Building Evolvable Systems: The ORBlite Project

One critical requirement that HP has learned over the years from building large systems is the need for the system and its components to be able to evolve over time. A distributed object communication framework is described that supports piecewise evolution of components, interfaces, communication protocols, and APIs and the integration of legacy components.

by Keith E. Moore and Evan R. Kirshenbaum

Hewlett-Packard has been building distributed and parallel systems for over two decades. Our experience in building manufacturing test systems, medical information systems, patient monitoring systems, and network management systems has exposed several requirements of system and component design that have historically been recognized only after a system has been deployed. The most critical of these requirements (especially for systems with any longevity) is the need for the system and system components to be able to evolve over time.

The ORBlite distributed object communication infrastructure was designed to meet this requirement and has been used successfully across HP to build systems that have evolved along several dimensions. The ORBlite framework supports the piecewise evolution of components, interfaces, communication protocols, and even programming APIs. This piecewise evolution enables the integration of legacy components and the introduction of new features, protocols, and components without requiring other components to be updated, ported, or rewritten.

A vertical slice through the ORBlite framework forms the basis of HP's ORB Plus product, a strict implementation of the CORBA 2.0 standard.

The Problem of Evolvability

By definition, a distributed system is one that contains components that need to communicate with one another. In most practical systems, however, many of these components will not be created from scratch. Components tend to have long lifetimes, be shared across systems, and be written by different developers, at different times, in different programming languages, with different tools. In addition, systems are not static—any large-scale system will have components that must be updated, and new components and capabilities will be added to the system at different stages in its lifetime. The choice of platform, the level of available technology, and current fashion in the programming community all conspire to create what is typically an integration and evolution nightmare.

The most common solution to this problem is to attempt to avoid it by declaring that all components in the system will be designed to a single distributed programming model and will use its underlying communication protocol. This tends not to work well for several reasons. First, by the time this decision is reached, which may be quite early in the life cycle of this system, there may already be existing components developers desire to use, but which do not support the selected model or protocol. Second, because of the availability of support for the model, the choice of model and protocol may severely restrict other choices, such as the language in which a component is to be written or the platform on which it is to be implemented.

Finally, such choices tend to be made in the belief that the ultimate model and protocol have finally been found, or at least that the current choice is sufficiently flexible to incorporate any future changes. This belief has historically been discovered to be unfounded, and there does not appear to be a reason to believe that the situation has changed. Invariably, a small number of years down the road (and often well within the life of the existing system), a new "latest-and-greatest" model is invented. When this happens, the system's owner is faced with the choice of either adhering to the old model, which may leave the system unable to communicate with other systems and restrict the capabilities of new components, or upgrading the entire system to the new model. This is always an expensive option and may in fact be intractable (e.g., one HP test system contains an investment of over 200 person-years in legacy source code) or even impossible (e.g., when the source code for a component is simply not available).

An alternative solution accepts the fact that a component or set of components may not speak the mandated "common protocol" and instead provides proxy services (protocol wrappers or gateways) between the communication protocols. Under this scheme, the communication is first sent to the gateway which translates it into the nonstandard protocol and forwards it on to the component. This technique typically gives rise to the following issues:

Issue	Typical Cause
Degraded performance	Message forwarding
Resource use	Multiple in-memory message representations
Reliability	The introduction of new messages and failure conditions
Security, location, configuration, and consistency	Disjoint mechanisms used by different communications protocols

It is tempting to think that the problem of evolvability is merely a temporary condition caused by the recent explosion in the number of protocols (and things will stabilize soon) or that the problem is just an artifact of poor design in legacy components (and won't be so bad next time). It appears, however, that this problem of protocol evolution is intrinsic in building practical distributed systems. There will always be protocols that are claimed to be better, domain-specific motivations to use them, and legacy components and protocols that must be supported. Indeed, we consider it a truism that nearly any real distributed system will have at least three models: those of legacy components, the current standard, and the emerging latest-and-greatest model. The contents of these categories shift with time—today's applications and standard protocols will be tomorrow's legacy.

Dimensions of Evolution

The ORBlite architecture is concerned with multiple dimensions of evolution.

Evolution of Component Interface. A component's interface may evolve to support new features. The danger is that this evolution will require all clients of the component to be updated. For reasons cited in the previous section, there must be a mechanism whereby old clients can continue to use the old interface and new clients can take advantage of the new features.

Evolution of Component Implementation. A component's implementation may evolve independently of the rest of the system. This may include the relocation of a component to a new hardware platform or the reimplementation of a component in a new programming language. There must be a mechanism that insulates other components from these changes in the implementation yet maintains the semantic guarantees promised by the interface.

Evolution of Intercomponent Protocol. It is generally intractable to choose a single communication protocol for all components in the system. Different protocols may be more attractive because of their performance, availability, security, and suitability to the application's needs. Each communication protocol has its own model of component location, component binding, and often data and parameter representation. It must be possible to change or add communication protocols without rendering existing components inaccessible.

Evolution of Intercomponent Communication Model. The programming models used to perform intercomponent communication continue to evolve. They change over time to support communication of new types of data and new version communication semantics. At the same time, new programming models are frequently developed. These models are attractive because of their applicability to a particular application, because of their familiarity to programmers on a particular platform, or because they are merely in fashion or in corporate favor. It must be possible to implement components to a new model or a new version of an existing model without limiting the choice of protocols to be used underneath. It must also be possible to do so without sacrificing interoperability with existing components written to other models or other versions of the same model (even when those components will reside in the same address space).

Contribution of Distributed Object Systems

Distributed object systems such as the Object Management Group's CORBA (Common Object Request Broker Architecture)[1,2] and Microsoft's® OLE (Object Linking and Embedding),[3] like the remote procedure call models that preceded them, address the issue of protocol evolution to a degree by separating the programming model from the details of the underlying protocol used to implement the communication. They do this by introducing a declarative Interface Definition Language (IDL) and a compiler that generates code that transforms the protocol-neutral API to the particular protocol supported by the model (see Fig. 1). As the protocol changes or new protocols become available, the compiler can be updated to generate new *protocol adapters* to track the protocol's evolution. These adapters are shown as *stubs* and *skeletons* in Fig. 1.

Another benefit of IDL is that it forces each component's interface to be documented and decouples a component's interface from its implementation. This allows an implementation to be updated without affecting the programming API of clients and simplifies the parallel development of multiple components.

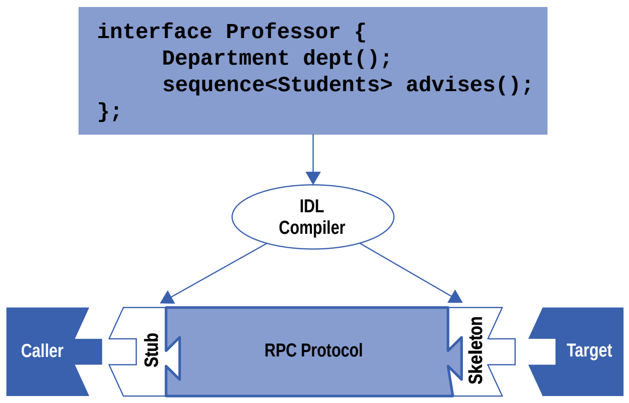


Fig. 1. *Generating stubs and skeletons from IDL. The stub and skeleton serve as software protocol adapters, which can be updated as a protocol evolves.*

In CORBA and OLE, interfaces are reflective—a client can ask an implementation object whether it supports a particular interface.* Using this dynamic mechanism, a client can be insulated from interface and implementation changes. Clients familiar with a new interface (or a new of an existing interface) ask about it, while old clients restrict themselves to using the old interface.

While such systems abstract the choice of communication protocol, none addresses the situation in which a system needs to be composed of components that cannot all share a single protocol or a single version of a protocol.** CORBA and OLE have each defined a protocol that they assert all components will eventually adopt. For reasons cited above, we feel that each is merely adding yet another (incompatible) protocol to the mix—a protocol that will continue to evolve.

Key Contributions of ORBlite

The ORBlite distributed object-oriented communication framework was designed with these concerns in mind. It takes the protocol abstraction provided by IDL a step further by allowing a single component to be accessed and to communicate over multiple protocols and multiple versions of the same protocol, simultaneously and transparently. Centered around the notion of the declarative interface, ORBlite also provides for different components to be written to different models, even when the components reside in the same process. The result is that programmers are presented with the illusion of the entire system adhering to their processing model regardless of whether this is true or in fact whether the component at the other end is even implemented using the ORBlite framework. It further enforces the notion that programming models and protocols have no knowledge of one another with respect to either existence or implementation, allowing the programmer complete freedom to mix and match.

ORBlite departs from the traditional client/server model by treating caller (client) and target (server) as merely roles relative to a particular call. Any process can contain objects that act as both callers and targets at different times or even simultaneously. Thus, ORBlite is fundamentally a peer-to-peer model even though a particular system may elect to follow a strict client/server distinction.

The main goal of the framework is to provide an efficient, thread-safe communication substrate that allows systems to be composed of components whose protocols, language mappings (i.e., object models), implementations, clients, interfaces, and even interface definition languages can evolve independently over time. It must be possible for protocols to evolve or be added without requiring recompilation of components, for object models to evolve without obsoleting existing components (or existing protocols), and for legacy components to be integrated without requiring reengineering. The reality of systems development is that components have different owners, different lifetimes, and different evolutionary time frames.

One further contribution of the ORBlite framework is that it treats local and remote objects identically. In most current systems, the syntax for a call to a remote object is quite different from a call to one located in the same process. As a result, once code has been written with the assumption that a particular object is local or remote, this decision becomes difficult to change. ORBlite, by contrast, encourages the programmer to talk in terms of *distributable references* (i.e. references to objects that may be local or remote), even when the referenced object is believed at coding time to be coresident. Application code that uses a distributable reference will not need to be changed if the referenced object is later moved to a remote process. The framework provides extremely efficient dispatching for calls when the object is detected to be coresident. The use of distributable references allows the assignment of objects to processes to be delayed well past coding time and to be adjusted based on performance or other requirements.

* In CORBA C++ this is a dynamic _narrow() mechanism. In OLE it is the IUnknown::QueryInterface() mechanism.

** The term protocol in this article refers to more than just the transport protocol. For example, the DCE protocol supports multiple string-binding handles so that objects can be accessible over connectionless and connection-based transports. However, programs based on the DCE RPC model cannot transparently communicate with programs based on the ONC RPC model.

The ORBlite Communication Framework

The ORBlite communication framework contains a core and three key abstraction layers: the language mapping abstraction layer, the protocol abstraction layer, and the thread abstraction layer (see Fig. 2). The core is responsible for behavior that is not specific to any particular protocol or language mapping. This includes the management of object references and the lifetime of target implementations, the selection of the protocol to use for a particular call, and the base data types used by the protocols and the language mappings to communicate.

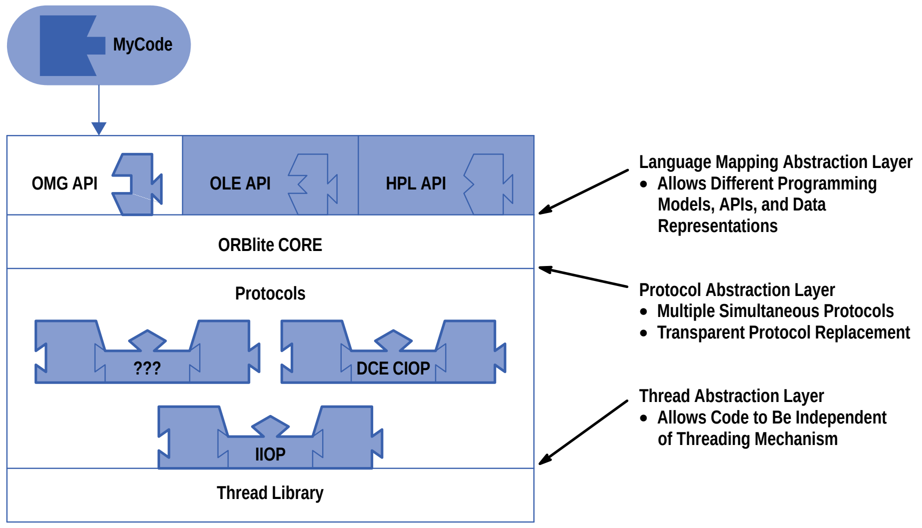

MyCode
OMG API
OLE API
HPL API
ORBlite CORE
Protocols
???
DCE CIOP
IIOP
Thread Library
Language Mapping Abstraction Layer
• Allows Different Programming Models, APIs, and Data Representations
Protocol Abstraction Layer
• Multiple Simultaneous Protocols
• Transparent Protocol Replacement
Thread Abstraction Layer
• Allows Code to Be Independent of Threading Mechanism

Fig. 2. *An overview of the ORBlite architecture.*

Language Mapping Abstraction Layer

This layer is designed to support evolution of the programming model presented to the application. Using the language mapping abstraction layer, each component views the rest of the system as if all other components (including legacy components) followed the same programming model. An OLE component, for example, views remote CORBA components as if they were OLE components, and a CORBA component views remote OLE components as if they were CORBA components.* This abstraction layer allows components to follow multiple programming models even when the components are located in the same address space.

Protocol Abstraction Layer

This abstraction layer is designed to support the evolution of protocols and the choice of protocol sets available in a particular system. In addition, it decouples the in-memory representation expected by a particular language mapping from the protocol used to communicate between components on a given call. For example, implementations of DCE RPC assume that the in-memory image for a structure has a particular memory alignment and member ordering. ONC RPC, on the other hand, has a different assumption about how memory should be layed out.[4,5] The protocol abstraction layer allows a given language mapping to transparently satisfy both without restricting its own layout decisions.

The protocol abstraction layer provides several features:

- Support for multiple simultaneous communication protocols—services can be shared across communication protocols and components can interact with objects simultaneously over multiple protocols.
- Support for transparent protocol replacement—one protocol can be replaced with another protocol without any change to application code. Available protocols are declared at link time or are dynamically loaded. No recompilation is necessary to change the available protocol set.
- Support for legacy integration—the framework does not need to be on both sides of the communication channel. Each protocol has full control over message representation, enabling a protocol to be used to communicate with non-ORBlite components.
- Support for multiple in-memory data representations—applications can choose the in-memory representation of data structures without incurring copy penalties from the protocols.

* The mapping between CORBA and OLE was standardized by OMG and is detailed in reference 3.

Thread Abstraction Layer

This layer is designed to provide a portability layer such that components can be written to be independent of platform-specific threading mechanisms. The thread abstraction layer also serves to coordinate the concurrency requirements of the various protocol stacks. When a protocol can be written in terms of the thread abstraction layer, it can coexist with other communication protocols in the same process. All parts of the ORBlite framework are written in a thread-safe

and thread-aware manner. The framework manages object lifetimes to ensure that multiple threads can be exploited and simultaneous calls can be executing safely in the infrastructure and in each object.

These three abstraction layers are strongly interrelated. A protocol that obeys the protocol abstraction layer will typically use the language mapping abstraction layer to marshal and unmarshal data structures. A language mapping, such as the OMG C++ mapping, will in turn use the protocol abstraction layer to allow the protocol to marshal the structure in the protocol's preferred representation.

Conceptual Overview of an ORBlite Call

In ORBlite, there are six major pieces involved in a distributed call. These pieces are shown in Fig. 3. In systems that include legacy components, two of these pieces might be purely conceptual. A legacy server might not have a discernible skeleton or an identifiable implementation, yet will honor the wire protocol. Likewise, a legacy client may not have a real stub.

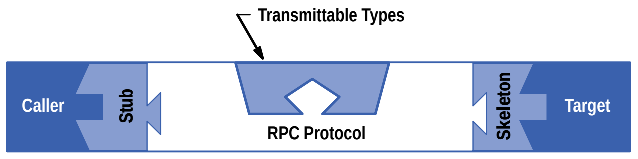
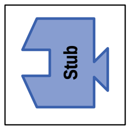


Fig. 3. *The pieces involved in a distributed call.*

The ORBlite model is similar to the CORBA and OLE models, except that in ORBlite an IDL compiler, for a given language mapping, emits stubs, skeletons, and types that are protocol-neutral. ORBlite further allows the caller and stub to follow a different language mapping from the skeleton and implementation.



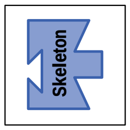
Stub. The stub is responsible for turning a client-side, language-mapping-specific call of the form:**

```
result = object.foo(a,b,c);
```

into the protocol-neutral form:

```
ORBlite::apply(object, ”foo”, arglist);
```

Essentially, the stub is saying to the ORBlite core, "invoke the method named "foo" on the implementation associated with object using the list of arguments in arglist."



Skeleton. The skeleton is primarily responsible for the reciprocal role of turning a call of the form:

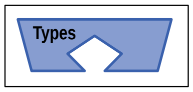
```
ORBlite::apply(object, ”foo”, arglist);
```

back into a call of the form:

```
result = impl.foo(a,b,c);
```

The stub can be viewed as the constructor of a generic call frame. The skeleton can be viewed as a call-frame dispatcher.

** The examples here use C++ syntax. The actual call syntax is a property of the language mapping. Also, note that the internal calls described here have been simplified.

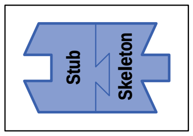


Transmittable Types. A language mapping defines one or more in-memory data representations or classes for each type (e.g., structure, union, interface, any,* etc.) describable in its IDL. For such data to be passed to a protocol, it must inherit from an ORBlite-provided base class TxType. Such classes are called *transmittable types* and support methods that allow protocols to request their instances to *marshal* themselves or to *unmarshal* themselves from a marshalling stream.** Occasionally, a language mapping may have a specification that precludes the types presented to the programmer from inheriting from TxType. In such cases, the IDL compiler often emits parallel transmittable classes that wrap the user-visible classes. These parallel classes are the ones presented to the core or to the protocols.

By convention, the marshalling methods are implemented in terms of requests on the stream to marshal the instance's immediate subcomponents. As an example, an object representing the mapping of an IDL sequence will marshal itself by first requesting the marshalling of its current length and then requesting the marshalling of each of its elements. ORBlite contains abstract transmittable base classes for each of the types specifiable in CORBA IDL, which implement the canonical marshalling behavior. Thus, the classes defined by a language mapping typically provide only methods that make a reference to or marshal the subcomponents

When a protocol's marshalling stream receives an instance of a transmittable type, it typically responds by simply turning around and asking that instance to marshal itself. Occasionally, however, a protocol may have special requirements for the wire representation (as with DCE's padding requirements for structures). Transmittable types provide type-safe accessors (foreshadowing C++'s recent dynamic_cast() mechanism) which allow a marshalling stream to ask, for example, "Are you a structure?," and take action accordingly, often calling the transmittable type's subcomponent marshalling methods directly.

The marshalling capability also provides transmittable types with the ability to convert from one language mapping's inmemory representation to another's (or between a single language mapping's distinct in-memory representations for the same type). As long as the two data types assert that they represent the same external IDL type, they can use a highly optimized in-memory marshalling stream to perform the conversion with the source object marshalling and the sink unmarshalling.



Local Bypass Optimization. When the stub and the skeleton exist in the same process space, the stub can directly invoke the skeleton's methods and bypass the transformation to and from the apply() call. In this case, the call:

```
result = object.foo(a,b,c);
```

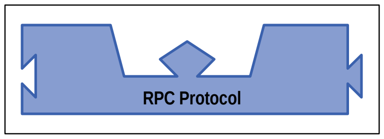
is directly forwarded through the skeleton using

```
result = impl.foo(a,b,c);
```

Note that the signatures for these two calls do not need to be identical.

An implementation object can disable this optimization. This is useful when an object wishes to ensure that a protocol has an opportunity to service every invocation, even those that are local. Certain logging, high-availability, and release-to-release binary compatibility mechanisms require this form of protocol intervention, even for the local case.

When the stub and skeleton reside in the same process but follow different language mappings, the stub may not know the target implementation object's calling conventions, or the argument data may not be in the appropriate form. When this happens, the local bypass is not taken. Instead, the call is routed through the protocol abstraction layer, which will use a very efficient local procedure call (LPC) protocol. This protocol behaves like a full RPC protocol (see below), but instead of marshalling its argument list, it merely tells the arguments to convert themselves from the caller's format to the target's.



RPC Protocol. The RPC protocol is primarily responsible for implementing a distributed apply() call. It works in cooperation with the transmittable types to migrate a call frame from one process space to another. ORBlite does not require that the protocol actually be an RPC protocol, only that it be capable of presenting the semantics of a thread-safe distributed apply( ) call. Asynchronous and synchronous protocols are supported, and it is common for more than one protocol to be simultaneously executing in the same process. The protocol may also be merely an adapter which is only capable of producing the wire protocol required for a particular remote interface but is not a full RPC implementation.

The separation between the *transmittable types' marshallers* and the RPC protocol means that transmittable types can be reused across different RPC protocols (see Fig. 4). An additional benefit is that adding a new custom protocol is fairly straightforward because almost all of the complex marshalling is handled outside of the protocol layer.

All RPC protocols have the same shape, meaning that each protocol obeys the protocol abstraction layer. There are well-defined interfaces for how a stub interacts with the protocol, how the protocol interacts with the marshallers, and how the protocol interacts with the skeleton.

* A self-describing type that can hold an instance of any IDL-describable type.

** Marshalling is the process of serializing a data structure into a buffer or onto a communication stream such that the resulting data stream is sufficient to recreate or initialize an equivalent object. Unmarshalling is the opposite process of reading the stream and creating or initializing the object.

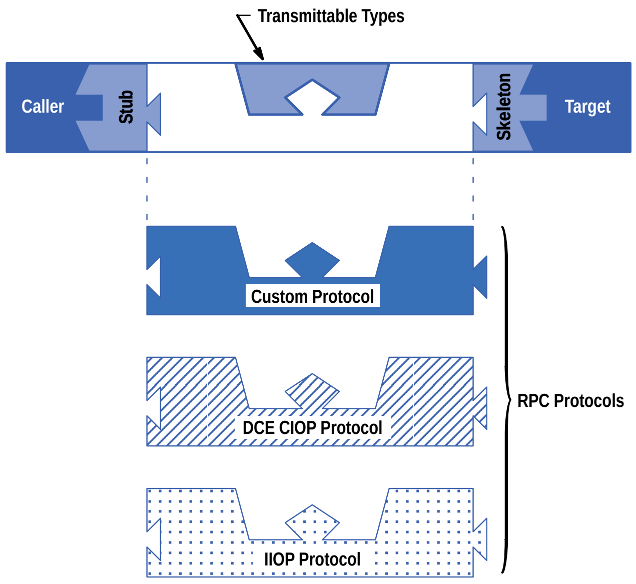


Fig. 4. *Alternate RPC protocols.*

These interfaces are, however, logically private in that they are not directly exposed to the client or to the implementation. Keeping these interfaces private means that the system can dynamically choose, based upon a variety of variables, which protocol should be used to connect a particular client to a particular implementation for a particular call. Examples of variables that may affect protocol selection would be the protocol's estimate of the time needed to bind to the implementation, a protocol's round-trip-time estimate for executing an apply( ) call, the security required on the communication, whether the channel should be rebound* on error, or the latency allowed for the call invocation.

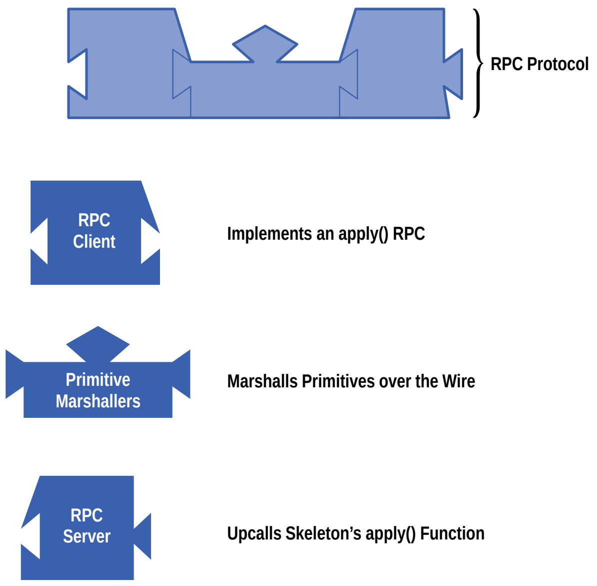

RPC Protocol
RPC Client
Implements an apply() RPC
Primitive Marshallers
Marshalls Primitives over the Wire
RPC Server
Upcalls Skeleton's apply() Function

Fig. 5. *The major transport components associated with protocols.*

Internal Structure of an RPC Protocol. Only the external interfaces for the RPC protocols are defined by ORBlite. The internal structure may vary considerably between protocols. ORBlite makes no statement on whether a protocol is connection-based or connectionless, which marshalling format is used (NDR, XDR, ASCII, etc.), whether the protocol represents data in big-endian or little-endian format, or even what physical medium is used by the underlying communication mechanism. In general, however, protocols will have the three major components shown in Fig. 5:

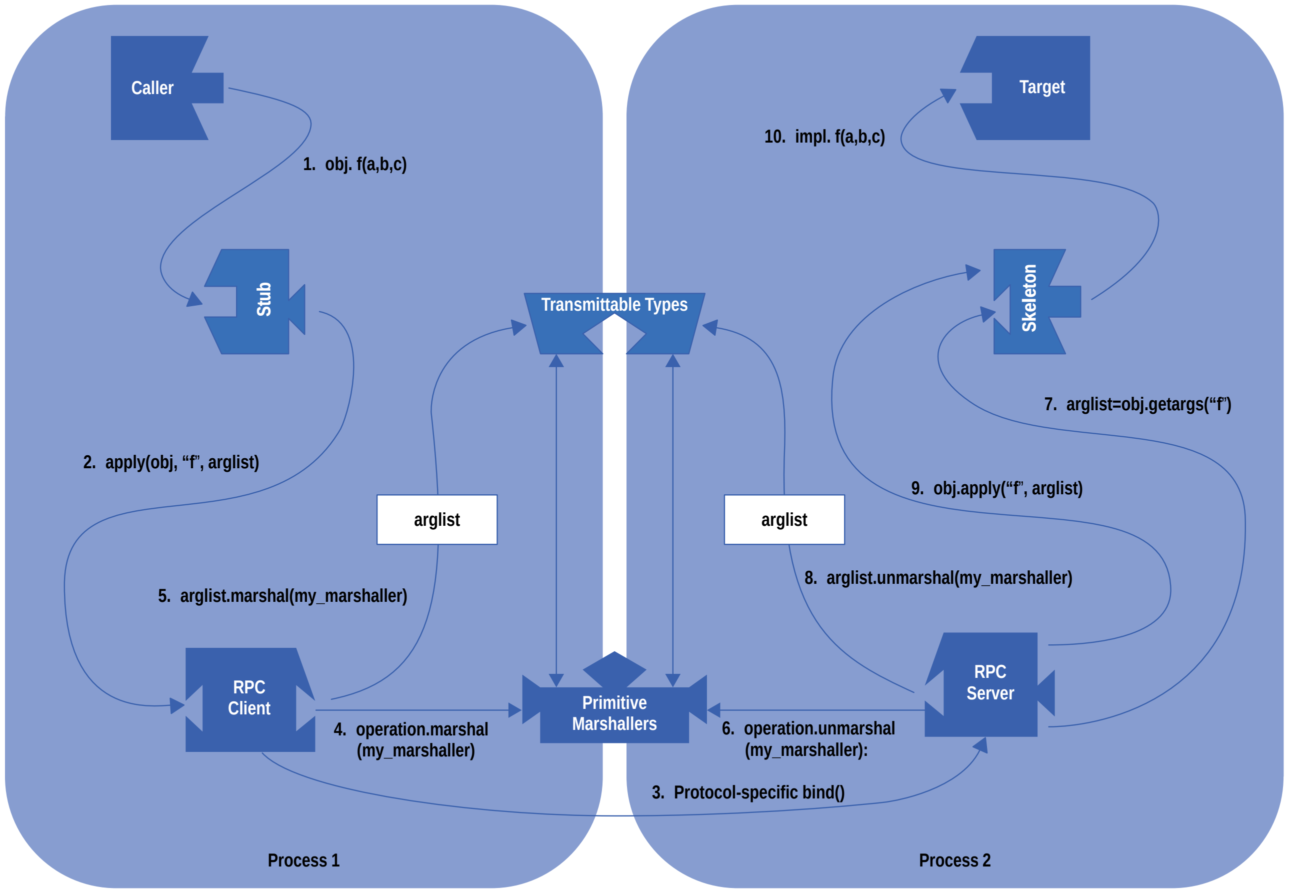
- The RPC Client implements the client side of the apply( ) call and is responsible for locating the target's implementation.

* Rebound means to reestablish a connection between a caller and a callee if an error occurs.

- The primitive marshallers support the transmission and reception of primitive data types in a protocol-specific manner.
- The RPC Server is responsible for receiving the call over the wire, using the ORBlite core to find the skeleton associated with the target of the invocation and forwarding the RPC Client's apply( ) call to the target skeleton.

Logical Call Flow

Given these pieces of the puzzle, the logical flow of control for a remote method invocation is shown in Fig. 6.



Caller
1. obj. f(a,b,c)
Stub
2. apply(obj, "f", arglist)
arglist
5. arglist.marshal(my_marshaller)
RPC Client
4. operation.marshal (my_marshaller)
Transmittable Types
Primitive Marshallers
Target
10. impl. f(a,b,c)
Skeleton
7. arglist=obj.getargs("f")
9. obj.apply("f", arglist)
arglist
8. arglist.unmarshal(my_marshaller)
RPC Server
6. operation.unmarshal (my_marshaller):
3. Protocol-specific bind()
Process 1
Process 2

Fig. 6. *The logical flow of a remote method invocation.*

Step 1. The caller executes the method f(a,b,c) on the stub object.

Step 2. The stub creates an arglist and calls the RPC Client's apply( ) function.

Step 3. If necessary, the RPC Client binds with the target's RPC Server using a protocol-specific mechanism.

Step 4. The RPC Client marshals the identifier for the target skeleton and then marshals the name of the operation to perform.

Step 5. The RPC Client marshals an identifier for the target skeleton, then marshals the name of the operation to perform, and finally tells the arglist to marshal itself (handing in the transport's primitive marshallers). The arglist will use its transport independent marshallers to turn composite data structures into primitives which can be marshalled using the transport's primitive marshallers.

Step 6. The RPC Server unmarshals the identifier for the target skeleton and then unmarshals the name of the operation to perform.

Step 7. The RPC Server then upcalls the skeleton to get the server-side arglist for the specified operation. This upcall is a critical component in decoupling the language API from the underlying protocol. Without this upcall, the RPC Server component would have to know the memory format that the skeleton is anticipating and therefore would be tied to a particular memory mapping.

Step 8. The arglist returned from the upcall, which is operation-specific, is told to unmarshal its arguments. Each argument is a transmittable type and will use the protocol independent unmarshallers to construct the arglist contents from primitives unmarshalled using the protocol's unmarshalling stream.

Step 9. The skeleton is upcalled to apply the unmarshalled arglist to the desired operation.

Step 10. The skeleton takes apart the arglist and invokes the actual method on the implementation.When the call on the skeleton completes, the RPC Server will ask the arglist to marshal its output parameters back to the client process. The RPC Client will unmarshal the output parameters and the stub will return the values back to the caller.

Dimensions of Evolvability

In this section we discuss how the ORBlite framework addresses the various types of evolvability.

Evolution of Object Implementation

ORBlite uses the IDL specification and the language mappings defined by CORBA and OLE to decouple an object's implementation from its interface. In this manner, an object's implementation can be updated without affecting any other part of the system provided that the interface is considered to specify not only syntax but also semantics and behavior.

ORBlite is not tied to a particular IDL or even the set of data types describable by a particular IDL. ORBlite requires that isomorphic parts of different IDLs be mapped to the same base type constructs, but model and IDL designers are free to experiment with extensions. Such extensions may, of course, impact interoperability. For instance, a server whose interface uses a non-CORBA IDL type such as an asynchronous stream cannot easily be called by a client whose model does not map this type.

Evolution of Object Interface

In ORBlite, objects can support multiple interfaces simultaneously, and the language mapping abstraction layer allows clients to inquire of a target object whether the target supports a particular interface (in the OMG CORBA C++ mapping, this is presented as the _narrow() and is_a() methods and in OLE C++ this is presented as QueryInterface()).

If an ORBlite object supports new functionality (or changes the semantics behind an interface) the object should export a new interface. Old clients can query for the old interface, and new clients can query for the new one. In this manner, the target object can support old clients as well as new clients.

Of course, with a strongly typed object model such as CORBA, such dynamic queries are often unnecessary since the received object reference may already have been received as a strongly typed reference to the new interface.

Evolution of Programming Model

From the standpoint of evolution, there are two aspects of model evolution that must be anticipated: support for the introduction of new data types and support for new implementations of existing data types.

Evolution of Language Mapping Types. The ORBlite framework defines a set of basic data types from which the transmittable types used by each language mapping are derived. At the root of the tree is an abstract class TxType which requires the derived classes to support _marshal() and _unmarshal() methods. These methods take a primitive marshalling stream parameter supplied by the protocol being used for a particular call. Framework-provided subclasses of this root define more interfaces for each of the basic types describable by CORBA IDL (e.g., structures, sequences, or enumerations). These subclasses provide default marshalling behavior in terms of (abstract) methods for marshalling and unmarshalling the object's components.

A language mapping can evolve in two different ways. Since it is responsible for providing the actual types used by the programmer, it is free to define and modify their interfaces as emitted by the language mapping's IDL compiler. Canonically, these types will derive from the ORBlite-provided base classes shown in Fig. 7, so an OMG C++ structure or COM array will be seen by a protocol as merely a generic structure or array, regardless of its internal representation.

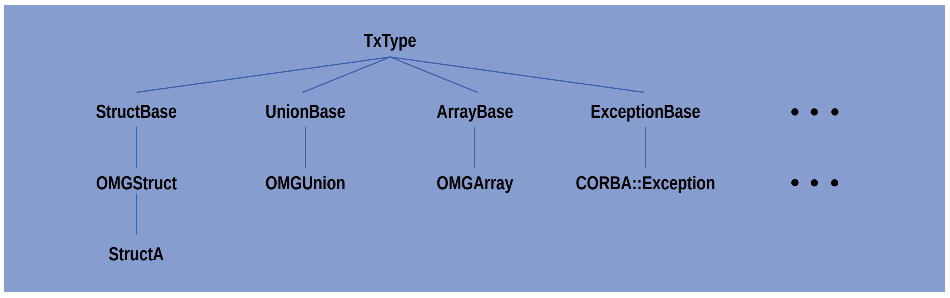


Fig. 7. *Data types derived from ORBlite's base class TxType.*

Note that there is no requirement that the actual types as presented to the programmer be transmittable. A language mapping merely has to guarantee transmittability of the data provided to a protocol. It is perfectly acceptable for a language mapping to use a transmittable wrapper class within argument lists and idiosyncratic classes (or even C++ primitives or arrays) in its API.

The other way that a language mapping can evolve is by adding types that are not directly supported by the ORBlite framework. The OLE mapping, for example, does this to create a VARIANT data type. The mapping can choose to implement the new data type in terms of one of the existing types (for instance, introducing a tree data type for use by the application but internally representing it using a sequence data type) and subclassing from a provided base. The language mapping can also choose a private representation for its contents and derive directly from TxType.

An additional attribute of ORBlite that supports a language mapping evolution is that the ORBlite framework makes no requirement that a language mapping have a unique class representing a particular IDL type. This allows a mapping to provide different representations of a type for different purposes. It also allows a later version of a language mapping to change to a new representation for a data type while remaining able to handle the old version's representation. For example, the ORBlite core uses two different mappings for strings: one optimized for equality comparison and the other for concatenation and modification. To the protocols, they behave identically.

Evolution of In-Memory Representation. There are two key issues involved in ensuring that the ORBlite core and the protocols are decoupled from the language mapping's data type representation. The first issue is ensuring that the RPC Client can marshal the parameters of a call, and the second is ensuring that the RPC Server can unmarshal the parameters without requiring excess buffering or parameter transformation. Essentially, we do not want to have to require that the language mapping translate from a protocol's in-memory data representation to its own.

The first issue is handled by the transmittable types' marshallers and accessors, which allow a protocol to marshal and retrieve composite data types without any knowledge of a language mapping's in-memory data representation.

The second issue is more complicated, and is shown as step 7 in Fig. 6, in which the RPC Server upcalls the skeleton to acquire the server-side default arglist. This upcall allows the RPC Server to offload memory management and in-memory representation for the incoming arguments to the portion of application code that actually knows the data type that is expected. A consequence of this is that the RPC Server can be reused across language mappings and is independent of the evolution of a particular language mapping.

The arglist returned from the upcall knows how to unmarshal itself. This means that the RPC Server does not need to buffer the incoming message and can allow the arglist to unmarshal its components directly into the language-mapping- specific memory representation. This is sometimes called zero-copy unmarshalling. The number of message copies is a major performance bottleneck in interprocess messaging.

Some language mappings, such as our experimental C++ mapping, allow an implementation to override the skeleton's default construction of the arguments. This is typically used when the implementation has a particular memory representation that is more convenient for the application than the default representation provided by the language mapping (e.g., the tree structure mentioned earlier). Overriding the construction of the default arguments removes the copy that would normally be required to switch representations. A language mapping can use this technique to support features not currently found in CORBA or OLE.*

The upcall is also used for two other features:

- Checking the per-object and per-method security policies
- Setting the thread-dispatch policy (e.g., thread priority and whether a new thread should be launched when executing the method).

A language mapping will typically allow the implementation to override the skeleton's default responses to the security policy or thread-dispatch mechanism.

Supporting Protocol Evolution

The principal obstacle to protocol evolution in most systems is the dependency of application code on protocol-specific APIs. In ORBlite, there are no references by the ORBlite core or by any of the language mapping components (i.e., the stub, the skeleton, and the transmittable types) to any specific protocol. Given this independence from a specific protocol, there is no need for visibility to the programmer.

This actually caused a rather interesting problem. It was not possible to just link a protocol into an ORBlite image as a normal C++ library. Since the core supports multiple protocols and there are no references by the language mapping or the core to any protocol, the linker does not have any unresolved symbols that would pull in a protocol built as a library. To overcome this obstacle we force the protocol to be loaded by creating an unresolved reference at link time.

* For instance, arbitrary graphs, migratable objects, or structures that support inheritance.

The protocols of a system evolve by dynamically or statically linking new protocols (or new versions of old protocols) into an ORBlite process. Updating or adding a protocol requires no change to the application code, the ORBlite core, or any language mapping.

To add a new protocol, the protocol developer derives from four abstract classes (the RPC Client, the RPC Server, the RPC primitive marshallers, and the RPC_Info class). The RPC_Info class registers the protocol with the ORBlite core and implements the bind() call for the protocol. The bind() call returns an instance of the RPC Client abstract interface that will be used to issue the apply() call for communication with a particular virtual process.

The RPC primitive marshallers will be used during the apply() call to choose the on-the-wire representation for the arguments of a call. They are called to marshal primitive data, such as integers and floating-point numbers, and are also given a chance to handle composite transmittable types. Normally, this last call merely hands marshalling responsibility back to the transmittable object, but the protocol can use this hook to satisfy special externally mandated padding, alignment, or ordering requirements as with DCE RPC's alignment requirements for structures and unions.

Managing Object References and Binding. Fig. 6 depicts the flow of a method invocation assuming an RPC Client has already been selected. In its simplest form, an RPC Client is selected when a client invokes a method on a stub. If the stub is not already bound to a suitable RPC Client, the stub asks the ORBlite infrastructure to find a protocol that can connect to the target object associated with an object reference. A bound RPC Client can become unsuitable if the client requires a particular quality of service (such as authentication or deadline-based scheduling). If the RPC Client is not suitable, a new RPC Client must be bound or an exception raised.

Each protocol registers with the ORBlite core a unique identifier and a binding interface. Each object reference contains a set of protocol tags and opaque, protocol-specific address information. The tags supplied in the object references are used by ORBlite to select a protocol that might be able to communicate with the target object.

If the target object is accessible over multiple protocols (i.e., both the client and the server support more than one protocol in common) then the protocol with the best quality of service is selected. The current selection criterion is based on a combination of the overhead involved for binding to the process associated with the reference plus the overhead for invoking the call. Assuming the process containing the object is activated, most RPC protocols have a 10-ms initial binding cost plus a 1-ms round-trip overhead per call. Protocols that can reuse connections across objects are generally selected in preference to connectionless protocols, which are selected in preference to protocols that require connection setup. The actual quality-of-service parameterization can get complicated. A named collection of collocated objects is called a *virtual process*. Fig. 8 shows the situation in which a process has exported two objects A and B in the virtual process VP1234. The virtual process is accessible over three protocols: IIOP (Internet Inter-ORB Protocol), ONC RPC, and the DCE-CIOP (DCE Common Inter-ORB Protocol).

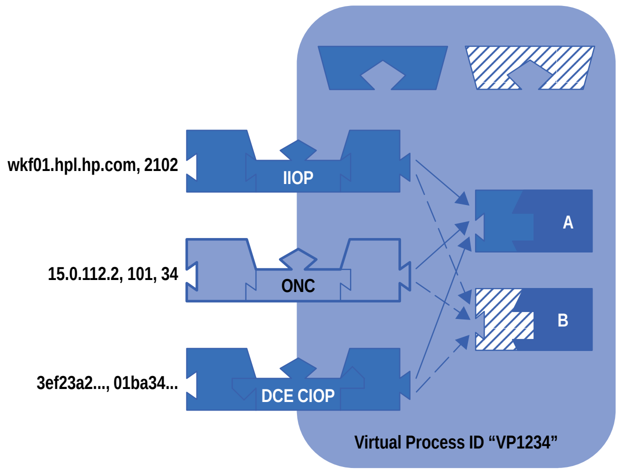


Fig. 8. *Using multiple profiles to locate object implementations.*

In ORBlite, protocols are encouraged to cache in the object reference the protocol-specific address of the last known location of the virtual process containing the object. While objects do move, the last known address is often correct and caching it can improve performance over using an external location mechanism.

Handling Common Scalability Issues. ORBlite was designed to support very large numbers of object references (more than 100,000) within a single process. To improve the scalability of location and per-object memory overhead, ORBlite provides support for protocols that wish to merge per-object cache information for objects located at the same address. In this model of object addressing, the address information held in an object reference is partitioned into two parts: an address associated with a virtual process identifier and an object identifier, which uniquely identifies the object within the virtual process.

In Fig. 8 the objects are named A@VP1234 and B@VP1234. A client that holds references to A and B can merge the cache information for the virtual process VP1234.

Often there are hundreds if not thousands of objects per process, and therefore, if location information for a protocol is based on a virtual process identifier, locating a single object in a process will have the side effect of refreshing the address information for all other objects at the same address. Some protocols will lose cache information for other protocols as the object reference is passed between processes. This is unfortunate because the cache information must be recreated if the object is to be accessible over other protocols. It is highly recommended that protocol designers allow object references to contain additional opaque information that may be used by other protocols.

ORBlite makes no requirement that a protocol use the virtual process abstraction, nor does it dictate how a protocol locates an object. ORBlite does expect, however, that the protocol's address information contained in an object reference is sufficient for that protocol to locate and, if necessary, activate the target object.

Supporting Legacy Protocols

In most cases, an object reference is created when an implementation is registered with the ORBlite infrastructure. When such an object reference leaves the process, the opaque, protocol-specific address information associated with each currently loaded protocol is marshalled along with it.

In the case of legacy components, it is likely that ORBlite is not in the server process. In this case, the binding information for the protocol must be added to the object reference via some other mechanism. Such ad hoc object references may be created by the legacy protocol, which obtains addressing information through an out-of-band mechanism. Alternatively, they may be acquired using normal protocols from a special-purpose server which creates the references from information kept in system configuration tables. However such constructed object references are obtained, they are indistinguishable from real object references and can subsequently be handed around in normal ORBlite calls.

When a stub attempts to bind the object reference, the protocol tag is matched to the protocols supported by the client process. If the process supports the protocol, an RPC Client is created that can interpret the request and communicate with the non-ORBlite server using the legacy protocol (see Fig. 9).

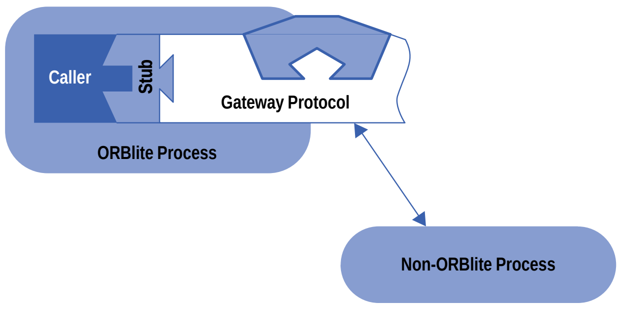


Fig. 9. *Using transport gateways.*

When ORBlite is not on both sides of the communication link, the protocol used is referred to as a *gateway protocol*. Note that gateway protocols are not only useful for communicating with legacy servers—an ORBlite process can publish itself on a legacy protocol so that it can be called by legacy, non-ORBlite clients. This form of publication is especially useful when a service needs to be accessible over both old protocols such as DCE RPC and new protocols such as IIOP.

Supporting Evolution of the ORBlite Core

In developing and deploying the ORBlite system, it became apparent that the typical owners of language mappings and protocols would not be the same as the typical owners of the ORBlite core. System developers from entities such as divisions building medical systems, test and measurement systems, or telecommunication systems were willing to own the portion that was particular to their domain, but each wanted the rest of the system to be someone else's responsibility.

This meant that the core itself needed to be able to evolve independently of the language mappings or protocols that plugged into it. It had to be simple to hook new protocols and mappings into old infrastructure and new infrastructure had to support old protocols and mappings.

The combination of the language mapping abstraction layer, the protocol abstraction layer, and the thread abstraction layer has made such independent evolution extremely straightforward.

Experience with the Framework

ORBlite was conceived in December, 1993 to support test and measurement systems. These systems contain computers and measurement instruments and are used in scientific experiments, manufacturing test, and environmental measurement. Analysis showed that the complexity of constructing the test and measurement system was the limiting factor in getting a product to market. Existing systems used a number of different communication mechanisms, and each component tended to have an idiosyncratic (and often undocumented) interface. Within HP, systems have used HP-IB, raw sockets, ONC/RPC, SNMP, NCS, and NFS.

At the time, there was a desire to move toward more stable, computer-industry-standard mechanisms, but it was unclear which proposed standard would win in the long run. The most likely contenders, CORBA and OLE, were still far from being well-specified. As we began publicizing our efforts within HP, we discovered that many others were facing a similar dilemma—notably those divisions responsible for medical systems and network management systems, each of which had its own set of legacy communication protocols.

The first version of ORBlite became operational in August of 1994. It supported the HyperDesk IDL/C++ language mapping[6] and two communication protocols: a thread-safe distribution protocol based on ONC RPC, and a gateway protocol designed to connect ORBlite services and clients to installed medical applications using the HP CareVue 9000 RPC protocol. The framework was extremely portable, thread-safe and reentrant, and because of the thread abstraction layer, it compiled without change on both UNIX® and Microsoft platforms. It was used in medical, test and measurement, analytical, financial, and telecommunication monitoring applications.

Over the past two years, dramatic changes have occurred in the specifications by OMG and in the OLE implementation by Microsoft. OMG has ratified a C++ language mapping,[7] two new standard communication protocols,[1] and recently an OLE language mapping for CORBA.[8] In addition, Microsoft has released a beta version of the DCOM (Distributed Component Object Model) protocol.[9]

In May, 1995, the ORBlite architecture began to make its way into external products. HP's Distributed Smalltalk was reimplemented to support the protocol abstraction layer, and the ORBlite code base was transferred to the Chelmsford Systems Software Laboratory to be turned into HP ORB Plus and released to external customers in April, 1996. HP ORB Plus, a strict implementation of CORBA 2.0, needed to support the new OMG standard C++ language mapping, which was previously unsupported by ORBlite. This pointed out the need for a well-defined language mapping abstraction layer and spurred its definition.

Since the transfer, the infrastructure has continued to evolve. We have experimented with new protocols to support high availability and legacy integration and new language mappings to support potential new IDL data types and to simplify the programmer's job. We are also investigating implementing an embeddable version of the architecture, which would have the same externally visible APIs but would be able to run in extremely memory-limited environments. Finally, we are looking into the declarative specification of protocol-neutral quality-of-service requirements and capabilities. This would assist in selecting the appropriate protocols to use and in guaranteeing the desired quality of service, where this interpreted to include performance, security, payment, concurrency, and many other dimensions. Following the ORBlite philosophy, we are attempting to design this mechanism in such a way that the set of available quality-of-service dimensions itself can evolve over time without impacting existing components.

The ORBlite infrastructure has allowed developers to build systems even as the standards evolve. The support of multiple language mappings, thread-safe distributed object communication, and multiple protocols has provided a unifying approach to building components and systems across the company. The key issues on the horizon will be ensuring that the standards from Microsoft, OMG, and others consider concurrency, streaming data types, and quality of service parameterization.

Acknowledgments

ORBlite would not have been possible without the input and feedback from our customer divisions and beta sites. We would especially like to thank Dean Thompson of the Network and System Management Division, Rob Seliger of the Medical Products Group, Horst Perner of the Böblingen Instrument Division, and Bill Fisher, Henrique Martins, and Paul Harry of HP Laboratories (Palo Alto and Bristol). We are also indebted to the ORB Plus team at the Chelmsford/Cupertino Systems Software Laboratory, especially Steve Vinoski, whose comments led to the idea of the language mapping abstraction layer, Mark Vincenzes, who was heavily involved in the design of the language mapping abstraction layer, Bob Kukura, who implemented the interoperable object references, and Bart Hanlon, who kept the product development on course. Others who assisted in the development include Kevin Chesney, who reimplemented HP Distributed Smalltalk according to the ORBlite framework, Walter Underwood and Lance Smith, who developed the thread abstraction layer, and Mark Morwood, who implemented the HP CareVue 9000 gateway protocol. We also wish to thank Mary Loomis, Joe Sventek, Jeff Burch, and Randy Coverstone for running interference for us, Harold Brown and Walter Underwood for picking up customer support when it got out of hand, Shane Fazzio for the NT build environment, and Dick Allen for general Microsoft expertise. And, of course, Lisa and Susan for putting up with all the long hours.

References

1. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*, Object Management Group, July 1995.
2. *The Common Object Request Broker: Architecture and Specification*, Object Management Group, Document Number 91.8.1, August 1991 (Draft).
3. *OLE 2 Programmer's Reference: Volume 1 & 2*, Microsoft Press, Redmond Washington, 1994.
4. *OSF DCE 1.0 Application Development Guide*, Technical Report, Open Software Foundation, December 1991.
5. *Network Programming Guide, Revision A*, Sun Microsystems Inc., March 27, 1990.
6. *Second Revised Submission in Response to the OMG RFP for C++ Language Mapping*, OMG Document Number 93-11-5, HyperDesk Corporation, November 1993 (Draft).
7. S. Vinoski, editor, *C++ Mapping 1.1 Revision*, OMG Document Number TC.96-01-13, January 1996.
8. J. Mischkinsky, editor, *COM/CORBA Part A Corrected Revised Submission*, OMG Document Number ORB.96-01-05, January 1996.
9. C. Kindel, *Microsoft Component Object Model Specification*, OMG Document Number 95-10-15, October 1995 (Draft).

▶ Go to Next Article

▶ Go to Journal Home Page