Frameworks For Component-Based Client/Server Computing

Scott M. Lewandowski
Department of Computer Science
Brown University
Providence, RI 02912-1910
scl@cs.brown.edu

Abstract

This paper presents two frameworks for client/server computing that rely on a new model involving distributed objects and Java. It introduces client/server computing and distributed objects and discusses how Java and distributed objects can be used to provide a robust system for client/server computing. After discussing the shift from objects to components, two popular component models (CORBA and DCOM) are presented, with an emphasis on a three-tier client/server model. The role of Java in a component-based client/server system is discussed, and the integration of Java and CORBA is presented as the optimal framework for component-driven client/server computing. A brief discussion of frameworks is then given and two frameworks are presented: a system for supplying business objects based on CORBA and a client model relying on the compound document paradigm.

Table Of Contents

Frameworks For Component-Based Client/Server Computing

1. Overview

This paper introduces the basics of client/server computing and component technologies and then proposes two frameworks for client/server computing using distributed objects. Topics affecting client/server frameworks are discussed, with a focus on the delegation of responsibilities between clients and servers and the stratification of client/server systems into levels. The component technologies discussed are CORBA, the Object Management Group's proposal for a distributed object framework and DCOM, Microsoft's system for creating and using objects on remote machines while maintaining the common paradigm for interaction among libraries, applications, and system software that COM provides [Chap]. ActiveXs, components built from COM-based technologies, are treated as an important example of DCOM parts in client/server systems. It should be noted that JavaBeans is not discussed, since it is a language-specific technology, not suitable for use in an environment when all participating components are not necessarily written in Java. The Java programming language is discussed in terms of its contributions to an effective framework for client/server computing using the distributed object services of CORBA. Java applications, which are suitable for downloading to a local machine from the World Wide Web, are discussed as a Java component unit providing a total integration framework for client/server computing while using distributed objects.

2. What is Client/Server Computing?

Client/server computing systems are comprised of two logical parts: a server that provides services and a client that requests services of the servers. Together, the two form a complete computing system with a very distinct division of responsibility. More technically, client/server computing relates two or more threads of execution using a consumer/producer relationship. Clients serve as the consumers in a client/server system. That is, they make requests to servers for services or information and then use the response to carry out the purpose of the client. The server plays the role of the producer, filling data or service requests made of it by clients.

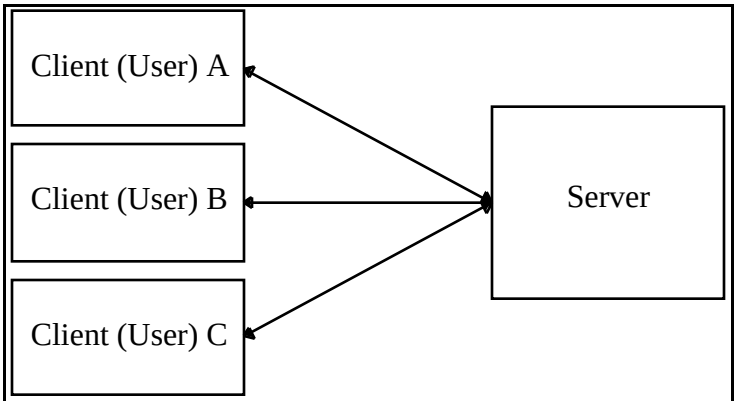

Figure 1: A traditional client/server system. Clients request services of the server independently but use the same interface.

Client/server computing attempts to leverage the capabilities of the networks used by typical corporations that are composed of many relatively powerful workstations and a limited number of dedicated servers. Client/server computing has gained popularity in the recent years due to the proliferation of low-cost hardware and the increasingly apparent truth of the theory that a model relying on monolithic applications fails when the number of users accessing a system grows too high or when too many features are integrated into a single system.

A good example of a client/server system is a simple automated teller machine (ATM) network. Users typically use ATMs as clients to interface with a small sub-server, which collaborates with a larger server that manages all of the smaller servers. In this model, the sub-servers are servers to the ATMs and clients to the master server. ATMs provide the user interface and can be customized (for multilingual support, as an example) as required, while the intermediate servers provide the application logic, such as checking on account balances and transferring money between accounts. The sub-servers allow the system to be scaled since adding servers allows an increased number of ATMs to be supported. However, the application logic can be provided only with the help of the centralized server. The results of these services are communicated to the user through the ATMs. The centralized server provides additional application logic, such as ensuring that concurrent transactions are handled correctly. It also serves as a central brokerage for all account information so that users can access their accounts from any ATM worldwide.

2.1 Clients

Many clients have a modern graphical user interface (GUI) that presents each resource accessible by the user as an independent object; the GUI is usually provided with the help of the OS so consistency across multiple applications is maintained. A typical example of a GUI is the common desktop metaphor in which each storage device, file, and printer is depicted as an independent entity. Clients can also offer services to other clients. Normal models of client/server computing place no limit on the number of servers a client can access simultaneously.

2.2 Servers

"Traditional" servers are entities that passively await requests from clients and then act on them, but recent research in this area encompasses systems fulfilling the theoretical organization of client/server systems in which servers can actively search out changes in the state of clients and take appropriate action. Servers typically fill one specific need and encapsulate the provided services so the state of the server is protected and so that the means by which the service is provided is hidden from the client. In order to accommodate workstation clients as first-rate network members, servers must handle peer-to-peer protocols that are used for "file sharing" on PCs, handle PC messages, and service PC resources using native formats. An important consideration in determining the granularity of services that a server will offer is the possibility of having servers act as clients to other servers. Using this model, a server can execute a task by dividing it into subtasks and then having other servers complete the subtasks.

2.3 Middleware

The distributed software required to facilitate client/server interaction is referred to as middleware. Transparent access to non-local services and resources distributed across a network is usually provided through middleware, which serves as a framework for communication between the client and server portions of a system. Middleware can be thought of as the networking between the components of a client/server system; it is what allows the various components to communicate in a structured manner. It is defined to include the APIs used by clients to request a service from a server, the physical transmission of the request to the network (or the communication of the service request to a local server), and the resulting transmission of data for the client back to the network. Middleware is run on both the client and server ends of a transaction.

Middleware is where most of the commercial competition and development in client/server computing has occurred. Examples of the proliferation of competing domain-specific standards include database middleware such as ODBC, SQL, and Oracle Glue; groupware middleware such as Microsoft Exchange and Lotus Notes; Internet middleware such as HTTP and Secure Socket Layer (SSL); and object middleware such as CORBA and DCOM. “Generic” or “fundamental” middleware forms the basis of client/server systems; the domain-specific middleware serves to harness the capabilities of the fundamental middleware for a more specialized purpose. Network authentication services, queues (such as for peripherals or tasks), network procedure calls, distributed file systems, and network time services are all considered part of the fundamental middleware. The most common network communication protocol used by middleware is TCP/IP, although IPX is also popular for some applications.[1] Fundamental middleware is becoming a standard part of modern operating systems such as WindowsNT, reducing the importance of systems such as Novell’s NetWare; this should help to standardize the availability of middleware.

2.4 Fat Servers vs. Fat Clients

Although clients and servers both play an important role in a successful client/server system, most systems are flexible with regard to the distribution of authority, responsibility, and intelligence. Information systems specialists dub a part of a system with a disproportionate amount of functionality “fat”; a “thin” portion of a system is a part with less responsibility delegated to it [OHE1]. The server portion of a client/server system almost always holds the data, and the client is nearly always responsible for the user interface; the shifting of application logic constitutes the distinction between fat clients and fat servers. Fat server systems, such as groupware systems and web servers, delegate more responsibility for the application logic to the server, whereas fat client systems, such as most database systems, place more responsibility on the client. Distributed object models of client/server computing are unique in that either the client or the server can be fat while still maintaining the approximately the same degree of flexibility and power. However, shifting intelligence from the client to the server or vice versa shifts the capabilities and strengths of the system. For example, if a fat server is being used, it usually is easy to update application logic since a new client does not need to be distributed. However, if fat clients are being used, the server need not be touched and system stability is not jeopardized.

---

[1] Most domain-specific middleware is designed for use with TCP/IP, and TCP/IP is supported by nearly all operating systems. Since Windows95 and the MacOS support TCP/IP, almost all desktop computer users have access to TCP/IP services.

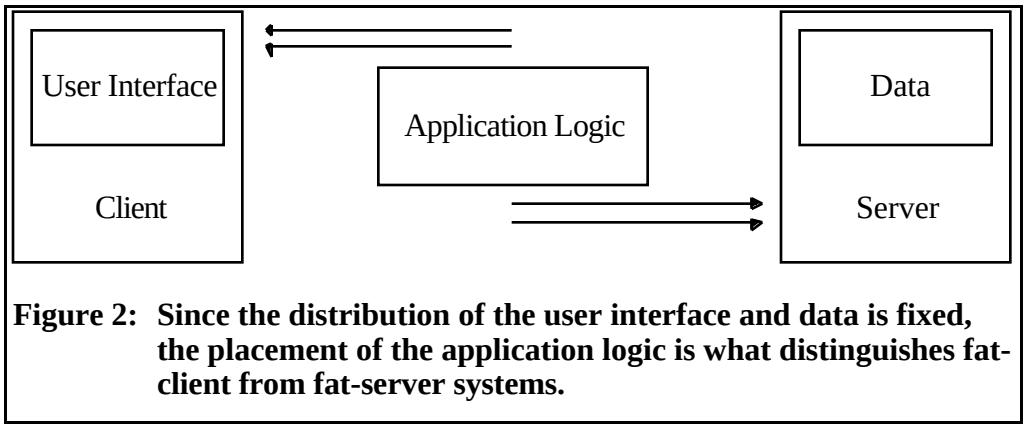

Figure 2: Since the distribution of the user interface and data is fixed, the placement of the application logic is what distinguishes fat-client from fat-server systems.

Although systems relying on fat servers have comprised the majority of client/server systems in the past, many programmers now favor fat-client systems because they are generally easier to program. Fat clients let users create applications easily and modify complex front-ends to systems easily but comes at the price of reduced encapsulation of data; as more responsibility is placed on a client, the client requires more intimate knowledge regarding the organization of data on the serving end. However, the continued development of fat-server systems has been significantly influenced by the industry trend towards greater object orientation, which favors a high degree of data encapsulation. By encapsulating the data better, more abstract services can be provided by the server, hiding raw data from the client. Instead of returning unprocessed data, meaningful responses to service requests are communicated back to the client by the server.

The use of fat servers has also increased because of their recently exploited efficiencies. Fat servers are generally easier to manage and deploy since the data and code exist in a centralized location. Instead of being coordinated across a network, debugging can all be done from one machine; unfortunately, as mobile servers and processes become the norm, this benefit will become less important. Fat servers reduce the problem of limited bandwidth by carrying out more of the work where the data resides, reducing the need for costly data transfers over the network. Mission-critical applications requiring the highest degree of fault tolerance and stability use fat servers for ease of maintenance.

The fat server model is often used to ensure greater compatibility between clients and servers: the more work the server does, the less dependent it is on the client. For example, a web page designed under the fat server model would assume that no plugins, ActiveX, or Java capabilities are available (because the user is using a thin client, a basic web browser) and would be restricted to the HTML 2.0 standard. Using this thin-client model ensures that all users see an

"acceptable" page in a predictable manner, although little flexibility for page design or functionality is possible without the use of the advanced technologies. If a more robust user experience is needed (in this example, this would be derived from using plugins and the like), the fat client model could be used at the expense of universal compatibility.

2.5 N-Tiered Systems

The canonical client/server model assumes exactly two discrete participants in the system. This is called a "two-tier system"; the application logic must be in the client or the server, or shared between the two. It is also possible to have the application logic reside separately from the user interface and the data, turning the system into a "three-tier system". In an idealized three-tier system, all application logic resides in a layer separate from the user interface and data. This rarely happens in actual systems; usually the bulk of the application logic is in the middle tier, but select portions of it are the responsibility of the client and/or the server.

The three-tier model is more advanced and flexible than the traditional two-tier model because the separation of the application logic from the client and the server gives application logic processes a new level of autonomy. The processes become more robust since they can operate independently of the clients and servers. Furthermore, decoupling the application logic from the data allows data from multiple sources to be used in a single transaction without a breakdown in the client/server model. This advancement in client/server architecture is largely responsible for the notion of distributed data.

Standard Web applications are the most common examples of three-tier systems. The first tier is the user interface, provided via interpretation of HTML by a browser. The embedded components being displayed by the browser reside in the middle tier; these could be Java applets, ActiveXs, or some other kind of entity that can provide the application logic pertinent to the system. The final tier is the data being served from a Web server. Quite often this is a database-style system, but it could be a data-warehousing or groupware system.

Many advanced applications can benefit from the use of more than three tiers. For example, when multiple data sources are being integrated (as in a data warehousing application), four possible tiers would be the individual data repositories, a server that unifies the view of this data, an application server that performs queries based on the unified view of the data, and the front end. The development of efficient and reliable systems with more than three tiers is still an imprecise science, but research in distributed computing is continuing to increase the availability and usefulness of such systems.

2.6 Functions and Benefits of Client/Server Systems

Although client/server systems are generally characterized by their components, they must serve essential functions in order to meet the demands of network computing. To prevent inflexible local system organization, a client/server system must keep parts of the application loosely coupled; of course, communication via message passing must continue to occur to support the granularity of communication (most often, method invocation) required between the client and the server. Access to shared resources must be regulated to handle cases in which the client and the server attempt to access the same resource (this occurs most often when the client and server are running on the same machine) or multiple clients attempt to access the same resource via the server. In addition, a client/server system must provide a standardized system through which network services can be utilized to provide location transparent access to all services.

As client/server systems have grown more robust, the computing community has acknowledged their many distinct advantages. Perhaps the most important is that many applications map naturally into a client/server framework. A typical example of this is an electronic phonebook system. Since the data is relatively static (meaning that it should be uniform for all users of the system) and the data repository needs to be able to respond to queries, it makes sense to construct this portion of the application as a server. A thin client is a logical match since it would be difficult to update every user's database of phone numbers, the optimal search algorithm could change at any time, and the space required for the amount of data being manipulated would be prohibitive for many user's workstations.

As a result of the availability of compatible middleware for multiple platforms and recent advances in binary interoperability, client/server systems can usually relate a client on one platform to a server on another. Technologies such as Java and object request brokers (ORBs) promise to provide seamless integration among all platforms within a few years. If portions of a client/server system encapsulate a single function and follow a rigid interface, the parts of a client/server system providing the services can be interchanged without upsetting other portions of the system. This allows users to customize their environments for maximum efficiency and system administrators to upgrade transparently to more powerful, capable, or less expensive servers without notifying the users (clients). Application development is simplified since a client and server each fill a specific need, and each properly designed server supports an interface directly related to the realization of one common goal.

Client/server models leverage the advantages of commodity-like hardware prices since resource-intensive applications can be designed to run on multiple low-cost systems. Systems can grow since client/server systems scale both horizontally and vertically, meaning that clients can be added with little performance penalty and that extra performance can be extracted from a client/server system by adding faster server hardware.

Despite maintainability issues arising from having data and code distributed throughout a network and the difficulties vendors have "keeping up" with competing standards, the client/server model is extremely well suited for many applications.

3. Distributed Objects and Components

As computing systems have evolved, new paradigms of constructing software applications have been developed and the paradigm of algorithmic computation has been replaced by the use of interacting objects.

3.1 From Objects to Distributed Objects

Classical objects can be viewed as self-contained entities that encapsulate data and a set of operations that can act on that data. The operations supported by an object (often called "methods") may rely on the internal state of the object. Objects provide a clean way of separating related data and functionality from other parts of a system in order to ease system construction and maintenance.

Whereas regular objects "reside" in a single program and do not even exist as separate entities once the program is compiled, distributed objects are extended objects that can reside anywhere on a network, continue to exist as a physical standalone entity, and can be accessed remotely by other objects. Robust distributed object systems allow objects written in different languages and compiled by different compilers to communicate seamlessly via standardized messaging protocols embodied by middleware. Such object frameworks allow higher levels of transparency of interoperability between distributed objects.

3.2 Benefits of Distributed Objects

Distributed objects promise to revolutionize the construction of scaleable client/server systems by providing modularized software that features interchangeable parts. Advanced architectures will offer the end user the ability to add components, allowing simple customization of applications.

Objects designed for self-management are the most easily utilized types of components since they impose little burden on the application programmer. Self-managing distributed objects take responsibility for their own resources, work across networks, and interact with other objects. These capabilities are frequently given to objects through a distributed object framework that provides middleware to regulate the inter-object communications necessary and provides a resource pool for each object that is deleted when the object ceases to exist. An example of such self-managing objects is ActiveXs; the DCOM middleware ensures interoperation between remote objects and handles resource allocation.

Self-managing objects are easily used by other objects since no management burdens are imposed on the client object: it is receiving object services “at no cost”. Objects crafted to these specifications can rely on a solid event model that allows objects to broadcast specific messages and to generate certain events. These events, which can be likened to an alarm signaling a human to take some action, can be “listened for” by other objects, which can then take action based on them. Each listening object can respond to a given event in its own manner. By using object-oriented techniques such as polymorphism, closely related objects can react differently to the same event. These capabilities simplify the programming of complex client/server systems and also help provide an accurate representation of the real-world system being modeled.

Objects can generate events to notify other objects that an action should take place. In this sense, events can be viewed as synchronization objects that allow one thread of execution to notify another thread of execution that something has happened. Using this model, an event can notify a component that it should take a certain action. Objects that can listen for events provide a more robust framework for interaction between objects than a model that forces objects to wait for the next instruction. For example, a word processor might generate a “finished printing” event when it has spooled a document to a printer. If someone wanted to add a display dialog alerting the user that the printing was done, an object listening for the “finished printing” event would need to be written; the word processor would never know of the existence of the alert object. Under a traditional model, the alert object would need to be explicitly told to show the dialog; this would require modifications to the word processor source code and subsequent recompilation, which is not always feasible.

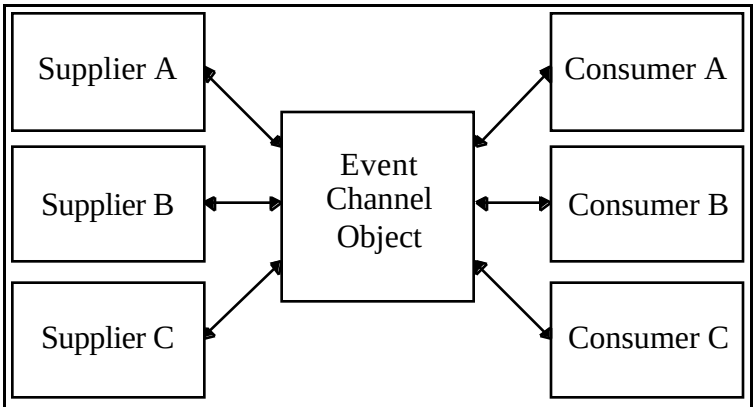

Figure 3: A sample event management system. Depicted is CORBA's Event Service, which is important because it extends CORBA's otherwise synchronous communication model.

Because of the strict encapsulation that objects provide, distributed objects are a fundamentally sound unit from which to build client/server applications in which separation of data is important. Cooperating objects form the logic portion of most substantial client/server systems since they offer rich interaction services [We1, We2]. The flexibility of granularity offered by components should not be overlooked. Objects can be as small as required to provide the correct degree of "mixing" of services, or as large and complex as required to encapsulate completely the logic of a particular system segment without unwarranted reliance on other objects.

Since distributed objects allow applications to be split up into lightweight pieces that can be executed on separate machines, less powerful machines can run demanding applications. Advanced applications for which distributed objects are well suited include roaming agents (autonomous objects that can logically or physically move within a network performing various tasks at specified locations) and objects that can adjust dynamically to their execution environment to provide optimal performance in any given situation.

3.3 Components

Components are the smallest self-managing, independent, and useful parts of a system that can work in multiple environments. Such parts may not even be objects; ActiveX controls (Microsoft's notion of a standalone software component that performs a common task in a standard way) are one such example. The promise of components is rapid application development

and a high degree of customizability for end users, leading to fine-tuned applications that are relatively inexpensive to develop and easy to learn.

Components are most often distributed objects incorporating advanced self-management features. Such components rely on robust distributed object models so as to maintain transparency of location and implementation. Components may contain multiple distributed or local objects and are often used to centralize and secure an operation. For example, a function provided on a Web page through JavaScript may be moved to a component to protect the logic of the application or to allow the operation to execute on a more powerful server.

The interface of a component should be the primary concern of its developer. Since components are designed to be used in a variety of systems and need to provide reliable services regardless of context, developers attempting to use a component need to be able to identify clearly what the component does and how to invoke the correct behavior from it.

4. A New Model for Client/Server Computing

Using distributed objects in client/server systems provides numerous benefits to end users and system developers. If distributed objects are to be used, choosing the most appropriate distributed object model and programming language is essential to leveraging the advantages of the client/server paradigm.

4.1 Client/Server Using Distributed Objects

Due to market forces, only CORBA and DCOM provide viable long-term solutions to the need for a standardized framework for object-based client/server systems, since other technologies do not support a large enough feature set or have as great an installed base and momentum.

4.1.1 Client/Server with CORBA

The computer industry has rallied to create an open standard for distributed object computing, known as the Common Object Request Broker Architecture (CORBA).[2] The most ambitious middleware undertaking ever, CORBA manages every detail of component interoperability, ensuring the possibility of interaction-based systems that incorporate components

[2] An open standard is traditionally a standard designed and supported by many companies that provides for interoperation between specific vendor's implementations of similar systems and allows continued innovation and development of the system by all companies working on the standard; the key benefits are ready availability of information about the standard and assurance that no one company will gain a significant competitive edge by having the "best" technology.

from multiple sources. Because an object's service specification is completely separated from its implementation, CORBA is able to provide a self-specifying system that allows the discovery of other objects on the network; this capability opens up new possibilities for interactive systems.

CORBA objects can exist anywhere on a network, and their location is completely transparent. Details such as the language an object is written in or the operating system it is currently running on are also hidden to clients. The implementation of an object is of no concern to an invoking object; the interface is the only consideration a client needs to make when selecting a serving object.

CORBA provides fundamental system and application services to the objects that rely on it for management. The inclusion of these services at the middleware level eliminates the need for most "mixin classes" (small classes providing generic functionality, such as garbage collection or persistence, to all objects inheriting from them), which tend to distort the object model and unnecessarily complicate the inheritance hierarchy of systems.[3]

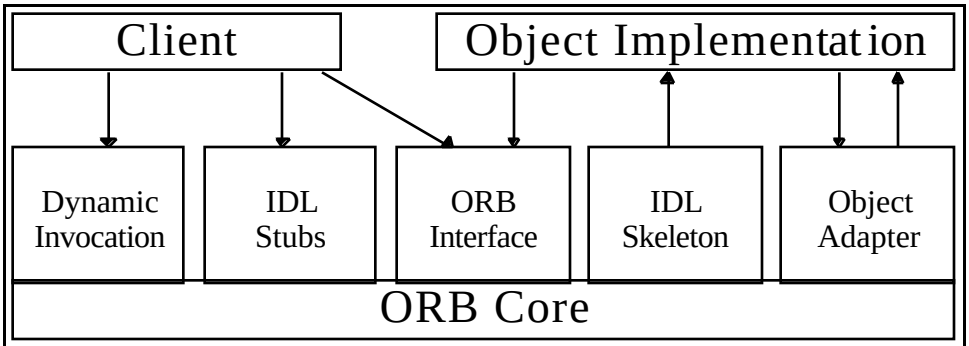

Figure 4: The CORBA architecture. Shown are the major ORB components used by the client and the implementation of the object.

The most important part of a CORBA system is the Object Request Broker (ORB). The ORB defines the object model and provides bidirectional location-transparent object access. The ORB is what shields clients from needing to deal with the complexities of remote object communication; the ORB handles all of the difficulties in coordinating the task. The CORBA 2.0 specification mandates intervendor ORB compatibility, which is accomplished via the required Internet Inter-ORB Protocol (IIOP). IIOP provides a common communication backbone between

[3] In fact, the CORBA model does use mixins to provide these services, but they do not interfere with the "traditional" object interface, since they are mixed in at run time. They could also be included at build time, but this limits the flexibility of the model.

different ORBs by adding several CORBA-specific messages to the TCP/IP schema already widely used today. The ORB provides most of the middleware-like services that a robust distributed object system should provide. Many CORBA features are drawn from proven models such as Remote Procedure Calls (RPC) and Message-Oriented Middleware (MOM).

CORBA uses the ORB to establish a client/server relationship between components. The ORB intercepts method invocations from client objects and routes them to an appropriate server. The serving component could be a specific object or a general server that can deliver the services required to meet the demands of a generic client request. By using an ORB with such capabilities, CORBA shields the programmer from implementation details (such as the language used to write that a cooperating component) as well as run-time variables (such as which machine is hosting a given component). The ORB does not bind a given component to a client or a server role: the same component can act as a client to other objects yet still deliver requested services, making it a server also.

At the heart of CORBA's strong interoperability is a language-neutral Interface Definition Language (IDL). The IDL, used to specify the services that an object can provide, was designed to be completely independent of the implementation language, tools, operating system, and other factors that normally affect interoperability. Important issues such as error handling are accounted for by this completely declarative language. Programmers write interactive sequences of code using native language constructs and rely on the IDL only to express object interfaces and related traits, such as its inheritance tree, the events that the object can trigger, and the exceptions that the object can raise (for error-handling purposes). When the interface is specified, the programmer specifies parameter and return types to ensure the appropriate invocation of the object's methods. To make the IDL easy to learn, it is a superset of C++; it even includes support for preprocessor directives.

```
module <identifier> {     /* define a naming context */
   <type declarations>;
   <constant declarations>;
   <exception declarations>;

   interface <identifier> [:<inheritance>]{ /* class definition */
      <type declarations>;
      <constant declarations>;
      <attribute declarations>;
      <exception declarations>;

      [<op_type>]<identifier>(<parameters>)/* method declaration */
         [raises <exception>] [<context>];
      ...
      [<op_type>]<identifier>(<parameters>)/* method declaration */
         [raises <exception>] [<context>];
   }

   /* more interfaces here */

}
```

Figure 5: The structure of a CORBA IDL file.

Client IDL stubs communicate the static interfaces that allow a client to be written as if it were simply invoking local methods; in reality, the ORB is routing the method invocations to the appropriate server objects. This flexibility comes at the expense of equipping each client component with an IDL stub for each server it wishes to use. In addition to static method invocation, CORBA supports dynamic method invocation, which is handled through the Dynamic Invocation Interface (DII). DII allows a component to learn about the methods of other components at run time. It serves as a generic interface that does not require stubs; instead, it supports the dynamic construction of method invocations as needed by the client at run time. Standardized methods support the transfer of data in a metadata format; however, the full utility of DII is yet to be exploited due to the complexities inherent in writing a component to exploit services that are not yet known to exist.

The server side of a client/server transaction does not need to know whether the method invocation it is handling is static or dynamic. Serving objects have Server IDL Stubs, similar to the Client IDL Stubs, that denote the static interface of the serving component. To accommodate components that do not have IDL-based stubs, CORBA provides a Dynamic Skeleton Interface (DSI) that binds incoming method calls for such objects at run time. Server demands are met via

an Object Adapter, which provides the core run-time functionality required by servers. It supports the registration of components, and can thus return object references to client components. The Implementation Repository gives server components a repository for run-time information about a variety of a server's services, such as the supported interfaces, audit trails, security details, and other administrative or "housekeeping" information.

Related to these two interfaces is the Interface Repository API, which allows components to obtain and modify the interfaces of registered components it has access to. The Interface Repository contains all of the IDL definitions, which describe the attributes, operations, user-specified types, and exceptions supported by server objects. Using the Interface Repository API, components can update their published interfaces, making CORBA a self-describing system. By supporting these flexible interface services, CORBA provides both the safety and speed of static method invocation as well as the flexibility afforded by dynamic method invocation.

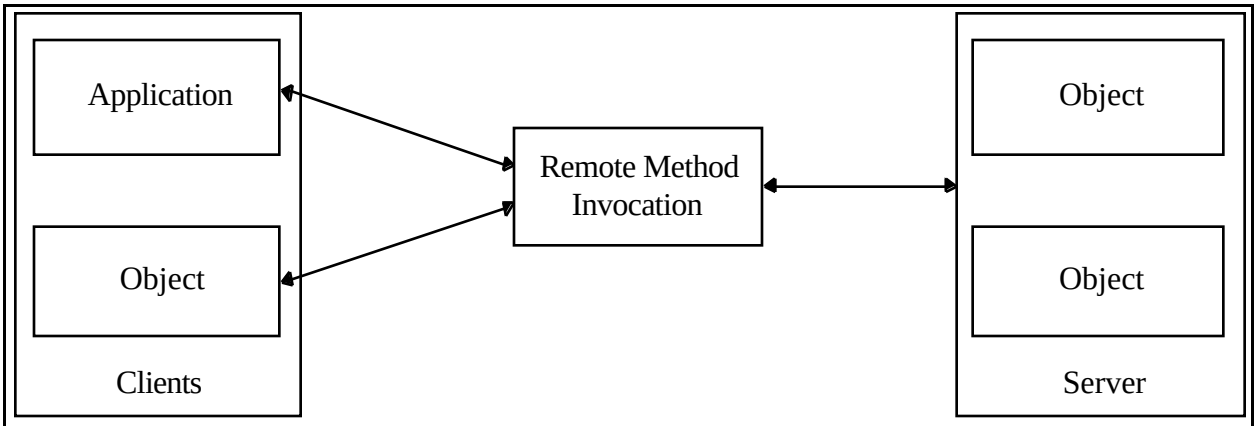

Figure 6: Client/server computing using distributed objects. Communication between components (denoted by arrows) is facilitated through ORBs (which have been omitted for clarity).

CORBA specifies two means by which an object can locate another object in a system. The first is the Naming Service. This service is analogous to the white pages in a phone book: an object looks up another object by the name under which the object registered itself with the ORB when it was initialized. This method of finding an object relies on unique signatures: a client must know the exact name a server gave when it registered to use it. The second service is the Trader Service, which is like the yellow pages: objects can ask the Trading Service which objects have registered with certain service characteristics. The trading repository then returns references to salient objects and gives the client information regarding the properties of the services. The client can then choose a server to contact for the needed services.

CORBA also provides CORBAservices, which define system level object frameworks that extend the CORBA model, and CORBAfacilities, which provide horizontal and vertical application frameworks used by business objects. Although quite important to the design and implementation of a successful CORBA-compliant system, these facilities are not important in the use of CORBA to support a robust client/server environment [OHE2].

4.1.2 Client/Server with DCOM

Microsoft is touting DCOM, which first shipped with WindowsNT 4.0, as the future model for Internet computing, manifesting itself primarily through the use of ActiveX components, which are DCOM objects.[4] Furthermore, software releases from Microsoft (such as Visual J++) reveal that Microsoft is intent on providing a workable platform on which Java objects can be implemented. The integration of Java with DCOM through Visual J++ is even greater than the integration of C++ and DCOM provided through Visual C++. This is particularly interesting in light of the fact that C++ is the language originally intended to be used with DCOM. DCOM has migrated from a system for binary interoperability to the more accepted (and needed) system of providing high-level bindings to popular languages. The bindings provided with Visual J++ are strong enough so that ActiveXs written in other languages can be made to look like remote Java objects.

DCOM is like CORBA in that it cleanly separates interface from functionality using an IDL. Microsoft has chosen to use an IDL based on the Distributed Computing Environment (DCE). The IDL is neither CORBA- nor DCE-compliant; this severely limits the potential for interoperability. In addition, separate interface functionality for Microsoft's Object Linking and Embedding technology is provided using the Object Definition Language (ODL).[5]

---

[4] Microsoft's WindowsDNA initiative strives to integrate DCOM further with Web-based client/server models.

[5] OLE no longer refers to Object Linking and Embedding, but rather to any COM-based technology. The ODL support for Object Linking and Embedding is to ensure backwards compatibility [Chap].

```
[ object, uuid(E7CD0D00-1827-11CF-444553540000) ]
interface ISpellChecker : IUnknown {
     import “unknown.idl”;
     HRESULT ChkWord([in] OLECHAR word[31],[out] boolean *found);
     HRESULT AddToDict([in] OLECHAR word[31]);
     HRESULT RemoveFromDict([in] OLECHAR word[31]);
}
```

Figure 7: Sample DCOM IDL code defining the interface for *ISpellChecker*. The compiler will generate proxies and stubs along with other related code. The non-standard parameter types are required by COM.

DCOM does not support the traditional notion of an object. DCOM “objects” do not have a state; rather, they are collections of interfaces. One could liken this to a collection of algorithms, suggesting that DCOM “objects” are inherently not as powerful computing machines as CORBA objects [We2]. When a reference to a DCOM object is requested, the handle is arbitrary; a subsequent request for the same object could yield a different handle connecting the client with an equivalent interface. DCOM supports a registry of available interfaces that objects can reference, and even provides information on the meanings and types of the parameters that should accompany a service request.

The client/server contract can be fulfilled in DCOM by using interfaces of objects. As in CORBA, DCOM interfaces make the implementation language and the location of objects transparent. This assumes that a language binding for the language in question is available. For a DCOM client to access the methods of an object, it must use a virtual lookup table to obtain a pointer to that function. There can be multiple virtual tables for each object and they can be modified so that interfaces can be altered while the application executes.

These differences from the traditional notion of an object may leave the reader wondering what a DCOM object really is. A DCOM object can best be described as a software component supporting at least one interface; no knowledge of state is provided. Since DCOM objects have no unique object identification, there are no naming or trading services provided by the DCOM runtime. Since individual objects cannot be referenced while they exist, persistence is not supported, limiting the problem domains for which DCOM objects are a viable solution. Furthermore, since objects have no way to be identified, a client object cannot request to be connected to a given server; it can only ask to be connected to an arbitrary server supporting the needed services. Since DCOM does not support multiple inheritance of interfaces (proven to be the purest model for interface composition in [We2]), DCOM objects cannot automatically support

multiple interfaces. If an object contains an object supporting a given interface, then the containing object can tell requesting clients that it supports that same interface; the calls need to be forwarded to the contained object. There are two ways in which this can be done:

- Containment/Delegation: The containing object is required to request the same service of its inner object as was requested of itself. The top-level object is delegating service requests to more capable objects.
- Aggregation: This model directly exposes the interfaces of the objects to requesting clients when they ask for the interface of the top-level object. Put another way, the top-level object presents the interfaces of its component objects as its own. This has the side effect of allowing a client to communicate directly with the objects contained by the top-level object. Performance and efficiency can benefit from such an approach, but some flexibility, control, and encapsulation may be lost.

By providing only for the functionality mandated by the notion of inheritance and thus ignoring inheritance on a conceptual level, Microsoft has limited users of DCOM to a "flat" object world. In addition, programmers are forced to use containment in place of inheritance to reuse code. This has a host of theoretical and practical ramifications, that are beyond the scope of this paper.[6]

When a DCOM client needs a server, it issues a request for an object supporting the required interface to be instantiated so that the interface is available. As a result, all DCOM servers must provide for a class factory supporting a specialized interface (IClassFactory) that can instantiate the class. In addition, the server has to tell the Windows registry which classes it supports; this is often done when the server component is first installed on a system. Numerous problems with Windows registries make this a poor model for advertising system services.[7] A major disadvantage of this model is that it does not accommodate "newly" supported clients nicely (for example, compatible clients).

[6] These issues should be treated adequately in most texts on software engineering and design.

[7] Most of these problems involve registry integrity. Because the Windows registry is used so often, it is very susceptible to corruption.

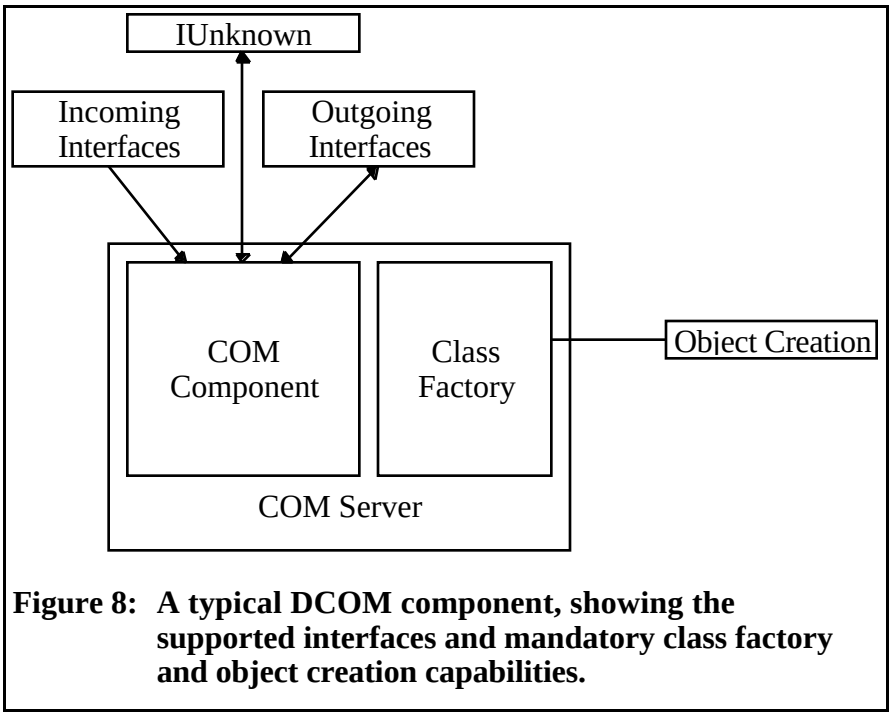

Figure 8: A typical DCOM component, showing the supported interfaces and mandatory class factory and object creation capabilities.

Using interface pointers, DCOM supports transparent communication between clients and remote server objects. The DCOM runtime does some behind-the-scenes work to ensure that the interface pointer is local to the invoking process (using a proxy, if necessary), but the view to the client is the perpetual availability of services. When proxies are created, they communicate with stubs on the serving end. This model is quite similar to the mechanism employed by CORBA to provide location-independent access to objects.

Although Microsoft has taken many liberties with the object model that should improve performance, benchmark tests indicate otherwise: benchmarks incorporating network communication and simple method invocation show DCOM to be almost 20% slower than CORBA.[8]

[8] A simple benchmark involved two 120-Mhz Pentiums running WindowsNT Workstation 4.0 connected by a 10 Mb/s Ethernet LAN. The client (written in C++) invoked a method on a server (also written in C++) 1000 times and measured the average response time of remote pings. The DCOM setup gave an average execution time of 3.9ms; the CORBA setup, which used the VisiBroker ORB, gave an execution time of 3.2ms.

4.1.3 CORBA as the Dominant Model

With DCOM running only on Microsoft operating systems, there is little question that CORBA is a more portable system.[9] CORBA is now in its third generation, and the standards body governing its evolution serves as assurance that it will remain an open system. In contrast, DCOM is currently a Windows-only technology (although some experimentation with porting the system to flavors of UNIX has been done) and is apparently governed by its maintainer, Microsoft. Several barriers to porting DCOM to other platforms exist. For example, DCOM relies on the WindowsNT security model to provide system security; it is unclear what will provide security when DCOM is used on other platforms. In contrast, CORBA uses a universal security mechanism that will work on all platforms, regardless of operating-system-level security.

CORBA provides a superior object model by supporting the foundations of the popular classical object model. By supporting unique object references, capabilities such as persistence are easily supported by CORBA, allowing a whole domain of specialized applications to leverage this strength to provide more robust end user services. Native state handling provided by CORBA frees the client from the burden of providing a managed-state system to handle the needs of interacting objects.

Support for legacy systems is essential to the success of any new computing technology. CORBA provides seamless integration for the most popular object oriented languages, and robust support for all languages. DCOM language support is limited to Java, C, C++, and Visual Basic. CORBA hides the IDL from the programmer, whereas an understanding of the DCOM IDL is essential to DCOM programming.

Although CORBA is clearly a superior technology, one should never underestimate any technology backed by Microsoft and its huge installed base and resource pool. However, the rest of the industry, including giants such as Hewlett Packard, IBM, Novell, and Apple, along with quickly moving newcomers such as Netscape, Oracle, and JavaSoft, are rallying behind CORBA to make it the distributed object standard.

4.2 Using Java For Client/Server Applications

Client/server systems developed using Java employ a model of interaction more advanced than that offered by other languages. Software systems developed in Java rely on applications that

---

[9] As of mid-1997, Microsoft had pledged to support DCOM on the MacOS and on the most popular UNIX variants.

have many component-like qualities. Using Java applications, one can distribute executable content as easily as one can deliver traditional static content.

Java supports a mobile code system; this makes it an excellent choice for meeting the needs of mobile agents, which need to share (or distribute) both code and data across multiple servers and clients. Mobile code is distinguished from traditional application code in that it is dynamically executed by an application that provides the run-time environment. In Java, the assisting application is quite often a web browser or some other stand-alone application, although operating systems are probably a better choice for this task. For code to be considered mobile, it must be portable, meaning the code can be run on a variety of platforms; code bound to a specific machine architecture or operating system is not very mobile. Since the security provided by the runtime varies, mobile code systems must provide security. Since Java applications can "roam" and be executed on any client machine and since we would like users to be able to download and execute programs without the threat of harm to their computing environment, the mobile code system must protect host resources such as memory and files. Other considerations for mobile code systems are mechanisms for loading, unloading, and discarding applications, and a way to transfer applications to hosts that guarantees their integrity.

Java's mobile code system is provided through bytecodes, which elegantly solves the portability and security problems. The bytecode system calls for programs to be "compiled" until they are at the point where they can be run on a Java Virtual Machine, which translates the bytecodes into actual machine instructions on the fly. Certain factors, such as the size of data types and the behavior of arithmetic operators, are standardized across virtual machines to ensure that a Java program executes in the same manner on any machine.[10] Before this translation and execution, the Java Verifier checks all code scheduled to be executed for malicious code (such as forged references or access violations). Security is augmented by four other levels of verification, ranging from native language design features to regulated access to system facilities.

Although Java is often compared to C++, it is quite different in many ways. Java supports only single inheritance of implementation, which solves many semantic problems arising in complex inheritance hierarchies, and allows objects to support multiple interfaces. Interface inheritance is not considered harmful and is generally recognized as providing a more clean and

---

[10] Unfortunately, this works better in theory than in practice; numerous incompatibilities between virtual machines still exist.

accurate object model. Java interfaces are quite similar to CORBA IDL files in that they both specify the services available from an object without revealing its implementation.

Namespace semantics are provided by Java packages. Packages allow the dynamic linking of classes with methods that can be overridden at runtime. Java also provides automatic garbage collection, array-bounds checking, does not allow pointers, enforces strong type checking, and treats exceptions as essential language constructs. Native support for multi-platform multithreading is provided; some simple features to provide for thread synchronization needs are also included. These language features assist developers writing client/server components that must interoperate easily and provide a high degree of security and reliability.

Much of Java's appeal comes from the core classes that are part of standard Java implementations. Since much of the application functionality is provided externally, application sizes are minimized. Since a lot of "busywork" code has already been written, debugged, and approved as multi-platform-compatible, program development is greatly simplified. Some of these pre-built classes are provided through packages such as java.lang, java.io, and java.util, while other parts come from established frameworks. Some of the core frameworks that are relevant to client/server computing include:

- Java Applet Framework: Provides the "basics" needed for Java applets (small applications meant to be downloaded from the web and usually executed within a web browser), such as base objects, native types, threads, exceptions, stream and file support, sockets, generic data structures (hash tables, stacks, etc.), a portable GUI layer (which supports events), animation, and wave audio.
- Java Commerce Framework: Provides secure monetary transactions, including online purchasing and financial management.
- Java Enterprise Framework: Provides object- and database-access services for distributed systems. Incorporated here are APIs that support Java-to-CORBA communication, Java-to-Java communication, and Java-to-Java-Database-Connectivity (very similar to ODBC) communication.
- Java Server Framework: Simplifies the development of Internet servers by providing APIs specifying simple and uniform access to servers. It also supports servlets, which are miniature servers that end users can deploy on a network to handle simple client requests.
- Java Media Framework: Supports 2D graphics and animation, synchronization services, and audio. It is being extended to support MIDI, 3D graphics, telephony, conferencing, and

video. Using the media framework allows applications with robust interaction semantics to work across multiple platforms with modification.

- Java Security Framework: Provides support for authentication, digital signatures, and encryption.
- Java Beans Framework: Serves as a component model native to Java. It extends the application model by allowing flexible event handling across multiple components, discovery of methods supported by other objects, persistent objects, and user interface allocation among multiple objects. This framework is quite important as it allows objects on a web page to communicate and occupy overlapping screen areas.

These frameworks suggest that Java is not simply a language, but rather it is a portable object platform.

By introducing a new way to develop applications, Java has created a new paradigm for managing and deploying client/server systems. By ensuring multi-platform compatibility, Java allows a developer to deliver an application to millions of users by making it available for downloading from a web server. A formal installation procedure usually is not required, and updates are provided by updating the single copy of the application on the web server. Server technology also stands to benefit from Java, most notably from the use of mobile code. With no modification to the application or its source code, servers can start processes on any machine on the network, and can move itself or its child processes to the most appropriate machine at any time. The potential for rapid application development also makes Java a good client/server programming language since effective client/server development requires constant feedback from end users.

4.3 Client/Server Using Java and CORBA

Java has transformed the World Wide Web into an interactive system supporting objects, but it is not a sufficient solution to the problem of creating transparently interoperating objects for client/server systems. However, when coupled with distributed object technology, it forms a strong basis for the development of a robust system that supports client/server computing. A platform for universal network computing can be created using Java as a programming language and mobile code system and CORBA as an integration technology.

CORBA can replace the current web-based system of providing client/server services, which uses HTTP/CGI. Doing this would bring three significant benefits to Java:

- Superior performance and flexibility: CORBA allows clients to invoke methods on a server directly, with parameters being passed by precompiled stubs or being generated dynamically. This system is more flexible than the system supported by HTTP since the client component is not limited to the predefined methods supported by HTTP; any IDL-defined method can be invoked. In addition, CORBA allows parameters of any type to be passed, whereas CGI accepts only strings as parameters. CORBA also avoids rerunning a program each time a client request for it is made, and can store state between client invocations; CGI requires executing the program for each request and does not support state.
- Scalability: The CORBA ORB can dispatch incoming client requests to a server of its choosing, allowing load balancing of client requests (this is further facilitated by inter-server communication via the ORB). CGI applications have no way of distributing client requests to other servers; one CGI application must handle all the requests sent to it.
- Component infrastructure: Java does not provide native support for method invocations across address spaces, meaning that Java applications cannot request services of remote Java objects. Using CORBA would allow communication not only among Java applications, but also among Java applications and components written in other languages. Of course, CORBA would also supply its standard component services, such as transactions and persistence.

Since it was designed to naturally support three-tier client/server systems, CORBA is a natural extension to Java's object model that brings robust distribution services to Java objects. Using CORBA with Java would allow Java components to be split into client and server side components in many cases, making the web client/server model even more attractive since download times would be decreased (only the client component would need to be downloaded).

Java can be used to build on the strengths of CORBA. By using CORBA to provide network transparency and Java to provide implementation transparency, a system that provides for total transparency of components can be created. Some ways in which Java improves on CORBA include:

- Simplified code distribution: Using Java, code is easily deployed from a central server to be downloaded on demand. This ensures that all updates are "installed" (instead of requiring system administrators to update each workstation manually).

- Mobile code: Using Java's native mobile code capabilities, functionality can be moved dynamically between machines or between the client and server components of a system.
- Agenting: Since it is assumed that every machine a roaming agent will visit will have a Java virtual machine installed on it, CORBA applications requiring agents can use Java's mobile code system to move behavior around, with CORBA providing state persistence services and Java providing the behavior.
- Additional component services: CORBA defines "visual containers" (based on Apple Computer's OpenDoc) for components, and Java applications can serve as the portable components that exist within these containers. CORBA's mobile container structure, based on OpenDoc's "Bento" system for storing structured data, can be used as a mechanism for moving Java agents around.
- Superior language features: Java is well suited for writing robust client and server code because of its native multithreading, garbage collection, and error management. Java works well with CORBA since both systems separate interface from implementation.

Java and CORBA complement each other quite well, with CORBA providing a distributed object infrastructure and Java providing a strong mobile code system. Together they let any two objects on the same network communicate under all circumstances.

5. Frameworks

One of the goals of object technology is to let programmers construct software systems in the same way as the analogous real-world systems are structured. Frameworks are a tool to help programmers achieve this goal.

5.1 What Are Frameworks?

Frameworks typically provide a way to manage a system of interacting objects and to develop objects that will integrate into the framework seamlessly. Frequently, frameworks will suggest patterns of collaboration between the objects that constitute the framework, although a well designed framework will allow flexible channels of collaboration that suit the application at hand. Components that emerge from a finished framework share consistent design attributes, and may even share common implementations. The benefit is more maintainable and consistent software systems.

In contrast to the traditional approaches to software reuse, which are built on the paradigm of libraries containing many small building-blocks, object-oriented frameworks allow the highest common abstraction level between a number of similar systems to be captured in terms of general concepts and structures. The result is a generic design that can be instantiated for each object system constructed.

Frameworks are ideally suited for capturing the elements common to a family of related systems. In this sense, the framework is essentially a large design pattern capturing the essence of one specific kind of object system. The bulk of the functionality of the system can be captured in the framework, which is maintained as a single entity. Each software system that uses the framework is an instantiation of the framework.

Frameworks provide a high degree of design reuse for interactive systems composed of collaborating objects and ensure that "products" of the framework will work within it. Thus, frameworks are a valuable tool for ensuring the availability of object services.

5.2 Business Objects as a Client/Server Framework

Business objects are self-managing components that can be used to represent key objects or processes in a real-life system. Business objects are "shippable" products that usually have a user interface and the ability to cooperate with other objects to meet a certain user need. They can be used across single or multiple enterprises. Business objects allow application-independent concepts to be described at a high level, minimizing the importance of languages, tools, and application-level concepts. Business objects represent a major breakthrough in the modeling of business events since they can describe both a portion of a real-world business system and the executing piece of the information system that supports that portion of the business [OHE1, OHE2].

Perhaps the most significant advantage of using business objects is the capability to model accurately the corresponding real-life business processes. Collaboration among business objects is essential to most robust systems since few business events involve only one object. For example, billing a customer involves the invoice, the purchaser, the goods being sold, and the seller. While some systems are relatively easily modeled using basic objects, business objects allow collaboration at a higher degree of semantic accuracy, reducing application development costs. When reengineering a business's computer systems, the result of the modeling should be a high-level group of components that can be immediately configured to run distributed across a network [OHE2]. With this infrastructure in place, modeling any event affecting the business simply

requires the instantiation of a new business object. For example, if an order is placed, a new invoice object needs to be instantiated. This new object then has its own life-cycle and could support itself by collaborating with other objects using the underlying ORB. (It should be noted that “business objects” need not apply to business processes. Any system relying on concrete objects can be modeled using business objects. A more accurate name would thus be “domain objects”; however, the mainstream client/server community has adopted the term business objects already.)

Like other components, business objects should support late binding, so that they can be interchanged easily and immediately interact with existing components, and they should also support the standard component features, such as event handling and state maintenance. Business objects are relatively easy to develop, since they can be based on CORBA objects, which already provide a means for inter-component collaboration and resource management. By using CORBA as a framework for constructing business objects, much of the work of building a robust component is eliminated from the development cycle.

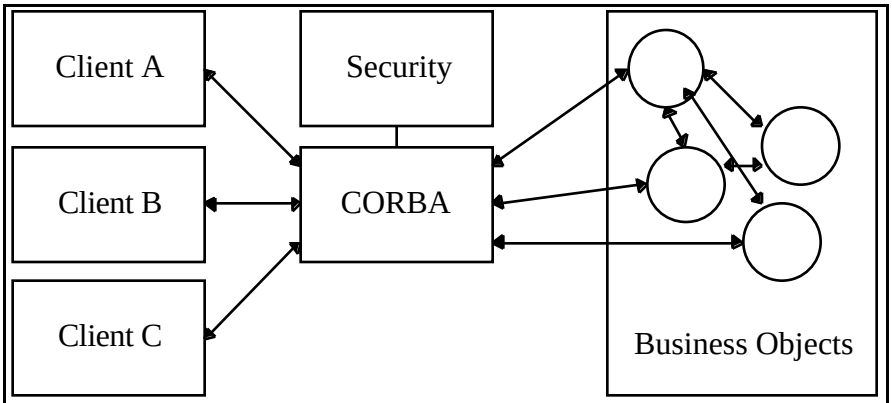

Figure 9: CORBA and business objects work together to provide client services.

Business objects provide the same benefits to system developers that traditional objects do; however, the benefits are specialized to a particular domain. Business objects can be reused, so the same invoice object could be used in both the accounts receivable and shipping portions of a company’s order management system. In fact, the component could be shippable to the purchaser of the goods and work with the accounts payable module of the purchaser’s computer system. Business objects can be specialized to meet the unique demands of a business. A business could purchase an accounting package and then specialize the invoice object to accommodate some special need, such as verifying that ordered items are compatible.

The Business Object Model Special Interest Group (BOMSIG) has proposed a standard for business objects. The standard calls for each business object to be composed of three types of cooperating objects:

- Business Logic Object (BLO): Defines how the object reacts to certain events and is responsible for the business logic of the component as well as storing the relevant business data.
- Business Process Object (BPO): Helps maintain the business logic for the entire system. The primary difference between a BPO and a BLO is the logical lifetime of the unit of logic: BPOs traditionally handle long-lived processes or processes related to the system as a whole.
- Presentation Objects: Provides the user with a representation of the component, that is usually but not necessarily visual.

A normal business object is likely to have multiple Presentation Objects, but should usually have one BLO and BPO. Because these three objects are managed by one object, collaborating components see only one object that provides the aggregate services of its constituent objects.

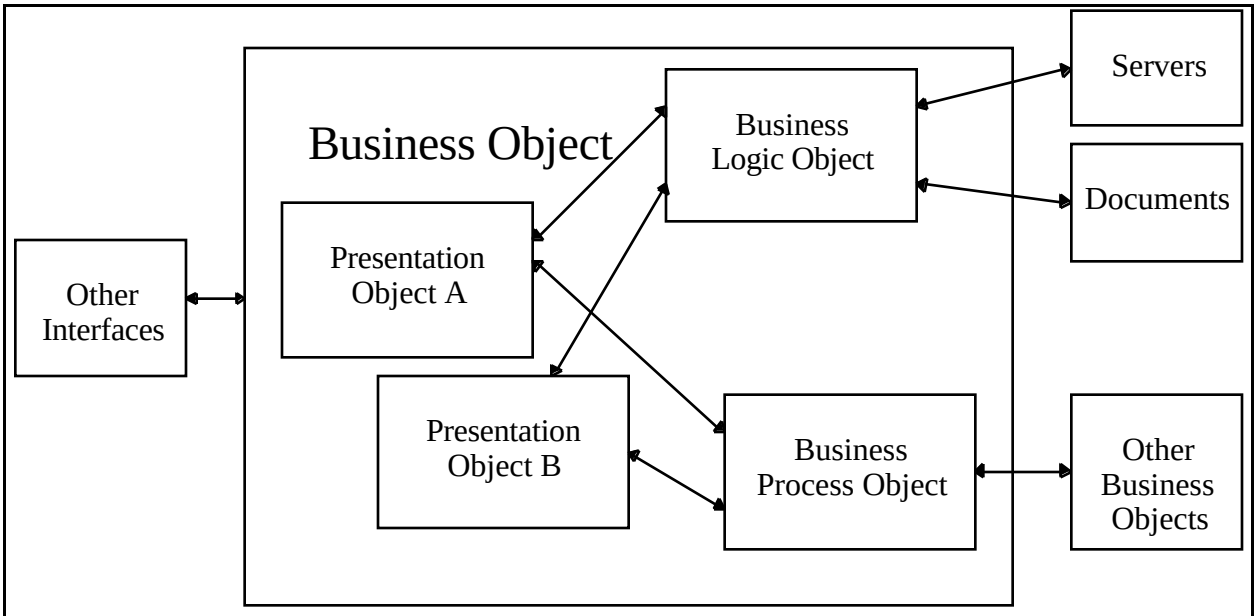

Figure 10:The parts of a business object and their communication with other system objects. [OHE2]

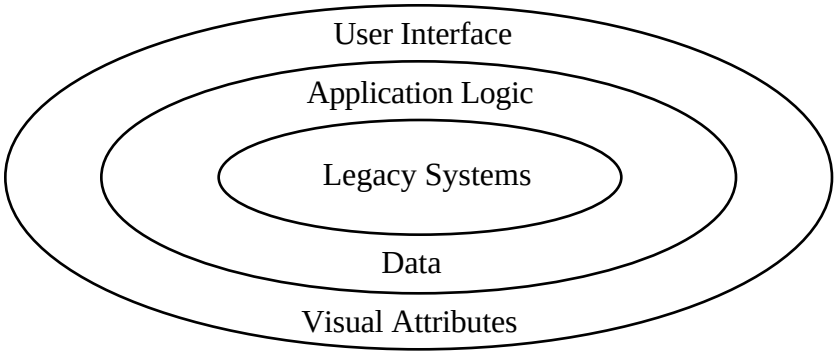

Figure 11:Three tiers in a business object.

This three-object construction can be viewed as a three tier client/server system:

- Tier 1: Visual aspects of a system, usually handled by a client system.
- Tier 2: Data for the object and the application logic required to meaningfully act on it.
- Tier 3: Data and application logic required to integrate the business object with other business objects and existing systems, such as legacy servers or databases.

The middle tier plays the largest role in this organization scheme. Tier-two objects communicate directly with the tier-one objects to provide feedback to the user and they also provide the logic for the entire business object. Furthermore, they communicate with multiple data repositories (tier three) and collaborate with other business objects in order to provide their services. This model separates the client from data for which it is not logically responsible. By channeling all requests for information through the tier two servers, major changes (such as the implementation of a new database system) remain completely transparent to the user. If ORBs are used for communication between the clients and the tier two objects, robust system services such as load balancing and event exchanges can be easily implemented and applications remain scaleable.

The business object model works well with CORBA because CORBA allows the constituent objects to reside on any machine. For example, the objects managing the business logic could reside on a server, while the client-side Presentation Object runs on a local workstation. This setup provides an inherent client/server relationship between components. Business objects can collaborate in a semantically and technically rich manner by using the facilities provided to every CORBA component. By leveraging CORBA's frameworks for data transfer and collaboration and the powerful mobile code system provided by Java, business objects can interoperate in a stable

manner, while allowing components to be added to or removed from the system. By keeping the framework for collaboration fixed, the object lifecycle can continue without affecting end users.

5.3 Compound Documents as a Client Framework

A compound document is a tool for organizing components that serves as a framework for integrating the visual and containment relationships between cooperating components. Compound documents can be composed of components from a wide variety of sources, making them universally applicable. The familiarity of documents to end users makes compound documents an obvious choice for introducing object technology on a widespread basis. The extension of the desktop metaphor, which incorporates application services and operating system services to provide a seamless user experience, has demonstrated the viability of such compound document systems.[11]

The compound document framework calls for containers that can contain components. The containers themselves are components; this allows recursive system construction. That is, a spelling-checker "container" could be constructed from dialog boxes, buttons, a dictionary file, and a database engine. This container could then become a component in a word processor, along with components for stylized text, graphics, grammar checking, and the like. Components can represent traditional programming entities or common data types. In fact, any data type can be extended to become a component. By melding each data type into a component mold, compound documents can accommodate all user demands, since they can contain anything the user would like. Since data is managed by its containing component, new data types can be added immediately to documents; there is no need for the document to know about the specific data type.

[11] The most notable compound document system is Apple's now-canceled OpenDoc project; many of the concepts and principles of current compound document models are derived from it. Interested readers are encouraged to examine some of the literature available both online and in print. For a fairly detailed introduction to OpenDoc, see [FM].

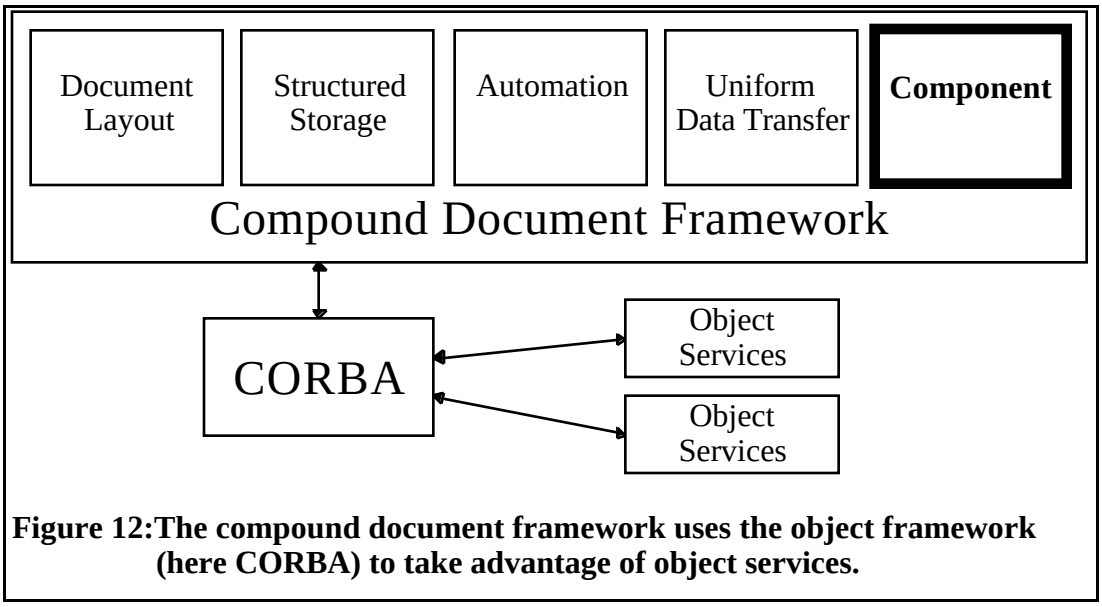

Figure 12:The compound document framework uses the object framework (here CORBA) to take advantage of object services.

In order to allow components with no pre-existing knowledge of other components to collaborate within a compound document, a system of communication is required between the components managing a document's data and the component managing the document itself. The component framework provides this communication and manages shared resources, such as files and windows on the user's screen. A compound document framework is built on top of an object bus, such as CORBA, which is built from object services. The framework provides four core services to objects serving as run-time members of the framework: [OHE2]

- Document Layout: This service allows independent components to share a common display area (usually a window). Components are required to collaborate via the document layout service to present themselves to the user. Once a display area is allocated, components that are also containers must recursively allocate space to the components that they contain. Each individual component is responsible for interacting with the user and displaying its data in the space allocated to it. Containers distribute events to the components they contain and manage the resources they share.
- Structured Storage: Since compound documents consist of many pieces of data managed by separate components, the storage needs of the framework are unique. A file storing a compound document consists of separate blocks of data, each managed by a component. Much like the display, files must be partitioned into spaces that each component can control without adversely affecting other components. A recursive structure similar to that used by

the document layout manager is employed. In addition to containing embedded data, files can also contain links to external data.

- Automation: Automation allows users to create their own relationships between components and customize their documents to create robust applications. Scripts can be used to replace traditional code, allowing the client component of a client/server system to be implemented as a compound document; intelligent active compound documents can also be crafted using scripts [We1]. The possibilities for scripts are wide-ranging: password protection for documents, personalized document views, and active data gathering are just a few examples. Using scripts in conjunction with compound documents allows users to create documents that can manage themselves and be self-sufficient. Scripts can be activated upon user request or in response to the occurrence of a document or system-level event.
- Uniform Data Transfer (UDT): Current user environments allow data transfer via the clipboard, linking, and drag-and-drop; compound documents must do likewise, while taking into account the added technical complexities of the new metaphor. UDT provides a single mechanism that works with the transfer mechanisms mentioned and intelligently handles containers by copying not only the container itself but also its components.

Compound documents offer enhancements to older user interface metaphors. Since any component can be immediately switched to without launching a different application (usually by clicking on it), editing multiple components “simultaneously” is not as burdensome as the current model, which requires users to switch applications. This metaphor allows components to give the user the tools required for the contained data in a context-sensitive way. This improves usability by allowing things such as positioning palettes next to the data being edited instead of at some “default location”. The components of a document are further integrated by eliminating the need for separate files for each component of the document. Everything contained by a document is stored within a single file (links to the components could be maintained instead) so that the user need not track the individual data items separately.

As compound document frameworks become embedded more seamlessly into the operating system, everything a user deals with will be considered a component. Initially, the transition from the WIMP user interface metaphor will not be apparent. Users will still move folders, and the files in them will follow. But users will truly be moving components, which have

more intelligence than today's simple folders (which are really just icons representing directories). They will be able to collaborate to share resources and to accomplish complex tasks together.

Compound documents have the potential to usher in a new era of mobile documents. Mobile documents can be edited, printed, and shared just like regular documents. Their added power comes from the ability to exploit the compound nature of the document. Since documents can contain executable code, they can perform functions, such as routing themselves automatically. In addition, the document can contain not only the salient data, but also the user interface for the document, a record of changes to it, a method for controlling who can access which portions of it, and other advanced functionality.

The compound document model can be extended to create new user environments. As compound documents begin to replace the traditional desktop metaphor, collaborative environments tailored to real-world models can be constructed. Different environments for children, doctors, and salespeople can form the basis of a person's interactions with local and remote services. Creating such a virtual environment requires merely assembling a collection of components into a container document, so the environment can be easily downloaded by users who do not wish to craft their own environment. To accommodate fee-based service access, companies can charge to download an environment. Only the environment would contain the components needed to transact with the remote components, so non-paying consumers would be unable to access the service. Once an environment has been constructed, it can serve as the client for all network services, including access to business objects.

Compound documents provide a framework for Web browsing, being a visual component foundation for a new generation of open Web browsers as well as a concrete container metaphor for storing and distributing data. By downloading a pre-built compound document, a user can have access to an entirely new and customized operating environment. Using compound documents in place of the older models of Web browsing has several distinct advantages. Compound documents can integrate the visual experience by allowing components on a Web page (including Java applets or applications and traditional content) to share window real estate and allow users to edit a component in place, without a separate application. New documents can be easily created by combining high-level components from existing documents in a new way. The metaphor would also allow a user to rearrange a Web "page" to his or her tastes. In addition, the components would automatically share common menus, palettes, colors, and clipboards so the user experience would be seamless. Components could be dragged onto other components, or components could

realize that they are in the same document and work together to take some action. Once web documents become componentized, new methods of displaying information will abound.

Because of its unique advantages, the compound document model is likely to become a dominant force on the client side in the growing client/server market. The model provides a convenient way of grouping related objects and manages the difficulties of displaying them, storing them, and moving them via networks. Since a compound document can maintain links to servers, it can serve as a universal client. Client applications can be built with ease by developers or end users by incorporating ready-made components and pre-written application logic, and then customized for each individual user. The user's entire workspace can be stored in one document if desired.

6. Conclusions

Distributed objects promise to revolutionize the stagnant client/server market. CORBA's object references provide a clean way of gaining an object's interface. Callbacks allow servers to control clients and allow clients to receive new content to be added to compound documents. In addition to being quite fast, the CORBA ORB is interoperable with C++ objects and integrates smoothly with Java. By supporting a three-tier client/server system, CORBA allows data from multiple sources to be encapsulated and allows pools of servers to be created. The provisions for dynamic discovery of object interfaces make CORBA components self-describing, allowing flexible binding and easy interoperation between components. The development of ubiquitous middleware available on all platforms will lead to true location transparency, and the fact that CORBA is an open standard will ensure continued innovation and evolution of the system.

Leveraging CORBA and Java provides the most notable advantages of component technology in the domain of client/server computing. CORBA works well with Java applications, which make very good downloadable clients because of their small size and Java's mobile code system. Using just-in-time (JIT) compilers, Java applications can deliver acceptable speed for all but the most demanding of client/server applications, and as processors specialized for Java are introduced, this performance will increase further. The Abstract Window Toolkit (AWT) provided with Java allows multipanel applications to be easily created and deployed under a variety of windowing systems with no need for rewriting UI-specific code.

Business processes can be modeled naturally and efficiently using distributed objects. The interactive and collaborative relationships between processes or divisions of businesses can be

modeled especially well by business objects. The potential for the reuse of key portions of business objects can decrease development costs substantially, and the mapping of business objects into a three-tier client/server system brings many advantages in the areas of efficiency and the transparent integration of data.

Compound documents can serve as a new metaphor for the client-side of client/server applications and can be adapted to be the next generation of web browsers. By leveraging the capabilities of cooperating components and the component management facilities of compound documents, a new generation of active documents can change the way in which client/server applications are viewed. The capability for interaction among components within a document will drastically change the limits of client-side data manipulation. Compound documents can also form the basis for dynamic customizable user environments crafted from components.

It is clear that components can usher in a new wave of client/server computing that will bring new capabilities to the masses. The only question that remains is whether the superior technologies will reign over Microsoft's well-funded DCOM and how soon ORBs will be integrated at the operating system level, creating a market environment conducive to the development of many interactive components.

7. References

[AG] Arnold, Ken and James Gosling. *The Java Programming Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1996.

[Bro] Brockschmidt, Kraig. *Inside OLE*, 2e. Redmond, Washington: Microsoft Press, 1995.

[Chap] Chappell, David. *Understanding ActiveX and OLE.* Redmond, Washington: Microsoft Press, 1996.

[Dol] Dolgicer, Max. "Inside CORBA Services." *Application Development Trends*, June 1997, 63-71.

[FM] Feiler, Jesse, and Anthony Meadow. *Essential OpenDoc*. Reading, Massachusetts: Addison Wesley Developers Press, 1996.

[GHJV] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Publishing Company, 1995.

[GM] Gosling, James and Henry McGilton. *The Java Language Environment: A White Paper*. Mountain View, California: Sun Microsystems, Inc., 1995.

[GO] Guttman, Michael and Viktor Ohnjec. “Distributed Object Computing: The Brave New World.” *Application Development Trends*, June 1997, 29-36.

[JCJO] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use-Case-Driven Approach*. Reading, Massachusetts: Addison Wesley Publishing Company, 1992.

[OHE1] Orfali, Robert, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide.* New York: John Wiley & Sons, Inc., 1996.

[OHE2] Orfali, Robert, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons, Inc., 1996.

[RBPE] Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1991.

[We1] Wegner, Peter. *Frameworks For Active Compound Documents*. Providence, Rhode Island: Brown University Department of Computer Science, 1997.

[We2] Wegner, Peter. “Why Interaction is More Powerful Than Algorithms.” *Communications of the ACM*, May 1997, 80-91.