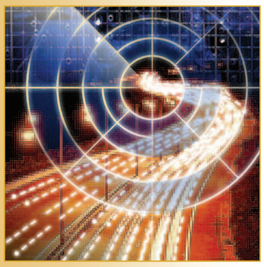
Middleware Track



Editors: Doug Lea • *lea@cs.oswego.edu*
Steve Vinoski • *vinoski@ieee.org*

A New Approach to Object-Oriented Middleware

Ice is a new object-oriented middleware platform that allows developers to build distributed client–server applications with minimal effort. While similar in concept to Corba, Ice breaks new ground by providing an object model that is both simpler and more powerful, by getting rid of inefficiencies that plagued middleware in the past, and by providing new features such as user datagram protocol (UDP) support, asynchronous method dispatch, built-in security, automatic object persistence, and interface aggregation. This article discusses design decisions, contrasts the Corba and Ice approaches, and outlines the advantages that result from a better design.

Michi Henning
ZeroC

Corba was, at its inception in the early nineties, a landmark advance in communications infrastructure.[1] For the first time, developers had an object-oriented distribution platform that could run on several different operating systems and implementation languages. Information technology professionals deployed Corba widely across numerous diverse industries, and people are still building new Corba systems today.

Despite its success in establishing the basic architecture of distributed object systems, Corba has many limitations. It can be difficult to learn and complex to use, suffers from several design and protocol inefficiencies, and lacks support for some frequently needed features. These weaknesses have prevented Corba's wider adoption and continue to limit its reach.

The Internet Communications Engine (Ice) represents a new approach to middleware[2] that builds on Corba's strengths while avoiding its weaknesses. The resulting middleware platform is easier to use, performs and scales better, and provides several new features.

Object Model

An object model is a set of definitions about the properties of computational entities, such as the available types and their semantics, rules for type compatibility, behavior in case of errors, and so on. While these rules are broadly similar for many middleware platforms, the devil lies in the detail – seemingly minor differences among object models have a large impact on system design and performance.

The Corba Object Model

Experience with the Corba model reveals several problem areas:

- opaque object references,
- weak object identity,

Ice Services

Apart from core run-time features, Ice also provides several object services that ease application development by providing access to frequently used functionality. Among these are

- a security service that permits secure communication over public networks and allows secure traversal of firewall and network address translation (NAT) boundaries;
- an object persistence service that automatically stores object state in a database using application-defined transaction boundaries;
- an event service for efficient event distribution to large numbers of users, with both TCP/IP and UDP support;
- a scalable and efficient implementation repository;
- an application server that allows central monitoring and control of application components; and
- a software patching service that permits secure and automatic distribution of software updates to clients.

For more details on these services, visit the Ice online manual.[2]

- lack of multiple interfaces,
- lack of exception inheritance,
- overly complex type system, and
- no notion of const-ness of operations.

Corba's object references are opaque. As a result, programmers cannot directly construct an object reference and must rely on additional services, such as a bootstrap or naming service, to obtain references in a portable way.

Object reference comparison in Corba has weak semantics. If two references compare as equal, they denote the same object; however, two references that compare as unequal might or might not denote the same object. To perform reliable object identity comparison, Corba requires a remote invocation on each of the objects compared.

Each Corba object has exactly one most-derived interface. This causes problems when a developer must upgrade a deployed application without losing backward compatibility, because Corba provides no way to add an interface for the newer version to an existing object without breaking already deployed application components. Similarly, Corba does not permit developers to change the number of parameters of an operation, add a field to a structure, or add a new exception to an operation.

Corba's interface definition language (IDL) does not support exception inheritance. As a result, even for languages that support it, structured exception handling is impossible. (Using an exception that contains a value type that derives from some other value type is possible, but prevents integration of exception inheritance with structured exception handling in implementation languages such as C++ and Java — the approach prevents catching derived exceptions directly.)

IDL provides many data types. Object request brokers (ORBs) sometimes do not support these types — fixed-point integers, extended floating-point types, or unsigned types, for example — if the underlying CPU or language lacks native support for them. Other IDL types are unnecessary or redundant. For example, an interface definition language need not provide bounded sequences and arrays as well as unbounded sequences; the minor gain in static type-safety is more than offset by the complexity they cause for language mappings. Similarly, value types can provide the functionality of Corba's `any` and `union` types, so they are redundant.

Corba provides at-most-once semantics, which state that an invocation can succeed or fail, but in no case can a single invocation by a client result in more than one invocation in the server. IDL does not permit operations to indicate whether they modify the target object's state. In the absence of such an indication, to preserve at-most-once semantics, compliant implementations must use very conservative error-recovery strategies. As a result, Corba treats potentially recoverable errors as unrecoverable.

The Ice Object Model

The Ice object model improves on the Corba object model in several ways. The "Ice Services" sidebar summarizes the unique object services that ease application development, but the Ice object model offers other advantages.

Proxies, which are handles to remote objects, are not opaque; given knowledge of the machine and port number at which a server runs and knowledge of an object's identity, a developer can construct a proxy at any time. This obviates the need for a bootstrap or naming service because clients can construct proxies from well-known names. As a result, applications require less code and the finished system has fewer dependencies on external services that might fail.

Ice provides strong object identity. Applications can reliably compare proxies for equality, and proxy equality is equivalent to object equality. That is, an Ice client does not need to send a remote message to compare object identities. This has performance benefits for system components that must efficiently perform object identity comparisons, such as a transaction service. A consequence of strong object identity is that transparent interposition (for example, using the Facade pattern[3]) becomes impossible: the facade and the object behind it have different identities, thus enabling a client to distinguish them. In practice, this is rarely a problem because clients almost always care more about object behavior than object identity. In the rare case in which an application uses interposition, the application can easily maintain consistency by ensuring that all system components access objects only via their facades.

Ice provides not only interface inheritance but also interface aggregation, just as the Component Object Model (COM) does.[4] A client can ask the proxy for a different interface to the object it represents. If the object supports the requested interface, the Ice run time creates a new proxy to that interface. Although an object can provide multiple interfaces, there is only a single object and hence, a single object identity. Interface aggregation solves the versioning problem: because a single object can have multiple unrelated interfaces while retaining a single object identity, developers can add newer interfaces to preexisting objects without violating the client–server contract. This lets developers add new versions to existing systems over time with no impact on deployed clients and avoids the contortions of using derivation to back-patch a versioning mechanism.

Ice provides single exception inheritance as a built-in feature. Language mappings preserve exception hierarchies, so hierarchical exception handling integrates cleanly with the native mechanisms of languages such as C++ and Java. For languages without hierarchical exception handling, language mappings provide APIs to perform safe down-cast at run time.

Slice. Similar to Corba's IDL, Ice provides the Specification Language for Ice. Slice provides a minimal number of built-in primitive types:

- integer types `short` (16-bit), `int` (32-bit), and `long` (64-bit);
- floating point types `float` (32-bit) and `double` (64-bit);
- `byte` (8-bit uninterpreted type);
- `string` (UTF-8 encoded Unicode string);
- types `Object` and `Object*` (the base type for classes and proxies, respectively);
- and `bool`.

Apart from these, Slice supports several user-defined types, such as constants, enumerations, sequences, structures, and modules. These work similarly to the equivalent Corba constructs.

In addition, Slice provides new constructs. A Slice *dictionary* defines collections of key-value pairs. The target language presents the dictionaries as associative arrays, such as C++ or Java maps. In contrast, Corba developers must emulate dictionaries using value types, for example, which are more complex to specify and implement, and which impose a performance penalty due to extra data copying.

Slice *classes* resemble Corba value types, though in greatly simplified form. Classes are like structures in that they are passed by value and can contain several members of arbitrary type. Unlike structures, however, classes support (single) implementation inheritance and (multiple) interface inheritance. Classes implicitly inherit from a common base type `Object`.

A program can supply a derived class where an operation signature expects a base class. If the receiver of a derived class instance knows the instance's derived type, the unmarshaling code creates a derived instance of the class in the receiver's address space. A safe down-cast lets the receiver convert a parameter of base type to the derived type. If the receiver of a derived class instance does not know the instance's derived type, however, the unmarshaling code truncates the instance to the most-derived type that the receiver understands.

Classes are more flexible than structures because they support pointer semantics: a class instance can have members that point to other class instances. This permits the construction of arbitrary graphs of nodes, wherein passing a single node as a parameter marshals all reachable instances in the graph and reconstructs the corresponding graph in the receiver's address space. The unmarshaling code preserves the identity relationships of instances, so the receiver's graph exactly matches the sender's graph even if some nodes have an in-degree greater than one (that is, have more than one incoming arc).

Unlike structures, classes can have operations. Operation invocations on a class are ordinary method calls and so execute in the caller's local address space. However, a developer can create a proxy to a class in a remote address space, in which case, invocations via the proxy are remote invocations.

What Slice omits. Slice is interesting not only for what it provides but also for what it leaves out. Slice includes no

- character, unsigned, fixed-point, extended integer, or extended floating-point types,
- distinction between narrow and wide strings (all strings are Unicode strings),
- `typedef` keyword (and therefore, no aliased types),
- anonymous types (the syntactic rules force all types to have names),
- nested types (Slice defines types only at global and module level),
- `any` type or unions,
- arrays or bounded sequences,
- constant expressions (it includes only constant definitions),
- attributes (it includes only operations),
- `inout` parameters, and
- type-level untyped name-value pairs (known as IDL contexts in Corba).

For classes, Slice also omits constructors, destructors, and public or private sections, as well as the notions of abstractness, boxing, and truncatability. Despite the long list of omissions, Slice is as functional as Corba IDL. Some of the missing features are simply unnecessary or redundant, such as constant expressions and attributes. For other missing features, developers can easily use alternative constructs, such as unbounded sequences to model bounded sequences and arrays and Slice classes to model IDL unions and `any` types.

Still other features, such as nested type definitions or anonymous types, cause more harm than good and are best left out entirely. Nested types cause name clashes if a target language's scoping rules differ from those of Slice — resolving the name clashes leads to complex and ugly APIs due to mangled names or the need for artificial scopes. Anonymous types make it impossible to, for example, declare a parameter or a variable of that type.

Error recovery and Slice. Slice provides two operation qualifiers, `nonmutating` and `idempotent`. The `nonmutating` qualifier indicates that the implementation of an operation does not modify the state of its target object; `idempotent` indicates that the effect of two or more successive invocations of an operation is the same as the effect of a single invocation. (Intuitively, developers can use `idempotent` to qualify operations that have simple assignment semantics.)

These operation qualifiers permit more aggressive error recovery from network failures because, for `nonmutating` and `idempotent` operations, a retry after an error can never violate at-most-once semantics. Additionally, if the target language permits, `nonmutating` causes the compiler to generate APIs that enforce the invocation's constant nature. (For example, `nonmutating` operations map to C++ `const` member functions.)

To use comparatively simple functionality, developers often face large numbers of interfaces with a bewildering array of operations.

Run-time APIs

Corba's overly complex APIs diminish its popularity. To use comparatively simple functionality, developers often face large numbers of interfaces with a bewildering array of operations.

A good example is the Corba Portable Object Adapter API, which requires 211 lines of IDL specification. Ice provides an equally functional object adapter that supports all the POA's implementation techniques — default servants, servant locators, many-to-one mappings of objects to servants, and so on — but the Ice object adapter API is defined in only 29 lines of Slice definitions.

Similar savings in API size exist for other parts of the Ice run time. This not only makes the platform easier to learn and use than Corba, it also results in smaller libraries and memory footprint at run time, with smaller working set sizes and concomitant performance gains.

Language Mappings

Ice currently provides language mappings for Java, C++, and PHP. The Ice Java mapping resembles the Corba Java mapping, but it is smaller and simpler because of Slice's simpler type model.

The Corba C++ mapping has suffered much criticism for its complex and difficult-to-use APIs. Programmers need to exercise extraordinary care, particularly with respect to memory management, to avoid creating hard-to-locate bugs. The following Corba C++ code fragments all contain errors:

```
void f(CORBA::Octet o);
void f(CORBA::Boolean b); // Ambiguous
cout << ref->getString(); // Leak
r1->putString(r2->getString()); // Leak
char *p = getString();
delete p; // Heap corruption
std::string s(ref->getString()); // Leak
if(ref1 == ref2) ... // Undefined
```

Code that compiles fine can contain many different types of latent bugs that might not manifest themselves until developers change the underlying operating system or compiler. The Corba C++ mapping contains many kinds of problems:

- The APIs pass dynamically allocated memory as raw pointers, which easily leads to memory-management errors.
- Parameter passing rules are complex. Whether a parameter is passed on the stack or in heap-allocated memory depends on the parameter's direction and type.
- Responsibility for allocation or deallocation of dynamic memory sometimes rests with the caller and sometimes with the callee, depending on the parameter's direction and type.
- Usually, the caller is responsible for deallocating variable-length return values. However, the callee retains ownership of memory for some APIs, so the memory-management rules are internally inconsistent.
- Allocating and deallocating many data types often requires the use of special functions — as opposed to C++'s `new` and `delete`. Failure to use these special functions can work out just fine or lead to heap corruption, depending on the underlying operating system.
- Writing exception-safe code requires considerable effort due to the use of raw pointers throughout the mapping.
- The mapping ignores threading. The extent to which operations on data types are thread-safe depends on the implementation.
- The APIs for many data types are large and complex. For example, the mappings for `union`, `any`, `TypeCode`, and `sequence` types are difficult to use correctly.
- The Corba C++ mapping predates the ISO C++ specification and, hence, does not use standard types such as `string`, `vector`, or `map`.
- If value types contain circular references, the Corba C++ mapping simply leaks their memory.

The Ice C++ mapping avoids the Corba C++ mapping's pitfalls:

- The mapping is free from memory-management artifacts. Dynamically allocated instances can be passed only by value as smart pointer types, so the programmer never needs to deallocate anything. (Ignoring return values is safe, for example, and the mapping never leaks memory in the presence of exceptions.)
- The mapping is fully thread-safe.
- Slice strings map to C++ `string`.
- Slice dictionaries map to C++ `map`.
- Slice sequences map to C++ `vector`.
- All Slice built-in types map to C++ types that are distinguishable for overloading.
- A garbage collector ensures that classes with circular references do not cause memory leaks.

Overall, the Ice C++ mapping is intuitive to use, requires fewer lines of code for equivalent functionality, integrates well with the standard template library, and is both thread- and exception-safe.

Invocation Models

Both Ice and Corba support invocations with time-outs, either as a global setting or for individual invocations: operations that do not complete within the specified time return with time-out exceptions. Corba also offers per-thread time-outs; however, Ice does not provide these due to the high cost of accessing thread-specific storage.

Corba supports synchronous, asynchronous, and one-way invocation modes. Ice covers those kinds of invocations and adds datagram and batching capabilities.

Synchronous Invocation

For synchronous invocation, Corba provides at-most-once semantics, which requires very conservative error-handling. In particular, if an invocation fails while the reply for a request is outstanding (for example, due to connection failure), the client-side run time has no choice but to propagate the failure to the application code. (The run time cannot retry the request because that might violate at-most-once semantics.)

Ice also provides at-most-once semantics, but it improves on the Corba situation by adding the `nonmutating` and `idempotent` operation qualifiers. The former operation does not change state, while the latter only sets the state to a defined value (independent of the previous state) that the run time can safely resend in the presence of network failures. This lets the Ice run time recover transparently from network failures that, in Corba, lead to hard errors.

Asynchronous Invocation

The Ice asynchronous invocation model resembles the Corba asynchronous method invocation (AMI) callback model: the client provides a callback object that the server uses to deliver an invocation's results. Ice discards the polling model because the majority of AMI applications use the callback model. Because the polling model adds little functionality to the callback model, a separate polling API would only add to the generated code's size and complexity.

One-way Invocation

Like Corba, Ice lets clients make one-way invocations for any operation that does not have a return value, an `out` parameter, or an exception. The run time dispatches one-way invocations like asynchronous operations: the thread of control returns as soon as the client-side stub has written the corresponding request to the local transport. One-way invocations are reliable in the sense that only catastrophic events, such as connection loss, cause the loss of one-way invocations. In particular, one-way invocations are flow-controlled just as two-way invocations are, so the client cannot overrun the server.

Datagram Invocation

Corba is silent about datagrams and their semantics. (Some ORBs provide a user datagram protocol [UDP] transport as a proprietary feature, but the semantics and APIs are proprietary and, of course, cross-vendor interoperability is not provided.)

Ice includes built-in UDP support, permitting clients to use datagram invocations. These resemble one-way invocations in that they apply only to operations that do not return values. At the transport level, the run time sends these invocations as proper UDP datagrams, which are unreliable and subject to size limitations. (Ice does not promise at-most-once semantics for datagram invocations because duplication of UDP packets makes it impractical to provide such a guarantee.)

Datagram invocations are useful in situations that require the distribution of large numbers of events via local area networks. In such situations, UDP offers substantial performance gains and allows applications to consume fewer operating system resources and to scale much further than with a connection-oriented transport.

Batched Invocation

Corba has a built-in one-to-one correspondence between client invocations and requests. Each client invocation results in a separate request–reply interaction on the wire. For fine-grained interfaces, this one-to-one correspondence can degrade performance significantly. For example, setting 20 attributes on an object requires 20 separate, and synchronous, round trips on the network.

Ice allows both one-way and datagram invocations to be *batched*. The run time queues batched invocations in a client-side buffer instead of sending them immediately. The buffer accumulates invocations until the client application invokes an explicit `flush` operation, which sends the accumulated invocations as a single message across the network. This is not only more efficient but also better decouples interface design from synchronous RPC's limitations.

If an invocation in a batch raises an exception, Ice does not inform the client of the error, so the failure of one invocation in a batch does not affect the remainder of the invocations in the batch.

Dispatch Models

As Corba does, Ice offers a *synchronous call dispatch* model for the server side. A client invocation results in a function call into the server-side application code; the client's call completes once the server-side function returns and the run time has marshaled the return values back to the client.

Synchronous dispatch is inappropriate in some situations. For example, consider a blocking `read` operation that returns data to clients. (Such operations are common in Corba — for example, both the Event and the Notification services offer such operations to deliver events to clients.) As Figure 1 (next page) illustrates, for every client that is waiting for data to arrive blocked in an invocation of `read`, the server loses an execution thread because the server-side operation cannot complete without also completing the client-side operation. As a result, such blocking operations are difficult

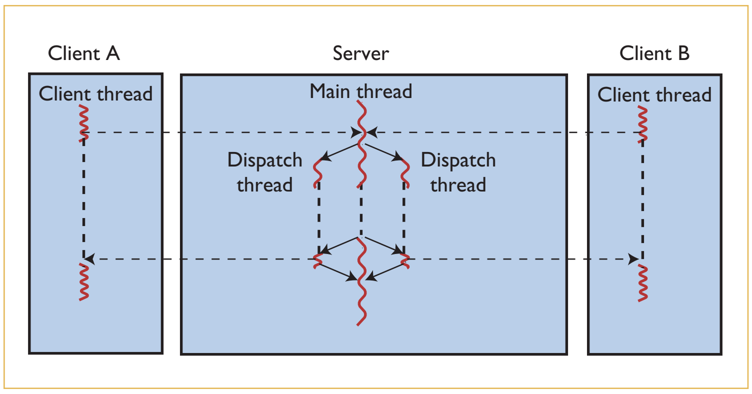


Figure 1. Synchronous call dispatch. Because the server uses a dedicated thread for each concurrent invocation, the clients tie up two additional threads for the entire duration of their blocking calls.

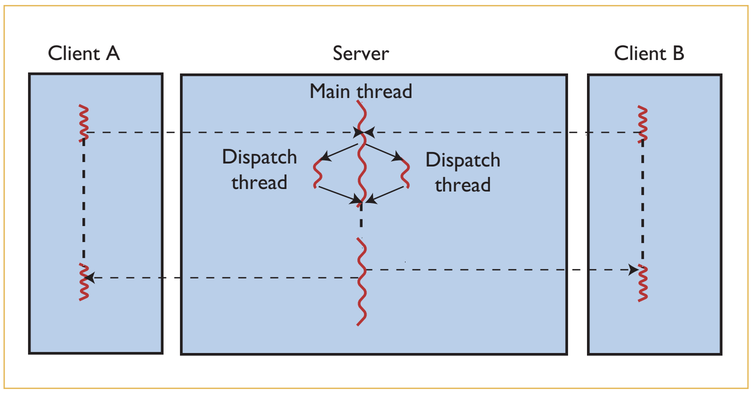


Figure 2. Asynchronous call dispatch. The server can implement any number of concurrent blocking calls using a single thread.

to scale because of the high resource consumption for each thread.

To deal with this problem, Ice provides an *asynchronous call dispatch* mode. As for synchronous dispatch, the Ice run time calls an application-defined function for each incoming operation invocation. However, as Figure 2 shows, that function can complete without completing the invocation in the client, thereby releasing its thread of control. To complete the operation, the server later invokes a callback function in the Ice run time. (The server can invoke the callback from a different thread than the one that received the invocation originally.) When the server makes the callback, the Ice run time receives the return results (or exception) for the invocation and marshals the results back to the client.

To clients, synchronous and asynchronous dispatch are indistinguishable, and the on-the-wire information is identical for either dispatch mode.

Asynchronous method dispatch permits a server to multiplex a large numerous concurrent invocations onto a small number of threads and so enables designs that could not scale with synchronous dispatch.

Threading

Corba has long suffered from its weak threading model. The specification provides no threading model on which a developer can rely. Developers must choose between single-threaded operation and vendor-specific models that might not provide threads at run time. If an ORB uses threads, Corba does not specify the threading model. Coupled with the lack of a portable threading API, this makes it difficult to write portable Corba code.

In contrast, Ice is inherently multithreaded. On the server side, a thread pool using a leader–follower model[5] dispatches incoming operation invocations. The thread pool's size is configurable; setting the thread pool size to 1 allows single-threaded server operation. Servers can create additional thread pools of arbitrary size. This permits servers to partition incoming operation invocations over several thread pools to, for example, prevent thread starvation.

By default, the Ice client side uses a thread pool containing two threads. One thread sends outgoing invocations while the other listens for incoming requests. This permits nested callbacks to proceed without deadlock. The nesting depth of callbacks is limited to the size of the thread pool, which developers can configure at run time.

The Ice run time itself is fully thread-safe, so programmers need not protect any Ice-related data structures from concurrent access. The only critical regions that developers must explicitly implement concern application-related data. Ice provides a portable threading and signal-handling API that lets developers write code that is portable across different platforms.

Protocol and Transports

Corba specifies the General Inter-ORB Protocol (GIOP), which requires a stream-oriented transport. The Internet Inter-ORB Protocol (IIOP) is a concrete specification of the abstract GIOP protocol for TCP/IP. In other words, the only standardized protocol that Corba provides is for TCP/IP. Corba does not provide a UDP-based protocol.

In contrast, the Ice protocol can run over a variety of stream and datagram transports. Currently, Ice supports TCP/IP and SSL as stream transports, and UDP as a datagram transport. The protocol engine is extensible, so programmers can add new transports via a plug-in API without modifying the Ice source code. (The Ice SSL transport is implemented as such a plug-in.) For connection-oriented transports, Ice provides an optional active-connection management feature that automatically reclaims connections that have been idle for a configurable amount of time.

The Problems with IIOP

IIOP suffers from several design flaws. For starters, the protocol state machine and encoding rules are not separately versioned. As a result, any changes to the encoding — of which there have been many in the past — break backward compatibility with deployed applications.

The encoding rules require padding in an attempt to align data on boundaries that match CPU alignment requirements. Unfortunately, these padding rules are severely misdesigned; they waste bandwidth and actually slow down marshaling instead of speeding it up.

The encoding of object references is complex and expensive to marshal and unmarshal. Object references marshal orders of magnitude slower than other structured data in many ORB implementations.

IIOP uses a *receiver-makes-it-right* byte-ordering scheme: the sender of a message marshals data in its native byte order and the receiver swaps the byte order if necessary. This approach results in a more complex protocol without tangible performance benefits. Moreover, because intermediate nodes can end up unnecessarily reordering data, this design hinders the efficient implementation of message switches that forward messages or parts of messages to down-stream consumers.

Addressing information is embedded in an IIOP byte stream at locations that depend on the type of data being marshaled. As a result, IIOP cannot cross network address translation (NAT) boundaries because addresses are not at fixed locations for patching by a protocol proxy or firewall. (Even if a proxy knew the address locations, it could still not patch the addresses because a change in an address's length would result in violating the IIOP alignment rules.)

Much of the data that IIOP sends is not encapsulated: to correctly unmarshal the data, the receiver must know the type of data in advance. This makes it difficult to provide application-level proxy servers and firewalls because a developer must configure these with type knowledge for the data they need to pass; IIOP does not allow generic proxies that are ignorant of application-level type definitions.

IIOP does not encapsulate values of type `any`, which requires the receiver to completely unmarshal values in detail, even if the receiver only wants to forward the data to a down-stream consumer. This prevents the efficient implementation of event and notification services.

Finally, developers can't extend GIOP to new transports without access to an ORB's source code. (The Object Management Group's Extensible Transport Framework RFP has been lingering for several years without progress.) Clearly, Corba's IIOP offers much room for improvement, which Ice provides.

Ice Improvements

The Ice protocol consists of two major parts: a set of encoding rules for the various data types and a protocol state machine that defines how clients and servers exchange messages. The encoding and the state machine carry separate version numbers to allow the protocol to evolve over time, with well-defined backward-compatibility rules and a reliable way to diagnose incompatible versions.

Encoding. The encoding rules emphasize simplicity and compactness. Ice marshals all data in fixed little-endian byte order and is always byte-aligned (that is, without any padding). This provides substantial advantages in terms of the marshaling code's simplicity and size and minimizes bandwidth consumption (which is especially important for wide area networks and low-bandwidth links).

All message payloads exchanged via the Ice protocol are encapsulated; that is, Ice sends each payload as a byte count followed by a blob of data. This enables the implementation of efficient message switches (such as an event service): intermediate nodes can forward incoming messages to any number of receivers by doing a simple block copy, without having to unmarshal and remarshal messages. In addition, the fixed byte order ensures that byte-reordering takes place only at the original sender and ultimate receiver if these run on big-endian machines; intermediate nodes that forward messages do not need to perform byte-swapping of the forwarded data.

Message switches can take advantage of UDP as a transport, making event distribution on LANs extremely efficient. Switches can use multicast and broadcast by simply choosing an appropriate address for UDP.

State machine. The protocol state machine is simple, with only five message types: `Request`, `BatchRequest`, `Reply`, `ValidateConnection`, and `CloseConnection`.

`ValidateConnection` and `CloseConnection` apply only to connection-oriented transports. For connection-oriented transports, the responding server sends a `ValidateConnection` message back to the client after accepting an incoming connection. That message informs the client that the server is ready to accept incoming requests and serves to negotiate a protocol version between client and server. `ValidateConnection` messages also eliminate a race condition that is inherent in GIOP: without explicit connection validation, a client can send a message to a server that is in the process of shutting down; when the client realizes that the server has closed the connection, it cannot retry the sent request because that might violate at-most-once semantics.

Ice connections are inherently bidirectional: if a client has established a connection to a server and the server invokes a callback operation on an object the client provides, the Ice run time can be configured to send the request over the connection the client has previously established. This feature is particularly important when operating through firewalls, which frequently do not permit clients to receive incoming connection requests. Ice also provides a built-in mechanism that lets applications transparently operate across NAT boundaries.

The Ice protocol supports on-the-wire compression; clients and servers indicate via a flag in the protocol header whether they are willing to accept compressed messages. In addition, developers can configure clients and servers at run time to enable or disable compression. With compression enabled, Ice compresses messages larger than 100 bytes using bzip2 (see http://sources.redhat.com/bzip2). This feature is most useful over low-bandwidth links and can provide substantial performance gains for applications that run, for example, over the public Internet. For high-speed LAN links, compression actually does more harm than good: the cost of the CPU cycles to compress and uncompress messages outweighs the savings in bandwidth.

Binding Modes

Corba provides both direct and indirect binding modes. With direct binding, object references carry the address information of the server endpoint, whereas for indirect binding, object references carry the address of an implementation repository. The implementation repository knows about servers' actual locations and replies to a client request with a redirect message containing the current server address — the client then resends its original request to the new address. Ice also provides both direct and indirect binding modes, but improves on the design of indirect binding.

Corba Binding

Forwarding via a redirect message has several disadvantages. For starters, a Corba client can't distinguish a directly bound reference from an indirectly bound one. (The client learns that a reference is indirectly bound only if it receives a redirect message in response to a request.) For large requests, this is wasteful because the client sends a request only to find that it has sent it to the wrong address and then has to resend the request in its entirety.

To alleviate this problem, GIOP includes a locate-request message that lets clients obtain a server's current address without sending an actual request. However, because clients cannot know the binding mode of references, they cannot know whether they should use that particular message: for directly bound references, it would be better to send the actual request, whereas for indirectly bound references, it would be better to send a locate request. The net effect is that, no matter what clients do, their actions are wrong some of the time and result in needless network traffic and delays.

Also, Corba's design encourages clients to store object references externally, for example, using the naming service. For indirect references, this poses problems. Changing the implementation repository's physical location invalidates all extant references to it and causes binding failure. This makes it very difficult to migrate implementation repositories if, for example, they exceed their scalability limits. In addition, this design severely limits the ability to migrate servers across machines: administrators can migrate servers only if the same implementation repository services both the source and target machines.

Corba implementation repositories must exchange messages with their servers. However, Corba does not standardize the protocol for doing this. As a result, servers written for a particular ORB cannot use another ORB's implementation repository, and administrators cannot federate repositories across vendor boundaries because no specification exists for doing so.

Ice Binding

For indirect binding in Ice, each proxy carries a symbolic name. To resolve the name to a physical end point, the client-side run time consults a lookup service that maps names to end points. (Configuration determines the lookup service's address.) This is analogous to the way that TCP/IP resolves domain names to IP addresses using the Domain Name System, and it has several advantages.

To start, the Ice protocol does not need to specifically support indirect binding. Instead, the Ice run time can resolve a name to an endpoint by invoking operations on Slice-defined interfaces as usual. This allows developers to implement third-party lookup services without modifying the Ice source code or protocol.

Another advantage is that the Ice design is extensible in the same way as the DNS. Developers can implement fault-tolerance and replication without additional Ice run-time support, achieving scalability by distributing state over several federated administrative domains.

Also, when an Ice client sends an invocation over the wire, the run time does not send the symbolic name that identifies a physical transport end point with the request — only the object identity that determines the target object. This allows objects to migrate freely among servers without breaking existing proxies, because all location information is external to both the proxy and the servers. Combined with the appropriate choice of object identity, such as a universally unique identifier (UUID), this permits free physical migration of objects down to the granularity of a single object.

Finally, indirect proxies contain no information that would identify the physical end point at which the lookup service runs. This makes it possible to migrate the lookup service without invalidating the proxies that clients use for binding. Because Ice keeps all location information externally, changing the lookup service's address requires updating only a single configuration item.

Summary

Developing Ice provided many lessons:

- A simple object model and type system contribute substantially to ease of use and performance of middleware.
- It pays to expend effort on designing run-time APIs that are both minimal and sufficient. Developers appreciate the simplicity.
- Simple language mappings that are thread-safe and do not burden developers with memory-management responsibilities substantially reduce development time and defect count.
- Rich invocation and dispatch models contribute to more scalable applications and let developers better decouple interface design from implementation.
- Keeping the protocol small and efficient improves the performance and scalability of applications and lets them use lower-bandwidth links.
- Built-in security and the ability to coexist with firewalls and NAT let applications use non-secure public networks rather than separate virtual private networks.

My colleagues and I are currently working on further improvements to Ice. These include a native implementation in C# that runs on both .NET and Mono; real-time extensions; a version of Ice suitable for embedded environments; and mappings to scripting languages such as Python, Perl, and Ruby.

References

1. M. Henning and S. Vinoski, *Advanced Corba Programming with C++*, Addison-Wesley, 1999.
2. M. Henning et al., *Distributed Programming with Ice*, ZeroC, 2003; www.zeroc.com/Ice-Manual.pdf.
3. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1999.
4. D. Box, *Essential COM*, Addison-Wesley, 1998.
5. D.C. Schmidt et al., "Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching," tech. report no. wucs-00-29, *Proc. 7th Pattern Languages of Programs Conf.*, Washington Univ., St. Louis, Mo., 2000; www.cs.wustl.edu/~schmidt/PDF/lf.pdf.

Michi Henning is chief scientist of ZeroC. From 1995 to 2002, he worked on Corba as a member of the Object Management Group's Architecture Board and as an ORB implementer, consultant, and trainer. With Steve Vinoski, he wrote *Advanced Corba Programming with C++* (Addison-Wesley, 1999). Since joining ZeroC, he has worked on the design and implementation of Ice and in 2003 coauthored *Distributed Programming with Ice* for ZeroC. He holds an honours degree in computer science from the University of Queensland. Contact him at michi@zeroc.com.