Windows Open Services Architecture (WOSA)

The Windows Open Services Architecture (WOSA) was developed by Microsoft to "...provide a single, open-ended interface to enterprise computing environments." The concept of WOSA is to design a way to access extended services from the Windows operating system that require having only a minimum amount of information about the services. For example, the MAPI (Message API) model is designed to allow programmers to develop applications that use the message services without having to understand the complexities of the hardware and software routines that implement messaging on various Windows platforms.

The WOSA model goes beyond the idea of exposing services in a uniform way across Windows operating systems. WOSA is also designed to work in a mixed operating system environment. For example, the Microsoft Rpc (Remote Procedure Call) interface is a WOSA service that is designed to work with the Open Software Foundation's DCE (Distributed Computing Environment) Rpc model. The design of Microsoft Rpc allows programmers to design software that will safely interface with any product that uses the DCE model, regardless of the operating system with which the software must interact.

In order to attain this flexibility, the WOSA model defines two distinct interfaces-the Client API and the Server SPI. These interfaces are linked by a single interface module that can talk to both API and SPI applications. As a result, all client applications need to do is conform to the API rules and then to the universal interface. All server applications need to do is conform to the SPI rules and then to the universal interface. No matter what changes are made to the client or server applications, both software modules (client and server) will be compatible as long as they both continue to conform to the API/SPI model and use the universal interface.

The WOSA Model

The WOSA model consists of three distinct pieces. Each of these pieces plays an important and independent role in providing programming services to your applications. The three WOSA components are:

- The Client API-the application programming interface used by the program requesting the service.
- The Server SPI-the service provider interface used by the program that provides the extended service (for example, e-mail, telephony, speech services, and so on).
- The API/SPI Interface-the single module that links the API and SPI calls. This is usually implemented as a separate DLL in the Windows environment.

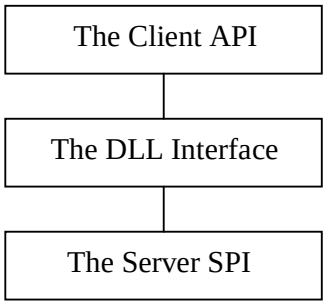

Figure-1: WOSA Model

Each of the components has an important job to do. Even though they perform their tasks independently, the components work together to complete the service interface. This is the key to the success of the WOSA model-distinct, independent roles that together provide the whole interface.

The Client API Makes Requests

The Client API is the interface for the application requesting the service. API sets are usually implemented at the Windows desktop. The Message API (MAPI) is a good example of a WOSA client API. Each client API defines a stable set of routines for accessing services from the back-end service provider. For example, the operations of logging into an e-mail server, creating an e-mail message, addressing it, and sending it to another e-mail client are all defined in the MAPI set. These services are requested by the client. The actual services are provided by the server-side application.

The key point is that the client application interface allows programs to *request* services from the server-side service provider but does not allow the client software to access the underlying services directly. In fact, the request is not even sent directly to the server-side application. It is sent to the DLL interface that sits between the API and SPI.

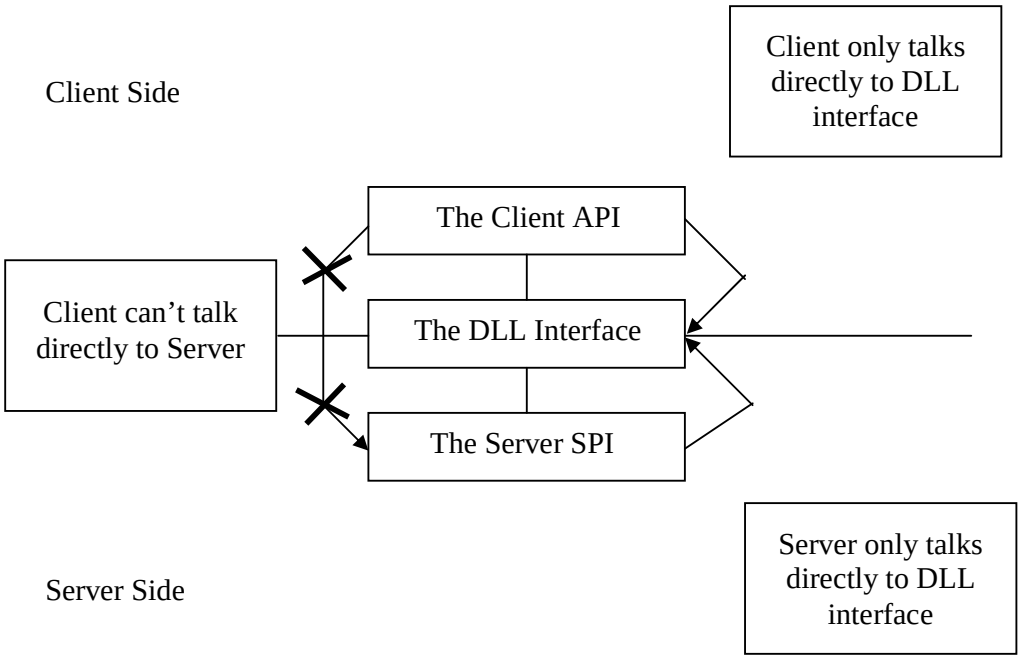

Figure-2: Communication between Client API & Server SPI

The Server SPI Responds to Requests

The Server SPI (Service Provider Interface) accepts requests for services and acts upon those requests. The SPI is not designed to interface directly with a client application. Most SPI programs are implemented on network servers or as independent services running on desktop PCs. Users rarely interact with these service providers, except through the client API requests.

A good example of an SPI implementation is the Open Database Connectivity (ODBC) interface. Even though programmers use API calls (or some other method of requesting ODBC services) in their programs, these calls merely request services from an external program. For example, ODBC calls to Microsoft's SQL Server are simply requests to SQL Server to perform certain database operations and to return the results to the client application. When making an ODBC request to SQL Server, the client actually performs very few (if any) database operations. It is SQL Server that performs the real database work.

As mentioned earlier, service providers rarely display interfaces directly to client applications. Their job is to respond to requests. These requests do not even come directly from the client program. In the WOSA model, all requests come directly from the interface DLL. The SPI talks only to the interface DLL. Any information that the SPI needs to supply as a part of the response to the request is sent directly to the interface DLL. It is the DLL's job to send the information on to the client that made the initial request.

Another important point should be highlighted here. The service provider portion of the WOSA model allows for multiple clients to request services. The DLL interface tells the SPI which client is making the request. It is the SPI's responsibility to keep the various clients straight. SPIs must have the ability to handle multiple requests from the same client and from multiple clients.

The Interface DLL Talks to Both the API and SPI

Since a key design aspect of WOSA is the isolation of the client API and the server SPI, a single interface between the two components is required. This single interface is usually implemented as a Windows dynamic link library (DLL) that allows programs to link to existing services at run-time instead of at compile time. The advantage of using DLLs is that programs need not know everything about an interface at compile time. Thus, programmers can upgrade DLL modules without having to recompile the applications that access the interface.

The DLL works as broker for the requests of the client API and the responses of the server SPI. The DLL does not actually perform any real services for the client and makes no requests to the SPI.

Note
Actually, the interface DLL may request basic information from the SPI at startup about the underlying SPI services, their availability, and other information that may be needed to support requests from clients.

The interface DLL is the only application in the Windows environment that actually "speaks" both API and SPI. It is the DLL's job to act as translator between the client and server applications.

In the past (before WOSA), these DLLs were written as translators from the client API directly to a specific back-end product. In other words, each DLL interface understood how to talk to only one back-end version of the service. For example, early implementations of the MAPI interface involved different DLLs for each type of MAPI service provider. If a program needed to talk to a Microsoft MAPI service provider, the MAPI.DLL was used as an interface between the client and server. However, if the program needed to talk to another

message server, another DLL had to be used to link the client request with the back-end provider.

In the WOSA world, interface DLLs can speak to any service provider that understands the SPI call set. This is an important concept. Now, a single interface DLL can be used for each distinct service. This single DLL is capable of linking the client application with any vendor's version of the service provider. This is possible because the service provider speaks SPI rather than some proprietary interface language.

WOSA Services

Microsoft has been championing the WOSA model for several years and promotes three types of WOSA services:

- Common Application Services
- Communication Services
- Vertical Market Services

Each type has its own purpose and its own core of services. The following sections describe the WOSA service types and give examples of services currently available for each.

Common Application Services

Common Application Services allow applications to access services provided by more than one vendor. This implementation of WOSA focuses on providing a uniform interface for all Windows applications while allowing programmers and/or users to select the vendor that provides the best service option for the requirement. In this way, Microsoft can encourage multiple (even competing) vendors to provide their own versions of key service components for the Windows operating system.

By defining a single set of APIs to access the service, all third-party vendors are assured equal access to the Windows operating environments. Since the interface is stable, vendors can concentrate on building service providers that expose the services that customers request most often. These vendors can also be confident that, as Windows operating systems change and evolve, the basic model for service access (the WOSA model) will not change.

The list of Common Application Services available for Windows operating systems is constantly growing and changing. Here is a list of some of the services provided under the WOSA model:

- *License Service Application Program Interface* (LSAPI) provides access to software license management services.
- *Messaging Application Program Interface* (MAPI) provides access to e-mail and other message services.
- *Open Database Connectivity* (ODBC) provides access to database services.
- *Speech Application Program Interface* (SAPI) provides access to speech and speech recognition services.
- *Telephony Application Program Interface* (TAPI) provides access to telephone services.

Communication Services

Communication Services provide access to network services. This set of WOSA services focuses on gaining uniform access to the underlying network on which the Windows pc is

running. The Communications Services also provide uniform access to all the network resources exposed by the underlying network. By defining a universal interface between the pc and the network, Windows applications are able to interact with any network operating system that conforms to the WOSA model.

The following list shows some examples of WOSA implementations of Communication Services:

- *Windows SNA Application Program Interface* provides access to IBM SNA services.
- *Windows Sockets* provide access to network services across multiple protocols, including TCP/IP, IPX/SPX and AppleTalk.
- *Microsoft Rpc* provides access to a common set of remote procedure call services. The Microsoft Rpc set is compatible with the Open Software Foundation Distributed Computing Environment (DCE) model.

Vertical Market Services

The last category of WOSA services defined by Microsoft is Vertical Market Services. These are sets of API/SPI calls that define an interface for commonly used resources in a particular vertical market. By defining the interfaces in this way, Microsoft is able to work with selected vertical markets (banking, health care, and so on) to develop a standard method for providing the services and functions most commonly used by a market segment. In effect, this allows users and programmers to invent Windows-based solutions for an entire market segment without having to know the particular requirements of back-end service provider applications.

As of this writing, Microsoft has defined two Vertical Market Services under the WOSA umbrella:

- *WOSA Extensions for Financial Services* provide access to common services used in the banking industry.
- *WOSA Extensions for Real-Time Market Data* provide access to live stock, bond, and commodity tracking data for Windows applications.

Benefits of WOSA

There are several benefits for both users and programmers in the WOSA model. The three key benefits worth mentioning here are

- Isolated Development
- Multi-vendor support
- Upgrade protection

The next three sections describe the details of the benefits of the WOSA model as it relates to both client and server programmers and application users.

Isolated Development

In one way or another, all WOSA benefits are a direct result of the model's ability to separate the details of service providers from the application running on users' desktops. By keeping the details of hardware and software interface locked away in the SPI-side, programmers can concentrate on providing a consistent interface to the services, rather than concerning themselves with the low-level coding needed to supply the actual services.

The isolation of services from user applications has several other benefits. With WOSA services, developers can limit their investment in understanding the details of a service technology, where appropriate. Those focused on developing client-side applications can leave the details of server-side development to others. They can concentrate their efforts on developing quality client-side software knowing that, as long as the service providers maintain WOSA compatibility, the client software will be able to take advantage of new services as they become available.

Of course, this works the same for developers of server-side software. They can concentrate their efforts on providing the most efficient and effective means for exposing and supporting requested services and leave the client interface details to others. Service provider developers are assured equal access to all Windows clients because the WOSA model ensures that all links to the services are the same, regardless of the client software used.

Multi-vendor Support

In addition to allowing programmers to focus on the client-side of the equation instead of the server-side, WOSA implementations provide benefits to application programmers. With a common interface for the service, application programmers can build software solutions that are independent of vendor-specific implementations. The WOSA model allows programmers to build programs that interact with any vendor's service implementation as long as that vendor adheres to the WOSA model.

This is a key benefit for both client-side and provider-side developers. Now service provider vendors can be assured that, as client-side applications change, the interface to their services will remain the same. At the same time, client-side developers can be assured that as service providers upgrade their software; client applications need not be rewritten except to take advantage of new services. This feature allows client-side and service-side development to go forward independently. The result is greater freedom to advance the technology on both sides of the interface.

Upgrade Protection

Another benefit of WOSA-compliant systems is the protection it provides during service or application upgrades or platform migrations. Users can more easily plan and implement software and hardware upgrades when the service access to uniform calls is isolated to a single DLL. Since the WOSA model ensures that service provision and access is standardized across vendors and platforms, changing software and hardware has a minimal effect on users who access services from WOSA-compliant applications.

Thus, when users decide to move their primary database services from an IBM VSAM system to a DB2 or SQL Server environment, the client applications see minimal change as long as the WOSA model was implemented for database access. This protects users' software investment and allows greater flexibility when selecting client and server software.

At the same time, this approach provides protection for commercial software providers because a single application can be designed to work with multiple service providers. Developers can focus on creating full-featured applications without tying their software to a single service provider. As the market grows and changes and service providers come and go, client-side applications can remain the same when the WOSA model is used as the route for service access.

Leveraging WOSA in Your Own Applications

Now that you understand the concepts behind the WOSA model, you can use this information in your own development efforts. For example, when accessing WOSA services from your client applications isolate those calls in your code. This will make it easier to modify and enhance your application's WOSA interface in the future. As the services available change and grow, you'll need only to make changes in a limited set of your code.

Also, when designing your application, plan for the future from the start. Assume that the various program services will be provided via WOSA-compliant tools-even if they are not currently available as true WOSA components. This approach will make it much easier to add true WOSA components to your client application once they become available and will increase the life and flexibility of your software.

The same holds true for back-end service provider developers. If there is a WOSA SPI defined for the services you are providing, use it. This will make your software client-ready for virtually all Windows desktops. If no WOSA interface is yet defined for your service, code as if there is one. Limit the code that talks directly to hardware to a single area in your program. If you are using vendor-specific calls, isolate these, too. This way, when a WOSA model *is* defined for your service, you'll have an easier time converting your software to comply with the WOSA model.

You can learn about the specific API calls that are used to access extension services. You can see a consistent pattern throughout all three extensions. Each of the extension services is divided into layers. The first layer provides a simple interface-sometimes as simple as a single function call-to the target service. The second level provides a more extensive access to a complete feature set. Finally, you'll find a third level of service that allows sophisticated programmers access to all the nifty little details of the extension service. When you're coding your own applications, it is a good idea to use this same approach. Where possible, give users a single function or subroutine that provides access to the extension service. For more advanced applications, create a function library or class object that encapsulates all the extension service calls. By doing this, you'll make it easier to make modifications to your program logic without touching the extension routines. It will also be easier to update the extension routines in the future without damaging your local program logic.