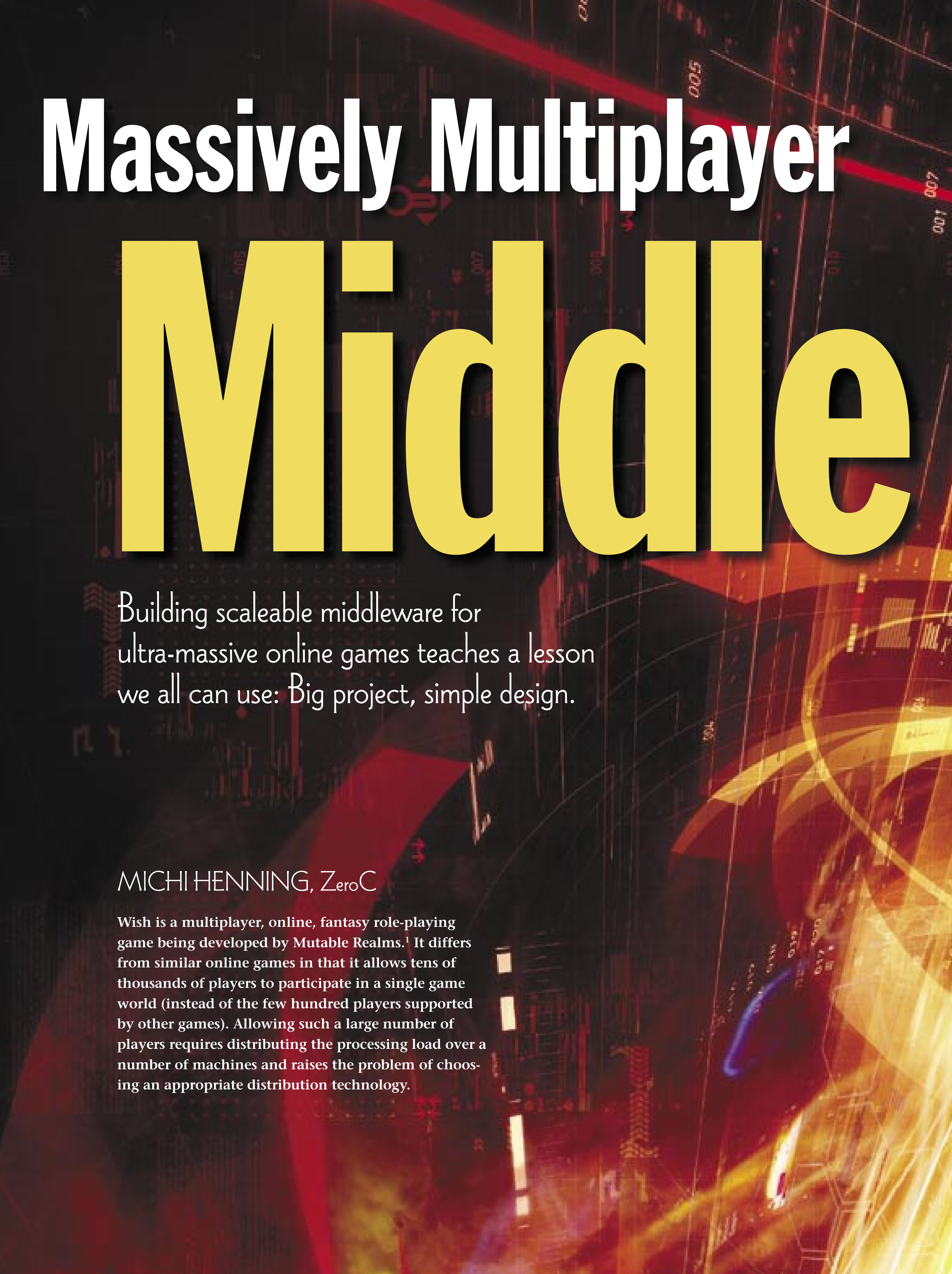


Massively Multiplayer Middle

Building scaleable middleware for ultra-massive online games teaches a lesson we all can use: Big project, simple design.

MICHI HENNING, ZeroC

Wish is a multiplayer, online, fantasy role-playing game being developed by Mutable Realms.[1] It differs from similar online games in that it allows tens of thousands of players to participate in a single game world (instead of the few hundred players supported by other games). Allowing such a large number of players requires distributing the processing load over a number of machines and raises the problem of choosing an appropriate distribution technology.

ware

ILLUSTRATION BY ANTHONY KYRIAZIS

Massively Multiplayer Middleware

DISTRIBUTION REQUIREMENTS

Mutable Realms approached ZeroC for the distribution requirements of Wish. ZeroC decided to develop a completely new middleware instead of using existing technology, such as CORBA (Common Object Request Broker Architecture).[2] To understand the motivation for this choice, we need to examine a few of the requirements placed on middleware by games on the scale of Wish and other large-scale distributed applications.

Multi-Platform Support. The dominant platform for the online games market is Microsoft Windows, so the middleware has to support Windows. For the server side, Mutable Realms had early on decided to use Linux machines: The low cost of the platform, together with its reliability and rich tool support, made this an obvious choice. The middleware, therefore, had to support both Windows and Linux, with possible later support for Mac OS X and other Unix variants.

Multi-Language Support. Client and server software is written in Java, as well as a combination of C++ and assembly language for performance-critical functions. At ZeroC we used Java because some of our development staff had little prior C++ experience. Java also offers advantages in terms of defect count and development time; in particular, garbage collection eliminates the memory management errors that often plague C++ development. For administration of the game via the Web, we wanted to use the PHP hypertext processor. As a result, the game middleware had to support C++, Java, and PHP.

Transport and Protocol Support. As we developed the initial distribution architecture for the game, it became clear that we were faced with certain requirements in terms of the underlying transports and protocols:

- *Players connect to ISPs via telephone lines, as well as broadband links.* While broadband is becoming increasingly popular, we had decided early on that the game had to be playable over an ordinary modem. This meant that communications between clients and server had to be possible via low-bandwidth and high-latency links.
- *Much of the game is event driven.* For example, as a player moves around, other players in the same area need to be informed of the changes in the game world around them. These changes can be distributed as simple events such as, "Player A moves to new coordinates <x,y>."

  Ideally, events are distributed via "datagrams." If the occasional state update is lost, little harm is done: A lost event causes a particular observer's view of the game world to lag behind momentarily, but that view becomes up-to-date again within a very short time, when another event is successfully delivered.
- Events in the game often have more than one destination. For example, if a player moves within the field of vision of five other players, the same positional update must be sent to all five observing players. We wanted to be able to use broadcast or multicast to support such scenarios.
- Communications between clients and game servers must be secure. For an online subscription-based game, this is necessary for revenue collection, as well as to prevent cheating. (For example, it must be impossible for a player to acquire a powerful artifact by manipulating the client-side software.)
- Clients connect to the game from LANs that are behind firewalls and use NAT (network address translation). The communications protocol for the game has to be designed in a way that accommodates NAT without requiring knowledge of application-specific information in order to translate addresses.

Versioning Support. We wanted to be able to update the game world while the game was being played—for example, to add new items or quests. These updates have to be possible without requiring every deployed client to be upgraded immediately—that is, client software at an older revision level has to continue to work with the updated game servers (albeit without providing access to newly added features). This means that the type system has to be flexible enough to allow updates, such as adding a field to a structure or changing the signature of a method, without breaking deployed clients.

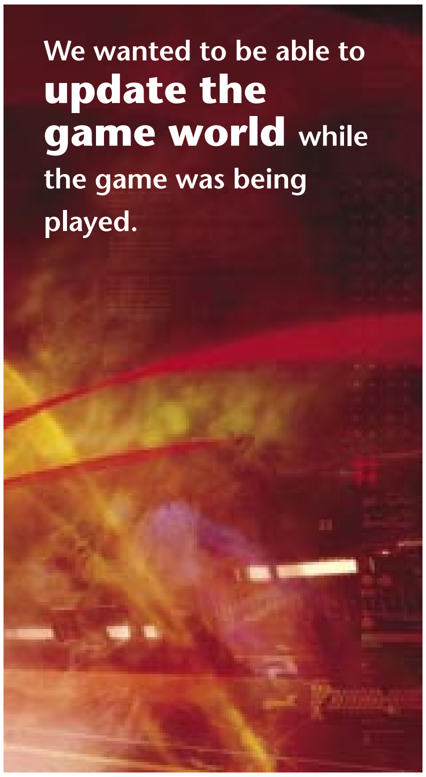
Ease of Use. Although a few of the Wish game developers are distributed computing experts, the majority have little or no experience. This means that the middleware has to be easy for nonexperts to use, with simple, thread-safe and exception-safe APIs (application programming interfaces).

Persistence. Much of the game requires state, such as the inventory for each player, to be stored in a database. We wanted to provide developers with a way to store and retrieve persistent state for application objects without having to concern themselves with the actual database and without having to design database schemas. Particularly during development, as the game evolves, it is prohibitively time consuming to repeatedly redesign schemas to accommodate changes. In addition, as we improve the game while being deployed, we must add new features to a database and remove older features from it. We wanted an automatic way to migrate an existing, populated database to a new database schema without losing any of the information in the old database that was still valid.

Threading. Much of the server-side processing is I/O-bound: Database and network access forces servers to wait for I/O completion. Other tasks, such as pathfinding, are compute-bound and can best be supported using parallel algorithms. This means that the middleware has to be inherently threaded and offer developers sufficient control over threading strategies to implement parallel algorithms while preventing problems such as thread starvation and deadlock. Given the idiosyncrasies of threading on different operating systems, we also wanted a platform-neutral threading model with a portable API.

Scalability. Clearly, the most serious challenges for the middleware are in the area of scalability: For an online game, predicting realistic bounds is impossible on things such as the total number of subscribers or the number of concurrent players. This means that we need an architecture that can be scaled by federating servers (that is, adding more servers) as demands on the software increase.

We also need fault-tolerance: For example, upgrading a server to a newer version of the game software has to be possible without kicking off every player currently using that server. The middleware has to be capable of automatically using a replica server while the original server is being upgraded.



We wanted to be able to **update the game world** while the game was being played.

Other scalability issues relate to resource management. For example, we did not want to be subject to hardwired limits, such as a maximum number of open connections or instantiated objects. This means that, wherever possible, the middleware has to provide automated resource management functions that are not subject to arbitrary limits and are easy to use. Simultaneously, these functions have to provide enough control for developers to tune resource management to their needs. Wherever possible, we wanted to be able to change resource management strategies without requiring recompilation.

A common scalability problem for distributed multiplayer games relates to managing distributed sets of objects. The game might allow players to form guilds, subject to certain rules: For example, a player may not be a member of more than one guild, or a guild may have at most one level-5 mage (magician). In computing terms, implementing such behavior boils down to performing membership tests on sets of distributed objects. Efficient implementation of such set operations requires an object model that does not incur the cost of a remote message for each test. In other words, the object identities of objects must be visible at all times and must have a total order.

In classical RPC (remote procedure call) systems, object implementations reside in servers, and clients send remote messages to objects: All object behavior is on the server, with clients only invoking behavior, but not implementing it. Although this approach is attractive because it naturally extends the notion of a local procedure call to distributed scenarios, it causes significant problems:

• Sending a remote message is orders of magnitude slower than sending a local message. One obvious way to reduce network traffic is to create “fat” RPCs: as much data as possible is sent with each call to better amortize the cost of going on the wire. The downside of fat RPCs is that performance considerations interfere with object modeling: While the problem domain may call for fine-grained interfaces with

many operations that exchange only a small amount of state, good performance requires coarse-grained interfaces. It is difficult to reconcile this design tension and find a suitable trade-off.
- Many objects have behavior and can be traded among players. Yet, to meet the processing requirements of the game, we have many servers (possibly in different continents) that implement object behavior. If behavior stays put in the server, yet players can trade objects, before long, players end up with a potion whose server is in the United States and a scroll whose server is in Europe, with the potion and scroll carried in a bag that resides in Australia. In other words, a pure client–server model does not permit client-side behavior and object migration, and, therefore, destroys locality of reference.

We wanted an object model that supports both client- and server-side behavior so we could migrate objects and improve locality of reference.

DESIGNING A NEW MIDDLEWARE

Looking at our requirements, we quickly realized that existing middleware would be unsuitable. The cross-platform and multi-language requirements suggested CORBA; however, a few of us had previously built a commercial object request broker and knew from this experience that CORBA could not satisfy our functionality and scalability requirements. Consequently, we decided to develop our own middleware, dubbed Ice (short for Internet Communications Engine).[3]

The overriding focus in the design of Ice was on simplicity: We knew from bitter experience that every feature is paid for in increased code and memory size, more complex APIs, steeper learning curve, and reduced performance. We made every effort to find the simplest possible abstractions (without passing the "complexity buck" to the developer), and we admitted features only after we were certain that we absolutely had to have them.

Object Model. Ice restricts its object model to a bare minimum: Built-in data types are limited to signed integers, floating-point numbers, Booleans, Unicode strings, and 8-bit uninterpreted (binary) bytes. User-defined types include constants, enumerations, structures, sequences, dictionaries, and exceptions with inheritance. Remote objects are modeled as interfaces with multiple inheritance that contain operations with input and output parameters and a return value. Interfaces are passed by reference—that is, passing an interface passes an invocation handle via which an object can be invoked remotely.

To support client-side behavior and object migration, we added classes: operation invocations on a class execute in the client's address space (instead of the server's, as is the case for interfaces). In addition, classes can have state (whereas interfaces, at the object-modeling level, are always stateless). Classes are passed by value—that is, passing a class instance passes the state of the class instead of a handle to a remote object.

We did not attempt to pass behavior: This would require a virtual execution environment for objects but would be in conflict with our performance and multi-language requirements. Instead, we implemented identical behavior for a class at all its possible host locations (clients and servers): Rather than shipping code around, we provide the code wherever it is needed and ship only the state. To migrate an object, a process passes a class instance to another process and then destroys its copy of the instance; semantically, the effect is the same as migrating both state and behavior.

Architecturally, implementing object migration in this way is a two-edged sword because it requires all host locations to implement identical (as opposed to merely similar) behavior. This has ramifications for versioning: If we change the behavior of a class at one host location, we must change the behavior of that class at all other locations (or suffer inconsistent behavior). Multiple languages also require attention. For example, if a class instance passes from a C++ server to a Java client, we must provide C++ and Java implementations with identical behavior. (Obviously, this requires more effort than implementing the behavior just once in a single language and single server.)

For environments such as Wish, where we control both client and server deployment, this is acceptable; for applications that provide only servers and rely on other parties to provide clients, this can be problematic because ensuring identical behavior of third-party class implementations is difficult.

Protocol Design. To meet our performance goals, we broke with established wisdom for RPC protocols in two ways:

• Data is not tagged with its type on the wire and is encoded as compactly as possible: The encoding uses no padding (everything is byte-aligned) and applies a number of simple techniques to save bandwidth. For example, positive integers less than 255 require a single byte instead of four bytes, and strings are not NUL terminated. This encoding is more compact (sometimes by a factor of two or more, depending on the type of data) than CORBA's CDR (common data representation) encoding.

• Data is always marshaled in little-endian byte order. We rejected a receiver-makes-it-right approach (as used by CORBA) because experiments showed no measurable performance gain.

The protocol supports compression for better performance over low-speed links. (Interestingly, for high-speed links, compression is best disabled: It takes more time to compress data than to send it uncompressed.)

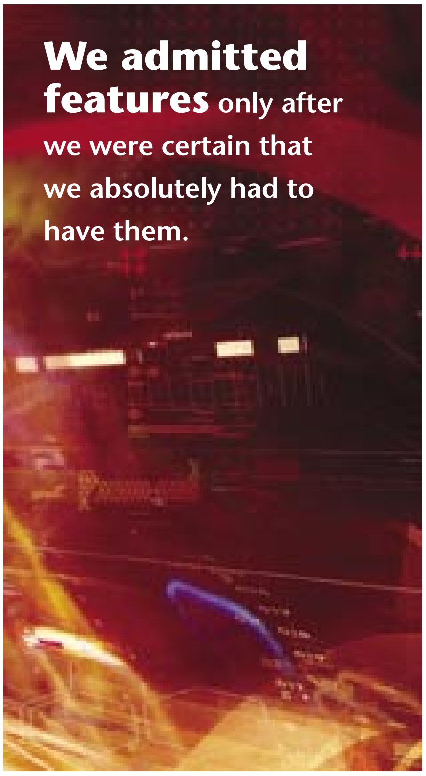
The protocol encodes request data as a byte count followed by the payload as a blob. This allows the receiver of a message to forward it to a number of downstream receivers without the need to unmarshal and remarshal the message. Avoiding this cost was important so we could build efficient message switches for event distribution.

The protocol supports TCP/IP and UDP (user datagram protocol). For secure communications, we use SSL (secure sockets layer): It is freely available and has been extensively scrutinized for flaws by the security community.

The protocol is bidirectional, so a server can make a callback over a connection that was previously established by a client. This is important for communication through firewalls, which usually permit outgoing connections, but not incoming ones. The protocol also works across NAT boundaries.

Classes make the protocol more complex because they are polymorphic: If a process sends a derived instance to a receiver that understands only a base type of that instance, the Ice runtime slices the instance to the most-derived base type that is known to the receiver. Slicing requires the receiver to unmarshal data whose type is unknown. Further, classes can be self-referential and form arbitrary graphs of nodes: Given a starting node, the Ice runtime marshals all reachable nodes so graphs require the sender to perform cycle detection.

The implementation of slicing and class graphs is surprisingly complex. To support unmarshaling, the protocol sends classes as individually encapsulated slices, each tagged with their type. On average (compared with structures), this requires 10 to 15 percent extra bandwidth. To preserve the identity relationships of nodes and to detect cycles, the marshaling code creates additional data structures. On average, this incurs a performance penalty of 5 to 10 percent. Finally, for C++, we had to write a garbage collector to avoid memory leaks in the presence of cyclic class graphs, which was nontrivial. Without slicing and class graphs, the protocol implementation would have been simpler and (for classes) slightly faster.



We admitted features only after we were certain that we absolutely had to have them.

Versioning. The object model supports multiple interfaces: Instead of having a single most-derived interface, an object can provide any number of interfaces. Given a handle to an object, clients can request a specific interface at runtime using a safe downcast. Multiple interfaces permit versioning of objects without breaking on-the-wire compatibility: To create a newer version, we add new interfaces to existing objects. Already-deployed clients continue to work with the old interfaces, whereas new clients can use the new interfaces.

Used naively, multiple interfaces can lead to a versioning mess that forces clients to continuously choose the correct version. To avoid these problems, we designed the game such that clients access it via a small number of bootstrap objects for which they choose an interface version. Thereafter, clients acquire handles to other objects via their chosen interfaces on bootstrap objects, so the desired version is known implicitly to the bootstrap object. The Ice protocol provides a mechanism for implicit propagation of contextual information such as versioning, so we need not pollute all our object interfaces by adding an extra version parameter.

Multiple interfaces reduced development time of the game because, apart from versioning, they allowed us to use loose coupling at the type level between clients and servers. Instead of modifying the definition of an existing interface, we could add new features by adding new interfaces. This reduced the number of dependencies across the system and shielded developers from each others' changes and the associated compilation avalanches that often ensue.

On the downside, multiple interfaces incur a loss of static type safety because interfaces are selected only at runtime, which makes the system more vulnerable to latent bugs that can escape testing. When used judiciously, however, multiple interfaces are useful in combating the often excessively tight coupling of traditional RPC approaches.

Ease of Use. Ease of use is an overriding design goal. On the one hand, this means that we keep the runtime APIs as simple and small as possible. For example, 29 lines of specification are sufficient to define the API to the Ice object adapter. Despite this, the object adapter is fully functional and supports flexible object implementations, such as separate servant per object, one-to-many mappings of servants to objects, default servants, servant locators, and evictors. By spending a lot of time on the design, we not only kept the APIs small, but also reaped performance gains as a result of smaller code and working set sizes.

On the other hand, we want language mappings that are simple and intuitive. Limiting ourselves to a small object model paid off here—fewer types mean less generated code and smaller APIs.

The C++ mapping is particularly important: From CORBA, we knew that a poorly designed mapping increases development time and defect count, and we wanted something safer. We settled on a mapping that is small (documented in 40 pages) and provides a high level of convenience and safety. In particular, the mapping is integrated with the C++ standard template library, is fully threadsafe, and requires no memory management. Developers never need to deallocate anything, and exceptions cannot cause memory leaks.

One issue we repeatedly encounter for language mappings is namespace collision. Each language has its own set of keywords, library namespaces, and so on. If the (language-independent) object model uses a name that is reserved in a particular target language, we must map around the resulting collision. Such collisions can be surprisingly subtle and confirmed, yet again, that API design (especially generic API design, such as for a language mapping) is difficult and time consuming. The choice of the trade-off between ease of use and functionality also can be contentious (such as our choice to disallow underscores in object-model identifiers to create a collision-free namespace).

Persistence. To provide object persistence, we extended the object model to permit the definition of persistence attributes for objects. To the developer, making an object persistent consists of defining those attributes that should be stored in the database. A compiler processes these definitions and generates a runtime library that implements associative containers for each type of object.

Developers access persistent objects by looking them up in a container by their keys—if an object is not yet in memory, it is transparently loaded from the database. To update objects, developers simply assign to their state attributes. Objects are automatically written to the database by the Ice runtime. (Various policies can be used to control under what circumstances a physical database update takes place.)

This model makes database access completely transparent. For circumstances in which greater control is required, a small API allows developers to establish transaction boundaries and preserve database integrity.

To allow us to change the game without continuously having to migrate databases to new schemas, we developed a database transformation tool. For simple feature additions, we supply the tool with the old and new object definitions—the tool automatically generates a new database schema and migrates the contents of the old database to conform to the new schema. For more complex changes, such as changing the name of a structure field or changing the key type of a dictionary, the tool creates a default transformation script in XML that a developer can modify to implement the desired migration action.

This tool has been useful, although we keep thinking of new features that could be incorporated. As always, the difficulty is in knowing when to stop: The temptation to build better tools can easily detract from the overall project goals. ("Inside every big program is a little program struggling to get out.")

Threading. We built a portable threading API that provides developers with platform-independent threading and locking primitives. For remote call dispatch, we decided to support only a leader/followers threading model.[4] In some situations, in which a blocking or reactive model would be better suited, this decision cost us a little in performance, but it gained us a simpler runtime and APIs and reduced the potential for deadlock in nested RPCs.

Scalability. Ice permits redundant implementations of objects in different servers. The runtime automatically binds to one of an object's replicas and, if a replica becomes unavailable, fails over to another replica. The binding information for replicas is kept in configuration and is dynamically acquired at runtime, so adding a redundant server requires only a configuration update, not changes in source code. This allows us to take down a game server for a software upgrade without having to kick all players using that server out of the game. The same mechanism also provides fault tolerance in case of hardware failure.

To support federating logical functions across a number of servers and to share load, we built an implementation repository that delivers binding information to clients at runtime. A randomizing algorithm distributes load across any number of servers that form a logical service.

We made a number of trade-offs for replication and load sharing. For example, not all game components can be upgraded without server shutdown, and a load feedback mechanism would provide better load sharing than simple randomization. Given our requirements, these limitations are acceptable, but, for applications with more stringent requirements, this might not be the case. The skill is in deciding when *not* to build something as much as when to build it—infrastructure makes no sense if the cost of developing it exceeds the savings during its use.

SIMPLE IS BETTER

Our experiences with Ice during game development have been very positive. Despite running a distributed system that involves dozens of servers and thousands of clients, the middleware has not been a performance bottleneck.

Our focus on simplicity during design paid off many times during development. When it comes to middleware, simpler is better: A well-chosen and small feature set contributes to timely development, as well as to meeting performance goals.

Finally, designing and implementing middleware is difficult and costly, even with many years of experience. If you are looking for middleware, chances are that you will be better off buying it than building it. Q

REFERENCES

1. Mutable Realms (Wish home page): see http://www.mutablerealms.com.
2. Henning, M., and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading: MA, 1999.
3. ZeroC. Distributed Programming with Ice: see http://www.zeroc.com/Ice-Manual.pdf.
4. Schmidt, D. C., O'Ryan, C., Pyarali, I., Kircher, M., and Buschmann, F. Leader/ Followers: A design pattern for efficient multithreaded event demultiplexing and dispatching. *Proceedings of the 7th Pattern Languages of Programs Conference* (PLoP 2000); http://deuce.doc.wustl.edu/doc/pspdfs/lf.pdf.

LOVE IT, HATE IT? LET US KNOW
feedback@acmqueue.com or www.acmqueue.com/forums

MICHI HENNING (michi@zeroc.com) 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), the definitive text in the field. 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 honors degree in computer science from the University of Queensland, Australia.