eXcentrix
A Client-Server Architecture for Language Services

Francis Malka, Éric Forget, and Patricia Watt

Abstract

Semantix developed the eXcentrix client-server architecture as a solution to the problems encountered when integrating its translation server into various text applications. At the core of the architecture is the eXcentrix engine which can communicate with clients and servers written in C, C++, Pascal, Java, AppleScript, and Visual Basic. These languages have different ways of handling pointers, references, function name mangling, memory allocation, garbage collection, exceptions, and callback functions. This paper explains how the eXcentrix engine handles each of these issues. eXcentrix was developed for linguistic services but can be applied to any domain in which an abundance of servers and clients cause compatibility problems for users and endless reprogramming for developers.

1.The Birth of an Architecture

Over the last few years, programmers and linguists at Semantix have been developing an advanced translation engine. Strangely enough, our biggest challenges so far have not arisen from translation itself, but from integrating the translation software into other applications. We started to draw a diagram of all the parts involved in the translation process. We identified the client application, the translation user interface, the translation server, the dictionary server, and the transport layer. Then we defined common APIs so all the parts could communicate in a standard way.

At first, it seemed like an elegant way to divide the work among the members of the development team and to encapsulate the complexity of every part. But we soon realized that the architecture could be applied to other language services, such as spell checking, grammar checking, thesaurus, and hyphenation. New objects were created to support all these services and even new services that might appear further down the road.

A question then came to everyone's mind: why not publish this architecture and let anyone design parts that would be compatible? eXcentrix was born. We then added filters and plug-ins so eXcentrix could work from virtually any application. We also created different glues so that third party developers could write code in C, C++, Java, Pascal, AppleScript, and Visual Basic.

This paper gives an overview of the different parts of the architecture. It explains how we dealt with certain programming challenges. And it also shows how easy it is to implement eXcentrix-compatible language servers and text applications on Mac OS, Windows, and Linux.

2. The Ground Rules

As we worked through our difficulties with the way existing language servers interact with text applications, we determined some basic requirements for our architecture:

Standard Server API. Currently, developers of language servers must adapt their programs to suit many different APIs. We decided to have a standard server API that would make it possible to write a single language server that can plug into any client application.

Standard Client API. Similarly, we needed a

standard client API so that client text applications could be written to support different language servers.

Clients should able to Access Remote Services. Currently, client applications cannot access a service on a remote language server. We decided to include features that would make remote access possible:

- Client/server communication using standard protocols like TCP/IP and AppleEvents
- Filters for stream and object data

Support for Different Programming Languages and Frameworks. Developers should be able to write their applications using whichever language or framework they like. Initially, we are supporting C, C++, Pascal, Java, AppleScript, and Visual Basic, and the PowerPlant and MFC frameworks.

Since we were figuring out ways to simplify the lives of programmers, we thought we may as well simplify the lives of end users as well. Here are a few more basic principles related to usability:

Standard User Interface. Currently, each text application has a different user interface for language services even though the services are very similar. We decided to have a standard interface for each language service. Furthermore, the interfaces should have a platform-specific look-and-feel.

Services Menu. We decided to add a standard Services Menu to the menu bar of all client text applications. The Services Menu would list all the available language services so the user can choose any service from within any text application.

Standard User Dictionaries. Currently, if a user creates a personal dictionary in one text application, he or she cannot use the same dictionary in any other text application. We decided to have a standard format for user dictionaries so they can be used in all text applications and for all language services.

Centralized Preferences in one Control Panel. Currently, users must set their language preferences for each text application individually rather than globally as they would do to change their printer, desktop colors, and date/time formats. We decided to put all preferences for language services together in one control panel.

In addition, we decided to support features that Mac users expect to see:

- All parts are scriptable (client, server, interface, etc.)
- Mac look-and-feel
- Contextual menus
- Control Strip module
- Location Manager-awareness
- Drag-and-drop

3. The Finished Product

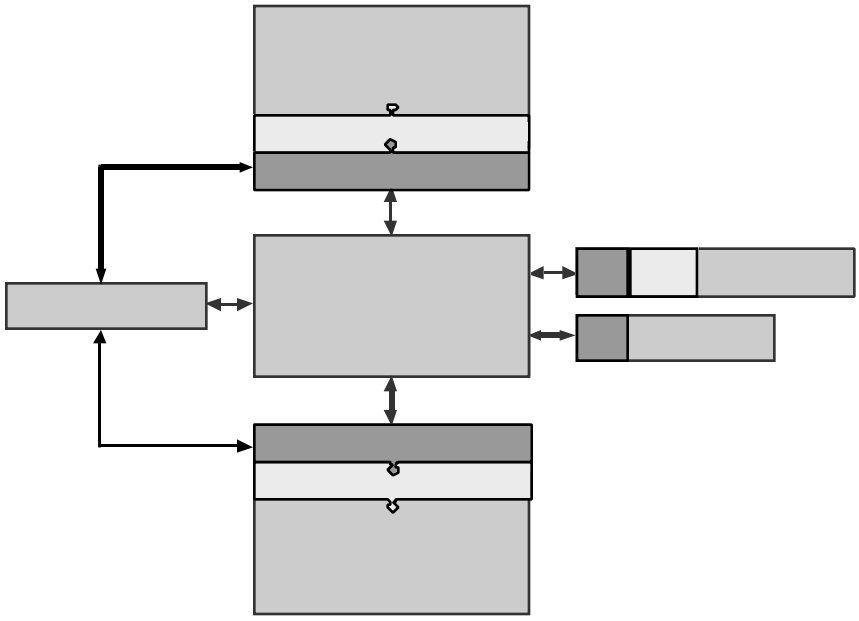
Here's how the architecture turned out. Basically, it consists of three layers:

Client Application. Any application that uses text such as a word processor, an email application, a Web browser, or a database.

eXcentrix. The Engine and various Parts, Glue, and Wrappers

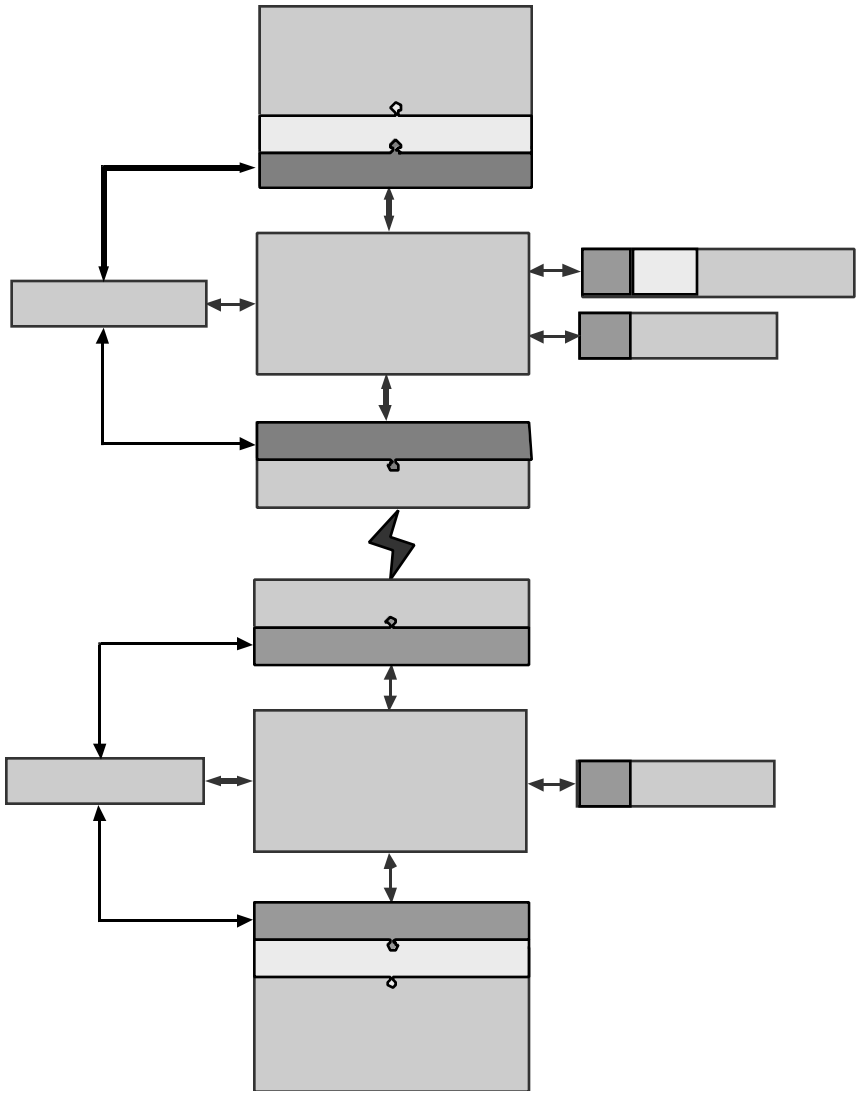
Server Part. Any text-based application or library that provides language services such as grammar checking or translation to client applications

The architecture can be implemented with or without a communication protocol. If the client and the server are on the same machine, the architecture is usually implemented as follows:



eXcentrix architecture implemented without a communication protocol

If the client and the server are on separate machines or if it is desirable to use a communication protocol even though the client and server are on the same machine, the architecture looks like this:



eXcentrix architecture using a communication protocol

Let us introduce you to the various parts:

eXcentrix Engine. The part that makes it possible for clients and servers of all sorts to communicate. It receives requests from a client and translates them into a request comprehensible to a server. Likewise, it receives responses and results from a server and translates them into a form that the client can understand.

Server Part. The part that does the real processing—for example, a spell checker or a translation server. This part does not communicate with the end user directly since it may be located on a remote server machine. If it has any GUI components, they would be for network administration tasks and would be located on the server machine.

Filter Part. Accepts stream or object data as input, transforms it in some way, and returns the modified data to its source.

Transporter Part. Carries binary and object data between client applications and language servers if they reside on different machines. Transporters have been developed to support the TCP/IP and AppleEvents protocols, but developers can create new transporters to

support other protocols.

Interface Part. The means of communication between an end user and eXcentrix. The interface can include menus, dialog boxes, and other graphical objects.

Service Part. Though we originally focused on language services, the architecture can handle other types of services—for example, financial services. A Service Part groups all the classes required for a particular category of service. For language services, the Language Service Part includes classes that relate to words, text, and services like grammar checking and synonym finding.

Glue. A "thin" layer that makes it possible for an application written in any language (C, C++, Java, Pascal, AppleScript, Visual Basic) to communicate with the engine. The glue is transparent to the application developer who simply uses the eXcentrix classes and methods.

Wrapper. Classes that can be inserted between the engine and an application developed with a framework like PowerPlant. The wrapper classes for a particular framework conform to the nomenclature and functionality of the framework. Usually, you would derive your classes from these wrapper classes rather than deriving them directly from the framework classes.

4. Interfacing with Any Programming Language or The Art of Glue

The engine is able to communicate with clients written in C, C++, Pascal, Java, AppleScript, and Visual Basic. These languages all have different ways of handling pointers, references, function name mangling, memory allocation, garbage collection, exceptions, and callback functions.

In this section, we take a look at the problems we faced interfacing with these languages and explain our solutions.

4.1 The Name Mangling Problem

The engine offers an object-oriented API. All the classes inside the engine are written in C++. Exporting engine functions in C++ would have been a nightmare since the C++ compiler would have generated very long and hard-to-understand mangled names. It would have been impossible to trace these names from a language other than C++. Worse, different C++ compilers do not use the same name mangling algorithm, so every application developer would have needed to use the compiler that we used to compile the engine.

To get around the name mangling problems, we renamed all engine functions by concatenating the class name with the method name and exported them as simple "extern C" functions. For instance, the xs::Connection::GetName() method was exported as xsConnectionGetName. This meant we couldn't overload functions with different parameters. Fortunately we had enough imagination to come up with a different name for each function. To offer an object-oriented API to application developers, we then had to wrap the functions in classes outside the engine. This process may seem to generate a lot of extra work, but it allowed us to wrap the classes in the best way for each programming language.

4.2 Fragile Classes

Another problem associated with exporting an object-oriented API is fragile classes. If class definitions change between versions of the API, all applications using the API need to be recompiled to take into account the new size and the new positions of member variables, and virtual functions.

If we were to include our class definitions in our public header files, our classes would be fragile. To avoid this, we published an API consisting of object-oriented C functions. This apparent oxymoron simply means that the first parameter of every function is a pointer to the class to which the function belongs (or "this" as we would call it in C++). This allowed us to wrap these functions easily in glue classes.

4.3 The Solution to the Pointer Problem: No Pointers

The problem can be stated simply: some programming languages, such as C, C++, and Pascal, support pointers and others, such as Java, AppleScript, and Visual Basic, don't.

The main idea behind the solution we've implemented is to get rid of pointers. Instead, for each new object instantiated in the engine, the engine returns a unique ID to the calling application. This ID is a 32-bit integer, a basic type well supported by all languages and scripts. The same ID is used to free the object when it is no longer needed.

The use of IDs also allows for extra validation inside the engine. IDs are allocated within a predefined range by the engine and are validated every time an API function is called. The validation has no overhead cost since it consists of comparing two integers for each object received as a parameter. This mechanism makes it impossible to crash the engine by passing it dangling pointers. If the calling application passes an invalid ID, an error code is returned. If an application passes a random ID that turns out to be valid, the engine checks that the object type of that ID is the expected object type for the given parameter. If the object type isn't valid, an error code is returned. And even if the object type turned out to be valid, the engine would still be manipulating an object of the correct type and would not crash.

Object IDs are visible to the programmer working in procedural languages such as C and Pascal. Objects are created by calling the New() function and are deleted by calling the Dispose() function. In object-oriented languages such as C++, Java, AppleScript, and Visual Basic (dare I call VB an object-oriented language?), object IDs are enclosed in glue classes. For these languages, the programmer need only use the glue classes which manipulate the IDs transparently. The New() and the Dispose() functions are called respectively by the constructor and destructor of the glue classes.

4.4 The Solution to Garbage Collection: Reference Counts

The engine maintains a reference count for each object. When an object is created, its reference count is set to 1. Each time the calling application makes a new reference to a given object, its reference count is incremented. When a reference to an object is no longer needed, the object's reference count is decremented. When the reference count reaches 0, the engine destroys the object.

In procedural languages, the reference count is handled explicitly by calling the Own() and the Disown() functions. Own() increments the reference count; Disown() decrements it. In object-oriented languages, Own() and Disown() are called automatically in the assignment operator and the copy constructor. These operators disown the original object that was referenced (decrement its reference count) and own the new one (increment its reference count). Here's a simplified version of the assignment operator:

```
const MyClass &operator=(
    const MyClass   &inOriginal)
{
    Disown(mID);
    mID = inOriginal.get_mID();
    Own(mID);
}
```

Copying a glue class object by using its assignment operator or its copy constructor creates a new reference to the same object inside the engine. If you modify the original object, you modify the copy and vice-versa. This behavior is desirable in some cases, but in other cases a real duplication may be necessary. To obtain a real duplicate of an object—that is, a separate object—you must call the Duplicate() method. It is more time-consuming for the engine, but the two objects then lead separate lives.

The engine doesn't have to worry about when to collect garbage. The glue class operators look after incrementing and decrementing the reference counts of all objects and whenever an object's reference count drops to 0, the engine deletes the object. Developers working in object-oriented languages don't have to call delete explicitly. For that matter, they never have to call new either. When a developer instantiates a glue class object, the engine instantiates an object and returns an ID—the only member of the glue object. Because glue objects take up so little space (4 bytes), they should always be declared directly on the stack as local variables or inside another class as an aggregate member. No allocation, no pointers, no memory leaks.

4.5 Asynchronous Notifications: A Crash-safe Implementation

Every time an unsolicited event occurs, such as the reception of a TCP/IP message, the engine needs to notify whichever application is interested in the particular type of event. These asynchronous notifications are implemented in the engine using callback functions. Each application has callback functions for each type of event it expects to receive. When an application is active, it registers as a listener for the events in which it is interested. The callback functions are hidden in glue classes whenever a programming language provides a more elegant way to solve the problem.

The first problem we faced when implementing callback functions was to avoid calling an application when it is no longer expecting messages. If this were to happen, the engine would pass the application pointers to deleted objects which would cause a crash.

Our goal was to know when an application is no longer ready to receive messages. This is accomplished by encapsulating callback functions inside classes. The pointers to the callback functions are stored in the engine in instances of objects created by the listening application. When an object is no longer used by the application, its reference count reaches 0 and it is destroyed by the engine. The callback function pointers are destroyed along with the object data members, so it is impossible for the engine to call a callback function once its object is no longer referenced by the listening application.

Callback functions need to be implemented in totally different ways in various languages. Procedural languages, such as C and Pascal, support callback functions directly, without any enhancement. They must be used as is, by supplying the engine with the address of a function that is in the listening application's address space. The application may also supply an application-defined pointer along with the object that contains the callback functions. This pointer is passed as a parameter by the engine every time it calls the callback function. This allows the application to associate its own data structure with the object and to retrieve it instantly when the callback function is called.

Object-oriented languages provide virtual methods, which gave us a neat way to encapsulate callback functions. The implementation in glue classes is a bit tricky. It is done by implementing three different methods for each callback function in the encapsulating class. The first one is a static method that is, in fact, the callback function as seen by the engine. This method does nothing except cast the application-defined pointer into a pointer to the current class and call a virtual method using that pointer. It would look something like this in C++:

```
class MyClass
{

...

protected:

    static xsErr            StaticCallback(
                                    void        *inThis,
                                    long        inParam)
                            {
                                return reinterpret_cast<MyClass *>(inThis)
                                                ->RealCallback(inParam);
                            }
};
```

The second method to implement is the real callback method. It consists of a virtual method that has the same parameters as the static one except for the application-defined pointer:

```
class MyClass
{

...

protected:

    virtual xsErr           RealCallback(
                                    long        inParam)
                            {
                                //  you must overload this function
                                ASSERT(false);
                                return xsErrNone;
                            }
};
```

This method is like an abstract method except that the compiler will let you instantiate the class to which it belongs. We did not make it a pure virtual method in order to allow the engine to instantiate any class, even those that contain callback methods. This was necessary for object streaming—a subject we won't go into here. The method does contain an assertion, so if an object receives a notification and the programmer has not overloaded the corresponding method, it will assert in debug mode but will not crash in release mode.

The third method to implement is a special method that registers the callback function. It sometimes requests a service from the engine at the same time. It is often a constructor, since some objects can be notified without ever

requesting something from the engine. The method would look something like this:

```
MyClass::MyClass()
{
    EngineFunction(this, StaticCallback);
}
```

In pseudo-object-oriented languages like Visual Basic, virtual methods are not supported. Still, VB provides a way of handling notifications by raising events. Raising an event is like throwing an exception, only it is less exciting. It basically calls the functions that are registered for an event and for the given instance of the class (you've read it right: not for all the instances of the class). So you need to know the exact instance of the object at compile time to be able to receive events. The implementation is even trickier than in C++. You must first implement a global function outside the class (in another file). The global function plays the same role as the static function in C++. It is the callback function as seen by the engine. This function cannot raise the event itself because it is not a member of a class, but it must not be a member of the class so you can obtain its address by calling AddressOf (I suddenly feel like I'm writing a legal document). The global function must instead call a member function whose only purpose is to call RaiseEvent which fires the event. This whole process is initiated by passing the address of your callback function to the engine using the AddressOf keyword in the class constructor or in another method. If you try to use the application-defined pointer to keep the address of an object, the Visual Basic runtime won't recognize it in the callback function and will throw a C++ exception!

4.6 Exceptions: Only When Possible

Exceptions are not handled by all programming languages, so the engine returns an error code instead of throwing an exception when an error occurs. When a glue class method receives an error code and the programming language includes a try-catch mechanism (like C++), the method throws an exception:

```
void
MyClass::Function(
    long    inParameter)
{
    xsErr       err;

    if ((err = EngineFunction(inParameter)) != xsErrNone) {

        throw err;
    }
}
```

In other cases, the error code is simply returned to the calling application.

5. How To Use the Architecture

If you want to make a language server or a text application eXcentrix-compatible, you only have to do a modest amount of coding. The classes and functions needed are already defined in the Language Service Part. Since it would take too much time to give examples in all programming languages, the examples below are in C++.

If you want to adapt the architecture to other types of service (or for language services not already covered in our Language Service Part), you have a little more work to do. The required labor is divided as follows:

- You define the data to be transported
- You create the necessary resources in the Part Editor, a drag-and-drop IDE provided with our SDK
- You define classes and / or functions for the type of service you wish to provide
- eXcentrix handles all the low-level stuff

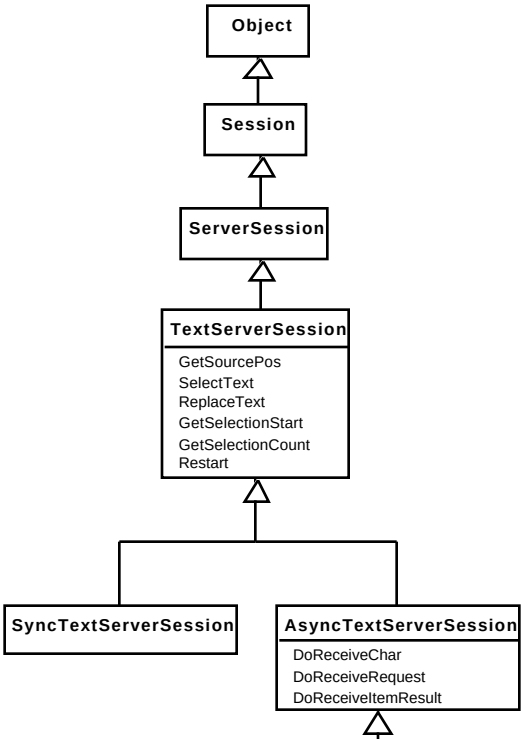
5.1 Coding Required for a Language Server

Making a language server eXcentrix-compatible involves these coding tasks:

- Deriving a subclass from an existing class
- Writing the necessary initialization code
- Writing the necessary clean-up code

To demonstrate the coding effort required, we will explain what must be done to accomplish the first task mentioned above: Deriving a class needed for a language server.

eXcentrix provides a number of classes, all of which are derived from its base class Object. For a typical single-threaded server, the developer must derive a subclass from the class AsyncTextServerSession. The following diagram shows the relevant branch of the class hierarchy as well as MyTextSession, the derived class.



Server developers must derive a subclass from eXcentrix class AsyncTextServerSession

If you want to receive the text character by character, you must overload the DoReceiveChar() function in the derived class. This function should do the work of reading characters, processing them, and replacing them. It would typically call various functions inherited from the class TextServerSession, such as GetSourcePos() and ReplaceText() as well as the server's own processing functions.

For example, DoReceiveChar() might be defined as follows:

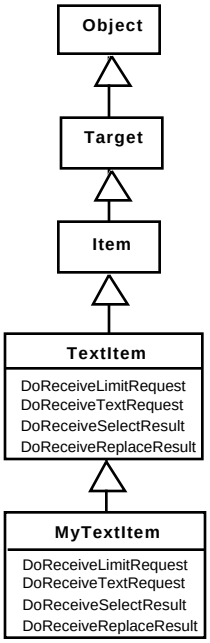
```
bool
MyServerSession::DoReceiveChar(
    bool    inMoreText,
    xsChar  inChar)
{
    //  if there is text
    if (inMoreText) {

        //  Calls to the server's own processing functions
        //  until the end of a sentence or a paragraph
        if (mTextPos >= mTextLength) {

            //  Replace the text
            ReplaceText(mBufferPos, mTextPos, mTextPos, mText);
            mTextPos = 0;

        } else if (mTextPos > 0) {  //  Send the final results

            //  Replace the text
            ReplaceText(mBufferPos, mTextPos, mTextPos, mText, false);
            mTextPos = 0;
            inMoreText = false;
        }
    }
    return inMoreText;
}
```

You can also receive text in blocks—for example, 4 kb. In this case, you would overload the method DoReceiveTextResult() which is a method of TextServerSession. But you would have to take care of preserving text formats yourself.

5.2 Coding Required for a Text Client

If you use TextEdit or WASTE, you don't have to do any coding!

If you have your own text engine, you must derive a subclass from the eXcentrix class, TextItem. The following diagram shows the relevant branch of the class hierarchy as well as MyTextItem, the derived class:



Client developers must derive a subclass from the eXcentrix class TextItem

In the eXcentrix scheme of things, sessions are server-driven. Apart from the client's initial request for a service, the client is passive. It waits for various requests from the server and then it waits until the server sends the results. The methods that you must overload in MyTextItem are:

- DoReceiveLimitRequest
- DoReceiveTextRequest
- DoReceiveSelectResult
- DoReceiveReplaceResult

DoReceiveLimitRequest() must return the boundaries of the document and of the selected text to the server. The method would be overridden something like this:

```
xsErr
MyTextItem::DoReceiveLimitRequest(
    LimitRequest    &inLimitRequest)
{
    //  Definition of local variables:
    LimitResult theResult;      // instance of an eXcentrix class
    uint32      docCount;       // for storing length of document
    uint32      selectStart;    // for storing start of selection
    uint32      selectEnd;      // for storing end of selection

    // Get the length of the document with call(s) to the
    // client's own methods and store in docCount

    // Set the document boundaries
    theResult.SetItemStartingPos(0);    // always zero
    theResult.SetItemCount(docCount);

    // Get start and end positions of the selection with
    // call(s) to client methods and store in selectStart &
    // selectEnd

    // Set the selection boundaries
    theResult.SetSelectionStartingPos(selectStart);
    theResult.SetSelectionCount(selectEnd);

    SendItemResult(theResult);

    return xsErrNone;
}
```

DoReceiveTextRequest() must return the portion of text requested by the server. The method might be overridden like this:

```
xsErr
MyTextItem::DoReceiveTextRequest(
    TextRequest     &inTextRequest)
{
    // Calls to the client's own methods that
    // can return a portion of text
}
```

DoReceiveSelectResult() must select the text in the document as requested by the server—the text would be highlighted. The method would be overridden as follows:

```
xsErr
MyTextItem::DoReceiveSelectResult(
    SelectResult    &inSelectResult)
{
    // Calls to the client's own methods that
    // can select text
}
```

DoReceiveReplaceResult() must replace text in the document based on information received from the server:

- start position in the document
- length of original text
- replacement text
- length of replacement text

The method would be overridden as follows:

```
xsErr
MyTextItem::DoReceiveReplaceResult(
    ReplaceResult   &inReplaceResult)
{
    // Definition of local variables:
    // Replacement and TextRun are eXcentrix classes
    // xsPosition and xsPositionFirst are eXcentrix types
    Replacement     theReplacement = inResult.GetReplacement();
    xsPosition      currentPosition = xsPositionFirst;
    TextRun         aRun;

    // Get replacement text and insert it in original document
    while (theReplacement.GetNextTextRun(currentPosition,aRun)) {

        // Calls to the client's own methods that
        // can replace a portion of text in the document
    }

    return xsErrNone;
}
```

6. Conclusion

We developed a client-server architecture in order to make our lives, as developers of a translation service, simpler. We now think that it could also simplify the lives of end users and of other programmers who develop text applications and language servers. The architecture allows any server to plug into any client. It could be applied to any domain in which an abundance of servers and clients causes compatibility problems for users and

endless reprogramming for developers.