merely implementations of methods defined by the interfaces that class supports. You always use the interface.

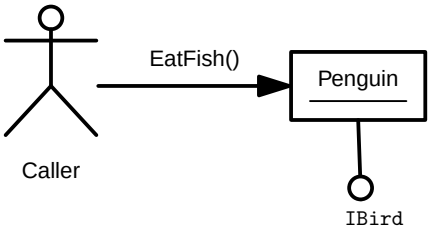

Figure 2: A local COM call

In order to call a method, you first obtain a handle to the relevant interface on your object. That handle is unique for that specific interface on that specific object on the server host and is called a Interface Pointer ID (IPID). You can obtain such a handle by calling the $QueryInterface$ method, which is part of the $IUnknown$ interface.

1.6 Creating an object

In the COM world, there are three ways for obtaining an instance of an object:

- Binding directly to the "Class Object". This can be used to do either a "static" call or to call $CreateInstance$ (if the class object supports the $IClassFactory$ interface) which returns a new class instance
- Create an instance based on the CLSID. Windows looks up the CLSID in the registry, finds the associated DLL or EXE and loads it. ($CoGetInstanceFromFile()$)
- From Persistent State Object

Objects returned by other calls all use one of the three ways described above in way or another.

ActiveX controls embedded into browsers can also use URL's to get access to an instance of an object.

2 Defining Interfaces

In the Windows world, COM structures, interfaces and classes are specified in ODL (Object Definition Language). ODL is basically DCE/RPC IDL with a few extensions. Originally, these ODL files were compiled by a tool called *mktyplib*[5] but later support for ODL was merged into the *midl*[4] tool which is used on Windows to compile DCE/RPC IDL files. During this process some ODL features were lost [2].

The main extensions ODL makes to IDL are:

- inheritance for interfaces. Base classes can be specified in a fashion similar to C++ class inheritance.
- "object" interfaces, i.e. interface that can have instances. Specified by adding *object* to the properties of the interface.

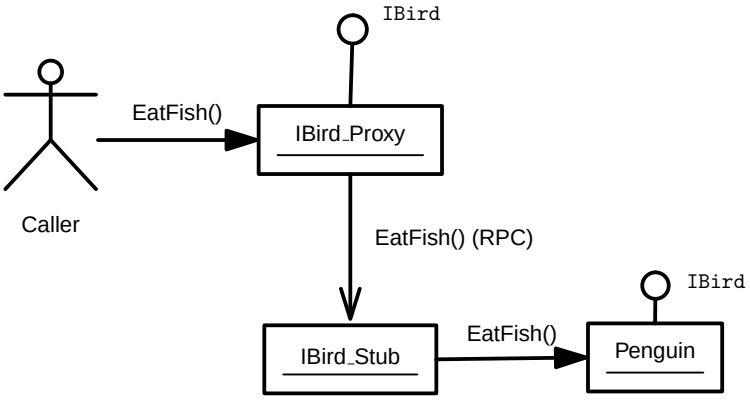

Figure 3: A remote COM call using DCOM

- passing pointers to interface instances. Either an explicit interface name can be specified or the $iid_is()$ property can be used to pass a pointer to an interface (run-time) identified by a custom IID.
- writing "type library files", i.e. binary files that contain a binary version of the interface definition (extension: *.tlb*

3 Distributed COM

Distributed COM is nothing more then a 'hack' to let standard COM run remotely. It was added later as a way to allow the distribution COM objects over multiple computers. It is basically the glue between DCE/RPC [10] and COM.

DCOM is not as closed as one might think — there are several documents available from Microsoft explaining the wire format [7, 8]. There has even been an internet draft on DCOM [9].

3.1 Proxies and stubs

When doing COM remotely, instead of directly calling the object one would like to use, the client does the call on a 'stub' object that takes care of the network call.

The stub, which is usually autogenerated, then does one or more DCE/RPC calls and passes along the data it receives.

On the server side (where the 'real' object remains) the DCE/RPC subsystem forwards the RPC call to a proxy object (also autogenerated, usually) that then makes a call to the real object.

In the Windows world, MIDL generates these proxies and stubs. In Samba, these classes are generated by pidl.

3.2 Object Exporters

Every object lives in a certain context, known to Windows programmers as an *apartment*. Calls crossing apartment boundaries need marshalling. Usually, an

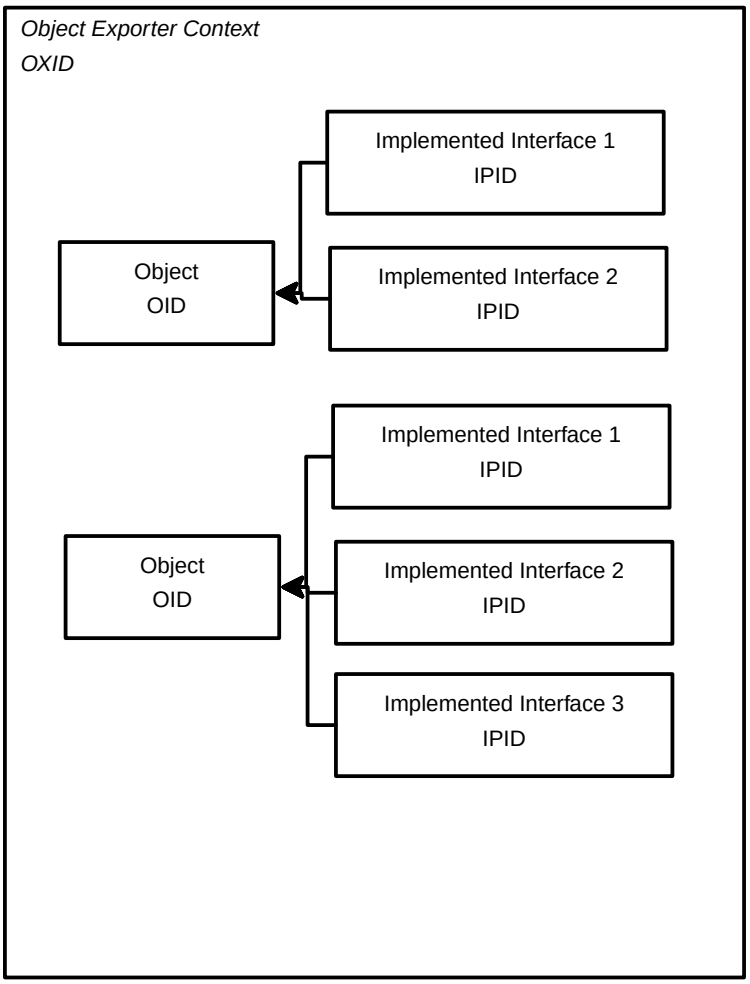

Figure 4: The object exporter

the term apartment maps to a thread. Apartments are identified remotely by OXIDs[2].

3.3 ORPC

DCOM is implemented on top of DCE/RPC[10] and uses the optional *object* field in the DCE/RPC header, in which it stores the IPID. This extension is usually referred to as ORPC (Object RPC).

Other then this minor change, there is no need for modifications to the client-side DCE/RPC implementation.

All DCOM calls have a mandatory extra `[in]` and a mandatory extra `[out]` argument that always occur before the other arguments. These arguments (ORPCTHIS and ORPCTHAT) contain COM version and extension information.

[2]I think these are just thread IDs, which are unique to the system on Windows.

3.4 Marshalling

When an interface pointer is sent across the wire using NDR, it is transformed into a OBJREF struct by the proxy, after which it is sent as a normal NDR structure.

The proxy can marshall interface pointers into the OBJREF struct using three possible methods:

- Standard marshalling
- Handler marshalling (Requires the interface to implement IStdMarshalInfo)
- Custom marshalling (Requires the interface to implement IMarshal)

3.5 Activation

Before an object can be used, it has to be activated (created). Older DCOM implements use the "plain" *IRemoteActivation* interface for this purpose while newer implementations (2000 and above) use the object interface *ISystemActivator*[3].

This interface can be called with a CLSID and a list of interfaces the client would like to use on the new object. The server will then (locally) do a number of calls:

1. GetClassObject(CLSID)
2. IClassFactory::CreateInstance()
3. IUnknown::QueryInterface() (once for each interface in the list)
4. IUnknown::Release()

Next to a list of interface pointers, the server also returns an OXID and a list of SECURITYBINDINGS and STRINGBINDINGS belonging to this OXID.

3.6 OXIDs

Objects don't have to live at the same endpoint or even the same host they were activated at. For this purpose, every interface pointer that is sent over the wire contains an OXID (Object Exporter ID) and a resolver address.

If the client wants to do a call on an interface pointer, it asks the remote server listening on the resolver address where the object with the given OXID can be reached. The server will respond with a list of SECURITYBINDINGS and STRINGBINDINGS.

The STRINGBINDING structs as returned by *IRemoteActivation* and *IOXIDResolver* can easily be converted to DCE/RPC binding strings while the SECURITYBINDING structs contain possible account names that can be used to authenticate.

[3] I suspect this interface is implemented by the *System.Activator* class in .NET

3.7 Garbage Collection

In order to make sure objects aren't kept running long after the client that created them crashed horribly, clients have to regularly send pings to the host they are living on. Pinging is done based on OID (object ID's) rather then IPID's. Objects can be grouped together in *PingSets*, which makes it possible to ping thousands of objects with a just a very small packet.

Objects are generally presumed dead if none of their clients have pinged them for 3 minutes.

3.8 Running Object Table

Every machine keeps a list of the objects it exports.

The running object table can be accessed using the IROT[3] interface, which is available at the same endpoint as the endpoint mapper (which corresponds to the rpcss.exe process on Windows).

4 Integration in Samba 4

The first attempt to implement DCOM in Samba 4 failed because it was aimed at implementing both DCOM and COM together, ignoring the concept of proxy classes. This attempt failed because it made matters very complicated.

The final attempt seperates the COM and DCOM implementations (somewhat similar to Windows), making the code simpler.

- *lib/com* contains a very simple (but sufficient) COM implementation.
- *lib/com/interfaces* contains pidl generated structure files for the COM interfaces
- *lib/dcom* contains the glue between DCE/RPC (NDR) and COM.
- *lib/dcom/proxies* contains the pidl-generated proxy files

4.1 COM implementation

Since Samba is written in C, which is not object-oriented, object-oriented calls are 'emulated'. This works pretty much in the same fashion as the GTK+[4] project uses it.

For example, the following Windows code:

```
IClassFactory *pcf = 0;
CoGetClassObject(CLSID\_Penguin, CLSCTX\_ALL,0, IID\_IClassFactory, (void**)\&pcf);
IBird *pBird = 0;
pcf->CreateInstance(0, IID\_IBird, (void**)\&pBird);
pcf->Release();
pBird->EatFish();
pBird->Release();
```

would translate to :

[4] http://www.gtk.org/

```
struct IClassFactory *pcf = NULL;
struct IBird *pBird = NULL;
com_get_class_object(CLSID_Penguin, IID_IClassFactory, &pcf);
IClassFactory_CreateInstance(pcf, mem_ctx, 0, IID_IBird, &pBird);
IUnknown_Release(pcf, mem_ctx);
IBird_EatFish(pBird, mem_ctx);
IUnknown_Release(pBird, mem_ctx);
```

in Samba.

4.1.1 Pidl extensions

Samba 4 will support ODL files in it's regular IDL compiler rather then creating a seperate compiler for ODL files. Reason for this is the fact that ODL files are really not that different from IDL files. Also, the extensions made by ODL do not require large amounts of changes in the pidl code nor do they cause additional code complexity in pidl.

The additional pidl module *com$_h$eader.pm* generates C header files that contain structs with function pointers for each object interface. They also generate wrapper macros for easier use of the interfaces.

For example, a header file generated by *com$_h$eader.pm* for an ODL file containing the IUnknown and IStream object interfaces would look like this:

```
  /* Interface */

\#define IUNKNOWN\_METHODS  \
int (*Release) (struct IUnknown *, TALLOC\_CTX *mem\_ctx); \
int (*AddRef) (struct IUnknown *, TALLOC\_CTX *mem\_ctx); \
void (*QueryInterface) (struct IUnknown *, TALLOC\_CTX *mem\_ctx, struct GUID *iid, void *


   struct IUnknown\_vtable {
   IUNKNOWN\_METHODS
   };

\#define ISTREAM\_METHODS \
IUNKNOWN\_METHODS \
void (*Read) (struct IStream *, TALLOC\_CTX *mem\_ctx);

struct IStream\_vtable {
ISTREAM\_METHODS
};

/* Instance */
struct IUnknown {
struct com\_context *ctx;
struct IUnknown\_vtable *vtable;
void *object\_data;
};

and wrapper functions:
```

```
\#define IUnknown\_QueryInterface(i,m,p1,p2) (i->vtable->QueryInterface(i->object\_data, m
...
```

4.2 DCOM implementation

4.2.1 Pidl extensions

A new Pidl module, *proxy.pm*, generates one proxy class (implementation) per interface. The functions int this class each simply call their DCE/RPC equivalents (similar to Windows).

Another new Pidl module, *stub.pm*, generates one stub with a custom dispatch function for each interface. This dispatch function looks up the interface pointer in the running object table and then calls the apprioriate function on the interface pointer.

```
 - Problem:
  Need MInterfacePointer in NDR-related stuff, but
need struct IUnknown / void in com\_header, etc. (with IID)

- Always set iid\_is() for Interface names occurring in IDL
- Need different treatment in com\_header.pm and proxy.pm
- For each object interface, define a
ndr\_{push,pull}\_IUnknown() that just does a push/pop of an MInterfacePointer
- unknown interfaces are a different story, but always require a iid\_is(), so always enco
- in the proxy, do a conversion from "real interface pointer" <-> MInterfacePointer
 - when iid\_is() was set
 - What when iid\_is() not set?
   - See if \$NAME is registered as an interface or as a struct. If it's
   not registered at all, use something like this:

\#ifdef COM\_IUNKNOWN\_UUID
/* Conversion from MInterfacePointer */
\#else
/* Plain assignment
\#endif

TODO:
Split out NDR-related and other stuff in ndr\_*.h
```

5 Future integration

5.1 Mono

(Ideas)

- support incoming DCOM calls from Windows clients
- support outgoing DCOM calls to Windows hosts

5.2 Wine

(Ideas)

- ability to disable either rpcss or Samba's epmapper support
- use epmapper and rot pipe internally as well as externally
- support for ncacn_unix_ds: in wine or ncalrpc: in the same fashion as Samba?
- Proposal: Keep all the object management stuff in Samba, but allow Wine (thru the ROT interface) to add new objects. This allows the Wine folks to use the Samba infrastructure for now, while giving them the opportunity to move to a Wine-specific implementation of this stuff later on, if necessary.

5.3 Mozilla (ActiveX)

There is a plugin for windows only at the moment, available in Mozilla CVS [5]. Maybe extend to support *nix ?

References

[1] URL `http://www.codeguru.com/Cpp/COM-Tech/activex/tutorials/article.php/c5567/`.

[2] Differences between midl and mktyplib. . URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/differences_between_midl_and_mktyplib.asp`.

[3] Irunningobjecttable. URL `http://msdn.microsoft.com/library/en-us/com/htm/cmi_q2z_8go5.asp`.

[4] Microsoft interface definition language. . URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_start_page.asp`.

[5] Mktyplib command-line tool. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/com/htm/ctrans_8a7g.asp`.

[6] Don Box. *Essential COM*. Addison Wesley Longman, Inc., 1998. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnesscom/html/msdn_essential_com.asp`.

[7] Guy Eddon and Henry Eddon. Understanding the dcom wire protocol. 1998. URL `http://www.microsoft.com/msj/0398/dcom.aspx`.

[8] Markus Horstmann and Mary Kirtland. Dcom architecture. 1997. URL `http://msdn.microsoft.com/library/en-us/dndcom/html/msdn_dcomarch.asp`.

[5] http://www.iol.ie/ locka/mozilla/plugin.htm

[9] Charlie Kindel Nat Brown. Distributed component object model — dcom/1.0. 1998. URL http://www.grimes.demon.co.uk/DCOM/DCOMSpec.htm.

[10] Open Group. *DCE 1.1: Remote Procedure Call*, 1997. URL http://www.opengroup.org/onlinepubs/9629399/toc.htm.