Com Corner: Distributed Computing

by Steve Teixeira

When I was a kid, one of my favorite breakfast cereals was Alphabits. For those of you who have never seen Alphabits, it consists of little pieces of sweetened cereal shaped like individual letters of the alphabet. With each spoonful of Alphabits, I would try to scoop out the right chunks of cereal to spell interesting words, like 'COOL' or 'BIKE' or the ever-elusive 'STEVE'. While I don't usually eat breakfast cereal now, I think I chose to work in the software development field to fill that deep-seated need I have to try to make sense out of various bunches of letters. Take the issue of DCOM versus CORBA, for example. If you've kept up with these articles written by myself and Dave Jewell, you probably already know that DCOM stands for *Distributed Component Object Model* (no, not *Common Object Model*), and it's Microsoft's 'standard' specification and implementation for interoperable software objects over a network. CORBA, if you've read the back of the Delphi 4 box, stands for *Common Object Request Broker Architecture*, and it represents a competing standard for software objects that work together over a network. Given the confusing mix of acronyms associated with these technologies, how do you decide which is right for you? Perhaps more pragmatically, why should you be interested in distributed computing in the first place? This article takes a high-level look at these issues in hopes of increasing your comfort level with distributed computing.

The Next Logical Step

In order to see where software development is heading, it's important to understand where we have been. In the early days of business computing, most companies employed a two-tiered model where under-powered dumb terminals would run applications and access data on a mainframe, which provided the storage and processing for the entire environment.

When the age of the PC dawned, developers quickly flocked to the modest hardware platform. It didn't take long for the demands of software to push the envelope of the hardware and operating systems. Overlays, protected mode, resources and DLLs were all devised as methods for handling the growing size and complexity of application code and data. Eventually the two-tier model came back into vogue, with moderately powered PCs accessing data stored on powerful database servers. In this model, enforcement of business rules was split between the client (with UI rules and application logic) and the server (with relationships, triggers and stored procedures). The complexity and inefficiency inherent in this division of business logic, combined with the demand for more and more power on the database servers, ushered in the age of multi-tier application development.

In the multi-tier development model, the client machine is as 'thin' as possible, and it communicates with one or more middle-tier machines that enforce business rules, which in turn communicate with the database servers. With the explosion of the internet and corporate intranets, multi-tier application developers soon found their middle-tier servers bombarded by greater volumes of users. In order to scale to intranet and internet volumes, it became necessary to componentize middle tier logic and split it among multiple machines. This architecture does away with the old concept that an application is just an EXE file and maybe some DLLs that run on a particular machine. An application can be any number of EXEs, DLLs, services, daemons, etc, running on any number of machines. Like any application, the components of a distributed application need to be able to communicate between one another, and there is therefore a serious need for intelligent middleware that enables such inter-component communication. This is where technologies such as DCOM and CORBA come in.

And don't make the mistake of thinking that distributed computing is 'that thing those enterprise people do.' Most of us use distributed applications almost every day on a little something we call the internet. Think about it: you hit a page on one of, say, Microsoft's web servers that launches an Active Server Page on another server, that uses DCOM to access data which lives on yet another server. Do you know or care that all this stuff is going on behind the scenes? No, you just want to get to the content you're looking for. However, the implementation is the way it is because that is what scales the best.

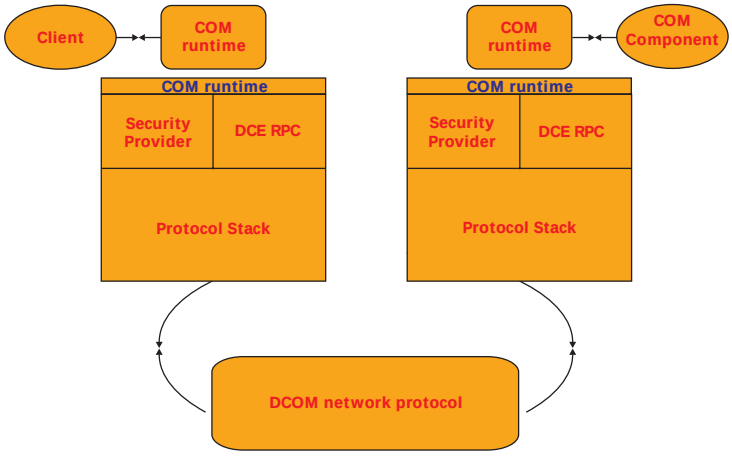
We are fast approaching a less machine-centric age in computing, where it doesn't matter what machine processes or stores what data. And all of this with a comparatively puny pipe between machines. Imagine what happens when the pipes connecting machines allow data to be passed at anything approaching the speed of the data bus *within* the machine!

DCOM And CORBA

It's important to understand that DCOM and CORBA are really out to solve the same fundamental problem: getting components to work together in a distributed, programming language independent

environment. However, for all their similarities, these two technologies come at the problem from very different points of view. DCOM follows Microsoft's agenda for enabling Windows to work well in a distributed environment, while CORBA follows an agenda of staunch platform-neutrality and standards-based development. Let's find out a little more about these technologies and the relative merits of each.

➤ *Figure 1*



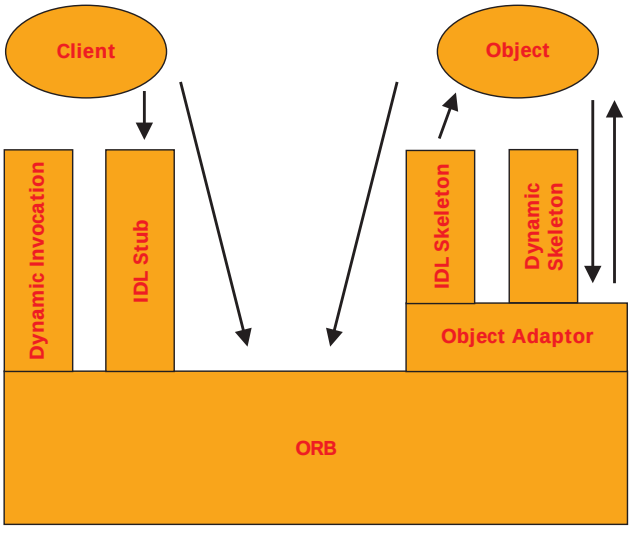
DCOM

DCOM is an extension of Microsoft's COM technology that enables COM clients and servers to communicate across machine bounds. For the purposes of this article, I'll assume a basic understanding of how COM works. Since DCOM is an evolutionary step in Microsoft's efforts for application and component interoperability, you can trace the genesis of DCOM to that quirky technology called OLE (Object Linking and Embedding) that was designed to allow applications to communicate with one another on the same machine. OLE matured into OLE 2, which employed COM as its basic underpinnings, and enabled in-place activation of one type of object within another (for example, an Excel spreadsheet within a Word document). Later came ActiveX, which, among other things, added a variety of technologies that support the internet to the COM runtime. DCOM then came on the scene as a means to broaden the machine-centric COM technology to entire networks and even the internet. Figure 1 shows a graphical representation of the steps involved in making a DCOM call from client to server. The client calls into the COM runtime, which handles details like marshalling and object creation. The data is passed across the wire using a special DCOM-enhanced version of the DCE RPC protocol, where it's received, unmarshalled, and passed into the COM server.

CORBA

CORBA is a platform-neutral middleware technology that is defined by the Object Management Group (OMG) and implemented by a number of different software vendors. While DCOM represents the evolution of Microsoft's OLE and COM strategies, CORBA was designed from the ground up with the goal of enabling software components to communicate with one another, no matter who wrote them or where they exist. The heart of a CORBA system is the Object Request Broker (ORB). The ORB is the functional equivalent of the DCOM runtime, security, and transport layers. CORBA objects are generally defined using Interface Definition Language (IDL). From the IDL, skeletons are generated for the object implementation and stubs are generated for the client to call object methods through the ORB. For Delphi users, this process is handled automatically when you define your CORBA interfaces in the Type Library Editor.

➤ *Figure 2*



Like COM, CORBA also provides a means for dynamic invocation of methods on the object from the client. All calls from client to server are routed through the ORB, and like COM, the ORB can be called directly by the client or the server when either wishes to take advantage of ORB services. Services such as security can be built onto the ORB using Object Adapters. Figure 2 shows an example of a CORBA client making a call into an object.

The purpose of this article isn't to drill down deeply into the gory details of DCOM and CORBA implementations, but rather take a high-level approach and look at some of the advantages and disadvantages inherent in each.

Some of the programming issues surrounding enterprise component development include: technology ownership, cost of entry, tools support, multi-platform support, performance, security, lifetime management and extensibility. I will provide a comparison of the two technologies for each of these topics.

Technology Ownership

A chief advantage of DCOM is that it is a Microsoft technology. Because DCOM is a strategic part of Microsoft's future plans, you can rest assured that the world's largest software company is staffing it with ample resources to ensure it

continues to be a viable technology. What's more, you can bet that Windows platforms will always be the best equipped to take advantage of DCOM. All that said, I also consider the fact that DCOM is a Microsoft technology is also considered a disadvantage. This is due to Microsoft's reputation for investing in technologies which further their agenda (sell more Windows!), without necessarily solving the customer's problems. Microsoft is free to modify DCOM whenever and wherever they see fit. What's more, Microsoft also has a reputation for making their own technologies obsolete in order to sell the Next Big Thing.

As I mentioned earlier, the OMG is responsible for the care and feeding of the CORBA specifications, while ORBs are implemented by several companies and individuals. Because the OMG employs a committee of various industry representatives that steer the direction of CORBA, it's unlikely to become too heavily influenced by the agenda of any one particular vendor. However, as we all know, anything designed by a committee isn't likely to be without compromises.

My analysis is: if you are a Microsoft shop, then you shouldn't consider the issue of DCOM ownership a bad thing, since you can be reasonably sure that DCOM will continue to work well with Windows for a long time to come. However, if your environment is heterogeneous, then the fact that CORBA relies on industry standards across multiple vendors will make it the clear choice.

Cost Of Entry

Perhaps the most compelling advantage of DCOM is that it is based on COM. If you already have a significant investment in COM development, then you will probably be able to get your COM object up and running over a network with DCOM with no programming and a little bit of configuration. Not only that, but you can leverage existing expertise in COM development so that your development efforts experience little down time as you transition into the distributed world. Also, DCOM is available for free.

Currently, the cost for entry into the CORBA world is high. To be an effective CORBA developer you must be proficient in a number of disciplines, including IDL, additional platforms and the specifics of the ORB you use. Delphi does reduce the cost of entry into CORBA by allowing you to expose your Automation objects as CORBA objects. However, to go beyond the basics still requires a significant amount of knowledge. Monetarily speaking, expect to pay ORB licensing fees.

My analysis is: the clear edge for Delphi developers is DCOM.

Tools Support

Between Microsoft's Visual Studio tools and Inprise's Borland tools, it's difficult to throw a rock into a software store without hitting a box of software that makes COM development easy. Necessity, as they say, is the mother of invention. Without development tools support, COM development is a tedious practice.

While bringing the world's best CORBA support to Borland brand tools is Inprise's mission, CORBA integration with development tools is still in the early stages. Delphi is currently the only RAD tool with support for CORBA, and while it shows a great deal of promise, the implementation isn't as complete as COM support.

My analysis is: DCOM is in the lead, with CORBA gaining ground.

Multi-Platform Support

Make no mistake, DCOM is a Microsoft technology. Microsoft has even gone as far as to state that they intend for Windows to be the best platform for DCOM. While DCOM implementations for other platforms such as various Unix flavors, OpenVMS and MacOS are in various stages of development or beta testing, release dates have yet to be announced.

Multi-platform is what CORBA is all about. There are ORB implementations for all major platforms and a whole bunch of obscure platforms, and CORBA handles the complexity of marshalling between plaforms.

My analysis is: no contest. If you intend for objects to operate on multiple platforms, then CORBA is the only way to go today.

Performance

Believe it or not, there is no clear winner as far as performance is concerned. DCOM is faster in some cases and CORBA in others. My advice here is, all other things being equal, write COM and CORBA test applications that come as close to your production use of the technology and benchmark them.

Security

DCOM uses Windows NT security and access is handled via the DCOM Configuration utility (DCOMCNFG). There isn't a lot of flexibility in the DCOM security model unless you undertake the significant task of writing your own client and server proxies, which could encrypt data, perform custom login or authentication, or employ some other security feature. One advantage to DCOM's security infrastructure is that you don't have to pay any extra for it; it comes with the OS.

CORBA's security architecture is designed to be extensible, and can even integrate with the Java 'sandbox' for client-side applications. The downside is the possible additional cost or configuration associated with a particular security mechanism. Examples of CORBA security products include Inrise's VisiBroker SSL Pack and VisiBroker GateKeeper.

My analysis is: CORBA is your choice if you need security capable of fulfilling the requirements for virtually any industry.

Lifetime Management

COM is most commonly used in situations where server objects are transient in nature, and the reference counting mechanism inherent in `IUnknown` isn't exactly ideal for distributed objects. For example, the enormous volume of `AddRef` and `Release` calls that COM

generates as a part of standard operating procedure could be murder on a network as well as the server receiving all those calls. To help alleviate this problem, DCOM tries to queue `AddRef` and `Release` calls on the client and send them to the server in batches. Speaking of reference counting, minor ref counting bugs are something every COM programmer fights: they are the frequent cause of objects going away while you are using them or objects hanging around forever because their ref count never reaches 0. On one machine it isn't that big a deal, but in a multi-user distributed environment, it's unacceptable. As an extra measure to guard against errors caused by clients going down, DCOM clients periodically ping their servers. If a server doesn't receive a ping from a client within a certain time, references to that server from the client will be released.

Unlike COM, CORBA objects are not generally designed to be transient. Typical CORBA objects go up and stay up for a long period of time. Because it was designed from scratch as a technology to reach across machine boundaries, it isn't plagued by the same lifetime management problems inherent in DCOM due to its COM roots.

My analysis is: if your architecture calls for objects to stay up for very long periods of time (days, weeks, months), then CORBA is the better choice. DCOM is a good choice for transient objects, but watch out for those ref count bugs!

Extensibility

DCOM is extensible only in the sense that you build infrastructure on top of COM or your application. It isn't possible to hook directly into the COM runtime provided by the operating system.

CORBA is designed to be extensible so that services, such as security, transaction processing, and object activation and deactivation can integrate with the ORB. Using the Internet Inter-ORB Protocol (IIOP), ORBs from different vendors can even communicate with one another.

My analysis: CORBA is definitely more extensible.

Summary

Hopefully, this article has provided you with some insight into the world of distributed computing and the differences between COM and CORBA. As you can see, DCOM is sort of the path of least resistance for the Windows developer, whereas CORBA should be the choice of those concerned with standards-based development and objects running in a heterogeneous environment. You'll now think of me the next time you find letters like DCOM, CORBA, OMG, IDL, ORB, or IIOP in your morning Alphabits.

Steve Teixeira works at DeVries Data Systems. He wishes to thank Lino, Helen and Phil for their help with this article. If you have a great idea for a COM article, email Steve at steve@dvdata.com