Microsoft®

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
(206) 882-8080

Microsoft® Object Linking and Embedding (OLE) Today and Tomorrow

Technology Overview

December 1993

Introduction

Microsoft's Object Linking and Embedding (OLE) specification offers a variety of ways to integrate application components, including features such as visual editing, Drag and Drop between applications, OLE Automation, and structured storage for objects.

The capabilities of OLE are powerful and compelling, and OLE 2.0 has recently received two prestigious industry awards: a Technical Excellence award from *PC Magazine* and the MVP award for software innovation from *PC/Computing*. Moreover, more than 30 applications using OLE 2.0 are shipping today, with hundreds more scheduled to appear during the next six to twelve months.

There is far more to OLE than desktop application integration. To support its award-winning integration features, OLE defines and implements a mechanism that allows applications to "connect" to each other as software "objects" — collections of data and accompanying functions to manipulate the data. This connection mechanism and protocol is called the *Component Object Model*. The many user-oriented and document-centric features of OLE are built on the Component Object Model's simple and fully extensible object architecture. In other words, OLE is a projection into the desktop space of a new and powerful general-purpose technology for building distributed, evolving object systems.

This paper looks briefly at the business reasons for the rapid movement in the computer industry to object-oriented software. It then reviews some of the capabilities of the Component Object Model, with special emphasis on its ability to provide a robust system model for seamless connections between software components running across multiple computers on a network. It concludes with information about the product direction of Microsoft and Digital Equipment Corporation which will result in products that allow OLE-based applications to interact with software objects running on a variety of UNIX®-based and other server platforms using the same programming model and communications layer used by OLE between Microsoft®-based systems.

The Business Benefits of Objects

As its name suggests, the OLE Component Object Model is based on the notion of a *component*. A component is a reusable piece of software that can be "plugged into" other components from other vendors with relatively little effort. For example, a component might be a spelling checker sold by one vendor that can be plugged into several different word processing applications from multiple vendors. Or it might be a specialized transaction monitor that can control the interaction of a number of database servers. In contrast, traditional applications are monolithic, which means that they come prepackaged with a wide range of features, most of which can't be removed or replaced with alternatives.

Component software provides a much more productive way to design, build, sell, use, and reuse software. It has significant implications for software vendors, end users, and corporations:

- **For vendors**, component software provides a single model for interacting with other applications and the distributed operating system. While it can readily be added to existing applications without fundamental rewriting, it also provides the opportunity to modularize applications and to incrementally replace system capabilities where appropriate. The advent of component software will help create a more diverse set of market segments and niches for small, medium and large vendors.
- **For users**, component software means a much greater range of software choices, coupled with better productivity. As users see the possibilities of component software, demand is likely to

increase for specialized components to be purchased at a local software retail outlet and plugged into applications.

- **For corporations**, component software can mean lower costs for corporate computing, helping IS departments work more efficiently, and enabling corporate computer users to be more productive. IS developers may spend less time developing general-purpose software components and more time developing "glue" components or components that solve business-specific needs. Existing applications do not need to be rewritten to take advantage of a component architecture. Instead, corporate developers can create object-based "wrappers" that encapsulate the legacy application and make its operations and data available as an object to other software components in the network.

Objects that conform to the Microsoft Component Object Model are known as *component objects*. As a long-term strategy, the concept of component objects consists of both current and future technologies that are designed to facilitate the development and use of component software. OLE 2.0 is the first step in the evolution of component objects. Future implementations of OLE will be designed to use the same basic mechanisms of the Component Object Model and be upward-compatible with OLE 2.0 while providing a range of additional features, including the ability for objects to communicate over the network. The investments that ISVs and IS developers make in implementing OLE technology will be protected in the long term by Microsoft's investments in future versions of OLE and Windows.

OLE Today and Tomorrow

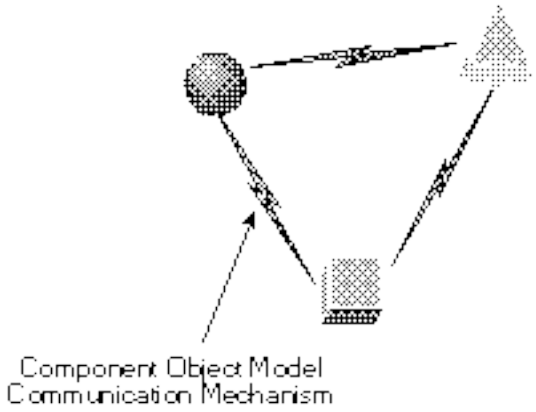
OLE and its Component Object Model are the first steps in a major path of innovation for Microsoft systems. OLE is critically important for the following reasons.

- **Binary standard for objects.** OLE defines a completely standardized way for objects to be created and to communicate with one another. Unlike traditional object-oriented programming environments, these mechanisms are independent of the applications that use object services and of the programming language used to create the objects. This binary standard will be used extensively in future versions of the Microsoft Windows™ operating system and will enable a wide market for component software.
- **Compelling collection of interfaces.** Software architectures become more interesting when useful products are shipping based on the architecture. While other vendors and some consortia have defined high-level object architectures and specifications for language-independent object systems, only ISVs using OLE have delivered to the mass market a compelling set of interoperable applications based on object technology.
- **True system object model.** To be a true system model, an object architecture must allow a distributed, evolving system to support millions of objects without risk of erroneous connections of objects and other problems related to strong typing or definition of objects. OLE's Component Object Model meets those requirements.
- **Distributed capabilities.** Many single-process object models and programming languages exist today, and a few distributed object systems are available. However, none provides an identical programming model for small, in-process objects and potentially large, cross-network objects. Moreover, security is required. Microsoft's OLE has these capabilities factored in.

Binary Standard for Objects

OLE enables interoperability among objects that are written by different programmers from different companies. For example, a spreadsheet object from one vendor can connect to a database object from another vendor and import database records into the cells of its spreadsheet. As long as both objects support a predefined interface for data exchange, the spreadsheet and database don't have to

know anything about each other's implementation, other than how to connect through the standard mechanism defined by the Component Object Model and exchange data through the common interface.



The OLE Component Object Model Provides a Standard Way to Communicate

Without a binary standard for interobject communication and a standard set of communication interfaces, programmers face the daunting task of writing a large number of procedures, each of which is specialized for communicating with a different type of application, or perhaps recompiling their applications depending on the other components with which they need to interact. Moreover, if the mechanism used for object interaction is not extremely efficient, PC software developers pressured by size and performance requirements simply would not use it. Finally, object communication must be language-independent since programmers cannot and should be forced to use a particular programming language to interact with the system and other applications.

The OLE Component Object Model meets these challenges. In OLE, applications interact with each other and with the system using collections of function calls or methods, called interfaces. An interface is a strongly typed "contract" between software components that is designed to provide a small but useful set of semantically related operations. All OLE objects support a method called QueryInterface that allows for very efficient negotiation or communication between components to find which interfaces they share. The small portions of functionality defined in the interfaces plus OLE's inherent interface negotiation protocol allow software components to interact with one another in simple or complex ways depending on the needs of the component. They also allow for change and graceful evolution within the object system, since new interfaces can be introduced and discovered safely and efficiently without disturbing existing patterns of interaction between components.

At the lowest level, OLE object interaction is extremely fast and simple. Once the connection between software components is established, method invocations on OLE objects are simply indirect function calls through two memory pointers. As a result, the performance overhead of interacting with an OLE object in the same address space as the calling code is negligible — only a handful of processor instructions slower than a standard function call, and the overhead is even less considering that a typical function call would have to contain some kind of parameter to identify the entity that the caller seeks to act upon. Thus, there is no performance barrier to using OLE objects pervasively, even on low-end PCs.

The simplicity of the OLE model also provides language independence. Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers — languages such as C, C++, Pascal, Ada, Small Talk®, and the Microsoft Visual Basic® programming system — can create and use OLE objects immediately. Other popular languages are being extended to provide direct OLE support since OLE-style programming is planned to be an inherent aspect of future versions of Microsoft Windows. Object-oriented languages can provide their own higher-level

mapping between language objects and OLE objects, and can also provide class libraries to make OLE programming easy.

Future versions of the Microsoft Windows operating system are planned to use OLE-style interfaces extensively in areas where new services are defined. Facilities as diverse as visual controls, multimedia services and distributed security are targeted to be defined and programmed through the Component Object Model. The existing Win32® Application programming interface will continue to be needed and fully supported; at the same time, object-based capabilities will gradually become pervasive, blurring the distinction between applications and the system and providing for easily replaceable components within the Windows operating system itself.

OLE: A Compelling Collection of Interfaces

Object models are interesting to theoreticians and software designers, but are unimportant to users. As discussed throughout this paper, OLE is based on a powerful, general-purpose object system. Microsoft's Component Object Model provides a language-independent binary standard for object interaction, and strongly typed components that can scale safely to distributed networks comprised of millions of objects. But the primary reason OLE will succeed in the marketplace where many object systems fail is that the immediate benefits of OLE to users are obvious and compelling.

OLE provides visual editing, Drag and Drop between applications, OLE Automation, and structured storage for objects. Visual editing allows two or more applications to cooperate in the editing of compound documents and display windows such that the user sees only a single document or window, with multiple editors loading themselves dynamically depending on which part of the document is in use. Drag and Drop allows users to select an application object such as a document or chart with the mouse and drop in into another application window where it will be copied or moved. OLE Automation provides a standard means for macro and script languages to drive one or more applications by viewing and manipulating a set of internal application-level objects, such as paragraphs, cells, rows, tables, forms, with methods for altering the object's state. Finally, structured storage allows applications to cooperate in the creation of compound files supporting a variety of native data types stored as nested objects within a standard file format.

OLE 2.0 has garnered awards and praise from the industry. According to Michael J. Miller, editor in chief of *PC Magazine, "*Object Linking and Embedding 2.0 offers a far easier, far better method of integration, and it will fundamentally change our expectations for the next generation of software" (*PC Magazine* Dec. 7, 1993, p.78). Columnist Jim Seymour said, "OLE 2.0 is a big, big win for both software developers and PC users. I'd go so far as to say that it's the most important development in PC software of 1993" (*PC Magazine* Dec. 7, 1993 p. 98).

A True System Object Model

Object technology is proliferating and moving outside the realm of object-oriented languages. One area of interest is the development of language-independent class library technology. This kind of technology solves the "C++ in a DLL" problem — the problem of recompiling all code that uses a class whenever changes are made to the class itself — and can be useful for application development. But it is not appropriate for a system object model.

There are several fundamental limitations of class library technology when used to build distributed, evolving object systems.

- **Strong typing.** Distributed object systems have potentially millions of interfaces and software components that need to be uniquely identified. Any system that uses human-readable names for finding and binding to modules, objects, classes, or methods is at risk. The probability of a collision between human-readable names is quite high in a complex system. The result of a name-based identification will inevitably be the accidental connection of two or more software components that were not designed to interact with each other, and a resulting error or crash — even though the components and system had no bugs and worked as designed.

By contrast, OLE uses globally unique identifiers — 128-bit integers that are virtually guaranteed to be unique in the world across space and time — to identify every interface, type and class. Human-readable names are assigned only for convenience and are locally scoped. This helps ensure that OLE components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects.

- **No implementation inheritance.** Implementation inheritance — the ability of one component to "subclass" or inherit some of its functionality from another component — is a very useful technology for building applications. But more and more experts are concluding that it can create problems in a distributed, evolving object system. The problem is well-documented in academic literature, which calls it "the fragile base-class problem." The problem with implementation inheritance is that the "contract" or relationship between components in an implementation hierarchy is not clearly defined; it is implicit and ambiguous. When the parent or child component changes its behavior unexpectedly, the behavior of related components may become undefined. This is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object system. So while implementation inheritance can be a very good thing for building applications, it is risky in a system object model.

  OLE does provide a code reuse mechanism called "aggregation." Using this model, a set of objects can work together in a well-defined manner to appear to other software components as a single object. Aggregation provides the benefits of code reuse while maintaining explicit relationships between all objects and avoiding the risks of implementation inheritance.

- **Single programming model.** A problem related to implementation inheritance is the issue of a single programming model for in-process objects and out-of-process/cross-network objects. In the former case, class library technology permits the use of features that don't work outside a single address space, much less across a network. For example, implementation inheritance typically does not work outside a single address space. In other words, the programmer can't subclass a remote object. Similarly, features like public data items in classes that can be freely manipulated by other objects within a single address space don't work across process or network boundaries. OLE's Component Object Model has a single interface-based binding model and has been carefully designed to avoid any differences between the local and remote programming model.

- **Security.** For a distributed object system to be useful in the real world it must provide a means for secure access to objects and the data they encapsulate. While OLE 2.0 does not implement security features, it has been designed to be upwardly compatible with future implementations of OLE that provide a full range of security features. Object servers can be modified to take advantage of secure object invocations, but unmodified OLE 2.0 clients can participate in fully secure distributed environments.

The issues surrounding system object models are complex for corporate customers and ISVs making planning decisions in this area. OLE meets the challenges, and is a solid foundation for an enterprise-wide computing environment.

Distributed Capabilities for OLE

Today's version of OLE supports a rich set of features for integrating information on a user's desktop computer. But imagine if users could easily integrate objects on different computers as if they were all local. With this capability, a user in San Francisco could, for example, link a range of spreadsheet cells on their desktop to a corporate database in New York. Each time data in the database was updated, the user's spreadsheet would automatically reflect these changes.

To support this level of object integration across different computers, Microsoft is developing a new implementation of OLE that takes full advantage of the inherent capabilities of the Component

Object Model. Distributed object communication is the next logical step for OLE 2.0. It provides the same kind of standard interobject cooperation as today's OLE 2.0, but allows this cooperation to take place *across networks* of computers. For example, an address-book application on a Windows-based computer can connect to a different address-book object on a UNIX-based system and import its address information without the user knowing that the interaction is taking place between different platforms over a network.

Remote Services With No Added Effort

Significantly, the new implementation of OLE requires no changes to existing applications. An existing OLE 2.0 application can immediately begin linking to other applications on other machines — without any changes the application's source code, and without recompiling the application. In other words, applications automatically receive these remote capabilities without any effort on the part of users or programmers.

Applications don't need to be changed because only the underlying object communication mechanism (which is transparent to applications) is being extended. The new version of OLE makes no changes to the OLE 2.0 application programming interface (API) and object interfaces, so applications call the same OLE 2.0 functions in the same way. If the services that support these function calls happen to be located on a different computer, the OLE infrastructure sends the request to the remote service automatically and invisibly.

How Distributed OLE Works

Distributed capabilities are a natural extension of the OLE Component Object Model. The most significant difference between current and future implementations of OLE is the remote procedure call mechanism used to transfer operations and data between objects. In OLE 2.0, a "lightweight" remote procedure call mechanism (LRPC) is used for interobject communication within a single computer. LRPC allows objects to pass information across the process boundaries that protect applications from each other within the operating system. In other words, LRPC is an interprocess communication facility that allows processes (such as objects) to talk to one another on the same machine. The new implementation of OLE extends OLE 2.0 by adding object communication across a network, using Microsoft RPC rather than LRPC.

RPC systems allow applications to call remote procedures as if these procedures were located within the same address space as the calling application. With RPC, an application calls a remote procedure in the same way it would call a local (in-process) procedure, but in reality, that procedure may be located in another process on the same machine, or on a different machine across the network. Since the transfer of data between the calling application and responding procedure is handled transparently, it is possible to build applications that run across multiple processes or computers without changing the programming model used by a non-distributed applications.

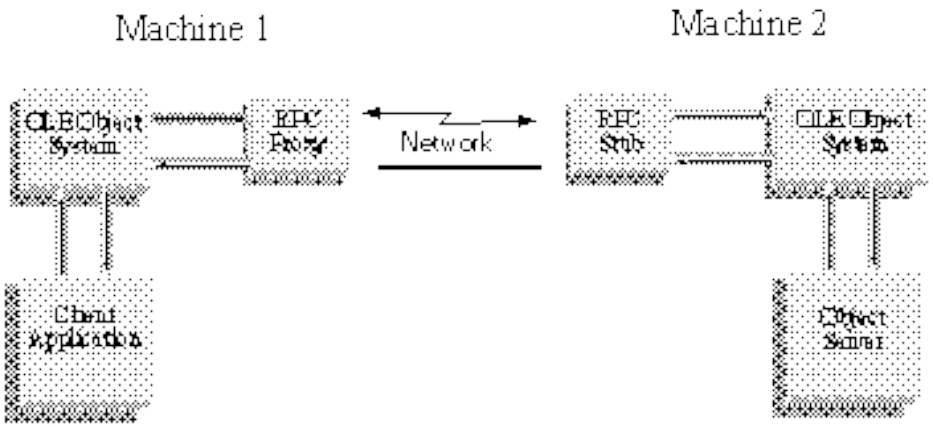
Microsoft RPC adds many capabilities to LRPC, the most notable of which is the ability to make RPC calls across the network. A more subtle but equally important aspect of Microsoft RPC is a set of accompanying tools that automatically generate "marshaling" code that packages data for transmission. This capability allows programmers to easily define new object interfaces that allow objects to communicate outside a single process. Moreover, Microsoft RPC is compatible with the RPC standard defined by the Open Software Foundation in its distributed computing environment (DCE) specification. Because Microsoft RPC is based on DCE RPC, systems that support Microsoft RPC can exchange information with a wide range of DCE-based systems, including OpenVMS™, MVS, AS/400®, and more than 20 varieties of UNIX. Microsoft RPC adds a few upward-compatible extensions to the DCE Interface Definition Language (IDL) used to define objects, but does not change the wire-level protocols.

While RPC itself provides a number of crucial capabilities that a distributed object system needs, such as network transport independence, security and

name service APIs, and data conversion between incompatible processor and operating system architectures, RPC is not enough. OLE's object layer above RPC provides further abstraction and simplification of the distributed programming issues as well as the unique polymorphic capabilities of objects.

At the object layer, OLE 2.0's LRPC architecture already provides transparency between local instances of objects and non-local instances (objects outside the address space of the calling program). Achieving this transparency is the most difficult aspect of building a distributed object system. While there is a substantial amount of work involved in adding a full RPC mechanism (and related capabilities such as distributed security), the new implementation of OLE does not change the overall architecture.

In the figure below, a client application connects to an object's server that is running on a different machine. An RPC proxy running on the client's machine enables the client to communicate with an object server as if it were on the same machine. The proxy simply packages function calls into a standard RPC message that can be transmitted over the network using the network transport running on the client's machine. A corresponding RPC stub on the server receives the RPC message from the client, unpackages it, and passes it to the object's server. To the server, the process is the same as if it were communicating with a local client.



RPC Allows Applications to Call Methods on Remote Systems

Note that the words "client" and "server" simply refer to the user and provider of object-based services, respectively. There is no connotation that the client is a desktop machine and the server is a large back-end machine. In fact, in OLE most interaction between applications involves a two-way relationship in which both software components are clients and servers to each other simultaneously.

Distributed Objects Will Redefine Computing

OLE with distributed object support allows a single application to be split into a number of different component objects, each of which can run on a different computer. Since OLE provides network-transparency, these components do not appear to be located on different machines. The entire network appears to be one large computer with enormous processing power and capacity. For example, a database application could be built as a set of components: a query engine, a report engine, a forms builder, and a transaction manager. Each of these components could run on a machine suited to the amount of processing power, I/O bandwidth and disk capacity required for it. As a result, computing can become much more efficient because software can be more closely matched with the exact hardware power required. Computing also becomes much more scalable,

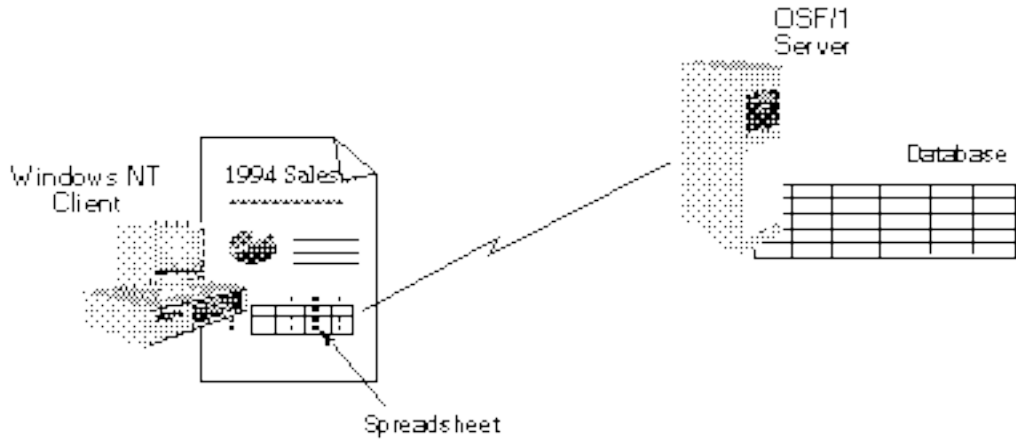
since the virtually unlimited resources of an entire network can be leveraged by a single application or a group of applications.

Common Object Model

Recognizing the need to allow objects on different types of operating systems to interact, Microsoft and Digital have developed an architecture to allow interoperation of OLE and Digital's multi-platform object system, ObjectBroker™. This architecture, called Common Object Model (COM), defines a common DCE RPC-based protocol and a subset of core OLE functions that will be supported by Digital and other interested companies within their products. The Common Object Model is a direct outgrowth of the Component Object Model and provides full upward compatibility with OLE. When OLE and ObjectBroker work together, the Windows, Windows NT™ and Windows NT™ Advanced Server operating systems can connect to objects running on a variety of platforms including OSF/1™, HP-UX®, SunOS™, IBM® AIX®, ULTRIX™, and OpenVMS.

To show how OLE operates across platforms using COM, Microsoft and Digital have demonstrated a sample application that connects an object running on the OSF/1 operating system with objects on a Microsoft Windows NT-based computer. The sample application supports a standard OLE 2.0-compatible "hot link" between Microsoft Excel and a stock quote object running on the OSF/1 server.

Transparently to the user, the OLE elements in ObjectBroker allow the Microsoft Excel spreadsheet to be linked to its source data on the OSF/1 machine. Together, OLE and ObjectBroker hide the new mechanism used to find the OSF/1 server and dynamically transfer stock quote data into the cells of the Microsoft Excel spreadsheet. Users watch the stock data update in real time on their screen, without having to know anything about the different sources of the data.

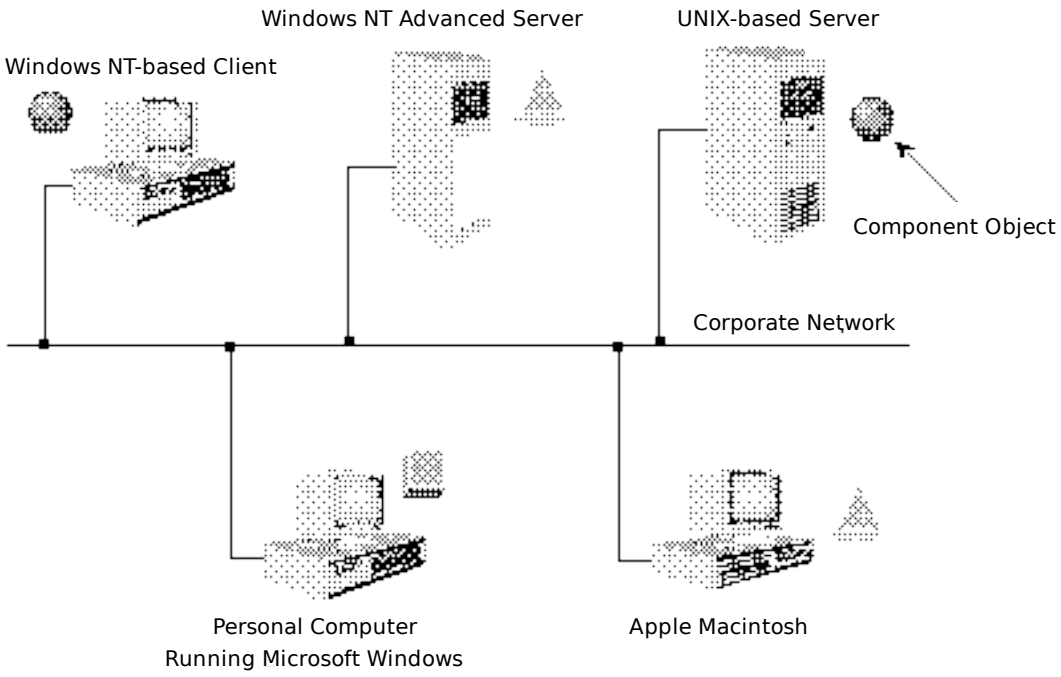


Demonstration: OLE Allows Object Links to Exist Across Heterogeneous Networks

For users, the benefits are numerous. With the Common Object Model, users can access information on virtually any platform in the enterprise without being concerned about the type of application they are connecting to, or the type of communication mechanism needed to reach that object. Using simple techniques such as OLE 2.0's Drag and Drop, users can manipulate objects throughout the enterprise without knowing or caring where they are located or which application was used to create them.

Microsoft and Digital are committed to following an open process for the Common Object Model that will address broad industry requirements. Design reviews for system integrators, corporate

application developers, independent software vendors, and other interested third parties are planned for the first half of 1994. Prior to these reviews, draft publications will be published for review and comment.



OLE and ObjectBroker Enable Enterprise Computing at the Object Level

An Evolutionary Path for Windows

Today's OLE 2.0 technology is a major foundation piece in Microsoft's strategy operating system direction. The next major releases of Microsoft Windows NT (code-named "Cairo") and Windows (code-named "Chicago") will build on OLE and be compatible with it, respectively. Cairo is intended to add a range of capabilities designed to make creating, accessing, manipulating, organizing and sharing information easier for computer users. It will offer an advanced object-oriented environment that focuses users on manipulating information through queries on content and properties, not on manipulating applications or groping around networks. To do this, Cairo will integrate a number of new and existing technologies that change the way people use computers while making computers much more intuitive to use. Similarly, Chicago will ship with full OLE 2.0 support, and future versions of Chicago will provide OLE with distributed capabilities as well as other aspects of the Cairo technology.

Although the evolution from OLE 2.0 to Cairo and Chicago offers dramatic changes in the way computers will be used, the path to this functionality will be smooth for existing applications. Much of the advanced technology that will be available in the future will simply be inherited by existing applications, with no changes to the applications themselves. Moreover, the OLE style of object programming will be used extensively in future Microsoft operating systems to implement replaceable and modular system services. Finally, the Microsoft and Digital Common Object Model helps ensure that OLE's distributed object capabilities will be available on a wide range of UNIX and OpenVMS platforms as well as desktop computers and Windows NT-based servers. Therefore,

independent software vendors and corporate IS engineers can begin implementing solutions today using OLE 2.0 and be assured that this solution can tap into the power of future distributed, heterogeneous object-based computing systems.

#########