Cover Feature

Component-Based APIs for Versioning and Distributed Applications

Constrained by the need to support legacy applications, current APIs tend to be large, rigid, and focused on a single host machine. Component-based APIs promise to solve these problems through strong versioning capabilities and support for distributed applications.

Robert J. Stets
University of Rochester

Galen C. Hunt
Microsoft Research

Michael L. Scott
University of Rochester

Operating system application programming interfaces (APIs) are typically monolithic procedural interfaces that address a single machine's requirements. This design limits evolutionary development and complicates application development for distributed systems.

An OS's functionality will change over its lifetime, and these changes must be reflected in the API. In an ideal world, obsolete API calls would be deleted, and calls with modified semantics (but unmodified parameters and return values) would remain the same. Unfortunately, since the OS must continue to support legacy applications, obsolete calls can never be deleted, and new call semantics are best introduced through new calls. The need for backward compatibility leads to bloat in the API and the supporting code.

In addition, typical OS APIs do not adequately address the needs of distributed applications: They have support for intermachine communication but lack high-level support for accessing remote OS resources. The primary omission is a uniform method for naming remote resources, such as windows, files, and synchronization objects.

ARE COMPONENTS THE ANSWER?

Component software methods, which have been primarily motivated by the desire for reuse, can eliminate these weaknesses. Software components are "binary units of independent production, acquisition, and deployment that interact to form a functioning system."[1] Component software environments are largely based on distributed platforms and, due to natural software evolution typically contain multiple active versions of components. Component infrastructures therefore have good distributed computing support and also have *versioning* support for managing multiple versions of code.

A *component-based* OS API is constructed entirely of software components, each modeling an OS resource. To facilitate independence, components necessarily encapsulate state and functionality. The component modeling an OS resource will therefore contain all necessary access and manipulation functions. This encapsulation allows the OS resource to be instantiated on a remote machine and also provides a natural unit for API versioning.

Only the API must be component-based. The underlying OS can be monolithic, a microkernel, or component-based. A component-based API controls access to the OS, which is sufficient to provide API versioning and to expose OS resources outside the host machine (*remoting*).

Our *component-based OS proxy* (COP), a prototype component-based API, acts like a traffic cop, directing OS requests to the appropriate version or resource location. COP currently targets the Win32 API and is implemented on top of Windows NT 4.0, covering approximately 350 Win32 calls. COP introduces a minimum of overhead in the local case while providing outstanding OS support for evolutionary development and distributed applications.

COMPONENT NUTS AND BOLTS

A component provides functional building blocks for a complex application:

- An *interface* is a contract that specifies how a component's functionality is accessed. Interfaces take the form of function or method calls, including parameter and return types.
- A *component instance* is a component that has been loaded into memory and is accessible by a client. All communication among component instances occurs through interfaces. Component software fundamentally maintains a strict separation between the interface and the implementa-

0018-9162/99/$10.00 © 1999 IEEE

tion, which is a key requirement because it forces components to encapsulate their functionality and guarantee component independence.

Several methodological concepts facilitate reuse in component software:

- *Independence* allows components to be composed without introducing implicit interactions, which may lead to subtle program errors.
- *Polymorphism* allows components with differing implementations of the same interface to be interchanged transparently. Allowing one component to substitute for another, as long as it provides equal or greater functionality than the original, enhances the ability to compose.
- Finally, *late binding* (delaying the point at which component choices are bound) allows an application to choose components dynamically.

Component infrastructures also contribute to reuse: All popular infrastructures provide mechanisms that allow development in multiple languages and execution across multiple hardware platforms.

Two of the more popular component infrastructures are Microsoft's Component Object Model (COM)[2] and the Object Management Group's Common Object Request Broker Architecture (CORBA).[3] Although originally motivated by different goals, these infrastructures have largely converged. Both promote software reuse, independent of development language, in both single-machine and distributed computing environments.

We built COP on top of COM.

Component Object Model

Microsoft developed COM to address application interaction. It introduced distributed COM (DCOM) extensions[4] to support distributed computing.

COM employs a binary standard to provide language independence. It implements component interfaces as a table of function pointers, which are called *vtables* because they mimic the format of C++ virtual function tables.

References to component instances are called *interface pointers*. Interface pointers are actually double-indirect pointers to the vtable. The extra level of indirection is an implementation convenience. For example, an implementation can attach useful information to the interface pointer that will then be shared by all references to the interface.

In keeping with component software methodology, COM maintains strict separation between a component interface and its implementation, addressing only the interface. Interfaces can be defined through single inheritance. Only the interface is inherited; the implementation is entirely separate. COM's real power comes from the ability of COM components to implement multiple interfaces regardless of inheritance hierarchy.

Globally unique class IDs (CLSIDs) identify COM components. Similarly, globally unique interface IDs (IIDs) specify all interfaces.

All COM interfaces must inherit from the IUnknown interface. IUnknown contains a QueryInterface() method and two methods for memory management. For our discussion, QueryInterface() is the most important. A client must use this method to obtain a specific interface pointer from a component instance.

Other Directions for Operating Systems

Kernel call *interposition* is the process of intercepting kernel calls and rerouting them to pieces of extension code that add or modify OS functionality.

COP uses interposition techniques, but our goal is a new style of API that provides versioning and distributed computing benefits. A general interposition system should be built on top of COP. Interposition agents[1] and SLIC[2] are full-featured interposition systems.

Other research efforts (for example, MMLite[3] and Jbed[4]) are considering entirely component-based OSs, which could be assembled dynamically to reflect the execution environment. To our knowledge, none of this work addresses API versioning or the naming of remote OS resources. These efforts require that the kernel be built from scratch, whereas our work can be easily applied to existing commercial OSs.

The work closest to our own is the Inferno distributed OS.[5] In this system, all OS resources are named and manipulated like files. This unique approach provides the advantage of a global, hierarchical namespace for all resources, with the disadvantage that the natural semantics for manipulating resources may not be maintained.

References

1. M. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface," *Proc. 14th Symp. Operating Systems Principles*, ACM Press, New York, 1993, pp. 80-93.
2. D. Ghormley et al., "SLIC: An Extensibility System for Commodity Operating Systems," *Proc. Usenix Ann. Tech. Conf.*, Usenix, Berkeley, Calif., 1998, pp. 39-48.
3. J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture," *Proc. Eighth ACM SIGOPS European Workshop*, ACM Press, New York, 1998, pp. 6-17.
4. *Jbed Whitepaper: Component Software and Real-Time Computing*, tech. report, Oberon Microsystems, Zürich, 1998.
5. S. Dorward et al., *Bell Labs Tech. J.*, Lucent Technologies Inc., Murray Hill, N.J., 1997.

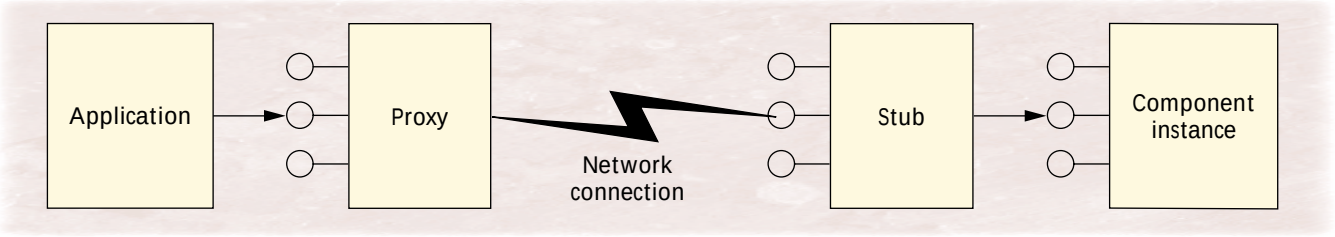

Figure 1. For a call to a remote component instance, the proxy marshals data arguments. The transport mechanism sends the request and data across the network. At the server, the stub receives the request, unmarshals the data, and invokes the requested interface function. The process is reversed for return values.

A client instantiates a component instance by calling the COM CoCreateInstance() function and specifying the desired CLSID and IID, and the COM runtime system returns a pointer to the desired interface. Given an interface pointer, the client can use QueryInterface() to determine if the component also supports other interfaces.

By convention, COM holds that all published interfaces are immutable in both syntax (interface method names and parameters) and semantics. If a change is made to an interface, a new interface with a new IID must be created. Immutable interfaces provide a very effective versioning mechanism. A client can request a specific interface and be assured that it supports the desired syntax and semantics.

Under COM, components can be instantiated in three different execution contexts: directly in the application's process (*in-process*), in another process on the same machine (*local*), or on another machine (*remote*). The ability to access instances regardless of execution context is called *location transparency*. COM provides location transparency by requiring that all instances be accessed through the vtable.

For in-process instances, the component implementation is usually held in a dynamically linked library (DLL) and is loaded directly into the process's address space. The vtable then points directly to the component implementation. For local or remote components, the component implementation is loaded into another process, and the application must engage in some type of interprocess communication (IPC). To handle these cases, COM instantiates a *proxy/stub pair* to perform the communication, as shown in Figure 1. The vtable is set to point directly to the proxy.

Before an IPC mechanism can be used, data must be packaged into a suitable transmission format (*marshaling*). The proxy is responsible for marshaling data and sending the data and the request to the component instance, where the stub receives the request, unmarshals the data, and invokes the appropriate method on the instance. The process is reversed for any return values.

A system programmer can customize the IPC mechanism. Otherwise COM defaults to using an optimized remote procedure call for the local case. For remote cases, COM uses an extension of the Open Group's Distributed Computing Environment remote procedure call (DCE RPC) facility.[5]

COM, CORBA, and a component-based API

COM and CORBA have many fundamental similarities, especially in the area of distributed computing. For remote communication, CORBA uses an architecture that is very similar to COM and offers essentially the same capabilities for remote component instances.

However, the systems differ greatly in their versioning capabilities. One CORBA implementation, IBM's System Object Model (SOM), builds interface specifications at runtime,[6] so interface methods can be added or reordered but not removed. SOM's strategy does not address semantic changes.

CORBA *repository IDs* could be used to uniquely identify interfaces in much the same manner as COM IIDs. However, repository IDs are only checked when an instance is created, not when an instance reference is obtained directly from another component instance. A more fundamental problem is that CORBA's conventional object model merges all inherited interfaces into the same name space, so it is impossible to simultaneously support multiple interface versions unless all method signatures are different. A component-based API built on top of CORBA would not be able to offer very robust versioning capabilities.

COP IMPLEMENTATION: FACTORING A MONOLITHIC API

The first step in constructing a component-based API is to split, or *factor*, the monolithic API into a set of interfaces. After factoring, the entire API should be modeled by the set of interfaces, with individual and independent OS resources and services modeled by independent interfaces. A good factoring scheme produces interfaces that are appropriately independent and provides the benefits of clarity, effective versioning, and clean encapsulation of resources.

Here we describe our factoring strategy for the Win32 API (the application of our strategy to a 1,000+ subset of Win32 is available elsewhere[7]). However, our strategy and techniques should be generally applicable to monolithic, procedural APIs.

Our factoring strategy involves three steps:

1. We factor the monolithic API calls into groups on the basis of functionality. For example, our strategy places all graphical window calls in an IWin32Window group (the IWin32 prefix denotes an interface to a Win32 API component).

API subset:

```
BOOL AdjustWindowRect(RECT *, DWORD, BOOL);
HANDLE CreateWindow(...);
int DialogBoxParam(...,HANDLE, ...);
int FlashWindow(HANDLE, ...);
HANDLE GetProp(HANDLE, ...);
int GetWindowText(HANDLE,...);
```

Final factorization:

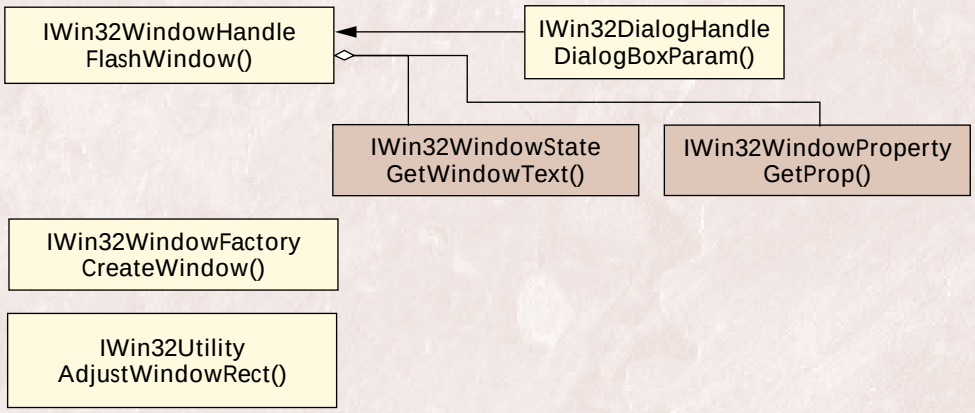

Figure 2. Factoring of a simple subset of the Win32 API. Proposed interfaces are listed in bold and prefixed with "IWin32."

2. We then factor the calls in each group into three subgroups, according to their effect on OS resources. A call's effect is easily identifiable through its parameters and return value. A loaded OS resource is exported to the application as an opaque value called a *kernel handle*. Calls that create kernel handles—that is, OS resources—are moved to a *factory* interface, and calls that query or manipulate these kernel handles are moved to a *handle* interface. Any calls that do not directly involve kernel handles are moved to a *utility* interface.
3. We further refine the factorization. For example, the first two steps do not capture a situation in which a monolithic API contains a set of calls that act on a number of different OS resources. We place the synchronization calls in an IWin32SyncHandle interface, the process calls in IWin32ProcessHandle, and the file calls in IWin32FileHandle. For correctness, the process and file interfaces should also include the synchronization calls. Because the process and file handles can be thought of as logically extending the functionality of the synchronization handle, we can model this relationship through interface inheritance: Both IWin32ProcessHandle and IWin32FileHandle inherit from the IWin32SyncHandle interface.

Figure 2 is an example of how we factored the Win32 window functions, focusing on a small but representative subset (six calls) of the 130+ window calls.

The AdjustWindowRect() call determines the necessary size of a window given specific settings. The second call, CreateWindow (), creates a window. The remaining calls execute a dialog box, flash the window's title bar, query various window properties, and return the current text in the window title bar.

These calls all operate on windows, so we first factor them to a windows group. Next, we further factor them on the basis of their use of kernel handles (HANDLE in Figure 2). Finally, we further factor the IWin32WindowHandle into IWin32WindowState and IWin32WindowProperty interfaces to make the API easier to read. These interfaces do not extend the IWin32WindowHandle interface but compose it. We model this relationship through interface aggregation. Also, we have factored the dialog calls into their own interface since the dialogs are logically extensions of plain windows. Again, this relationship is modeled through interface inheritance.

Properly applied, this factorization strategy will produce a set of interfaces, each with a tightly defined set of calls to access the appropriate underlying OS resource. The factorization will improve API clarity by clearly defining the specific methods for accessing each OS resource and the relationship between API calls. Versioning capabilities will also be improved, since modifications can be isolated within the affected interfaces. Finally, a good factorization inherently encapsulates functionality (and the associated state), which facilitates the remote instantiation of OS resources.

COP RUNTIME SYSTEM

At runtime, the application accesses the OS through the COP component layer (Figure 3). COP components implement the interfaces described above and, like them, can be roughly classified as factories, handles, or utilities.

Most applications will instantiate factory components during initialization and then use the factories to create OS resources during execution. A basic implementation of a factory component first invokes the OS to create the desired resource. The OS returns

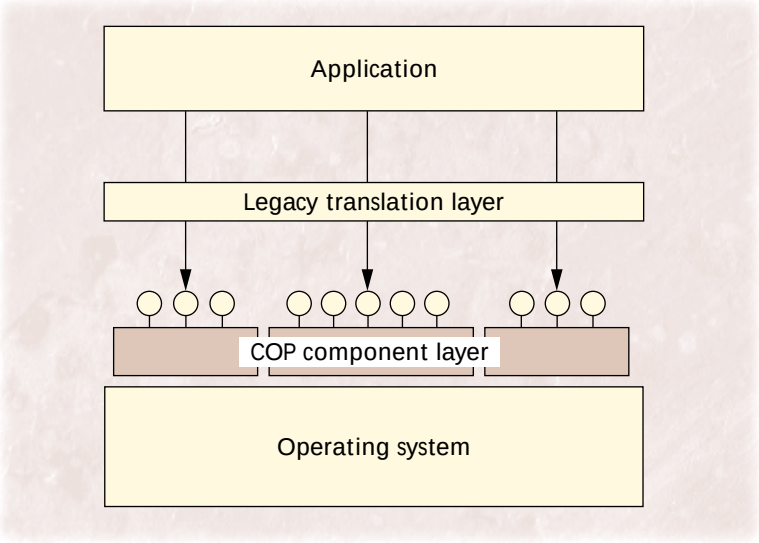

Figure 3. The COP runtime system consists of a component layer that presents the OS API and an optional legacy translation layer available for Win32 applications.

a kernel handle to identify the resource. This handle, however, is only valid on the local machine. To enable remote access to the resource, the factory also creates an instance of the associated handle component and stores the kernel handle in the instance's private state. Then, rather than returning the kernel handle, the factory returns a pointer to the instance of the handle component. The application makes subsequent accesses to the resource through the instance pointer.

Utility components do not directly manipulate loaded kernel resources but provide generic services. These components can be instantiated whenever necessary, anywhere in the system. Once they are instantiated, all accesses will occur through the instance pointer.

The instance pointer provides COP with one of its main advantages over typical modern OS APIs. Instance pointers uniquely name the loaded resource throughout the system and act as a gateway to the underlying remoting mechanism (COP/DCOM). With COP, applications can create resources throughout the system and subsequently use the instance pointer to access them in a location-transparent manner.

Versioning

COP's other main advantage over modern OS APIs—its versioning capability—follows directly from our factoring strategy and COM's robust versioning mechanism, including immutable interfaces and globally unique IDs.

To mark specific interfaces, an application can store the appropriate IDs in its data segment. Alternatively, the OS binary format could be extended to support static binding to a dynamic interface in the same way that current OSs support static binding to DLLs (or shared libraries). With such an extension, an application binary would declare a set of interfaces to which it should bind instead of a set of DLLs. Of course, COP-aware applications can query dynamically for special interfaces.

Location transparency

One of the main contributions of COP is the ability to instantiate OS resources anywhere throughout a distributed system, as shown in Figure 4. In-process components experience only the added overhead of an indirect function call. In the local case, a COM proxy/stub pair is used to marshal data across the process boundaries. (The local case is less efficient than the in-process case but provides better fault isolation.) The remote case uses the same general proxy/stub architecture. However, in the remote case, COP also includes an optional proxy manager that can be used to optimize remote communication.

A proxy manager, for example, caches remote data in an effort to avoid unnecessary communication. COP caches information to improve the redrawing of remote windows, for example. The Win32 call BeginPaint() signals the beginning of a redraw operation by creating a new drawing context resource. To be available remotely, this resource must be wrapped by a COP component. Rather than creating a new component instance on each redraw operation, COP caches a component instance in the proxy manager and reuses the instance for the redraw wrapper.

Although hidden from the application, extra state is obviously required to maintain location transparency. For example, handle components must store the value of their associated kernel handle, and optional proxy manager implementations may also require extra state. COM maintains this state automatically; COP components often have little extra state to maintain.

In a less common case, some components need extra state to maintain location-transparent results. The different execution contexts—in-process, local, or remote—may cause some calls to execute differently (of course we try to maintain the same operation as the normal Win32 API). For example, the call RegisterClass() registers a window class for use within a process and returns an error if the class is already registered. A naïve component implementation could report this error incorrectly in some cases. In COP, this call falls under the IWin32WindowUtility interface (since it does not target kernel handles).

Consider the case where two applications try to register the same class on the same remote machine. To access RegisterClass(), both applications would create an instance of IWin32WindowUtility. Since these instances will both be remote and on the same machine, COM creates the instances inside the same process to optimize performance. Note that the instances are separate COM instances but share the same process. The first application to register the class will succeed, but the second application will fail since the class has already been registered inside the COM process.

In attempting to mimic standard Win32 operation, this error would be incorrect since the application processes are separate. In COP, the IWin32WindowUtility implementation maintains a list of classes each

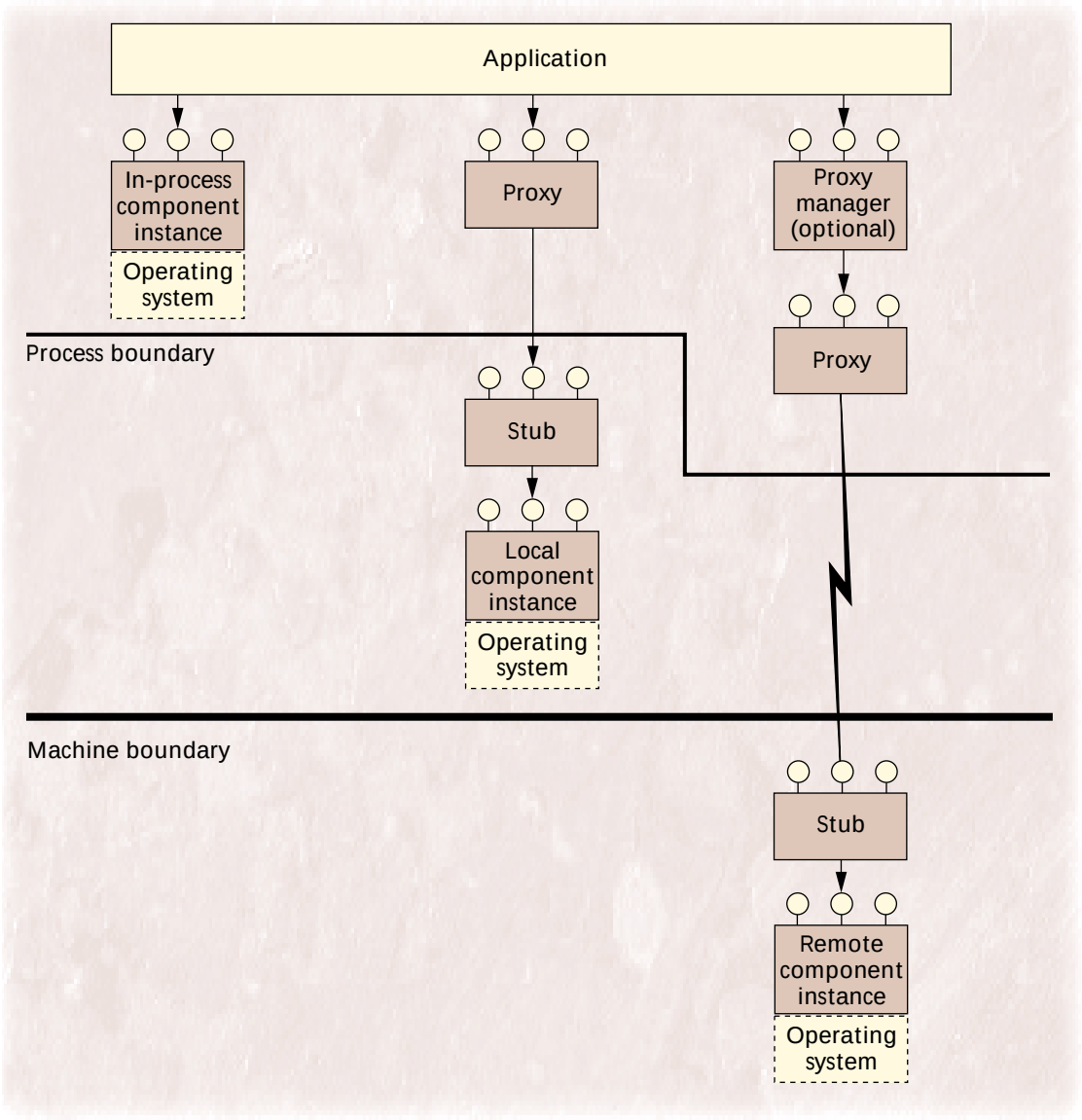

Figure 4. COP is able to instantiate OS resources in several locations: in-process, local, or remote. The client application can still access the resources in a location-transparent manner through the proxy manager, proxy, and stub components.

process has registered. The implementation can then determine if the caller has already registered the specified class and avoid spurious errors.

Obstacles to remote access of OS resources

OS callback functions are a significant obstacle to remote execution. Numerous API functions contain callback functions that the caller specifies and the OS invokes on certain events. This is a problem when the caller—that is, the location of the callback function—is on a different machine from the OS invoking the callback. COP solves this in the same way that it remotes OS resources: It wraps all callback functions with components. Instead of passing the address of the callback function, COP passes a pointer to the component instance wrapping the callback, which the OS can then use to invoke the callback function in a location-transparent manner.

Asynchronous events are the other main obstacle to remote execution. Some OS resources—such as windows, synchronization objects, and asynchronous I/O—must respond to asynchronous events. In all these cases, the OS assumes all involved parties reside on the same machine. COP therefore needs to provide extra support to remote these types of resources.

COP accomplishes this by creating a special *event agent* on the remote machine that is responsible for fielding asynchronous events and forwarding them to the client application. For example, COP supports remote windows. Windows must respond to asynchronous events such as mouse clicks and redraw events. In the normal case, the OS invokes an application-specified window procedure, a special case of a callback routine, on each window event. At window creation time, COP wraps the application's window procedure in a component instance. This wrapper allows the window procedure to be subsequently invoked in a location-transparent manner.

COP then creates an IWin32WindowHandle instance on the remote machine. This instance creates the window, stores a pointer to the component wrapping the application's window procedure, and attaches its own window procedure to act as an event agent. This agent forwards events back to the client simply

Component software provides excellent support for the evolutionary development of software and for distributed computing.

by invoking methods in the stored component wrapper.

COM delivers remote function call requests to COP components through a standard message queue. An idle component instance simply spins on the message queue, waiting for function call requests. Fortuitously, window events are delivered through the same message queue. In the course of polling for incoming requests, the IWin32WindowHandle instance will also discover pending window events and can then use the stored instance pointer to send the messages to the application's window procedure for processing.

Synchronization and asynchronous I/O can be handled in the same manner—an event agent can be instantiated on the remote machine. The agent will wait for the desired event and then forward notification to the application via a callback component.

Legacy translation layer

Our ultimate intention is for applications to write directly to the COP API. To ease the transition and to support legacy applications that cannot be rewritten, we have built an optional COP translation layer (shown in Figure 3). This layer is responsible for intercepting the procedural Win32 calls and translating them to COP. To help minimize translation overhead, we have designed the COP interface methods to use the same parameters as their Win32 counterparts.

Runtime interception is performed with the Detours package,[8] which has the ability to instrument an application's binary file by adding a specified DLL to the start of the list loaded at program initialization. This ensures that the specified DLL is the first loaded by the application.

We use Detours to place our COP start-up DLL at the start of the list. The start-up DLL then uses the Detours package to intercept and reroute Win32 calls to the legacy translation layer. Detours performs the interception by rewriting the first few instructions of a subroutine so that upon entrance, control is automatically transferred to a user-defined detour function. The replaced instructions are combined with a jump instruction to form a *trampoline* that can serve as the entrance to the original subroutine. The detour function can call the trampoline code to invoke the original subroutine, in our case the original Win32 call.

The legacy translation layer is responsible for creating the COP factory and utility instances as necessary (the handle instances are created by the factory instances). The layer of course caches pointers to interfaces to avoid unnecessary overhead. This approach works well for existing, single-machine Win32 applications and also allows the functionality of these applications to be transparently extended.

The translation layer can be configured to automatically create resources on remote machines. For example, all window resources can be started on a remote machine, very similar to X Window[9] remote displays. We have used this feature to remote the display of several existing Win32 applications. A remote display, however, only leverages a small amount of COP's most powerful feature—the ability to trivially connect to resources scattered throughout a distributed system.

The design of the translation layer is relatively straightforward, but one significant problem arises. Our translation layer intercepts all invocations of a specified call, even if the call is invoked from within another Win32 call, which could cause reentrance problems. The legacy translation layer tracks when an application is inside a Win32 call and avoids COP handling if an infinite recursion would start.

COP FULFILLS THE PROMISE

The initial goal for COP is to support the development of the Microsoft Research Millennium system (http://research.microsoft.com/research/sn/Millennium). Millennium will monitor the execution of a distributed component-based application and intelligently distribute the component instances to maximize performance. As components are distributed throughout the system, they still must be able to access remote OS resources. COP provides that capability.

To this end, we have developed remote access support for the registry, windows, graphic device interface (the low-level drawing routines), and file APIs. This subset consists of approximately 350 calls, enough to support the development of Millennium, including support for legacy applications provided by a legacy translation layer.

The Millennium system seeks to maximize application performance, so COP should not introduce significant overhead. To gauge the overhead, we performed two benchmark tests on a Gateway 2000 machine with a Pentium II processor running at 266 MHz. The machine has a 512-Kbyte off-chip cache and 64 Mbytes of RAM. Our benchmark timings were calculated based on the Win32 QueryPerformanceCounter() call, which has a resolution of approximately one microsecond on our machine.

Our first benchmark focused on estimating the overhead of our legacy translation layer. We measured the time required to call through the legacy translation layer to a COP component instance. For an in-process COP component, the call can be executed in 1.3 microseconds. If the component instance is instantiated as a local server (in another process), the call time jumps to 200 microseconds due to the crossing of process boundaries.

The second test was performed to examine the full overhead on the Win32 RegEdt32 application. The

plain application (with no COP overhead) starts in 0.833 seconds; using COP in-process components, it starts in 1.118 seconds, a 34 percent increase. A large amount of this overhead is due to the cost of instantiating the components, which would be amortized in a normal situation. Using COP local components, the application starts in 5.296 seconds, with the increase due to the frequent process boundary crossings.

We did not benchmark COP with remote components since the choice of network will have a strong influence on the results. These results show that in-process COP components add an acceptable amount of overhead while providing benefits in versioning management. When COP components are moved to remote machines, the overhead will be much higher, but network transmission time will be the dominant concern. Nevertheless, the functionality of the system will be much greater: An application can easily access scattered, remote OS resources.

Component software provides excellent support for the evolutionary development of software and for distributed computing. With an OS API based on components, a system can gain considerable leverage in these areas. The OS can export different versions of the API, allowing the API to be modified without jeopardizing legacy applications. Instead, the support for legacy applications can be dynamically loaded. By modeling the OS resources as components in the API, a global name space is created. An application can instantiate and manipulate any number of resources scattered throughout a distributed system. Natural access semantics for the remote resources is maintained by virtue of the encapsulation of functionality inherent in components. Applications will no longer have to rely on ad hoc methods to access remote resources.

Future work on COP will focus on increasing coverage of the Win32 API. We are also interested in methods to provide consistent, global view and management of resources throughout a cluster as well as fault tolerance and security throughout the system. ❖

Acknowledgments

The AST Toolkit from the Semantics-Based Tools Group at Microsoft Research was instrumental in building our component-based API.

References

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, ACM Press, New York, 1998.
2. *The Component Object Model Specification*, Microsoft Corp. and Digital Equipment Corp., Redmond, Wash., 1995.
3. *The Common Object Request Broker: Architecture and Specification*, Rev. 2.0., Object Management Group, Framingham, Mass., 1996.
4. *Distributed Component Object Model Protocol*, Ver. 1.0, Microsoft Corp., Redmond, Wash., 1998.
5. D. Hartman, "Unclogging Distributed Computing," *IEEE Spectrum*, May 1992, pp. 36-39.
6. I.R. Forman et al., "Release-to-Release Binary Compatibility in SOM," *Proc. 10th Ann. Conf. Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, New York, 1995, pp. 426-438.
7. R. Stets, G.C. Hunt, and M.L. Scott, *Component-Based Operating System APIs: A Versioning and Distributed Resource Solution*, Tech. Report MSR-TR-99-24, Microsoft Research, Redmond, Wash., June 1999.
8. G. Hunt, *Detours: Binary Interception of Win32 Functions*, Tech. Report MSR-TR-98-33, Microsoft Research, Redmond, Wash., 1998.
9. R. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Graphics*, Apr. 1986, pp. 79-109.

Robert J. Stets is a PhD candidate in the Computer Science Department at the University of Rochester. His research interests are in cluster-based operating systems and parallel computer architecture. Stets received a BSE in electrical engineering from Duke University and an MS in computer science from the University of Rochester.

Galen C. Hunt is a researcher at Microsoft Research. His research interests include inherently distributed systems, binary rewriting techniques, and application runtime environments. Hunt received a BS in physics from the University of Utah and a PhD in computer science from the University of Rochester.

Michael L. Scott is chair of the Computer Science Department at the University of Rochester, where he is the coleader of the Cashmere Shared Memory Project. His research interests include operating systems, programming languages, and program development tools for parallel and distributed computing. Scott received a PhD in computer science from the University of Wisconsin, Madison.

Contact Stets and Scott at the Department of Computer Science, University of Rochester, Rochester, NY 14627, {stets, scott}@cs.rochester.edu. Contact Hunt at Microsoft Research, One Microsoft Way, Redmond, WA 98052, galenh@microsoft.com.