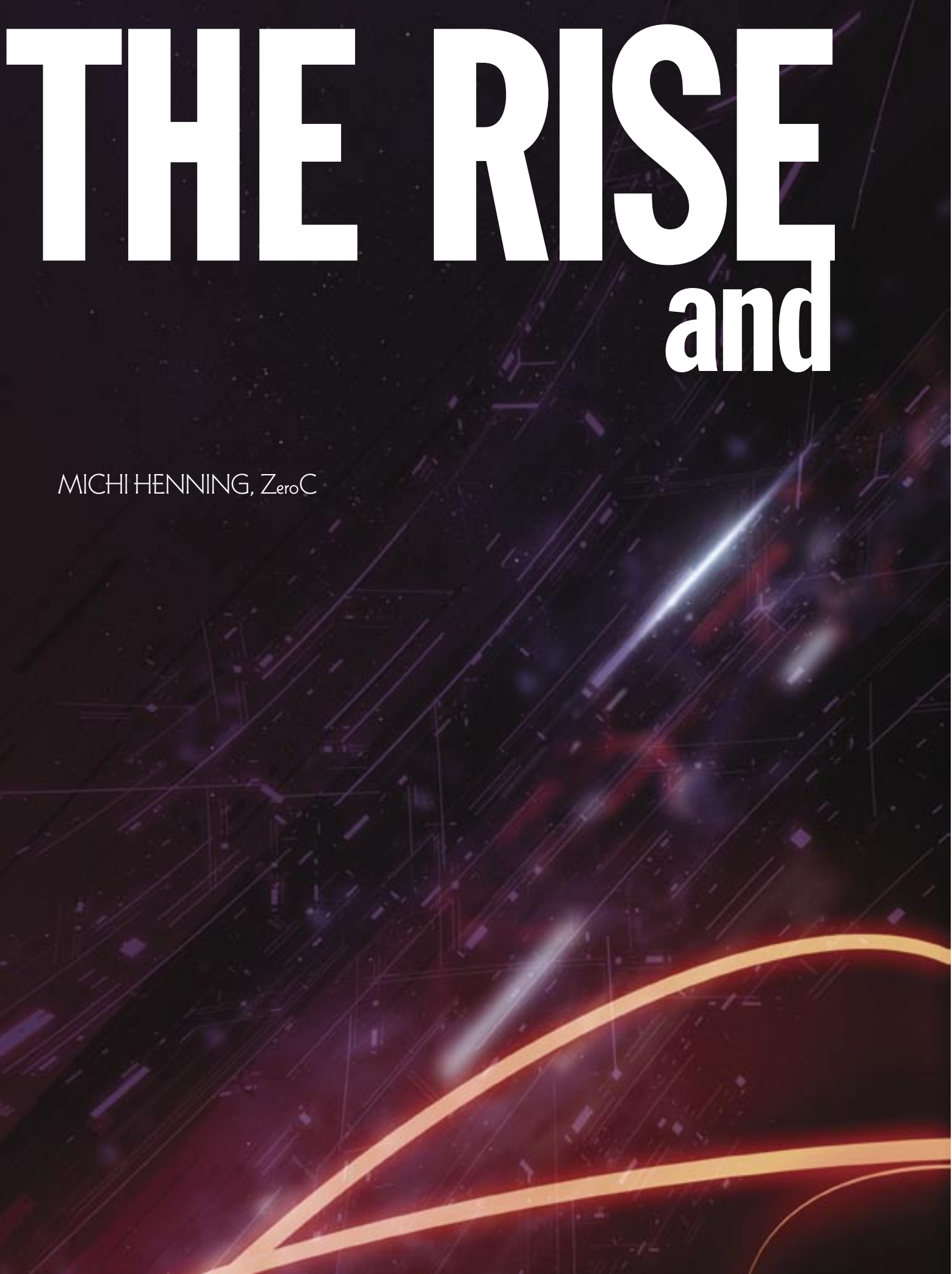# THE RISE
and

MICHI HENNING, ZeroC

Depending on exactly when one starts counting, CORBA is about 10-15 years old. During its lifetime, CORBA has moved from being a bleeding-edge technology for early adopters, to being a popular middleware, to being a niche technology that exists in relative obscurity. It is instructive to examine why CORBA—despite once being heralded as the "next-generation technology for e-commerce"—suffered this fate. CORBA's history is one that the computing industry has seen many times, and it seems likely that current middleware efforts, specifically Web services, will reenact a similar history.

A BRIEF HISTORY

In the early '90s, persuading programs on different machines to talk to each other was a nightmare, especially if different hardware, operating systems, and programming languages were involved: programmers either used sockets and wrote an entire protocol stack themselves or their programs didn't talk at all. (Other early middleware, such as Sun ONC, Apollo NCS, and DCE, was tied to C and Unix and not suitable for heterogeneous environments.)

After a false start with CORBA 1.0, which was not interoperable and provided only a C mapping, the OMG (Object Management Group) published CORBA 2.0 in 1997. It provided a standardized protocol and a C++ language mapping, with a Java language mapping following in 1998. This gave developers a tool that allowed them to build heterogeneous distributed applications with relative ease. CORBA rapidly gained popularity and quite a number of mission-critical applications were built with the technology. CORBA's future looked rosy indeed.

During CORBA's growth phase in the mid- and late '90s, major changes affected the computing landscape, most notably, the advent of Java and the Web. CORBA provided a Java language mapping, but it did nothing to cooperate with the rapidly exploding Web. Instead of waiting for CORBA to deliver a solution, companies turned to other tech-

There's a lot we can learn from CORBA's mistakes.

FALL of

CORBA

THE RISE and FALL of

CORBA

nologies and started building their e-commerce infrastructures based on Web browsers, HTTP, Java, and EJB (Enterprise JavaBeans).

In addition, developers who had gained experience with CORBA found that writing any nontrivial CORBA application was surprisingly difficult. Many of the APIs were complex, inconsistent, and downright arcane, forcing the developer to take care of a lot of detail. In contrast, the simplicity of component models, such as EJB, made programming a lot simpler (if less flexible), so calls for a CORBA component model became louder and louder. A component model was a long time in coming, however. Work was started in 1996 on a CBOF (Common Business Object Facility), but that effort got bogged down in political infighting and was eventually abandoned, to be replaced by the CCM (CORBA Component Model). A specification for CCM was finally published in late 1999 but turned out to be largely a nonevent:

• The specification was large and complex and much of it had never been implemented, not even as a proof of concept. Reading the document made it clear that CCM was technically immature; sections of it were essentially unimplementable or, if they were implementable, did not provide portability.

• No commercial CORBA vendor made a commitment to implement CCM, making it a stillborn child.

• Even if implementations had been available by the time CCM was finally published, it was too late. The horse had already bolted: EJB had become entrenched in the industry to the point where another component technology had no chance of success.

The failure of CCM did little to boost the confidence of CORBA customers, who were still stuck with their complex technology.

Meanwhile, the industry's need for middleware was stronger than ever. After some experience with e-commerce systems that used HTTP, HTML, and CGI, it had become clear that building distributed systems in this way had serious limitations. Without a proper type system, applications were reduced to parsing HTML to extract

semantics, which amounted to little more than screen-scraping. The resulting systems turned out to be very brittle. On the other hand, EJB had a proper type system but was limited to Java and so not suited for many situations. There were a few flies in the CORBA ointment, too:

• Commercial CORBA implementations typically cost several thousand dollars per development seat, plus, in many cases, runtime royalties for each deployed copy of an application. This limited broader acceptance of the platform—for many potential customers, CORBA was simply too expensive.

• The platform had a steep learning curve and was complex and hard to use correctly, leading to long development times and high defect rates. Early implementations also were often riddled with bugs and suffered from a lack of quality documentation. Companies found it difficult to find the expert CORBA programmers they needed.

Microsoft never embraced CORBA and instead chose to push its own DCOM (Distributed Component Object Model). This kept much of the market either sitting on the fence or using DCOM instead, but DCOM could not win the middleware battle either, because it worked only on Windows. (A port of DCOM to Unix by Software AG never gained traction.) Microsoft eventually dropped DCOM after several failed attempts to make it scale. By that time, the middleware market was in a very fragmented state, with multiple technologies competing but none able to capture sufficient mindshare to unify distributed systems development.

Another important factor in CORBA's decline was XML. During the late '90s, XML had become the new silver bullet of the computing industry: Almost by definition, if it was XML, it was good. After giving up on DCOM, Microsoft wasn't going to leave the worldwide e-commerce market to its competitors and, rather than fight a battle it could not win, it used XML to create an entirely new battlefield. In late 1999, the industry saw the publication of SOAP. Originally developed by Microsoft and DevelopMentor, and then passed to W3C for standardization, SOAP used XML as the on-the-wire encoding for remote procedure calls.

SOAP had serious technical shortcomings, but, as a market strategy, it was a masterstroke. It caused further fragmentation as numerous vendors clambered for a share of the pie and moved their efforts away from CORBA and toward the burgeoning Web services market. For customers, this added more uncertainty about CORBA's viability and, in many cases, prompted them to put investment in the technology on hold.

CORBA suffered another blow when the Internet bubble burst in early 2002. The industry's financial collapse drove many software companies out of the market and forced the survivors to refocus their efforts. The result was significant attrition in the number of commercial CORBA products. Before the collapse, several vendors had already dropped or deemphasized their CORBA products and, after the collapse, more followed. What in the mid- to late '90s had been a booming market with many competing products had suddenly turned into a fringe market with far fewer vendors, customers, and investment. By then, open source implementations of CORBA were available that partially compensated for the departure of the commercial vendors, but this was not enough to recover the lost mindshare and restore the market's confidence: CORBA was no longer the darling child of the industry.

Today, CORBA is used mostly to wire together components that run inside companies' networks, where communication is protected from the outside world by a firewall. It is also used for realtime and embedded systems development, a sector in which CORBA is actually growing. Overall, however, CORBA's use is in decline and it cannot be called anything but a niche technology now.

Given that only a few years ago, CORBA was considered the cutting edge of middleware that promised to revolutionize e-commerce, it is surprising to see how quickly the technology was marginalized, and it is instructive to examine some of the deeper reasons for the decline.

TECHNICAL ISSUES

Obviously, a number of external factors contributed to the fall of CORBA, such as the bursting of the Internet bubble and competition with other technologies, such as DCOM, EJB, and Web services. One can also argue that CORBA was a victim of industry trends and fashion. In the computing industry, the technical excellence of a particular technology frequently has little to do with its success—mindshare and marketing can be more important factors.

These arguments cannot fully account for CORBA's loss of popularity, however. After all, if the technology had been as compelling as was originally envisaged, it is unlikely that customers would have dropped it in favor of alternatives.

Technical excellence is not a *sufficient* prerequisite for success but, in the long term, it is a *necessary* prerequisite. No matter how much industry hype might be pushing it, if a technology has serious technical shortcomings, it will eventually be abandoned. This is where we can find the main reasons for CORBA's failure.

COMPLEXITY

The most obvious technical problem is CORBA's complexity—specifically, the complexity of its APIs. Many of CORBA's APIs are far larger than necessary. For example, CORBA's object adapter requires more than 200 lines of interface definitions, even though the same functionality can be provided in about 30 lines—the other 170 lines contribute nothing to functionality, but severely complicate program interactions with the CORBA runtime.

Another problem area is the C++ language mapping. The mapping is difficult to use and contains many pitfalls that lead to bugs, particularly with respect to thread safety, exception safety, and memory management. A number of other examples of overly complex and poorly designed APIs can be found in the CORBA specification, such as the naming, trading, and notification services, all of which provide APIs that are error-prone and difficult to use. Similarly, CCM configuration is so complex that it cannot be used productively without employing additional tool support.

Poorly designed interfaces and language mappings are a very visible part of any technology because they are the "coal face" of software development: They are the point at which developers and the platform meet, and their usability and safety have a major impact on development time and defect count. Obviously, any technology that suffers from endemic complexity does little to endear itself to developers, and does even less to endear itself to management.

Complexity also arises from architectural choices. For example, CORBA's IORs (interoperable object references) are opaque entities whose contents are supposed to remain hidden from developers. This is unfortunate for three reasons:
• Opaque references pretty much force the use of a naming service because clients cannot create object references without the help of an external service. This not only complicates system development and deployment, but also introduces redundant state into the system (with the concomitant risk of corrupting that state) and creates an additional failure point.
• Opaque references considerably complicate some APIs. For example, CORBA's interceptor APIs would be far simpler had object references been made transparent.
• Opaque references require remote calls to compare object identity reliably. For some applications, the overhead of these calls is prohibitive.

Another source of complexity is the type system. For example, CORBA's interface definition language provides a large set of types, among them unsigned integers,

THE RISE
and FALL of

CORBA

fixed-point and extended-precision floating-point numbers, bounded and unbounded sequences as well as arrays, and an "Any" type that can store values of arbitrary type.

Supporting these types complicates many APIs (in particular, the interfaces for introspection and dynamic invocation) and leads to subtle portability problems. For example, Java does not support unsigned types, so use of an unsigned integer in an interface can lead to overflow problems when a Java client communicates with a C++ server. Similarly, on platforms without native support for fixed-point or double-precision floating-point numbers, implementations must emulate these types. Emulations are hard to implement such that they behave identically across platforms, and they require additional APIs. This adds further complexity and is a source of hard-to-diagnose interoperability problems.

Finally, some of the OMG's early object services specifications, such as the life cycle, query, concurrency control, relationship, and collection services, were not only complex, but also performed no useful function whatsoever. They only added noise to an already complex suite of specifications, confused customers, and reinforced CORBA's reputation of being hard to use.

INSUFFICIENT FEATURES
CORBA provides quite rich functionality, but fails to provide two core features:

Security. CORBA's unencrypted traffic is subject to eavesdropping and man-in-the-middle attacks, and it requires a port to be opened in the corporate firewall for each service. This conflicts with the reality of corporate security policies. (Incidentally, this shortcoming of CORBA was a major factor in the rise of SOAP. Not having to open a port in the corporate firewall and sending everything via port 80 was seen as a major advantage, despite the naïvete of that idea.) The OMG made several attempts at specifying security and firewall traversal for CORBA, but they were abandoned as a result of technical shortcomings and lack of interest from firewall vendors.

Versioning. Deployed commercial software requires middleware that allows for gradual upgrades of the software in a backward-compatible way. CORBA does not provide any such versioning mechanism (other than versioning by derivation, which is utterly inadequate). Instead, versioning a CORBA application generally breaks the on-the-wire contract between client and server. This forces all parts of a deployed application to be replaced at once, which is typically infeasible. (This shortcoming of CORBA was another major factor in the rise of SOAP. The supposedly loosely coupled nature of XML was seen as addressing the problem, despite this idea being just as naïve as funneling all communications through port 80.)

For a commercial e-commerce infrastructure, lack of security and versioning are quite simply showstoppers—many potential e-commerce customers rejected CORBA for these reasons alone.

OTHER TECHNICAL ISSUES
A number of other technical issues plague CORBA, among them:

• Design flaws in CORBA's interoperability protocol make it pretty much impossible to build a high-performance event distribution service.
• The on-the-wire encoding of CORBA contains a large amount of redundancy, but the protocol does not support compression. This leads to poor performance over wide-area networks.
• The specification ignores threading almost completely, so threaded applications are inherently nonportable (yet threading is essential for commercial applications).
• CORBA does not support asynchronous server-side dispatch.
• No language mappings exist for C# and Visual Basic, and CORBA has completely ignored .NET.

This list of problems is just a sample and could be extended considerably. Such issues affect only a minority of customers, but they add to CORBA's bad press and limit its market.

PROCEDURAL ISSUES
Technical problems are at the heart of CORBA's decline. This raises the question of how it is possible for a technology that was produced by the world's largest software consortium to suffer such flaws. As it turns out, the technical problems are a symptom rather than a cause.

The OMG is an organization that publishes technology based on consensus. In essence, members vote to issue an RFP for a specification, member companies submit draft specifications in response, and the members vote

on which draft to accept as a standard. In theory, this democratic process is fair and equitable but, in practice, it does not work:

There are no entry qualifications to participate in the standardization process. Some contributors are experts in the field, but, to be blunt, a large number of members barely understand the technology they are voting on. This repeatedly has led to the adoption of specifications with serious technical flaws.

RFPs often call for a technology that is unproven. The OMG membership can be divided into roughly two groups: users of the technology and vendors of the technology. Typically, it is the users who would like to expand CORBA to add a capability that solves a particular problem. These users, in the hope that vendors will respond with a solution to their problem, drive issuance of an RFP. Users, however, usually know little about the internals of a CORBA implementation. At best, this leads to RFPs containing requirements that are difficult to implement or have negative performance impact. At worst, it leads to RFPs that are little more than requests for vendors to perform magic. Instead of standardizing best existing practice, such RFPs attempt to innovate without prior practical experience.

Vendors respond to RFPs even when they have known technical flaws. This may seem surprising. After all, why would a vendor propose a standard for something that is known to suffer technical problems? The reason is that vendors compete with each other for customers and are continuously jostling for position. The promise to respond to an RFP, even when it is clear that it contains serious problems, is sometimes used to gain favor (and, hopefully, contracts) with users.

Vendors have a conflict of interest when it comes to standardization. For vendors, standardization is a two-edged sword. On the one hand, standardization is attractive because it makes it easier to sell the technology. On the other hand, too much standardization is seen as detrimental because vendors want to keep control over the features that distinguish their product from the competition.

Vendors sometimes attempt to block standardization of anything that would require a change to their existing products. This causes features that should be standardized to remain proprietary or to be too vaguely specified to be useful. Some vendors also neglect to distinguish standard features from proprietary ones, so customers stray into implementation-specific territory without warning. As a result, porting a CORBA application to a different vendor's implementation can be surprisingly costly;

customers often find themselves locked into a particular product despite all the standardization.

RFPs are often answered by several draft specifications. Instead of choosing one of the competing specifications, a common response of OMG members is to ask the submitters to merge their features into a single specification. This practice is a major cause of CORBA's complexity. By combining features, specifications end up as the kitchen sink of every feature thought of by anyone ever. This not only makes the specifications larger and more complex than necessary, but also tends to introduce inconsistencies: Different features that, in isolation, are perfectly reasonable can subtly interact with each other and cause semantic conflicts.

Major vendors occasionally stall proceedings unless their pet features make it into the merged standard. This causes the technology process to degenerate into political infighting, forces foul compromises, and creates delays. For example, the first attempt at a component model was a victim of such infighting, as was the first attempt at a C++ mapping. Both efforts got bogged down to the point where they had to be abandoned and restarted later.

The OMG does not require a reference implementation for a specification to be adopted. This practice opens the door to castle-in-the-air specifications. On several occasions the OMG has published standards that turned out to be partly or wholly unimplementable because of serious technical flaws. In other cases, specifications that could be implemented were pragmatically unusable because they imposed unacceptable runtime overhead. Naturally, repeated incidents of this sort are embarassing and do little to boost customer confidence. A requirement for a reference implementation would have forced submitters to implement their proposals and would have avoided many such incidents.

Overall, the OMG's technology adoption process must be seen as the core reason for CORBA's decline. The process encourages design by committee and political maneuvering to the point where it is difficult to achieve technical mediocrity, let alone technical excellence. Moreover, the addition of disjointed features leads to a gradual erosion of the architectural vision. (For example, the architectural concept of opaque references was ignored by a specification update in 2000. The net effect is that references are no longer opaque, but APIs are still burdened with the baggage of treating them as opaque.)

CORBA's numerous technical flaws have accumulated to a point where it is difficult to fix or add anything without breaking something else. For example, every revision of CORBA's interoperability protocol had to make

THE RISE
and FALL of

CORBA

incompatible changes, and many fixes and clarifications had to be reworked several times because of unforeseen interactions with features that were added over time.

CAN WE LEARN FROM THE PAST?

A democratic process such as the OMG's is uniquely ill-suited for creating good software. Despite the known procedural problems, however, the industry prefers to rely on large consortia to produce technology. Web services, the current silver bullet of middleware, uses a process much like the OMG's and, by many accounts, also suffers from infighting, fragmentation, lack of architectural coherence, design by committee, and feature bloat. It seems inevitable that Web services will enact a history quite similar to CORBA's.

What steps should we take to end up with a better standards process and better middleware? Seeing that procedural failures are the root cause of technical failures, I suggest at least the following:

Standards consortia need iron-clad rules to ensure that they standardize existing best practice. There is no room for innovation in standards. Throwing in "just that extra little feature" inevitably causes unforeseen technical problems, despite the best intentions.

No standard should be approved without a reference implementation. This provides a first-line sanity check of what is being standardized. (No one is brilliant enough to look at a specification and be certain that it does not contain hidden flaws without actually implementing it.)

No standard should be approved without having been used to implement a few projects of realistic complexity. This is necessary to weed out poor APIs: Too often, the implementers of an API never actually use their own interfaces, with disastrous consequences for usability.

Interestingly, the open source community has done a much better job of adhering to these rules than have industry consortia.

Open source innovation usually is subject to a Darwinian selection process. Different developers implement their ideas of how something should work, and others try to use the feature and critique or improve it. That way, the software is extensively scrutinized and tested, and only the "fittest" version survives. (Many open source projects formalize this process with alternating experimental and production releases: The experimental releases act as the test bed and evolutionary filter.)

To create quality software, the ability to say "no" is usually far more important than the ability to say "yes." Open source embodies this in something that can be called "benevolent dictatorship": Even though many people contribute to the overall effort, a single expert (or a small cabal of experts) ultimately rejects or accepts each proposed change. This preserves the original architectural vision and stops the proverbial too many cooks from spoiling the broth.

At the heart of these open source practices are two essential prerequisites: cooperation and trust. Without cooperation, the evolutionary process cannot work; and without trust, no cabal of experts can act as an ultimate arbiter. This, however, is precisely where software consortia find their doom. It is naïve to put competing vendors and customers into a consortium and expect them to come up with a high-quality product—commercial realities ensure that cooperation and trust are the last things on the participants' minds.

Of course, software consortia contribute to an evolutionary process just as much as open source projects do. But it is the commercial marketplace that acts as the test bed and evolutionary filter, and it is the customers who, with their wallets, act as the (usually not so benevolent) dictator. This amounts to little more than an industry that throws up silver bullets and customers who leap after them like lemmings over a cliff. Until we change this process, the day of universal e-commerce middleware is as far away as ever. Q

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 OMG'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, ZeroC's next-generation middleware, and in 2003 co-authored *Distributed Programming with Ice*. He holds an honors degree in computer science from the University of Queensland, Australia.