The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

CS4/MSc

Distributed Systems

Jane Hillston

jeh@dcs.ed.ac.uk

Room 1604

(Lecture 8: DCOM and CORBA compared,
31st January 2002)

An overview of DCOM

- DCOM (Distributed Component Object Model) is a distributed object model which is native to the Windows operating system.
- DCOM extends the Component Object Model (COM) with an object remote procedure call (ORPC) layer which builds upon the Open Software Foundation's DCE-RPC specification.
- **COM servers** can create object instances of multiple **object classes**.
- A COM object can support multiple interfaces which represent different views of the object.
- Fuctionality to locate remote servers and manage RPC invocations is provided by the **Service Control Manager**.
- COM provides language interoperability by specifying that any interface must follow a *standard memory layout* (that of the C++ virtual function table).

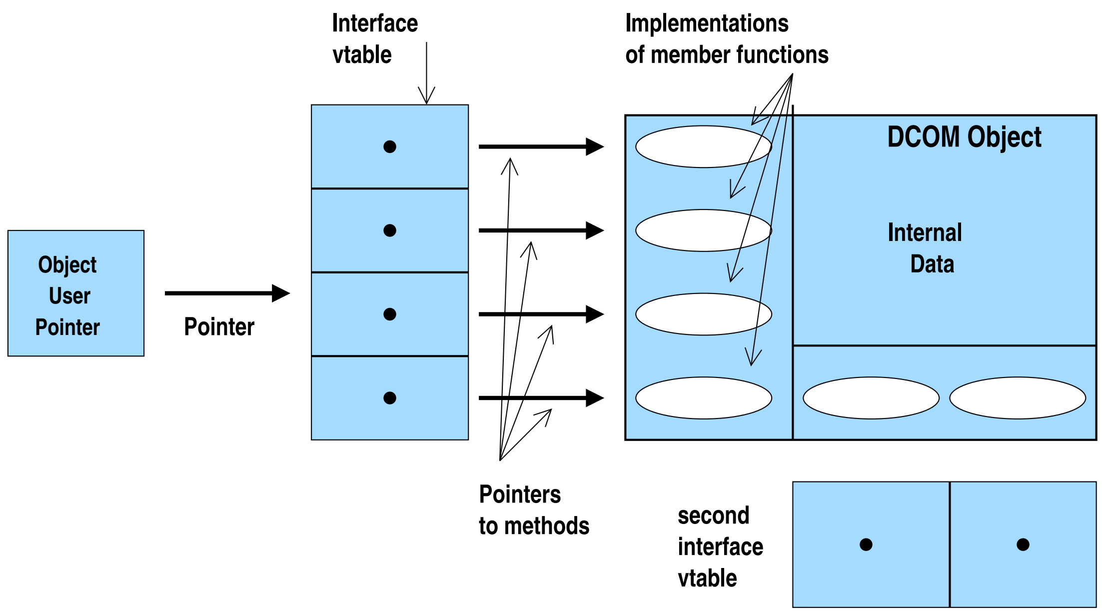
DCOM and CORBA similarities

- Like CORBA, the DCOM framework provides *client-server* communication.
- To request a service, a client invokes a method implemented by a remote object.
- The remote object acts as a server in the client-server model.
- In both DCOM and CORBA the client makes a call to the client stub. The stubs packs the call parameters into a request message, and invokes the *communication module* to send the message to the server.
- At the server side, the communication module delivers the message to the server stub, which unpacks the request and calls the actual function on the object.
- In DCOM the client stub and the server stub are referred to as the *proxy* and the *stub* respectively. (In CORBA they are called the *stub* and the *skeleton*.)

DCOM and CORBA differences

- A COM client interacts with a COM object by acquiring a pointer to one of the object's interfaces and invoking methods through it.
- This means that the client has a pointer to an interface to the object rather than the object itself.
- As a result COM objects do not have a persistent identity: a COM client cannot reconnect to exactly the same object instance with the same state at a later time.
- Re-use an interface pointer may lead to connection to another instantiation of the same class.
- Each object type has a primary interface which is the one provided by default, but also provide a function to query what other interfaces are available.
- Each interface pointer provides an array of function pointers, termed the **virtual table** or **vtable**. These pointers point to the actual object implementations.

A DCOM Interface



DCOM's Interface Definition Language

- Like CORBA, DCOM provides an Interface Definition Language.
- As with CORBA, the purpose of this is to serve as a *contract* between a server and its clients.
- Both IDLs support data encapsulation, polymorphism and single inheritance.
- CORBA IDL supports multiple inheritance but DCOM IDL does not. DCOM provides the notion of an object having multiple interfaces to achieve a similar purpose.
- CORBA IDL can express programmer-specified exceptions, but DCOM IDL cannot.
- Both CORBA IDL and DCOM IDL are processed with their respective IDL compilers to generate stub code and interfaces that are used by both the server and the client.

DCOM IDL features

- In DCOM, each interface is assigned a **globally unique identifier** (GUID) called the **interface ID** (IID). Each object class is assigned a unique **class ID** (CLSID).
- A GUID is a 128-bit number based on time and network address that is assumed to be universally unique.
- GUIDs appear in DCOM IDL as long number strings. They can be automatically generated by Microsoft Developer Studio.
- Every COM interface must inherit from the `IUnknown` interface which provides methods to navigate between different interfaces for an object and methods for reference counting.
- DCOM requires that all methods return a 32-bit error code called an `HRESULT`. For languages which support exceptions, system services allow failure codes to be converted into them.
- DCOM IDL associates multiple interfaces with an object class in a block describing the COM class (the `coclass` block).

Comparing CORBA IDL and DCOM IDL

- We provide a brief comparison of CORBA IDL and DCOM IDL with an example taken from *Client/Server Programming with Java and CORBA* by Robert Orfali and Dan Harkey.
- Consider the following CORBA IDL definition

```
// This file is CORBA IDL (Count.idl)

module Counter {
    interface Count {
        attribute long sum;
        long increment();
  };
};
```

This definition defines an interface for a counter (`sum`) which will have “get” and “set” methods. In addition it is possible to increment the counter (by 1).

DCOM IDL version

```
// This file is DCOM IDL (Count.idl)
[uuid(66892570-5F62-11d1-99E3-0060084A1358), version(1.0)]

library Counter {
    [ object, uuid(66892571-5F62-11d1-99E3-0060084A1358),
      pointer_default(unique), oleautomation ]
    interface ICount : IUnknown  {
        import "oaidl.idl";
        HRESULT set_sum([in] int val);
        HRESULT get_sum([out, retval] int* retval);
        HRESULT increment([out, retval] int* retval);
    };
    importlib("stdole32.tlb");
    [ uuid(66892572-5F62-11d1-99E3-0060084A1358),  ]
    coclass Count {  [default] interface ICount;  };
};
```

Type information in interfaces

- In CORBA, the IDL compiler generates type information for each method in an interface and stores it in an **Interface Repository** (IR).
- In DCOM, type information for interface methods is stored in a **type library** generated by the IDL compiler and assigned a GUID.
- The DCOM IDL compiler which is provided with Microsoft Developer Studio is `midl`. The type library which it produces is first registered with the system and then the Java Type Library tool (`javatlb`) uses it to produce a class summary file as well as the `ICount` and `Count` class files themselves.
- The base class files produced by `javatlb` only implement the `ICount` DCOM server, a default factory implementation that can create instances of the ICount component. The actual implementation of the ICount methods must be provided by hand in a Java file. The implementation of that class is shown next.

The Count implementation

```
import com.ms.com.ComException;
import Counter.*;
class Count implements ICount {
    private int sum;
    public int get_sum() throws ComException {
        return sum;
    }
    public void set_sum(int val) throws ComException {
        sum = val;
    }
    public int increment() throws ComException {
        return sum++;
    }
}
```

Both the constructor and `main()` are taken care of by DCOM.

DCOM server responsibilities

A DCOM server is a Java class which houses one or more object classes each with its own CLSID. It is sometimes termed a **serving agent** to emphasise that the server is not a COM object itself. Each serving agent must:

- Implement a class factory interface for each object class that it supports. Once an object is created, a pointer to its primary interface is returned to the client.
- Register a CLSID for each class that it supports with the registry. The CLSID is registered with information about the location (pathname) of the serving agent.
- Initialise the COM library and verify that it is compatible in terms of version. The library provides runtime services to objects.
- Provide an unloading mechanism to terminate itself when there are no active clients for its objects.
- Un-initialise the COM library upon termination.

DCOM in operation: reference counting

- C++ and Java programmers use the **new** operator to create a new instance of some class. COM programmers call `CoCreateInstance()` to create a new COM object.
- Unlike C++, COM does not provide an explicit delete operation. Unlike Java, it does not assume the presence of a garbage collector. COM relies on **reference counting** instead.
- With reference counting, each COM object counts how many clients hold references to its interfaces (clients add themselves by invoking the `AddRef()` method inherited from `IUnknown`).
- When a client is finished with an object it calls `Release()` on that object, decrementing the reference count.
- An object can delete itself when its reference count is zero.
- When using COM with Java these details are hidden and the language runtime takes care of reference counting. A Java client can treat an external COM object as if it was a Java object.

DCOM in operation: the *pinging protocol*

- DCOM uses reference counting to control the lifetimes of both local and remote objects.
- The **pinging protocol** is used to allow a server object to garbage-collect remote object references when a remote client fails. This means that objects are not kept for clients who have crashed.
- When a client obtains an interface pointer to a remote object it adds it to a **ping set** and periodically sends a ping to the server (a **keepalive packet**) to let it know that the client is still alive.
- Every two minutes a ping packet is sent to the server. If no such packets arrive for three ping periods—six minutes—the server assumes that the client is dead and calls `Release()` as needed.
- Without such a protocol in use, the servers in the system would grow without bound, storing no-longer-needed garbage objects, and would eventually need to be shut down and restarted to free those resources.

DCOM performance issues

Two aspects of DCOM's remote reference counting would cause performance problems if they had not been implemented well.

Firstly, a lot of network traffic could be generated with reference counting just keeping each remote reference count up-to-date. However, it turns out that *the remote reference count need not be up-to-date*—as long as a remote object thinks that it has one client it will not be released. Most calls to `AddRef()` and `Release()` go no further than the client proxy: they are not sent across the network. Only when the last `Release()` occurs is the message actually sent.

Secondly, all of the network bandwidth could have been wasted on ping packets if every client sent a ping packet to every remote object it holds a reference to. Instead, the DCOM infrastructure automatically creates ping sets. The entire ping set is kept alive with a single packet sent every two minutes between server and client. Pings can also be *piggy-backed* on normal messages.

From DCOM to .NET

- Microsoft's .NET initiative involves a number of technologies, the most important of which is **the .NET Framework.**
- The .NET framework provides a new approach to components based on a **Common Language Runtime** (CLR), shared between multiple programming languages (C++, C#, Visual Basic, ...)
- A program in a high-level programming language such as C# (similar to Java) is compiled into low-level **Microsoft Intermediate Language** (MSIL) code (similar to Java Byte Code).
- The MSIL code is then run in a managed execution environment, the Common Language Runtime (which is similar to the Java Virtual Machine).
- Both platforms support native compilation via *just-in-time* (JIT) compilers. JIT compilation is an option with the Java Virtual Machine but .NET code is always compiled to native code before execution.

.NET and COM development compared

- A goal of the .NET Framework was to make COM development easier.
- The .NET Framework has automated COM features such as reference counting and others (interface description, and registration).
- In addition, the .NET framework improves COM security and provides automatic memory management via mark and compact garbage collection.
- However, .NET Framework components can still interoperate with COM components, providing compatability with legacy systems implemented using COM components.

.NET and COM together