The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Obtuse, a scripting language for migratory applications

Robert P. Cook
Dept. of Computer and Information Science
University of Mississippi

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Obtuse, a scripting language for migratory applications

Robert P. Cook
Dept. of Computer and Information Science
University of Mississippi
www.cs.olemiss.edu/~bobcook; bobcook@cs.olemiss.edu

Abstract

This paper discusses the design and implementation of Obtuse, a scripting language for migratory applications. The paper reviews the pertinent ActiveX technology that provides the runtime object infrastructure. Then we discuss the Obtuse object model and present an overview of the language. Next, several sample programs are used to illustrate the concepts. Finally, we review some of the problems with DCOM, based on our experience.

Keywords: scripting language, Obtuse, migratory applications, Obliq, distributed systems

---

1. Introduction

Obtuse was designed by the author, inspired by Cardelli's Obliq [1] and Bharat's Visual Obliq[2] systems, and implemented as part of a summer research appointment at Microsoft Corporation. The goal was to explore the potential of several core ActiveX technologies [3,4,5], including COM (**C**omponent **O**bject **M**odel), Automation, and DCOM (**D**istributed **C**omponent **O**bject **M**odel).

Obtuse is unique in two respects; first, in its synergistic use of ActiveX technology and second, in its ability to transfer the state of a Visual Basic form from one machine to another. A **migratory application** is one that can transfer program state (including the user interface) to different Internet locations under program control. Other terms used in the literature are **mobile** or **transportable agents**. Obtuse is also a **scripting language**; that is, it defines sentences capable of being executed as fine-grained code fragments.

As an example of transferring UI state from one machine to another, consider the Visual Basic (VB) form in Figure 1, which consists of an edit control and a button. The form is used in a simple, routing-slip application. The user can type a command line, such as "obtuse poll m1 m2 m3" to initiate execution. The list (a routing slip) represents a sequence of computers to visit. The DCOM infrastructure is utilized by the Obtuse runtime to implement the remote activation that is necessary to support the routing-slip application.

The form is circulated to the machines in the order listed. The accumulated comments are available to each recipient and the completed form, with all comments, is returned to the source machine. When one user clicks the OK button, the form is moved to the screen of the next computer in the list.

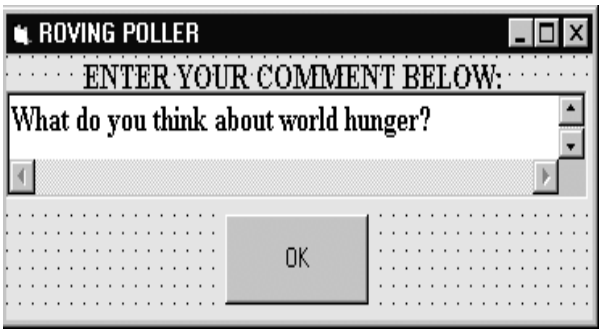

Figure 1. User Interface for a Roving Poller

We refer to programs, such as the routing-slip example, as **in-your-face** applications. When one user clicks the routing slip's OK button, the document appears instantaneously on the screen of the next recipient. Most word processors also support routing slips by using e-mail as the transport mechanism. However, users are only notified if an e-mail client is executing at a site and if they decide to read their mail.

In the routing-slip application, the code, together with its execution state, can also migrate from one machine to another with the form. Obtuse implements program migration by exposing threads and contexts (a program's global variables) as COM objects.

Figure 2 lists a simple Obtuse program that moves itself from one machine to another. In Obtuse, a running program is a collection of COM objects, which support an Automation interface. The sample program creates a thread and a context object on a remote machine. Next, the Fork method of the running-thread object is invoked to clone the program's state. At this

point, there are two threads executing, one locally and one remotely. However, since they have duplicate contexts, any object references in one are duplicated in the other. As a result, the remote thread can access objects on the parent machine in a location-opaque fashion.

```
// Note that variables are "typed" at runtime
var me, thread, context, where;
me := self;              // a reference to the executing
thread
print( "parent process starting");
where := "louie.cs.olemiss.edu";
                                  // create a remote thread
thread := object("Bob.Thread", where);
                                  //create a remote context
context := object("Bob.Context", where);
                                //duplicate myself at "where"
if me.Fork{thread, context}=1 then
    print(" parent stopping");
    quit;                        // 1 returned to parent; it
quits
end;
print( "we made it to", where);
quit;
```

Figure 2. A Sample Obtuse Program

Obtuse is unique in that the mechanisms to support the runtime (threads, contexts, stacks) are all COM objects. Another unique aspect is that Obtuse uses Visual Basic forms to implement user interfaces. These forms can also be marshaled in order to transport their state from one machine to another. Since Obtuse is based on COM, it can be used to manipulate any COM Automation object, which includes all Office applications and ActiveX controls. The DCOM infrastructure supports the remote location and activation of COM objects.

Other features of Obtuse include support for script-based execution, support for multiple threads, runtime strict typing, and a machine-invariant program representation. An Obtuse program can consist of a sequence of expressions with no variables, a series of statements on a set of global variables, or a collection of procedures. Furthermore, an Obtuse program can invoke an Automation object's methods and access its properties at runtime; it is not necessary to "import" or "include" interfaces.

Obtuse does not support compile-time type binding. The type checking in expressions and procedure calls is performed at runtime. However, Obtuse is "strict"; that is, types must match exactly on operations such as comparison or multiplication. Variables are bound to a type on runtime assignment. From that point on, until another assignment occurs, that variable must be type compatible with every operator that is applied to it.

Programs are UNICODE-based and compile to a machine-invariant representation that encodes the source program. That is, the object code can be inverted to recover the original source, including comments.

The paper first presents an overview of ActiveX technology. Then we discuss the Obtuse object model and present an overview of the language. Next, several sample programs are introduced to illustrate the concepts. Also, we present some performance measurements for Obtuse/ActiveX. Finally, we review some of the problems with DCOM based on our experience.

2. ActiveX—COM and DCOM

The two most important aspects of ActiveX for scripting support are its implementation of dynamic method binding and invocation, as well as self-describing types. Dynamic method binding is the technology (Automation) that enables Visual Basic applications to manipulate Office documents, such as spreadsheets or slide presentations. It also enables HTML scripting support (VBScript) in Microsoft's Internet Explorer 3.0.

The dynamic, or late, binding technology enables an object to expose methods and properties for use by other objects. The technology also supports the lookup of method and property names and a mechanism to build and execute a procedure call at runtime. It is a separate set of code from COM.

Self-describing types (or **variants** as they are termed in COM), are the key, underlying representation for data types in the Visual Basic language common to VBScript and VB. In the next sections, we present an overview of COM and DCOM.

2.1 COM – Component Object Model

An "object" in COM typically has a document type, such as .xls, .ppt, .doc. Each document type can have a registered server. For example, winword.exe is the server for *.doc objects. Objects also have a registered application name (e.g. "Microsoft Word Document") and a globally-unique identification number, called a **class id** (CLSID).

The association between a class and its server is maintained in a persistent store called the **registry**.

There is one registry per machine and there is currently no "yellow pages" server to support object lookup for distributed services, although one is reputed to be available shortly.

The COM model is language independent; it may have a concrete implementation in a particular language, such as C++, but the relationship between COM and different languages is orthogonal. For example, Obtuse is implemented in C++ but it uses COM objects, which are implemented in Visual Basic, to define its user interface.

A COM object is defined by its support for a collection of interfaces, each is which is tagged by a 128-bit globally unique interface identifier (IID). The interfaces that an object supports can vary over time; and the interfaces need not have any other relationship (such as inheritance). There is only one requirement i.e. **EVERY COM INTERFACE MUST INHERIT FROM THE IUnknown INTERFACE**, which is listed in Figure 3.

```
virtual HRESULT QueryInterface (
    InterfaceID & riid,
    LPVOID * ppvObj)=0;
virtual HRESULT AddRef(void) = 0;
virtual HRESULT Release(void) = 0;
```

Figure 3. COM IUnknown Interface

The power of COM derives from several of the requirements satisfied by the IUnknown implementation. First, QueryInterface must be used to obtain an object handle for an instance variable **x** (as in x.Queryinterface) that supports a particular interface (identified by the InterfaceID argument). If the object **x** does not support the interface, the HRESULT returned indicates an error. In C++, a COM object handle is a "pointer to a pointer to a vTable". The vTable is generated in C++ because the interface is "pure virtual", as are all COM interfaces.

Since every interface is required to inherit from IUnknown, any object handle can be used to retrieve a handle for any interface that the object supports by calling QueryInterface at any time. Further, the object handles are reference counted. QueryInterface increments an object's reference count and so does AddRef. A Release call decrements an object's reference count.

2.2 DCOM – Distributed COM

DCOM extends COM in a number of ways. First, objects can be remotely activated and a handle returned to the activating site. The returned object handle can be used by a program in a location-opaque fashion; that is, the programmer need not be aware of the object's location. Second, DCOM imposes location, security and identity restrictions on COM objects. Each site has total control over who can activate an object, how objects are activated, and with what permissions object servers can execute. Third, DCOM implements reference counting across machine boundaries and garbage collection. Finally, DCOM automatically remotes calls to COM interfaces that are supported by remote objects. For user-defined interfaces, an IDL compiler must be used to generate proxy stubs for the client/server sides. Typically, both stubs are included in a single DLL.

2.2.1 Remote activation

The DCOM method to activate (cause its server to be loaded) a remote object is **CoCreateInstanceEx**. The arguments to the method are the object's class id, a machine name, and a list of interface ids.

Machines are identified using the naming scheme of the network transport layer. By default, all UNC (\\chairpc) and DNS names ("chair.com" or "135.9.19.33") are supported. Object search is restricted to a single machine at present. DCOM has no notion of distributed scope or of distributed search paths.

To optimize network performance, the CoCreate call may specify a list of interface ids. Thus, N object handles can be retrieved in a single round-trip to the server site. Conceptually, this is analogous at runtime to the "import java.lang.*" convention in Java, which can be used to import all of the classes in a package at compile-time.

2.2.2 Access control

Access to objects can be regulated under program control using the NT security API; however for most Obtuse users, the utility program **dcomcnfg** is the point of control. This program lists the application objects that are "registered" on a particular machine. The **Location**, **Security**, and **Identity** of each object can be separately controlled. The Location options are "run here" or "run there". The latter option supports forwarding a requested activation from one computer to another. The Security option supports editing the access control lists (ACLs) for activation, access and configuration. NT provides very fine-grained access control so that individual users, or groups, can be specified.

The Identity option designates the protection domain in which a server is executed. The choices are the domain of the interactive user, the launching user, a particular user, or a system service. For example, the "particular user" option can be used to solve the "game accounting" problem, which is to let a user run a game program that can write its list of winners to a file that is not accessible to one of the players. The appropriate protection domains can be created by having one DCOM object (player's domain) to play the game communicating with another DCOM object (game's domain) to record the scores.

2.2.3 Reference counting

As we discussed in Section 2.1, COM defines a mechanism to reference count object handles. If a program fails, any cross-process links must be broken to properly release objects. Similarly for DCOM, the system must account for inter-machine links and must break links when processes terminate or fail. For distributed systems, there are also the possibilities of node crashes and communication outages. The DCOM implementation addresses these problems.

2.2.4 Remote procedure call

DCOM automatically remotes inter-node calls on COM interfaces. The arguments are marshaled through the normal remote procedure call (RPC) mechanism. RPC on user-defined interfaces requires the use of the IDL compiler to generate client- and server-side proxy stubs.

The automation interface (IDispatch) can be used to "late bind" a procedure call; that is, a program can build a procedure call at runtime. The automation interface provides the object-access infrastructure for any scripting language, such as Obtuse, VBScript, JavaScript or AppleScript.

COM supports the registration of type libraries that describe an object's properties and methods (also argument lists and return values). As a property example, a button object might have BackgroundColor and Text properties, which could be accessed or modified remotely using Obtuse. The IDispatch interface includes methods to "query" for the id of a method or property name and then to "invoke" that method or "access" that property. The Automation runtime builds the argument list in a format that is compatible with the target language and handles the call/return processing.

2.2.5 Variant data

Another aspect of the automation solution is a "universal" data type termed a **variant**. A variant is a "union" of about 40 different base types that also includes arrays of those types, and arrays of arrays. An array can be created with homogeneous elements of a particular type or with variant-type elements, each of which can be of any type.

IDispatch and IUnknown object handles are two of the possible variant-record base types. Since IDispatch is one of the "builtin" COM interfaces, it is remoted automatically by DCOM. In Obtuse, all argument lists to procedures, return values, and property values are encoded as variant data.

3. Obtuse Language Overview

The Obtuse system consists of a compiler and an interpreter, and a collection of COM objects. The compiler's output is a UNICODE text string that encodes the source program, including comments. The executable can be inverted to produce the original source program. Thus, after a program is initially compiled, there is only one representation, which can be used for both execution and symbolic debugging.

Obtuse supports only one data type (variant), so a variable declaration is just a list of identifiers. The type is implicit. A form of type checking is supported based on the notion of assignment-typing. Basically, every assignment statement binds a new type to an identifier as well as a new value. Expression evaluation is type checked at runtime. There is no implicit conversion as in VB; that is, type checking is "strict".

In Obtuse, the "object" built-in function maps a registered object name at a particular machine to an object reference. For example, the function call **object("Bob.Thread", "foo.univ.edu")** would check the registry on the designated machine and then load the server if necessary.

Once an object reference is obtained, the program can manipulate the properties of that object or invoke its methods. Assignment of object references copies the reference, not the value, even if the assignment crosses machine boundaries. The DCOM reference counting infrastructure tracks each copy. For assignment of other variant values, including arrays, Obtuse copies the value. The rule is simple: sharing can only be accomplished through COM objects.

Obtuse implements a common set of statements such as **if**, **loop**, **for**, **case,** in addition to variable and method

declarations. Pointer, structure and class declarations are not supported. A qualified reference can be used to access an object's properties. Since an object's methods are dynamically bound using the IDispatch automation interface, the compiler cannot perform checking for undefined names or mismatched argument lists. To facilitate some checking, calls to Obtuse procedures are delineated with the traditional "( )" and calls to an object's methods use "{ }".

As mentioned earlier, Obtuse programs are encoded as UNICODE strings. Sufficient information is retained in the encoding to invert the object code to the source. The opcodes were designed to use a character encoding so that program fragments could be embedded in documents, sent as mail messages, or be applied as drag-and-drop operators on user-interface objects. Figure 4 lists several example encodings. The blank, tab, and new-line opcodes are no-operations.

The " opcode designates a constant. Constants are translated at runtime so the opcode includes a type designator, the length of the string, and the text constant. This is not very efficient but it does avoid representation issues, such as for floating-point numbers. Small integers are encoded as individual opcodes. The opcode design also took into account the requirements for the next version of Obtuse in which type modules, such as Complex numbers, could be called upon to parse their own constant representation.

Code Fragment	Encoding
print(3/24/76);	**"**D073/24/76 **p**0 **q**
print(3+45);	**3 "**L0245 **+ p**0 **q**
if a>3 **then** y := 6;	**Y**3 **L**1 **3 :** > **I**005
	6 S2 **J**015
elsif a>2 **then** y := 8;	**Y**4 **L1 2 :** > **I**005
	8 S2 **J**007
else y := 9; **end**;	**Y**5 **9 S2 Y**6

OpCode	Key
"	Load Constant
Y	Syntax Marker
L	Load Variable
S	Store Variable
:	Compare
I/J	Forward Jumps

Figure 4. Program Encoding Example

The Y opcodes encode the syntax of the source program. Even the comments in the source are encoded, but the comment opcode is treated as a no-op at runtime. The compiler attempts to generate code for branch instructions so that syntax markers are not included in loops.

4. Obtuse Object Model

The initial Obtuse implementation supports FORM, FILE, MUTEX, THREAD, CONTEXT, and STACK objects. FORM objects are Visual Basic forms, which can contain any VB control. Each FORM object represents one VB form. Since there are hundreds of different VB controls, the Obtuse user interface model has a broad range of capabilities. As a result, forms can be constructed as the user interface for almost any application.

The FORM interface is implemented as a Visual Basic program. VB supports the creation of programs that support COM interfaces (particularly IDispatch, the Application Automation interface). As a result, these programs, can be activated remotely using DCOM. We implemented (in VB) a form-server object that supports Form, Item, Save and Restore methods. The Form and Item functions return object references to a form or to any of the controls on that form. Once an object reference is obtained, Obtuse can set or retrieve the properties of a form or control. For example, the "value" property of a scroll bar is a numeric quantity that can be used to get/set the thumb position.

In the current prototype, a programmer constructs a user interface with a VB program called GenForm, which is part of the Obtuse system. When GenForm is executed, it writes a file that contains a text array constant that "defines" a form. The array constant is then inserted into an Obtuse program as a "resource". The Restore method causes the VB form-server object to display the previously-saved "look". A VB form is encoded/decoded by Save/Restore as a text array. Figure 5 illustrates the encoding of the Roving Poller form that was displayed in Figure 1.

```
[12345, 12, 3, 15, 4, 5535, 1, 2145,5,
16776960, 2, "Roving Poller", 23456,
14,3, 72, 4, 144, 0, 89, 1, 41, 5,
-2147483633, 2, "OK", 45678, 12, 3, 0,
4, 88, 0, 125, 5, -2147483633, 2,
"Enter your comments below:",
56789, 12, 3, 16, 4, 0, 0, 361, 1, 49, 6,
""]
```

Figure 5. Array Constant for a VB Form

To save space when creating a new form, the GenForm program only saves the differences between a canonical

set of control values and those specified by the user. For example, the "top" and "left" properties are almost always changed; the "visible" property is rarely changed. Since VB has a large number of properties for each control, this convention saves considerable space.

Obtuse has the unusual property that the mechanisms of the language implementation are objects, in fact DCOM objects. Remember that a DCOM object can be activated on any machine. The Obtuse interpreter is only required to run Obtuse code, not remote objects. This is one of the main differences with Obliq and other distributed application systems, which require an instance of their interpreter at each node.

A FILE object supports I/O on files and directories anywhere in the Internet. Interestingly, DCOM can pass a file handle from one machine to another and the handle retains its validity. This is not possible with NT, the host operating system. Since activating a FILE object is necessary to access files and since DCOM implements per machine and per object access controls, the user has full control over the safety of the system.

A MUTEX object is used to implement critical section synchronization for shared variables or resources. The supported methods are Enter and Leave.

The Obtuse object model takes unique advantage of DCOM's capabilities. First, thread and context (a program's global variables) objects can be created on any DCOM machine on the Internet so that a thread on one machine can opaquely access variables on any other machine's context. Figure 6 lists the attributes of the three Obtuse program objects – Thread, Context, and Stack.

A context can be shared among any number of threads, local or remote. For example, a master debug console can easily be constructed to monitor the modification of contexts located all over the world. Finally, a thread can migrate by simply forking its state to a new machine and killing the parent thread. Since a context is one of the arguments to the Fork method, the new thread can be created with its own copy of the parent's context or it can share the parent's context. All object references are marshaled properly by DCOM on inter-machine transfers so that programs remain completely location opaque.

STACK objects are always co-located with their thread objects; however, they still are DCOM objects. There is no requirement that execution be "procedure based". The interpreter can evaluate formulas with only a stack and a code string. When a thread is marshaled to be transferred to another machine, the stack content, including return addresses, is converted to a portable format.

In the COM model, objects are normally created by server front-ends called class factories. The separation of request and creation on a per-type basis provides a way to create many different types of servers for each object type. Remember that in COM an object is defined by the interfaces that it supports, not by its data structures or algorithms.

4.1 The thread object

In the current implementation, a THREAD object contains a code string, an IDispatch object handle to a context object, two object handles to a stack object, and type information (used by IDispatch, described later). The IOperator interface defines methods such as Add and Subtract; the IProcedure interface defines methods such as Frame and Return (used for procedure call/return). The latter interface may be omitted for calculations that do not involve procedure calls.

	DATA	**OBJECT INTERFACES**
THREAD	Code string Context Stack Type Info	Com.IUnknown Com.IDispatch IThread
CONTEXT	Array Variants Type Info Persist flag	Com.IUnknown Com.IDispatch IObject
STACK	TopOfStack ProcFrameIndex FrameTopStack Array of Frames Array Variants	Com.IUnknown Com.IDispatch IOperator IProcedure

Figure 6. Obtuse Program Objects

When a thread starts execution, it binds to an IObject handle. For efficiency (since a context holds global variables), the IObject interface was compiled by the IDL compiler to generate proxy stubs. As a result, references to a remote context, must pass through a proxy DLL, which must be registered at that site. Accessing global variables using the IObject interface is much faster than using IDispatch.

Figure 7 lists the IThread interface, which contains methods for creating a thread, marshaling its state, and controlling its execution. Migrating a thread or object depends on support for marshaling its state. In theory,

any system, such as C++ or Java, could support migration.

4.2 The context object

A context object is a vector of global variables. Since all variables in Obtuse are represented using the variant data type, a context is just an array of variants. Further, since an array of variants is also a variant type, a context is marshaled automatically by DCOM when passed from one machine to another.

METHOD	ARGUMENTS	USE
Open	Code string	Start a thread with a default context
OpenEx	Code string Initial pc value Initial context State to restore Suspend flag	Restart a thread from a saved state
Fork	Thread object Context object	Clone the current thread and context
Join	Timeout value	Wait for a thread to terminate
Suspend		Stop a thread
Resume		Start a thread
Sleep	Delay value	Timed delay
Code		Retrieve code string
Context		Retrieve context handle
Stack		Retrieve stack handle

Figure 7. The IThread Interface

The IObject interface, which is listed in Figure 8, describes the methods to store and retrieve Obtuse variables. The Get/Put methods access simple variables; GetIndex/PutIndex access arrays. Since all Obtuse variable locations are the same size, variable addresses are just indices (e.g. 0,1,2 etc.).

Array access is implemented by passing the entire subscript list as an argument. This approach is more efficient for remote access than evaluating one subscript at a time. In Obtuse, array assignment is "by value". The only way to generate a "reference" in Obtuse is by creating a COM object.

The context class interface contains a number of helper functions (save, restore, persist, sweep) that are intended only for local use. The "save" and "restore" functions are used to clone a context. The "persist" helper function toggles a flag that indicates whether a context should be retained after its thread terminates. This option is useful for debugging and also for writing programs that inter-operate by passing contexts back and forth. Used in this way, a context is somewhat like a COMMON block in FORTRAN.

METHOD	ARGUMENTS	USE
Get	Index	Access a variable
GetIndex	Index Subscript list	X[a, b, c]
Put	Index	Store a variable
PutIndex	Index Subscript list	X[a, b, c] = 3

Figure 8. The IObject Interface

The "sweep" helper was introduced after we discovered that many of our early programs were leaving objects scattered all over the network. The program in Figure 1 illustrates the problem. A remote context is created and then the local context is "cloned" into it. However, the new context now has a reference to itself, since its handle was in the original context. As a result of this circular reference, DCOM never called the destructor for the context when the thread terminated. The "sweep" method addresses the problem by clearing all object handles in a context when its thread terminates.

4.3 The stack object

As mentioned earlier, there is a one-to-one relationship between a stack and a thread. By design, a stack can never contain a reference to itself so circular references are prevented. Stacks and threads are always co-located for efficiency. In Obtuse, a stack is a vector of variant values and may also have an associated vector of frames (if procedures are used by the code fragment).

This design is somewhat unconventional in that most systems embed the call chain within the evaluation stack. The disadvantage is that the chain typically uses pointers, which we avoid by inverting the list. As a result, the frame stack is a separate array. When a thread migrates, there are no restrictions on its state. It can be arbitrarily nested within procedure calls.

Each call frame contains an argument count for the procedure, a count of local variables, the evaluation stack index for the previous frame, and the index of the call point in the thread's code string. Another advantage of inverting the frame stack is that arguments and locals are adjacent so indexing is trivial.

Figure 9 lists the IOperator and IProcedure interfaces. The stack class includes two helper functions: Save and Restore. The "Save" procedure produces an array of variants that represents the "state" of the stack, including procedure nesting. The "Restore" procedure returns a stack to a previous state. Since program state is a first-class object in Obtuse, it should be possible to support fault-tolerant algorithms through various checkpointing schemes; however, we have not explored this idea further.

IOperator	
Add	3 + 4
Subtract	3 - 4
Multiply	3 * 4
Divide	3 / 4
Compare	< <= = >= >
Mod	3 % 4
Invert	- or !
And	3 & 4
Or	3 | 4
Load	Variant←constant string
Print	String←variant
Push	Stack←variant
Pop	Variant←stack
IProcedure	
Frame	Call procedure
Return	Return from procedure
Get	Get local/argument
GetIndex	Get local array
Put	Store local/argument
PutIndex	Store local array

Figure 9. The IOperator/IProcedure Interfaces

4.4 Type information

The power of the Obtuse object model derives from DCOM, particularly when combined with IDispatch. In this section, we illustrate how a dynamic procedure call can be accomplished. As is illustrated in Figure 1, the "object" statement in Obtuse can be used to create an object on any machine. In COM, an object handle is created by asking if the object supports a particular interface. Obtuse always queries for the IDispatch interface.

Thus, the "me" variable in Figure 1 is a variant record with a value that is an object handle of type IDispatch. The code "me.Fork{a, b}" translates to interpreter byte codes that indicate a method invocation. When the Obtuse interpreter encounters an "invoke" opcode (actually properties work the same way), it first has to bind the method name (Fork) to a method id code. Automation does not support a call by name option. As a result, it takes two round-trip calls to the server per method call.

After getting the method id, the interpreter calls the "Invoke" method in the IDispatch interface of the "me" object. The arguments to "Invoke" are a vector of variants (the argument list) and the method id. The return value from "Invoke" is a variant that encodes the result of the "Fork" call.

The remaining part of the puzzle is a discussion of how IDispatch actually constructs a method call to "Fork" in whatever language "Fork" is implemented, and with the appropriate calling convention. Figure 10 illustrates the solution used in Obtuse.

```
static PARAMDATA
            rgpdataCBobContextPersist[] =
{
   { "onOff",  VT_LONG }
};

static METHODDATA  rgmdataCBobContext[] =
{ // void CBobContext::Persist(long onOff)
   {
    "Persist",
    rgpdataCBobContextPersist,
    IDMEMBER_CBOBCONTEXT_PERSIST,
    IMETH_CBOBCONTEXT_PERSIST,
    CC_STDCALL,
    DIM(rgpdataCBobContextPersist),
    DISPATCH_METHOD,
    VT_VOID
   },
};

static INTERFACEDATA g_idataCBobContext=
{
   rgmdataCBobContext,
  DIM(rgmdataCBobContext)
};
```

Figure 10. Encoding an IDispatch Interface

The data structure encodes the "type information" referred to in Figure 6 for thread and context objects. The "interface data" structure contains a count of the number of methods, and pointers to method descriptors. Each method descriptor contains the name of the method, a pointer to a vector of parameter descriptors, a method id number, an index into the vTable for the class, a count of the number of

parameters, a code to indicate the calling convention, and the type of the return value.

Thus, for the "me.Fork{}" example, the arguments (encoded as an array of variants) are converted to the parameter types specified, pushed on the stack in a calling-convention and language-specific way, then the vTable index is used to make the call. The return value is removed from the stack (or registers) and is encoded as a variant. The Obtuse interpreter then pushes the return value on its stack and execution continues.

```
var resource;      //set from a VB form
var a, b, c, where, me, thread, context;

resource :=
[12345,12,3,-15,4,5535,1,2145,5,16776960,2,
"Roving Poller",23456,14,3,72,4,144,0,89,1,41,5,-2147483633,2,
"OK",45678,12,3,0,4,88,0,125,5,-2147483633,2,
"Enter your comments below:",56789,12,3,16,4,0,0,361,1,49,6,
""];
foreach where in argv do              //iterate command line arguments
  me := self;                             //a reference to the running thread
  thread := object("Bob.Thread", where);    //create a remote thread
  context := object("Bob.Context", where); //create a remote context
  if me.Fork{thread, context}=1 then   //duplicate myself at "where"
    quit;                               //1 returned to parent thread; it quits
  end;                                    //-------child thread starts here
  a := object("Bob.Form", "");            //create a VB form at the child site
  b := a.Restore{resource};             //method call to VB form server
  c := a.Item{"button0"};               //get object reference to the button
  loop
    if c.tag = "1" then exit; end;        //delay until a button click
  end;
  resource := a.Save{};                 //save the current look and content of form
end;                                      //loop until all the sites have been visited
quit;
```

Figure 11. A Migratory Application – Roving Poller

5. Sample Programs

We have identified three classes of distributed applications that can be programmed using Obtuse – Migratory, Synchronized, and Cooperative. A **migratory** application moves object state from one machine to another. This may involve moving all of a program, or only a part, such as the user interface. Implementing routing slips for documents is an example of a migratory application. A **synchronized** application is one in which the actions at one site are duplicated at another. For example, a debug console might be synchronized to the program or UI state of a distributed application. Finally, a **cooperative** application requires that multiple sites participate to solve problems. Decision-making tasks, such as preparing budgets for instance, are usually accomplished in a cooperative fashion.

Figure 11 lists the code for the Roving Poller example discussed in Section 1. It is an example of a migratory application. The program transfers itself to every machine in a list, which is specified on the command line, so that each user can enter comments in an edit control. The completed form, together with the accumulated comments, is displayed at the last machine in the list. A return-to-sender convention could be implemented by placing the name of the originating machine last in the argument list.

The second example, which is listed in Figure 12, is a simple, synchronized application that was written as a UI performance test. The idea was to synchronize a

```
var a, b, c, e, old, i, NBARS;
var aa, cc;
var resource;
resource :=
[12345,14,3,6120,4,6600,0,2880,1,1110,5,8454143,2,
"Scroll Test",23456,4,5,8454016,89013,8,3,16,4,16,0,153];
NBARS := 9;
aa := [0];                              //create an array dynamically
for i :=0 to NBARS-2 do                 // should probably be a function to do this
  aa := aa & [0];
end;
cc := aa;
a := object("Bob.Form");                //create the master scroll bar
for i:=0 to NBARS-1 do
  aa[i] := object("Bob.Form, "a-bobc-1"); //create N remote scrollbars
  b := aa[i];
  c := b.Restore{resource};
  c := b.Form{};                        //get an object reference to the form
  c.Top := (i%8)*1000;                  //place the scroll bars in a grid pattern
  c.Left := (i/8)*4000;
  cc[i] := b.Item{"hscroll0"};          //object reference to each scroll bar
end;
b := a.Restore{resource};
c := a.Item{"hscroll0"};
old := c.value;
loop
  e := c.value;
  if (e > 30000) then exit; end;        //exit when thumb moved to far right
  if e != old then                      //wait for a state change
    for i:=0 to NBARS-1 do              //update all the other scroll bars
      b := cc[i];
      b.value := e;
    end;
    old := e;
  end;
end;
quit;
```

Figure 12. A Synchronized Application – Master/Slave Scroll Bars

scroll bar on one machine with an arbitrary number of scroll bars on a second machine. The objective was to increase the number (NBARS) of scroll bars, and then to observe the impact on responsiveness.

The program actually does not scale very well, but the problem is with the algorithm, not DCOM. Distributing a signal to a large number of recipients should not be performed with a simple **for** loop, but rather with a distribution hierarchy.

The final example, which is listed in Figures 13 and 14, implements a cooperative application that supports a common decision-making task performed by three co-workers; that is, deciding where to go to lunch in a timely, and fair, fashion. When the application is initiated, it displays a form containing three edit controls at each site. The participants can type in their choice for lunch and can observe, but not modify, the other choices. Each user can change their mind arbitrarily, but at the instant that a majority has agreed on a choice, input is frozen and the consensus is displayed for all to see.

6. Performance Measures

Obtuse implements late binding of procedure calls and property access by using the Automation IDispatch technology. Obtuse marshals program and user interface state by encoding values in variant arrays. There is the question of what penalty is paid for the additional complexity.

We conducted performance tests in order to quantify some of the costs. The tests were conducted on two 166 Mhz Pentiums, which were on the same 10mb Ethernet segment, and which were running Windows NT 4.0. Table 1 lists the results of the tests. Each test program was run several times to verify that the results were stable and each test loop was repeated 100 to 10,000 times, depending on the amount of time involved. The test programs are listed at the Obtuse web site: obtuse.cs.olemiss.edu.

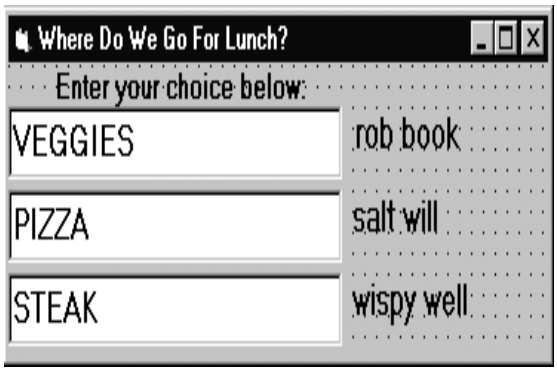

Figure 13. The Eat-Lunch User Interface

Obtuse has no program library to provide timing functions so the interpreter was modified such that a reference to the first global variable in a context returned the current time in milliseconds. Thus, two references to the same variable could be used as a timing function.

The test results require some explanation. First, there is a distinct time difference between variable access/function calls and invoking a method on an Automation object. The latter operation is slower because Obtuse must do a symbol table lookup to bind the method name to a method index. Access to Visual Basic properties or methods takes even longer, up to 1.6ms. Since there is no difference in the Obtuse runtime code between accessing a VB object and an Obtuse object (like Event), we can assume that the VB runtime contributes to the factor of 25 performance decrease.

The remote variable access only takes 4ms, versus 11ms for the function call, because context object access is routed though a proxy DLL on each machine whereas access to other Obtuse objects, such as EVENT or FILE, must use the IDispatch infrastructure. IDispatch requires two separate calls to the remote node for each method call: one call to bind the name and one to make the call.

Table 1. Obtuse Performance Measurements

Function	Timing	Software Layers
Global variable (in a Context)	0.016 ms	1:in-process proc call 2: array access
Local Obtuse function call	0.025	1:in-process proc call 2: variant copies
Local COM object function call	0.069	1:in-process call 2:COM runtime 3:IDispatch runtime
Local Visual Basic COM object function call	1.543	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Local VB property reference	1.610	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Local Clone task (one cycle)	4.400	1:variant copies(lots) 2:memory allocation 3:new OS thread 4:kill OS thread 5:memory free (all in process)
Remote global variable access (in a Context)	4.600	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:net transport

Remote COM object function call	11.020	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2)
Remote VB property reference	11.820	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Remote Visual Basic COM object function call	12.000	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Local Restore and Save of VB Form	43.760	1:cross-process call 2:COM runtime 3:IDispatch runtime 4:VB runtime
Remote Restore and Save of VB Form	81.100	1:cross-machine call 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:VB runtime
Remote Clone task and then remote clone back to the source (one cycle)	266.400	1:variant copies(lots) 2:COM runtime 3:DCOM runtime 4:IDispatch runtime 5:net transport (2) 6:new OS thread 7:kill OS thread 8:memory free

The final set of tests involved cloning a task to a target machine and then cloning that task back to the source. This provides an indication of the costs to migrate a process from one machine to another. The migration test was performed without also moving a user interface. The local test (in which the target and source were the same machine) took 4.4ms and the remote test took 266ms

7. Observations

The Obtuse system was designed and implemented by the author in eight weeks as part of a summer research appointment at Microsoft Corporation. As a result, both the language and runtime lack a number of features that can be found in more mature languages such as C++ or Java. Nevertheless, there are some lessons that can be shared based on the experience to date.

First, migrating a thread or object depends on support for marshaling its state. In theory, any system, such as C++ or Java, could support migration, and probably should.

The ability to transmit a user interface's state from one machine to another is also a useful capability. Visual Basic and its competitors could easily be extended to support persistence. The VB property model even lends itself to simple encoding strategies.

We found that support for marshaling the state of forms was essential to the implementation of migratory applications. Most Office applications, for example, marshal (or save) the current user interface state on exit, and restore it on startup. Java has provided entry points for marshaling UI state with the Applet methods init/destroy and start/stop. JDK 1.1 has additional support for serialization. Client-side Java has been joined by server-side Java. The next evolution should be mobile Java.

The most difficult implementation problems involved reference counting, which resulted in objects that never got released. For example, in testing the Roving Poller example, it was discovered that context objects were being activated, but never deleted, on machines all over our building. The problem turned out to be a circular reference; that is, the context objects had references to themselves. The problem was addressed by checking for circular references in both the stack and context object of a terminated thread. However, this approach would not handle indirect recursion through other context objects.

DCOM provides an excellent infrastructure for writing distributed applications. However, its OLE legacy is the source of a number of serious problems. The first, and most serious, problem is the OLE registry, which is used to store app/server/classId associations and access control information. As a result, Windows NT has two object information systems—the file system and the registry. Even worse, the DCOM configuration information can only be manipulated by the system administrator. Also, the tools that support changes to the registry are neither user friendly nor

fault tolerant. As a result, for multi-user systems, such as all the machines in the CS department, no students can create objects. This situation must be resolved before DCOM can find wide acceptance.

A less severe problem is the current specification of IDispatch/Variants, which were designed to support Visual Basic, and were later modified slightly to support Visual Basic for Applications. The design is not well-suited for supporting distributed applications. The concepts are correct, but the requirements have changed. As a result, the design needs to be upgraded. For example, the Variant value types are not extensible; variants only support what was required to implement Visual Basic. There is no reason to require reference counting for value-based types, such as complex numbers or x-y coordinates.

8. Related Work

Obliq depends on Modula-3 for its runtime support; Obtuse depends on COM/DCOM. Bharat's Visual Obliq also supports migrating user interfaces. Obtuse is unique in its integration of the COM object model in its design and in the support for persistent VB objects. Hirano's [6] HORB system is a Java superset that can be used to write distributed applications. Gray[7] has implemented a transportable agent system, Agent Tcl, and maintains an extensive "related work" site[8].

9. Acknowledgements

At Microsoft, Tony Williams conceived of, and initiated, the Obliq/DCOM summer project, which was supported by Nat Brown and Dennis Adler.

10. Availability

The Obtuse system is available for experimentation at the web site www.obtuse.cs.olemiss.edu.

References

1) Cardelli, L., Obliq: A language with distributed scope. Report No. 122, Digital Equipment Corporation, Systems Research Center, (1994).

2) Krishna Bharat and Luca Cardelli, Migratory Applications, Proceedings of ACM Symposium on User Interface Software and Technology '95, Pittsburgh, PA, (Nov 1995). http://www.cc.gatech.edu/gvu/people/Phd/Krishna/VO/Migration.html

3) Distributed COM, Microsoft Corporation (1996).http://www.microsoft.com/windows/common/aa2399.htm

4) The Component Object Model Specification, Microsoft Corporation (1996). http://www.microsoft.com/oledev/olecom/title.htm

5) Brown, Nat, and Kindel, Charlie. Distributed Component Object Model Protocol -- DCOM/1.0, (1996). http://ds.internic.net/internet-drafts/draft-brown-dcom-v1-spec-01.txt.

6) Hirano, Satoshi, The HORB System, (1996). http://ring.etl.go.jp/openlab/horb/

7) Gray, R.S. et al,. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Department of Computer Science, Dartmouth College, 1996.

8) Related Work, http://www.cs.dartmouth.edu/~agent/

```
var resource, a, b, c, d, e, f, g, h, i;
resource :=
[12345,12,3,-15,4,5535,1,2085,5,8454016,2,
"Where Do We Go For Lunch?",45678,12,2,
"Enter your choice below:",4,32,3,0,0,117,5,-2147483633,45679,14,2,
"   ",4,234,3,16,0,9,1,25,5,-2147483633,45680,12,2,
"",4,232,3,48,0,3,5,-2147483633,45681,12,2,
"",4,232,3,80,0,3,5,-2147483633,56789,12,3,16,4,0,0,225,1,25,6,
"",56790,12,3,48,4,0,0,225,1,25,6,
"",56791,12,3,80,4,0,0,225,1,25,6,""];
a := [ ["daddyo", "rob book"],  ["monroe", "salt will"],
    ["a-bobc-1", "wispy well"]];
g := [ ["label1", "text0"], ["label2", "text1"], ["label3", "text2"]];
b := [0,0,0];    i := [0,0,0];
for c:=0 to 2 do
  b[c] := object("Bob.Form", a[c,0]);          //create the form at each machine
  d := b[c];
  e := d.Restore{resource};
  e := d.Item{g[c,0]};                              //get an object reference to each label control
  e.caption := a[c,1];                         //set the "caption" property to the user's name
  for h:=0 to 2 do                             //lock all the edit fields except the user's
    f := d.Item{g[h,1]};
    f.locked := h!=c;
  end;
end;
loop
  for c:=0 to 2 do
    d := b[c];
    f := d.Item{g[c,1]};                            //retrieve the choices of the other users
    i[c] := f.text;
    for h:=0 to 2 do                           //update my controls to display their latest
      if h!=c then
        d := b[h];
        f := d.Item{g[c,1]};
        f.text := i[c];
      end;
    end;
  end;                                         //as soon as 2-of-3 match, let majority rule
  if (i[0]!="") & (i[0]=i[1]) then i[2] := i[1]; h := 2; exit; end;
  if (i[1]!="") & (i[1]=i[2]) then i[0] := i[1]; h := 0; exit; end;
  if (i[0]!="") & (i[0]=i[2]) then i[1] := i[0]; h := 1; exit; end;
end;
for c:=0 to 2 do                               //lock every control, display consensus everywhere
  d := b[c];
  f := d.Item{g[c,1]};
  f.locked := true;
  f.text := i[c];
  f := d.item{g[h,1]};
  f.text := i[c];
end;
```

Figure 14. Where-To-Eat-Lunch Application