Queue May/June 2007 Vol. 5 No. 4

Vol. 5 No. 4

acm

queue

architecting tomorrow's computing

May/June 2007

Association for Computing Machinery

Linux Security Sins

Standards-Based Middleware

API Design Matters

Interview: Stonebraker and Seltzer

www.acmqueue.com


Microsoft
Visual Studio
DEFY ALL CHALLENGES
Your challenge: create rich, dynamic PC or mobile apps. Defy it: deliver value, not just data with Visual Studio® and Windows Vista.™
More tips and tools at **defyallchallenges.com**

CONTENTS

MAY/JUNE 2007 VOL. 5 NO. 4

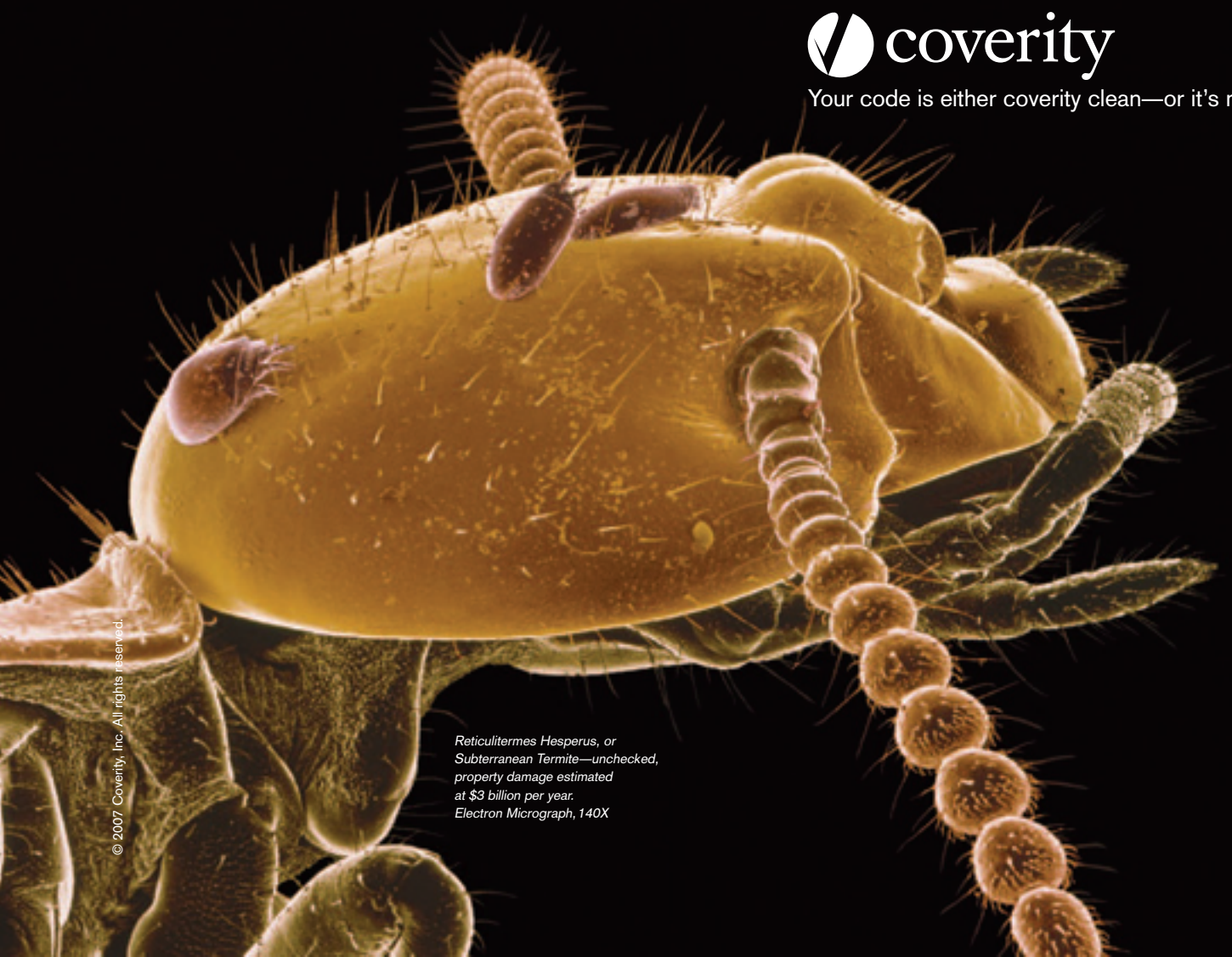
FEATURES

COVERITY FINDS THE DEADLY DEFECTS THAT OTHERWISE GO UNDETECTED.

Your source code is one of your organization's most valuable assets. How can you be sure there are no hidden bugs? Coverity offers advanced source code analysis products for the detection of hazardous defects and security vulnerabilities, which help remove the obstacles to writing and deploying complex software. With Coverity, catastrophic errors are identified immediately as you write code, assuring the highest possible code quality—no matter how complex your code base. **FREE TRIAL:** Let us show you what evil lurks in your code. Go to www1.coverity.com to request a free trial that will scan your code and identify defects hidden in it.

coverity

Your code is either coverity clean—or it's not.



Reticulitermes Hesperus, or Subterranean Termite—unchecked, property damage estimated at $3 billion per year. Electron Micrograph, 140X

CONTENTS

INTERVIEW

DEPARTMENTS

Swim with the Best

Cyberspace is an information feeding frenzy. Stay off the menu.
Black Hat USA brings together the most knowledgable and respected figures
in information and computer security to help you keep your edge.

Six days. Thirty Classes. Ninety presentations.

Black Hat®

Briefings & Training USA 2007

July 28–August 2 • Caesars Palace Las Vegas

www.blackhat.com

sponsors

diamond

RED LAMBDA

platinum

CISCO

Microsoft

gold

CENZIC

IOActive

CORE

ConfigureSoft

QUALYS

silver

ArcSight

FORTIFY

TippingPoint

Lancope

iCircle

NORMAN

radware

SAINT

SOURCEfire

SPIDynamics

StillSecure

BRIDGE

watchfire

Wireless

HAT

acm
queue
architecting tomorrow's computing

ACM Queue (ISSN 1542-7730) is published ten times per year by the ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701. POSTMASTER: Please send address changes to ACM Queue, 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA Printed in the U.S.A.

The opinions expressed by ACM Queue authors are their own, and are not necessarily those of ACM or ACM Queue. Subscription information available online at www.acmqueue.com.

BPA WORLDWIDE BUSINESS

OOPSLA

Montréal, Canada 2007

It's not about objects (only)!

Collaboration, diversity, and incubation / industry experts and their academic peers gathering to improve programming languages, refine the practice of software development, and explore new paradigms—*oo*psla's the premier conference for innovative and thought-provoking ideas, for seeking comment on works in progress, and (frequently, we're proud to say) for presenting Turing Award lectures on significant works. Contribute to *oo*psla and you will be enriched as you enrich the world of software.

CONFERENCE CHAIR
Richard P. Gabriel, USA
chair@oopsla.org

PROGRAM CHAIR
David F. Bacon, IBM
papers@oopsla.org

ONWARD! CHAIR
Cristina Videira Lopes, UC Irvine
onward!@oopsla.org

Critical Dates

March 19, 2007	Submission Deadline for Research Papers, Onward!, Essays, Practitioner Reports, Educators' Symposium, and proposals for Tutorials, Panels, Workshops, and DesignFest®
July 2, 2007	Submission Deadline for Posters, Demonstrations, Doctoral Symposium, Onward! Films, Student Research Competition, and Student Volunteers

For more information, visit: http://oopsla.org

Palais des congrès de **Montréal**
October 21–25, 2007

http://oopsla.org

For information, please contact
ACM Member Services Department
1.800.342.6626 (US & Canada)
+1.212.626.0500 (global)
email: info@oopsla.org

OOPSLA is sponsored by ACM SIGPLAN
in cooperation with SIGSOFT

KV the Loudmouth

A koder with attitude, KV ANSWERS YOUR QUESTIONS. MISS MANNERS HE AIN'T.

To buy or to build, that is the question. Of course, it's rarely that cut and dried, so this month Kode Vicious takes time to explore this question and some of its many considerations. He also weighs in on the validity of the ongoing operating system wars. Have an equally controversial query? Put your thoughts in writing and shoot an e-mail to kv@acmqueue.com.

Dear KV,

I was somewhat disappointed in your response to Unclear Peer in the December/January 2006/2007 issue of *ACM Queue*. You answered the question, but I feel you missed an opportunity to look at the problem and perhaps expand Unclear's professional horizons.

What requirement is being satisfied by having Unclear *build* a P2P file-sharing system? Based upon the answer, it may be more effective, and perhaps even more secure, to use an existing open source project or purchase commercial software to address the business need. Indeed, if the definition of P2P is loose enough, encrypted e-mail would meet your security criteria and might solve the business problem.

If Unclear is just a koding gnome, content to write kode as specified and not ask why, then I withdraw my concerns. Otherwise, it seems to me that an opportunity to teach Unclear, and your readers, was missed.

Sincerely,
Buyer not always a Builder

Dear BB,

Perhaps I've missed the marketing hype around this, or it has wound up in my spam box like all those ads for enlargement technology, but last I checked there wasn't an off-the-shelf P2P system one could buy. That being said, you bring up a good, if tangential, point—and one interesting enough to prevent your letter from winding up with all those aforementioned enlargement ads.

The buy-vs.-build, or as I like to think of it, the integrate-vs.-build question touches just about every part of a product. I like to say *integrate* because that can take into account using open source software, as well as buying software from a commercial vendor. Although many people might like to build everything from scratch—the Not Invented Here school of software construction—that is rarely an option in most projects because there is just too much to be done and never enough time. The problems that need to be addressed are the cost of integration and the risks.

Cost in this case is not just that incurred in buying a piece of software. Free or open source software often has high costs. The number of people on a local team required to maintain and integrate new releases of a component is definitely a cost that must be accounted for. Producing documentation is also a cost. For commercial products the costs include those just listed, as well as any money required to license the software in question.

In reality, the cost could be seen as just one of the risks involved when making the decision on whether to integrate or build a component of a system. The risks of integrating a component include the likelihood that the company or project that provides that component will continue to exist, and whether the component owner will change the system in a way that doesn't agree with your product over time. Plenty of people have been bitten by software that was changed underneath them.

It all comes down to control. If you can architect your system in such a way that the risks of integrating a component can be mitigated successfully, then integration, barring exorbitant costs, is probably a reasonable way to go. If you need absolute control over how a component works now and in the future, then you'll have to build it yourself. There is a spectrum of choices, but those are the two poles that you must navigate between.

KV

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue *coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!*

Your best source for software development tools!

programmer's paradise

LEADTOOLS Raster Imaging Pro

by LEAD Technologies

Raster Imaging Pro gives developers the tools to create powerful imaging applications. LEADTOOLS libraries extend the imaging support of the .NET framework by providing comprehensive support for image file formats (150+), 200 image processing filters, compression, TWAIN scanning, high-speed image display, color conversion, screen capture, special effects and more.

- .NET, API & C++ Class Library
- New Web Forms Control
- New Class Libraries for .NET
- Royalty Free

Now with 64-bit Support!

Paradise # L050569

$829.99

programmers.com/lead

dtSearch Web with Spider

New .NET Spider API

Quickly publish a large amount of data to a Web site

- Dozens of full-text and fielded data search options.
- Highlights hits in XML, HTML and PDF, while displaying links and images; converts other files ("Office," ZIP, etc.) to HTML with highlighted hits.
- Spider adds local or remote web sites (static and dynamic content) to searchable database
- Optional API supports SQL, C++, Java, and all .NET languages.

"Bottom line: dtSearch manages a terabyte of text in a single index and returns results in less than a second." —InfoWorld

Download dtSearch Desktop with Spider for immediate evaluation

Single Server Paradise # D290726

$888.99

programmers.com/dtsearch

DynamicPDF ReportWriter v4.0 for .NET

by ceTe Software

This easy-to-use tool integrates with ADO.NET allowing for the quick, real-time generation of PDF reports. The new GUI Report Designer makes laying out quality reports extremely simple.

- WYSIWYG Report Designer
- PDF Report Templates
- Recursive Sub-reports
- Automatic pagination, record splitting and expansion
- Full DynamicPDF Merger and Generator Integration
- Barcodes & PDF/X-1a
- Event Driven

New Release!

Pro. Server C3T005Y

$493.99

Ent. Developer C3T005Z

$1,482.99

programmers.com/cetesoftware

DevTrack

Powerful Defect and Project Tracking

by TechExcel

DevTrack, the market-leading defect and project tracking solution, comprehensively manages and automates your software development processes. DevTrack features sophisticated workflow and process automation, seamless source code control integration with VSS, Perforce and ClearCase, robust searching, and built-in reports and analysis. Intuitive administration and integration reduces the cost of deployment and maintenance.

Paradise # T340201

$487.99

programmers.com/techexcel

c-tree Plus®

by FairCom

With unparalleled performance and sophistication, c-tree Plus gives developers absolute control over their data management needs. Commercial developers use c-tree Plus for a wide variety of embedded, vertical market, and enterprise-wide database applications. Use any one or a combination of our flexible APIs including low-level and ISAM C APIs, simplified C and C++ database APIs, SQL, ODBC, or JDBC. c-tree Plus can be used to develop single-user and multi-user non-server applications or client-side application for FairCom's robust database server —the c-treeSQL™ Server. Windows to Mac to Unix all in one package.

64-bit SQL Available!

Paradise # F010131

$850.99

programmers.com/faircom

TX Text Control 13

Word Processing Components

New Release!

TX Text Control is royalty-free, robust and powerful word processing software in reusable component form.

- .NET WinForms control for VB.NET and C#
- ActiveX for VB6, Delphi, VBScript/HTML, ASP
- File formats RTF, DOC, HTML, XML, TXT
- PDF export without additional 3rd party tools or printer drivers
- Nested tables, headers & footers, text frames, bullets, numbered lists, multiple undo/redo
- Ready-to-use toolbars and dialog boxes

Professional Edition Paradise # T79023W

$739.99

Download a demo today.

programmers.com/theimagingsource

/n software Red Carpet Subscriptions

by /n software

/n software Red Carpet™ Subscriptions give you everything in one package: communications components for every major Internet protocol, SSL and SSH security, S/MIME encryption, Digital Certificates, Credit Card Processing, ZIP compression, Instant Messaging, and even e-business (EDI) transactions. .NET, Java, COM, C++, Delphi, everything is included, together with per developer licensing, free quarterly update CDs and free upgrades during the subscription term.

Paradise # D770148

$1,444.99

programmers.com/nsoftware

Compuware DevPartner Studio 8.1 Professional Edition

by Compuware

Compuware's award-winning DevPartner Studio Professional Edition lets you debug, test and tune your code in Microsoft Visual Studio applications, so you can deliver more reliable applications quickly and with ease. What else?

- Identify coding errors
- Find memory leaks in .NET and native code
- Pinpoint performance bottlenecks
- Automatically locate thread deadlocks
- Measure code complexity
- Analyze system configuration problems
- Ensure proper test coverage

Named User with Subscription Plus Paradise # N190S78

$2,315.99

programmers.com/compuware

VMware® Infrastructure 3

The most widely deployed software suite for optimizing and managing industry standard IT environments through virtualization—from the desktop to the data center. The only production-ready virtualization software suite, VMware Infrastructure is proven to deliver results at more than 20,000 customers of all sizes, used in a variety of environments and applications. The suite is fully optimized, rigorously tested and certified for the widest range of hardware, operating systems and software applications. VMware Infrastructure provides built-in management, resource optimization, application availability and operational automation capabilities, delivering transformative cost savings and increased operational efficiency, flexibility and service levels.

2 Processors Paradise # V55071H

$909.99

programmers.com/vmware

Adobe FlexBuilder 2

by Adobe

Adobe® FlexBuilder™ 2 software is a rich Internet application framework based on Adobe Flash® that will enable you to productively create beautiful, scalable applications that can reach virtually anyone on any platform. It includes a powerful, Eclipse™ based development tool, an extensive visual component library, and high-performance data services enabling you to meet your applications' most demanding needs.

Paradise # A14137P

$478.99

programmers.com/adobe

Intel® Cluster Toolkit

by Intel®

Create applications for Intel® processor-based cluster systems with performance-enhancing tools that include performance libraries, performance analyzers, and benchmark tests—integrated into one easy-to-install software bundle. Intel® Cluster Toolkit 3.0 for Linux adds more than 20 new features to the core libraries and tools to efficiently develop, optimize, run, and distribute parallel applications on clusters with Intel processors.

Paradise # I230ESL

$713.99

programmers.com/intel

Stylus Studio 2007 XML Enterprise Suite

by DataDirect

Stylus Studio 2007 XML Enterprise Suite, CRN Magazine's Product of the Year, adds powerful and intuitive modules for XML Pipelines and XML Reporting to an already robust tool set. It includes two-way conversion utilities for XML, cross-language debugging for XQuery, XSLT, Java, and more. Stylus Studio remains fully standards-based and standards-compliant, and built-in performance analysis tools and reports will help ensure that your XML applications are hitting on all cylinders.

Single User Paradise # P470239

$504.99

programmers.com/datadirect

800-445-7899

programmersparadise.com

Prices subject to change. Not responsible for typographical errors.

Dear KV,

I suspect that you don't get many letters from CFOs, but one of my people left a copy of *Queue* in my office the other day. I read your column and thought you might be interested in this question. Getting directly to the point, does the operating system still matter? I ask this because every time we initiate a project in my org, a small but loud group of people push me to pick an open source operating system for the project. It seems that they care more about that than about the application we're rolling out to our staff.

Reading over the trade press, I see claims and counterclaims about various operating systems, based on security and total cost of ownership, but all these claims seem to be written by proponents of one of the systems in question. At this point, it seems like the operating system doesn't really matter anymore, just so long as my application runs on it. What do you think? Should I just fire the loudmouths?

Tired of Zealots

Dear TZ,

You're right, I don't receive many letters from CFOs unless they're printed on pink paper and include words like "...please empty your desk by..." I also rarely condone firing the loud ones, for what must, by now, be obvious reasons.

Many pundits (i.e., people paid to have opinions) now claim that the operating system is a commodity that, in itself, has little intrinsic value. I don't get paid to have my opinion, but I claim that pundits have little intrinsic value.

Let me try to answer this question without going too deep into Operating Systems 101. The reason that the operating system matters, and will continue to matter as long as there are operating systems, is that the operating system is the ultimate arbiter between your application and the underlying computer. The operating system controls access to the CPU, memory, and all the devices. A good operating system is like good service in a restaurant: there when you need it and invisible when you don't. A poorly designed or implemented operating system is like the waiter who constantly asks, "Is everything all right?" when your mouth is full.

Two of the most important measures of operating system quality are security and efficiency. Does the operating system you want to use have a good security track record? No operating system, or piece of software, is perfect, but there are clearly classes of problems that may affect your application and these are the ones you, or likely your staff, need to study to make an informed decision on which operating system to put under your application.

Efficiency is also important. Although there are plenty of micro-benchmarks that show that one operating system is better than another, the speed of a context switch is unlikely to impress you—though I would be impressed if you knew what it meant. For an application, the question is one of a macro-benchmark. Simply put, "How much work can people do in the application in a given unit of time?"

Another question would be around how integral the operating system is to your product. If your company builds products where the operating system is an integral component, such as a consumer device or piece of networking equipment, then the quality of the code, your ability to modify it and distribute your changes, documentation, and how long you think the company or project that supports it will last all come into play. These concerns were addressed in the previous response to the letter from Buyer not always a Builder.

So, the short answer is, "Yes, the operating system matters." And, please, don't just fire the loudmouths. I might be one of them.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

Embracing Wired Networks

Mache Creeger, Emergent Technology Associates

Even at home, HARDWIRING IS THE WAY TO GO

Most people I know run wireless networks in their homes. Not me. I hardwired my home and leave the Wi-Fi turned off. My feeling is to do it once, do it right, and then forget about it. I want a low-cost network infrastructure with guaranteed availability, bandwidth, and security. If these attributes are important to you, Wi-Fi alone is probably not going to cut it.

People see hardwiring as part of a home remodeling project and, consequently, a big headache. They want convenience. They purchase a wireless router, usually leave all the default settings in place, hook it up next to the DSL or cable modem, and off they go. Ease and convenience are the selling points, but there are certainly tradeoffs to consider. As the IT expert of last resort for family, friends, and sometimes their family and friends, here are some of my experiences with Wi-Fi in the home.

ACCESSIBILITY AND AVAILABILITY

Wi-Fi signals usually do not reach every area of the home, or if they do, sometimes the service can be intermittent because of RF noise or conflict. As luck would have it, those areas are typically the most important: the home office, the master bedroom, or the kids' rooms. Often the culprit is too many walls or a wall that is too dense in the straight line between the router and its destination. A wireless telephone or microwave oven can sometimes be in conflict with the same unlicensed RF spectrum.

Usually Dad is dispatched to kludge together some kind of compromise. This typically means changing the router and/or client computer position to lessen outside interference and/or capture some finger of the transmitted Wi-Fi signal. Those who are more sophisticated will purchase a wireless access point, placing it at some midpoint to capture, amplify, and resend the signal, adding latency and decreasing bandwidth. As a last resort, the truly brave will string a long patch cable from the router to the problem area, usually to the horror of the significant other required to live with cables snaking along the baseboards.

Security. Most people use either no security at all or WEP (Wired Equivalent Privacy). More often than not they just plug the unit in and use it with the default settings. As a result, the majority of wireless LANs can be easily compromised. Just take a look at the scary discussions about how quickly a free utility called Aircrack can make an outside laptop a peer on a WEP-secured LAN.[1]

A quick scan of the surrounding homes in my relatively sophisticated Silicon Valley neighborhood showed a total of 14 operating Wi-Fi routers visible from the router in my closet. Even though WPA (Wi-Fi Protected Access) is currently considered to be the best available Wi-Fi security paradigm, only one was using it. The rest were either open or secured by WEP.

Why should you care? Not only do you want to surf the Internet in private, but you do not want unauthorized people using your Internet account or rummaging through your private information (think e-mail, medical records, tax returns, Quicken files, and PayPal and eBay passwords). Perhaps more importantly, you do not want to be liable to organizations like the RIAA (Recording Industry Association of America) for illegally downloading copyrighted material.

Bandwidth. Wi-Fi routers are rated at certain speeds, but the actual realized bandwidth is much less. The most popular standard, 802.11g, is rated at 54 megabits per second, but the effective rate realized in the real world is closer to 20-25 Mbps. These speeds are adequate for Web browsing and e-mail but are not nearly enough for home media, remote file sharing, or moving large files between machines. Newer, faster standards than 802.11g are now starting to take hold.

Cost. While most laptops and desktops produced today have Wi-Fi built in, older machines do not. Wi-Fi has only recently become standard on desktop machines. To become wireless, older systems will require the purchase, installation, and configuration of an additional Wi-Fi card. In my house everyone has his or her own computer, and I expect that my family is fairly typical. An upgrade to wireless would probably cost around $50 plus installation and configuration time for each older computer.

DEVELOPMENT LIFECYCLE PRACTICES

Managing Projects & Teams | Agile Development | Plan-Driven Development | Process Improvement & Measurement | Testing & Quality Assurance | Security & Special Topics

BETTER SOFTWARE CONFERENCE & EXPO

JUNE 18–21, 2007 LAS VEGAS, NEVADA THE VENETIAN

KEYNOTES BY INTERNATIONAL EXPERTS

Jeff Payne
Cigital, Inc.
What Better Software Means to the CEO

Payson Hall
Catalysis Group, Inc.
Risk Management—It's Not Just for Gamblers Any More

Sanjeev Verma and Kendra Yourtee
Microsoft
How to Design Frustrating Products

Alan Shalloway
NetObjectives
Using Lean Thinking to Align People, Process, and Practices

Jim Coplien
Nordija A/S
What Snake Oil is Your Organization Buying Today?

SOFTWARE QUALITY ENGINEERING
www.sqe.com

REGISTER NOW!
www.sqe.com/bettersoftwareconf

14 IN-DEPTH TUTORIALS receive the best strategic insight and tactical advice for your software development challenges

5 KEYNOTE SESSIONS on a variety of topics given by international industry experts from a range of backgrounds

42 CONCURRENT SESSIONS packed with information covering managing projects and teams, plan-driven development, agile development, process improvement and measurement, testing and quality assurance, and security and special topics to meet the professional needs of everyone

VISIT TOP INDUSTRY PROVIDERS discover the latest in software development technologies, trends, and practices

OVER 99% of 2006 attendees recommend the Better Software Conference & EXPO to others in the industry

The promise of Wi-Fi providing secure, available, and adequate network bandwidth anywhere in the home depends on what secure, available, and adequate mean to you. Wi-Fi plays on the fears that installing wiring in the home will cause great pain and cost, but many folks overlook the baggage that Wi-Fi brings with it. In my experience, what Wi-Fi really provides is limited network bandwidth, good for e-mail and Web browsing but not much else; it is easy to set up, but provides networking that is usually not very secure and is limited to unobstructed and interference-free environments.

WIRED INFRASTRUCTURE

With a wired infrastructure, you get a network utility that is highly available and full bandwidth (up to 1 gigabit per second, with 10 Gbps on the horizon). With access requiring a physical connection, security issues are limited to the capabilities of the firewall inside the router.

The wired infrastructure I built in my home centralizes all the wired services from wherever they enter the home to the central wiring closet. From there I project those services to other places around the house. This works not only for network services, but also for phone, alarm, audio, and video.

The cost of a wired infrastructure can be divided into two areas: the cost of the equipment, which is relatively inexpensive; and the cost of the installation, which can be variable depending on the architecture of your home and your skill set. You need network cable, a way to terminate the cable both at the central wiring closet and at the remote location, devices such as routers and network switches to centrally support and distribute services to the remote locations, and patch cables to attach those devices to the wiring infrastructure.

Cable. At a minimum, you should use CAT5e-rated UTP (unshielded twisted pair) cable. Its most economical form is a box with a 1,000-foot roll. Since the type of cable directly impacts overall performance and how long it will remain useful, I upgraded the cable in my home to one that is rated at 350 megahertz and is relatively immune to the kinks and twists that can occur during installation. Standard CAT5e cable typically costs around $50 per box, while the cable in my home costs a little under $200 per box.[2] In most cases one 1,000-foot box should suffice for an entire house.

Wire closet terminations. New or used 110-format punch-down or patch panels, rated CAT5e or greater, are relatively inexpensive and can terminate each UTP cable connecting a remote location to the wiring closet. They typically cost between $50 and $100. To connect the cable to either one of these panels, you may need to borrow or purchase a 110-format punch-down tool that forces each of the eight UTP wires into the panel connector.[3] From the panel you can connect each remote location to a network switch with a small patch cable.

It is important to keep the signals on each wire on a cable consistent through the connection. That means that each of the color-coded eight wires of a UTP cable must be attached to the same RJ45 connector pin on both ends of the cable. The assignment of each of these wires to a specific connector pin is defined by a standard. For my installations I stick to the 568A standard.[4]

Remote terminations. For remote network outlets in other rooms, most connector vendors[5] have a range of faceplate options that fit a standard-size, single-gang, plastic, old work outlet box[6] and range from one to six

connectors. Along with the normal CAT5e or better RJ45 network connector, the options include RJ11 telephone connectors, as well as RCA, F, and BNC coax connectors for audio and video.

INSTALLATION

Running cabling from a central location to a room outlet is the major challenge of installing a wired network. It requires some creativity in planning a path to get to its destination unseen. Ethernet allows for a cable length of up to 100 meters (328 feet). Taking a more roundabout route that allows the cable to remain hidden is encouraged and will rarely impact performance. The only rule is to avoid running parallel to a power cable. Either keep parallel runs at least three or four feet away or cross over them at a 90-degree angle.

Where to put the cable runs. My home is mostly one

story with an unfinished attic above and an unfinished crawl area below the living space. Both of those areas are available as cable pathways from the wiring closet to any network outlet in my house. Because of easy access, I opted for the unfinished crawl area for my cable runs, using large drive rings[7] (they are like big hooks) attached to exposed beams and floor joists every 18 inches or so to support the cable from the wiring closet to its destination directly underneath the room outlet.

To install a flush-mount room outlet, you must first make its location visible to the unfinished attic or crawl space. In my case, I gently removed the floor molding right below where I wanted to place the outlet and drilled a small signal hole through the floor directly next to the wallboard. I threaded a piece of wire through the hole so it was visible in the crawl area below and then cut a hole in the wallboard to accommodate the plastic workbox.

Back under the house the hanging wire became a guide to drill a one-inch hole through the flooring and into the empty space inside the wall. I inserted the cable into the hole and pushed it through so it was visible to the cutout in the wall in the room above. I then pushed the cable through the cutout and the opening in the back of a plastic workbox. The box could then be installed in the cutout flush to the wallboard. I attached the RJ45 connectors to the end of the cable and mounted them into the faceplate, which I then attached to the workbox in the wall. With the hanging wire removed and molding reinstalled, the signal hole was covered from view.

If you do not have an attic or crawl space, you do have other options for installing a flush- or surface-mount outlet in or on a wall. If you have wall-to-wall carpet, you can pull it off the tack strip that runs along the wall, place cable in the space between the tack strip and wall, and replace the carpet back onto the tack strip. Transitions across interior walls are easily done by drilling through the wall right at the level of the floor. If there are no carpets, you can try placing cable behind floor molding. Usually there is a space between the wallboard and the floor that will accommodate a cable. Similar strategies may work for door and ceiling molding. As a last resort, you can drill a hole through an exterior wall to the outside of your home, run wiring around the outside of the house under the eaves, and back through the wall to its final destination. Plug the holes with silicon caulk.

INVEST IN WIRE

Wi-Fi poses accessibility, availability, and bandwidth restrictions, as well as privacy and liability risks that I find unacceptable to my home networking needs. I often wonder at the efforts and expense people will go to in order to avoid installing wires when it is obvious that wires are the best way to transmit information. If you look at any commercial setting, structured wiring is the primary networking platform; wireless is secondary. Given that home networking demands usually lag what is needed commercially, people should embrace wired networks for the home.

With cable and wire closet terminations available from $150 and the parts costs for room outlets at between $5 to $10 per room, the equipment costs for hardwiring your home are relatively inexpensive. The variable is how difficult it is to hide the cable running from the wire closet to the room outlet. With minimal home improvement skills and a forgiving home architecture, a commercial-grade wiring plant can easily and inexpensively be installed to provide rock-solid and secure service to every room in your house. I see it much like the transition from terrestrial broadcast TV to cable TV. Given the increasing demands you will be making on your home network environment, investing in a wired infrastructure will eventually be as common as wiring for cable TV. Q

REFERENCES

1. http://www.google.com/search?hl=en&q=aircrack&btnG=Google+Search.
2. Belden Media Twist (part number 1872A) rated at CAT6 -; http://www.belden.com/pdfs/MasterCatalogPDF/PDFS_links%20to%20docs/11_Networking/11.4.pdf.
3. I used a Harris-Dracon D-814 punch-down handle with a 110 punch-down blade.
4. See the definition of 568A and 568B UTP Cable Termination Standards at http://www.ablecables.com.au/568avb.htm.
5. These vendors include Lucent, Panduit, Leviton, and many others.
6. You can find old plastic workboxes at any home improvement center for around $1.
7. I used 1¼ -inch drive rings priced at around 25 cents a piece.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MACHE CREEGER (mache@creeger.com) is a technology industry veteran based in Silicon Valley. He is the principal of Emergent Technology Associates, marketing and business development consultants to technology companies worldwide.

A Conversation with Michael Stonebraker and Margo Seltzer

Photography by Liesl Clark

Relating TO DATABASES

Over the past 30 years Michael Stonebraker has left an indelible mark on the database technology world. Stonebraker's legacy began with Ingres, an early relational database initially developed in the 1970s at UC Berkeley, where he taught for 25 years. The Ingres technology lives on today in both the Ingres Corporation's commercial products and the open source PostgreSQL software. A prolific entrepreneur, Stonebraker also started successful companies focused on the federated database and stream-processing markets. He was elected to the National Academy of Engineering in 1998 and currently is adjunct professor of computer science at MIT.

Interviewing Stonebraker is Margo Seltzer, one of the founders of Sleepycat Software, makers of Berkeley DB, a popular embedded database engine now owned by Oracle. Seltzer now spends most of her time teaching and doing research at Harvard, where she is full professor of computer science. She was kind enough to lend us her time and travel down the Charles River to

USENIX '07

2007

USENIX ANNUAL TECHNICAL CONFERENCE

Santa Clara, CA
June 17–22, 2007

Join us in Santa Clara, CA, June 17–22, for the 2007 USENIX Annual Technical Conference. USENIX Annual Tech has always been the place to present groundbreaking research and cutting-edge practices in a wide variety of technologies and environments. USENIX '07 will be no exception.

USENIX '07

USENIX '07 will feature:

- An extensive Training Program, covering crucial topics and led by highly respected instructors
- Technical Sessions, featuring the Refereed Papers Track, Invited Talks, and a Poster Session
- Plus BoFs and more!

http://www.usenix.org/usenix07/aq

speak with Stonebraker, her former Ph.D. advisor, at MIT's striking Stata Center.

MARGO SELTZER It seems that your rate of starting companies has escalated in the past several years. Is this a reflection of your having more time on your hands or of something going on in the industry?

MICHAEL STONEBRAKER Well, I think it's definitely the latter. What I see happening is that the large database vendors, whom I'll call the elephants, are selling a one-size-fits-all, 30-year-old architecture that dates from somewhere in the late 1970s.

Way back then the technological requirements were very different; the machines and hardware architectures that we were running on were very different. Also, the only application was business data processing.

For example, there was no embedded database market to speak of. And there was no data warehouse market. Today, there are a variety of different markets with very different computing requirements, and the vendors are still selling the same one-size-fits-all architecture from 25 years ago.

There are at least half a dozen or so vertical markets in which the one-size-fits-all technology can be beaten by one to two orders of magnitude, which is enough to make it interesting for a startup. So I think the aging legacy code lines that the major elephants have are presenting a great opportunity, as are the substantial number of new markets that are becoming available.

SELTZER What new markets are more amenable to what we'll call the small mice, as opposed to the big elephants?

STONEBRAKER There are a bunch of them. Let's start with data warehouses. Those didn't exist until the early 1990s. No one wants to run large, ad hoc queries against transactional production databases, as no one wants to swamp such systems.

So everyone scrapes data off of transactional systems and loads it into data warehouses, and then has their business analysts running whatever they want to run. Everyone on the planet is doing this, and data warehouses are getting positively gigantic. It's very hard to run ad hoc queries against 20 terabytes of data and get an answer back anytime soon. The data warehouse market is one where we can get between one- and two-orders-of-magnitude performance improvements from a very different software system.

The second new market to consider is stream processing. On Wall Street everyone is doing electronic trading. A feed comes out of the wall and you run it through a workflow to normalize the symbols, clean up the data, discard the outliers, and then compute some sort of secret sauce.

An example of the secret sauce would be to compute the momentum of Oracle over the last five ticks and compare it with the momentum of IBM over the same time period. Depending on the size of the difference, you want to arbitrage in one direction or the other.

This is a fire hose of data. Volumes are going through the roof. It's business analytics of the same sort we see in databases. You need to compute them over time windows, however, in small numbers of milliseconds. So, again, a specialized architecture can just clobber the relational elephants in this market.

I also believe the same statement can be made, believe it or not, about OLTP (online transaction processing). I'm working on a specialized engine for business data processing that I think will be about a factor of 30 faster than the elephants on the TPC-C benchmark.

Text is the fourth market. None of the big text vendors, such as Google and Yahoo, use databases; they never have. They didn't start there, because the relational databases were too slow from the get-go. Those guys have all written their own engines.

It's the same case in scientific and intelligence databases. Most of these clients have large arrays, so array data is much more popular than tabular data. If you have array data and use special-purpose technology that knows about arrays, you can clobber a system in which tables are used to simulate arrays.

SELTZER If I rewind history 20 years, you could imagine somebody else sitting in this room, saying, "Today people are building object-oriented applications, and relational databases aren't really any good for objects. We can get a couple of orders-of-magnitude performance improvement if we build a data model around objects instead of around relations."

If we fast-forward 20 years, we know what happened to the object-oriented database guys. Why are these domains different?

STONEBRAKER That's a great question: Why did OO (object-oriented) databases fail? In my opinion the problem with the OO guys is that they were fundamentally architecting a system that was oriented toward the needs of mechanical and electronic CAD. The trouble is, the CAD market didn't salute to their systems. They were very unsuccessful in selling to the engineering CAD marketplace.

The trouble was that the CAD guys had existing systems that would swizzle disk data into main memory, where they would edit stuff with very substantial editing

systems. Then they would reverse swizzle to put it back. If you were to go with object-oriented databases, the only thing you would save would be this swizzling code in both directions. There wasn't enough pain for them to think about switching to something else.

The OODB guys weren't faster than the CAD market's proprietary home-brewed systems. The CAD guys already had mountains of proprietary code to do all this editing. The OODB guys just didn't solve a big enough piece of their problem, and they weren't faster, so they were never very successful in the CAD marketplace.

They failed because the primary market they were going after didn't want them. I don't think that is true of the other markets I've talked about.

SELTZER Let me push on that point a little bit. The Wall Street guys have piles and piles of software that they've built in-house to do exactly what you're describing. What's the compelling reason for them to switch, when the CAD guys didn't think it was worthwhile?

STONEBRAKER There are two very simple answers. Answer number one is that feed volumes are going through the roof, and they're tending to break their legacy infrastructures. That gives them a compelling reason to try something new.

The second reason is that electronic trading has the characteristic that the "secret sauce" works for a while—and then it stops working, so you have to keep changing stuff. The current Wall Street folks are dying because of rickety infrastructure and an inability to change their hardcoded interfaces quickly to meet business needs.

One of the pilot projects that StreamBase [founded by Stonebraker in 2003] did was with a large multinational investment bank with bond-trading desks in Tokyo, New York, London, Paris, and a few other places. Each of these bond desks was using home-brewed software, written locally. What happens is that all of the bond desks reprice bonds on the fly. For example, a typical algorithm would be: "If two-year treasuries tick up by five basis points, then reprice five-year General Motors corporate bonds by three basis points." They have these built-in rules. So all of the bond desks are adjusting their prices and publishing them electronically. The internal traders inside this particular institution watch the same feeds that the bond guys are watching. If they can reach in and grab the bond that's about to be repriced, before the bond guys manage to reprice it, then, of course, they win.

It's basically a latency arms race. If your infrastructure was built with one-second latency, it's just impossible to continue, because if the people arbitraging against you have less latency than you do, you lose. A lot of the legacy infrastructures weren't built for sub-millisecond latency, which is what everyone is moving toward.

SELTZER Many people would argue that we solved the performance problem; processors are fast enough. You're saying, "No, there really still is a performance problem and a latency problem." The hardware guys are giving us processors with multiple cores, so they're increasing parallelism, but they're actually slowing down the single-threaded instruction execution rate. How does that interact with what you're seeing in the stream-processing world?

STONEBRAKER I can explain what's happening with a very simple example. Until recently, everyone was using composite feeds from companies such as Reuters and Bloomberg. These feeds, however, have latency, measured in hundreds of milliseconds, from when the tick actually happens until you get it from one of the composite-feed vendors.

Direct feeds from the exchanges are much faster. Composite feeds have too much latency for the current requirements of electronic trading, so people are getting rid of them in favor of direct feeds.

They are also starting to collocate computers next to the exchanges, again, just to knock down latency. Anything you can do to reduce latency is viewed as a competitive advantage.

Let's say you have an architecture where you process the data from the wire and then use your favorite messaging middleware to send it to the next machine, where you clean the data. People just line up software architectures with a bunch of steps, often on separate machines, and often on separate processes. And they just get clobbered by latency.

SELTZER So, it's not the latency of the instruction execution; it's the latency of the architecture?

STONEBRAKER Right.

SELTZER That argues that the software architectures we're building now are wrong.

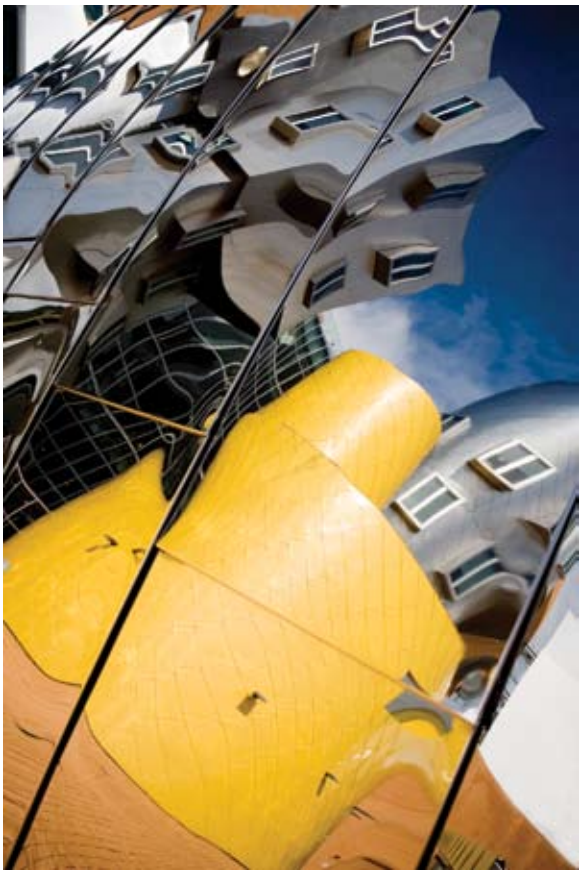
STONEBRAKER Well, as the founder of Sleepycat, you can readily relate to the following characteristic. If I want to be able to read and write a data element in less than a millisecond, there is no possible way that I can do that from an application program to any one of the elephant databases, because you have to do a process switch, a message to get into their systems. You've got to have an embedded database, or you lose.

In the stream processing market, the only kinds of databases that make any sense are ones that are embedded. With all the other types, the latency is just too high.

SELTZER You're preaching to the choir on that one. But

let's talk about that side of the world, where the elephants may be elephants, but they're not standing still. Can you really compete with the elephants in the long term? Are the elephants simply going to get smart and say, "OK, our big engine doesn't do this; so we'll build a little engine that does." Right? They've got lots of programmers.

STONEBRAKER I think of things in a much more holistic fashion. At least in the database world, the large vendors move quite slowly. So it seems the way technology transfer happens is that the elephants just don't do new ideas. They wait for startups to prove that they work. The good ideas go into startups first. Then the elephants pick and choose from them.



SELTZER So the startups are necessary for innovation, because the elephants can't innovate—is that really the answer?

STONEBRAKER I think so.

SELTZER Let's draw a distinction between emerging technology and disruptive technology. *Emerging technology* is anything that's new and may be different from the old stuff. *Disruptive technology* is an emerging technology that ultimately replaces the old technology. My question is whether these new database verticals that you've identified are emerging or disruptive?

STONEBRAKER Well, the elephants never had the text market, so that is simply somebody else's stuff.

Right now the elephants own the warehouse market, but they're selling the wrong technology, and it's not obvious how to morph from old to new. I think that will be very disruptive.

Stream processing is largely a new application. That's simply a green field that didn't exist 20 years ago, and now it does.

And I think if I'm successful in building an OLTP engine that's faster by a factor of 30, that would be very disruptive.

SELTZER Let's talk about how that disruption can occur, given that some people think that nobody actually buys databases anymore; people just buy applications. In order to truly disrupt, you've got to win the applications. How does a tiny startup do that?

STONEBRAKER It's clearest in the data warehouse space, where it turns out that Teradata is doing very well. There's a startup in Framingham, called Netezza, that's doing very well, too. It's selling proprietary hardware, which no one on the planet wants from the get-go, but it's very successful. Why would anybody buy lock-ins and proprietary hardware? The answer is, you have to be in considerable pain.

In the data warehouse market, people are in tremendous pain. There are several ways to talk about this pain. One way is ad hoc queries on data warehouses. The complexity of queries tends to go up at about the square of the database size. So, if you have a small warehouse, you're perfectly okay on Wintel and SQL Server.

But then, if you run out of gas on SQL Server, which doesn't scale anymore, you're facing a discontinuous forklift upgrade to something like, say, Sun Solaris and Oracle. That's different hardware, a different database, and a different operating system. In short, a forklift upgrade—a horrible transition to manage.

If you're staring at this wall, and the solution is a forklift upgrade, then you're in real pain.

Similarly, Oracle has scalability problems that limit its ability to scale in the multi-terabyte range. What usually happens is that people who have a terabyte-size warehouse that is growing are looking at the same kind of wall, and they are forced to go to something like Netezza or Teradata.

If you're looking at any one of these walls, you're faced with great pain in moving to the other side. And if you're in this kind of pain, it means you're willing to take a gander at new technology.

SELTZER I'm going to argue that you just, in fact, agreed with the point of my question, which is that people don't buy databases, they buy applications. The application that you just described is data warehousing. Each customer may run different queries on the warehouse, but the warehouse is still an application.

If you make that transition into the OLTP market, now suddenly OLTP is really a platform, and there are zillions of applications that run on top of it. How does a little guy disrupt the big technology?

STONEBRAKER An interesting way to answer that question is by looking at Tandem. It made a lot of hay by being a serious player in the OLTP market; the New York Stock Exchange runs Tandem. But Tandem didn't start out in OLTP; it started in the machine tool market. The NYSE is not about to trust its data to a 20-person startup.

You have to sneak into the OLTP market some other way, because the people who do serious OLTP are very cautious—they wear both a belt and suspenders. They're very risk-averse, and they're not going to trust a startup, no matter what.

If you started a company, it would behoove you to get two or three huge application elephants to be backers who would agree to go through the pain to give you legitimacy. For example, Dale Skeen's company, Vitria, in the beginning, had FedEx as its premier account. You need a pathfinder application.

Another alternative is if you're in the warehouse market and you're successful because there's so much pain there, then you move into the mixed market, which is partly transactions and partly warehouses. Once you're successful there, you just attempt to eat your way into the OLTP market.

SELTZER The classic disruptive technology approach.

STONEBRAKER All startups with disruptive technology have this problem. How do you get legitimacy in the enterprise software space, where stuff really has to work?

One of the things I find fascinating is that we've been writing software for 30 years and the tools we have to create reliable software are not significantly dissimilar from what we had a long time ago. Our ability to write reliable software is hardly any better now than it was then. That's one of my pet peeves.

SELTZER Does that mean you're going to become a languages guy or a tools guy?

STONEBRAKER I wish I knew something about that.

SELTZER That hasn't stopped others, before.

STONEBRAKER If you look at how you talk to databases right now, you use ODBC and JDBC, embedded in your favorite language. Those are the worst interfaces on the planet. I mean, they are so ugly, you wouldn't wish them on your worst enemy.

C++ and C# are really big, hard languages with all kinds of stuff in them. I'm a huge fan of little languages, such as PHP and Python.

Look at a language such as Ruby on Rails. It has been extended to have database capabilities built into the language. You don't make a call out to SQL; you say, "for E in employee do" and language constructs and variables are used for database access. It makes for a much easier programming job.

There are some interesting language ideas that can be exploited. If I knew anything about programming languages, I probably would attempt to do something.

SELTZER Now I'm really going to hold your feet to the fire. You were around not only at the birth of the relational stuff, but you were one of the movers and shakers that made it happen. Are you going to be one of the movers and shakers who helps lead to its demise, as well?

STONEBRAKER Let's look at Ruby on Rails again. It does not look like SQL. If you do clean extensions of interesting languages, those aren't SQL and they look nothing like SQL. So I think SQL could well go away.

More generally, Ruby on Rails implements an entity-relationship model and then basically compiles it down into ODBC. It papers over ODBC with a clean entity-relationship language embedding.

So you say, "Well, if that's true, is the relational model going to make it?" In semi-structured data, it's already obvious that it's not. In data warehouses, 100 percent of the data warehouses I've seen are snowflake schemas, which are better modeled as entity relationships rather than in a relational model.

If you get a bunch of engines for a bunch of different vertical markets, both the programming language interface and the data model can be thrown up in the air. We aren't in 1970. It's 37 years later, and we should rethink what we're trying to accomplish and what are the right paradigms to do it.

SELTZER One of the big arguments, if I recall correctly, was that you could prove things about the relational model. You could make strong mathematical statements. Is that important in building systems or in designing and developing this kind of database software?

STONEBRAKER If you look at what Ted Codd originally did with the relational model, and you compare it with SQL, you can prove almost nothing about SQL. In fact, there's a terrific paper by Chris Date (A Critique of the

SQL Database Language, *ACM SIGMOD Record*, 1984), that basically spent page after page, in area after area, explaining why SQL has terrible semantics. I think we've drifted far away from Ted Codd's original clean ideas.

SELTZER Have we drifted sufficiently far away from our roots that the roots no longer matter?

STONEBRAKER I think that's right, and I think with good reason: because Ted Codd's original idea was to clean up IBM's IMS (Information Management System) and business data processing. Now you want semi-structured data and data warehousing, and the problem is just vast, compared with what he was talking about 37 years ago. We've taken what started out as a simple standard and grown it into a huge thing, with layer upon layer of junk.

SELTZER Which no one understands.

STONEBRAKER Therefore, what the community does is "add only," which is why we just get more and more stuff. You don't create a skyscraper by growing it one floor at a time, year by year by year, by committee.

SELTZER I've always liked the attitude that we should start hiring programmers to *remove* lines of code, instead of hiring them only to *produce* lines of code.

I have one last question to ask: Now that you've done startups on both coasts, can you say there is a difference?

STONEBRAKER Having seen programmers, students, and technologists on both coasts, I have found that there are more of them on the west coast, but there sure are smart people everywhere.

In terms of the venture capital community, I think the east coast VCs are more conservative. You know, there are more of them who wear bowties.

I don't detect any difference in the intellectual climate. I think MIT has some of the smartest people on the planet. So does Stanford. So does Berkeley.

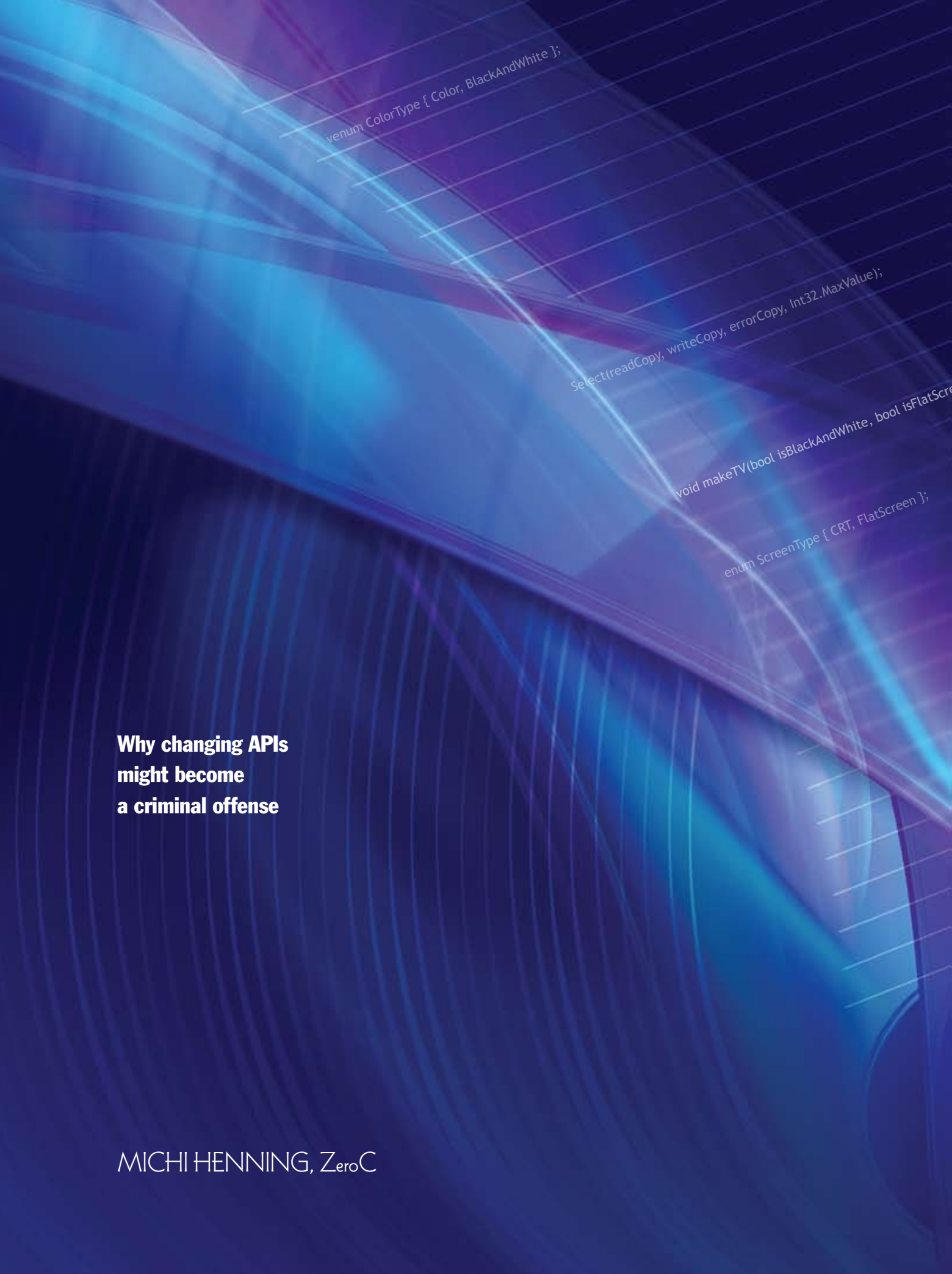
SELTZER There's another school up the river, Mike, that you're missing.

STONEBRAKER I applaud your efforts to improve computer science at Harvard, and I wish Harvard would get deadly serious about computer science because there's a tremendous upside that you can realize over time.

SELTZER Well, come meet our students! Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums



Why changing APIs might become a criminal offense

MICHI HENNING, ZeroC

API Design Matters

After more than 25 years as a software engineer, I still find myself underestimating the time it will take to complete a particular programming task. Sometimes, the resulting schedule slip is caused by my own shortcomings: as I dig into a problem, I simply discover that it is a lot harder than I initially thought, so the problem takes longer to solve—such is life as a programmer. Just as often I know exactly what I want to achieve and how to achieve it, but it still takes far longer than anticipated. When that happens, it is usually because I am struggling with an API that seems to do its level best to throw rocks in my path and make my life difficult. What I find telling is that, after 25 years of progress in software engineering, this still happens. Worse, recent APIs implemented in modern programming languages make the same mistakes as their two-decade-old counterparts written in C. There seems to be something elusive about API design that, despite many years of progress, we have yet to master.

GOOD APIS ARE HARD

We all recognize a good API when we get to use one. Good APIs are a joy to use. They work without friction and almost disappear from sight: the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions.

So, why are there so many bad APIs around? The prime reason is that, for every way to design an API correctly, there are usually dozens of ways to design it incorrectly. Simply put, it is very easy to create a bad API and rather difficult to create a good one. Even minor and quite innocent design flaws have a tendency to get magnified out of all proportion because APIs are provided once, but are called many times. If a design flaw results in awkward or inefficient code, the resulting problems show up at every point the API is called. In addition, separate design flaws that in isolation are minor can interact with each other in surprisingly damaging ways and quickly lead to a huge amount of collateral damage.

BAD APIS ARE EASY

Before I go on, let me show you by example how seemingly innocuous design choices can have far-reaching

API Design Matters

ramifications. This example, which I came across in my day-to-day work, nicely illustrates the consequences of bad design. (Literally hundreds of similar examples can be found in virtually every platform; my intent is not to single out .NET in particular.)

Figure 1 shows the interface to the .NET socket Select() function in C#. The call accepts three lists of sockets that are to be monitored: a list of sockets to check for readability, a list of sockets to check for writeability, and a list of sockets to check for errors. A typical use of Select() is in servers that accept incoming requests from multiple clients; the server calls Select() in a loop and, in each iteration of the loop, deals with whatever sockets are ready before calling Select() again. This loop looks something like the one shown in figure 1.

The first observation is that Select() overwrites its arguments: the lists that are passed into the call are replaced with lists that contain only those sockets that are ready. As a rule, however, the set of sockets to be monitored changes only rarely, and the most common case is that the server passes the same lists in each iteration. Because Select() overwrites its arguments, the caller must make a copy of each list before passing it to Select(). This is inconvenient and does not scale well: servers frequently need to monitor hundreds of sockets so, on each iteration, the code has to copy the lists before calling Select(). The cost of doing this is considerable.

A second observation is that, almost always, the list of sockets to monitor for errors is simply the union of the sockets to monitor for reading and writing. (It is rare that the caller wants to monitor a socket only for error conditions, but not for readability or writeability.) If a server monitors 100 sockets each for reading and writing, it ends up copying 300 list elements on each iteration: 100 each for the read, write, and error lists. If the sockets monitored for reading are not the same as the ones monitored for writing, but overlap for some sockets, constructing the

FIG 1

The .NET socket Select() function in C#

```
public static void Select(IList checkRead, IList checkWrite,
                          IList checkError, int microseconds);
{
```

```
//Server code
int timeout = ...;
ArrayList readList = ...; // Sockets to monitor for reading.
ArrayList writeList = ...; // Sockets to monitor for writing.
ArrayList errorList; // Sockets to monitor for errors.
while(!done)
{
   SocketList readTmp = readList.Clone();
   SocketList writeTmp = writeList.Clone();
   SocketList errorTmp = readList.Clone();
   Select(readTmp, writeTmp, errorTmp, timeout);
   for(int i = 0; i < readTmp.Count; ++i) {
      // Deal with each socket that is ready for reading...
   }
   for(int i = 0; i < writeTmp.Count; ++i) {
      // Deal with each socket that is ready for writing...
   }
   for(int i = 0; i < errorTmp.Count; ++i) {
       // Deal with each socket that encountered an error...
   }
   if(readTmp.Count == 0 &&
      writeTmp.Count == 0 &&
      errorTmp.Count == 0) {
       // No sockets are ready...
   }
}
```

error list gets harder because of the need to avoid placing the same socket more than once on the error list (or even more inefficient, if such duplicates are accepted).

Yet another observation is that Select() accepts a time-out value in microseconds: if no socket becomes ready within the specified time-out, Select() returns. Note, however, that the function has a void return type—that is, it does not indicate on return whether any sockets are ready. To determine whether any sockets are ready, the caller must test the length of all three lists; no socket is ready only if all three lists have zero length. If the caller happens to be interested in this case, it has to write a rather awkward test. Worse, Select() clobbers the caller's arguments if it times out and no socket is ready: the caller needs to make a copy of the three lists on each iteration even if nothing happens!

The documentation for Select() in .NET 1.1 states this about the time-out parameter: "The time to wait for a response, in microseconds." It offers no further explanation of the meaning of this parameter. Of course, the question immediately arises, "How do I wait indefinitely?" Seeing that .NET Select() is just a thin wrapper around the underlying Win32 API, the caller is likely to assume that a negative time-out value indicates that Select() should wait forever. A quick experiment, however, confirms that any time-out value that is equal to or less than zero is taken to mean "return immediately if no socket is ready." (This problem has been fixed in the .NET 2.0 version of Select().) To wait "forever," the best thing the caller can do is pass Int.MaxValue (2^{31}-1). That turns out to be a little over 35 minutes, which is nowhere near "forever." Moreover, how should Select() be used if a time-out is required that is not infinite, but longer than 35 minutes?

When I first came across this problem, I thought, "Well, this is unfortunate, but not a big deal. I'll simply write a wrapper for Select() that transparently restarts the call if it times out after 35 minutes. Then I change all calls to Select() in the code to call that wrapper instead."

The doSelect() function

```
public void doSelect(IList checkRead, IList checkWrite,
                     IList checkError, int milliseconds)
{
   ArrayList readCopy;   // Copies of the three parameters because
   ArrayList writeCopy;  // Select() clobbers them.
   ArrayList errorCopy;

   if (milliseconds <= 0) {
      // Simulate waiting forever.
      do {
         // Make copy of the three lists here...

         Select(readCopy, writeCopy, errorCopy, Int32.MaxValue);
      } while ((readCopy == null || readCopy.Count == 0) &&
               (writeCopy == null || writeCopy.Count == 0) &&
               (errorCopy == null || errorCopy.Count == 0));
   } else {
      // Deal with non-infinite timouts.
      while ((milliseconds > Int32.MaxValue / 1000) &&
             readCopy == null || readCopy.Count == 0) &&
             writeCopy == null || writeCopy.Count == 0) &&
             errorCopy == null || errorCopy.Count == 0)) {

         // Make a copy of the three lists here...

         Select(readCopy, writeCopy, errorCopy,
                (Int32.MaxValue / 1000) * 1000);
         milliseconds -= Int32.MaxValue / 1000;
      }
      if ((readCopy == null || readCopy.Count == 0) &&
          (writeCopy == null || writeCopy.Count == 0) &&
          (errorCopy == null || errorCopy == 0)) {
         Select(checkRead, checkWrite, checkError, milliseconds * 1000);
      }
   }
   // Copy the three lists back into the original parameters here...
}
```

FIG 2

API Design Matters

So, let's take a look at creating this drop-in replacement, called doSelect(), shown in figure 2. The signature (prototype) of the call is the same as for the normal Select(), but we want to ensure that negative time-out values cause it to wait forever and that it is possible to wait for more than 35 minutes. Using a granularity of milliseconds for the time-out allows a time-out of a little more than 24 days, which I will assume is sufficient.

Note the terminating condition of the do-loop in the code in figure 2: it is necessary to check the length of all three lists because Select() does not indicate whether it returned because of a time-out or because a socket is ready. Moreover, if the caller is not interested in using one or two of the three lists, it can pass either null or an empty list. This forces the code to use the awkward test to control the loop because, when Select() returns, one or two of the three lists may be null (if the caller passed null) or may be not null, but empty.

The problem here is that there are two legal parameter values for one and the same thing: both null and an empty list indicate that the caller is not interested in monitoring one of the passed lists. In itself, this is not a big deal but, if I want to reuse Select() as in the preceding code, it turns out to be rather inconvenient.

The second part of the code, which deals with restarting Select() for time-outs greater than 35 minutes, also gets rather complex, both because of the awkward test needed to detect whether a time-out has indeed occurred and because of the need to deal with the case in which milliseconds * 1000 does not divide Int.MaxValue without leaving a remainder.

We are not finished yet: the preceding code still contains comments in place of copying the input parameters and copying the results back into those parameters. One would think that this is easy: simply call a Clone() method, as one would do in Java. Unlike Java, however, .NET's type Object (which is the ultimate base type of all types) does not provide a Clone method; instead, for a type to be cloneable, it must explicitly derive from an ICloneable interface. The formal parameter type of the lists passed to Select() is IList, which is an interface and, therefore, abstract: I cannot instantiate things of type IList, only things derived from IList. The problem is that IList does *not* derive from ICloneable, so there is no convenient way to copy an IList, except by explicitly iterating over the list contents and doing the job element by element. Similarly, there is no method on IList that would allow it to be easily overwritten with the contents of another list (which is necessary to copy the results back into the parameters before doSelect() returns). Again, the only way to achieve this is to iterate and copy the elements one at a time.

Another problem with Select() is that it accepts *lists* of sockets. Lists allow the same socket to appear more than once in each list, but doing so doesn't make sense:

conceptually, what is passed are *sets* of sockets. So, why does Select() use lists? The answer is simple: the .NET collection classes do not include a set abstraction. Using IList to model a set is unfortunate: it creates a semantic problem because lists allow duplicates. (The behavior of Select() in the presence of duplicates is anybody's guess because it is not documented; checking against the actual behavior of the implementation is not all that useful because, in the absence of documentation, the behavior can change without warning.) Using IList to model a set is also detrimental in other ways: when a connection closes, the server must remove the corresponding socket from its lists. Doing so requires the server either to perform a linear search (which does not scale well) or to maintain

the lists in sorted order so it can use a split search (which is more work). This is a good example of how design flaws have a tendency to spread and cause collateral damage: an oversight in one API causes grief in an unrelated API.

I will spare you the details of how to complete the wrapper code. Suffice it to say that the supposedly simple wrapper I set out to write, by the time I had added parameter copying, error handling, and a few comments, ran to nearly 100 lines of fairly complex code. All this because of a few seemingly minor design flaws:

- Select() overwrites its arguments.
- Select() does not provide a simple indicator that would allow the caller to distinguish a return because of a time-out from a return because a socket is ready.
- Select() does not allow a time-out longer than 35 minutes.
- Select() uses lists instead of sets of sockets.

Here is what Select() could look like instead:

```
public static int
Select(ISet checkRead, ISet checkWrite,
       Timespan seconds,
       out ISet readable, out ISet writeable,
       out ISet error);
```

With this version, the caller provides sets to monitor sockets for reading and writing, but no error set: sockets in both the read set and the write set are automatically monitored for errors. The time-out is provided as a Timespan (a type provided by .NET) that has resolution down to 100 nanoseconds, a range of more than 10 million days, and can be negative (or null) to cover the "wait forever" case. Instead of overwriting its arguments, this version returns the sockets that are ready for reading, writing, and have encountered an error as separate sets, and it returns the number of sockets that are ready or zero, in which case the call returned because the time-out was reached. With this simple change, the usability problems disappear and, because the caller no longer needs to copy the arguments, the code is far more efficient as well.

There are many other ways to fix the problems with Select() (such as the approach used by epoll()). The point of this example is not to come up with the ultimate version of Select(), but to demonstrate how a small number of minor oversights can quickly add up to create code that is messy, hard to maintain, error prone, and inefficient. With a slightly better interface to Select(), none of the code I outlined here would be necessary, and I (and probably many other programmers) would have saved considerable time and effort.

THE COST OF POOR APIS

The consequences of poor API design are numerous and serious. Poor APIs are difficult to program with and often require additional code to be written, as in the preceding example. If nothing else, this additional code makes programs larger and less efficient because each line of unnecessary code increases working set size and reduces CPU cache hits. Moreover, as in the preceding example, poor design can lead to inherently inefficient code by forcing unnecessary data copies. (Another popular design flaw—namely, throwing exceptions for expected outcomes—also causes inefficiencies because catching and handling exceptions is almost always slower than testing a return value.)

The effects of poor APIs, however, go far beyond inefficient code: poor APIs are harder to understand and more difficult to work with than good ones. In other words, programmers take longer to write code against poor APIs than against good ones, so poor APIs directly lead to increased development cost. Poor APIs often require not only extra code, but also more complex code that provides more places where bugs can hide. The cost is increased testing effort and increased likelihood for bugs to go undetected until the software is deployed in the field, when the cost of fixing bugs is highest.

Much of software development is about creating abstractions, and APIs are the visible interfaces to these abstractions. Abstractions reduce complexity because they throw away irrelevant detail and retain only the information that is necessary for a particular job. Abstractions do not exist in isolation; rather, we layer abstractions on top of each other. Application code calls higher-level libraries that, in turn, are often implemented by calling on the services provided by lower-level libraries that, in turn, call on the services provided by the system call interface of an operating system. This hierarchy of abstraction layers is an immensely powerful and useful concept. Without it, software as we know it could not exist because programmers would be completely overwhelmed by complexity.

The lower in the abstraction hierarchy an API defect occurs, the more serious are the consequences. If I mis-design a function in my own code, the only person affected is me, because I am the only caller of the function. If I mis-design a function in one of our project libraries, potentially all of my colleagues suffer. If I mis-design a function in a widely published library, potentially tens of thousands of programmers suffer.

Of course, end users also suffer. The suffering can take many forms, but the cumulative cost is invariably high. For example, if Microsoft Word contains a bug that causes

API Design Matters

it to crash occasionally because of a mis-designed API, thousands or hundreds of thousands of end users lose valuable time. Similarly, consider the numerous security holes in countless applications and system software that, ultimately, are caused by unsafe I/O and string manipulation functions in the standard C library (such as scanf() and strcpy()). The effects of these poorly designed APIs are still with us more than 30 years after they were created, and the cumulative cost of the design defects easily runs to many billions of dollars.

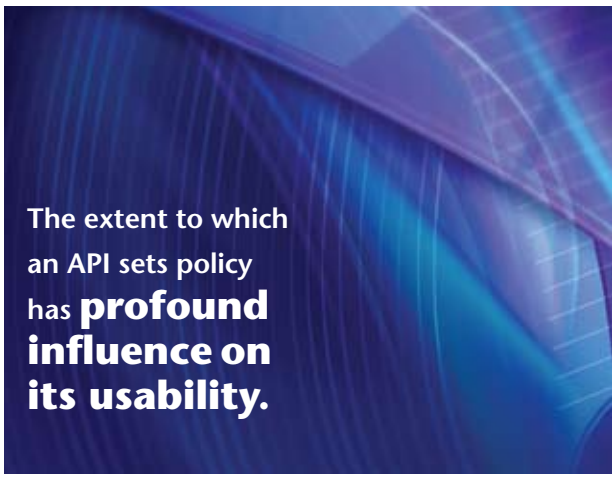
Perversely, layering of abstractions is often used to trivialize the impact of a bad API: "It doesn't matter—we can just write a wrapper to hide the problems." This argument could not be more wrong because it ignores the cost of doing so. First, even the most efficient wrapper adds some cost in terms of memory and execution speed (and wrappers are often far from efficient). Second, for a widely used API, the wrapper will be written thousands of times, whereas getting the API right in the first place needs to be done only once. Third, more often than not, the wrapper creates its own set of problems: the .NET Select() function is a wrapper around the underlying C function; the .NET version first fails to fix the poor interface of the original, and then adds its own share of problems by omitting the return value, getting the time-out wrong, and passing lists instead of sets. So, while creating a wrapper can help to make bad APIs more usable, that does not mean that bad APIs do not matter: two wrongs don't make a right, and unnecessary wrappers just lead to bloatware.

HOW TO DO BETTER

There are a few guidelines to use when designing an API. These are not surefire ways to guarantee success, but being aware of these guidelines and taking them explicitly into account during design makes it much more likely that the result will turn out to be usable. The list is necessarily incomplete—doing the topic justice would require a large book. Nevertheless, here are a few of my favorite things to think about when creating an API.

An API must provide sufficient functionality for the caller to achieve its task. This seems obvious: an API that provides insufficient functionality is not complete. As illustrated by the inability of Select() to wait more than 35 minutes, however, such insufficiency can go undetected. It pays to go through a checklist of functionality during the design and ask, "Have I missed anything?"

An API should be minimal, without imposing undue inconvenience on the caller. This guideline simply says "smaller is better." The fewer types, functions, and parameters an API uses, the easier it is to learn, remember, and use correctly. This minimalism is important. Many APIs end up as a kitchen sink of convenience functions that can be composed of other, more fundamental functions. (The C++ standard string class with its more than 100 member functions is an example. After many years of programming in C++, I still find myself unable to use standard strings for anything nontrivial without consulting the manual.) The qualification of this guideline, *without imposing undue inconvenience on the caller*, is



important because it draws attention to real-world use cases. To design an API well, the designer must have an understanding of the environment in which the API will be used and design to that environment. Whether or not to provide a nonfundamental convenience function depends on how often the designer anticipates that function will be needed. If the function will be used frequently, it is worth adding; if it is used only occasionally, the added complexity is unlikely to be worth the rare gain in convenience.

The Unix kernel violates this guideline with wait(), waitpid(), wait3(), and wait4(). The wait4() function is

sufficient because it can be used to implement the functionality of the first three. There is also waitid(), which could almost, but not quite, be implemented in terms of wait4(). The caller has to read the documentation for all five functions in order to work out which one to use. It would be simpler and easier for the caller to have a single combined function instead. This is also an example of how concerns about backward compatibility erode APIs over time: the API accumulates crud that, eventually, does more damage than the good it ever did by remaining backward compatible. (And the sordid history of stumbling design remains for all the world to see.)

APIs cannot be designed without an understanding of their context. Consider a class that provides access to a set of name–value pairs of strings, such as environment variables:

```
class NVPairs {
    public string lookup(string name);
    // ...
}
```

The lookup method provides access to the value stored by the named variable. Obviously, if a variable with the given name is set, the function returns its value. How should the function behave if the variable is not set? There are several options:

- Throw a VariableNotSet exception.
- Return null.
- Return the empty string.

Throwing an exception is appropriate if the designer anticipates that looking for a variable that isn't there is not a common case and likely to indicate something that the caller would treat as an error. If so, throwing an exception is exactly the right thing because exceptions force the caller to deal with the error. On the other hand, the caller may look up a variable and, if it is not set, substitute a default value. If so, throwing an exception is exactly the wrong thing because handling an exception breaks the normal flow of control and is more difficult than testing for a null or empty return value.

Assuming that we decide not to throw an exception if a variable is not set, two obvious choices indicate that a lookup failed: return null or the empty string. Which one is correct? Again, the answer depends on the anticipated use cases. Returning null allows the caller to distinguish a variable that is not set at all from a variable that is set to the empty string, whereas returning the empty string for variables that are not set makes it impossible to distinguish a variable that was never set from a variable that was explicitly set to the empty string. Returning null is necessary if it is deemed important to be able to make this distinction; but, if the distinction is not important, it is better to return the empty string and never return null.

General-purpose APIs should be "policy-free;" special-purpose APIs should be "policy-rich." In the preceding guideline, I mentioned that correct design of an API depends on its context. This leads to a more fundamental design issue—namely, that APIs inevitably dictate policy: an API performs optimally only if the caller's use of the API is in agreement with the designer's anticipated use cases. Conversely, the designer of an API cannot help but dictate to the caller a particular set of semantics and a particular style of programming. It is important for designers to be aware of this: the extent to which an API sets policy has profound influence on its usability.

If little is known about the context in which an API is going to be used, the designer has little choice but to keep all options open and allow the API to be as widely applicable as possible. In the preceding lookup example, this calls for returning null for variables that are not set, because that choice allows the caller to layer its own policy on top of the API; with a few extra lines of code, the caller can treat lookup of a nonexistent variable as a hard error, substitute a default value, or treat unset and empty variables as equivalent. This generality, however, comes at a price for those callers who do not need the flexibility because it makes it harder for the caller to treat lookup of a nonexistent variable as an error.

This design tension is present in almost every API—the line between what should and should not be an error is very fine, and placing the line incorrectly quickly causes major pain. The more that is known about the context of an API, the more "fascist" the API can become—that is, the more policy it can set. Doing so is doing a favor to the caller because it catches errors that otherwise would go undetected. With careful design of types and parameters, errors can often be caught at compile time instead of being delayed until runtime. Making the effort to do this is worthwhile because every error caught at compile time is one less bug that can incur extra cost during testing or in the field.

The Select() API fails this guideline because, by overwriting its arguments, it sets a policy that is in direct conflict with the most common use case. Similarly, the .NET Receive() API commits this crime for nonblocking sockets: it throws an exception if the call worked but no data is ready, and it returns zero without an exception if the connection is lost. This is the precise opposite of what the caller needs, and it is sobering to look at the mess of

API Design Matters

control flow this causes for the caller.

Sometimes, the design tension cannot be resolved despite the best efforts of the designer. This is often the case when little can be known about context because an API is low-level or must, by its nature, work in many different contexts (as is the case for general-purpose collection classes, for example). In this case, the strategy pattern can often be used to good effect. It allows the caller to supply a policy (for example, in the form of a caller-provided comparison function that is used to maintain ordered collections) and so keeps the design open. Depending on the programming language, caller-provided policies can be implemented with callbacks, virtual functions, delegates, or template parameters (among others). If the API provides sensible defaults, such externalized policies can lead to more flexibility without compromising usability and clarity. (Be careful, though, not to "pass the buck," as described later in this article.)

APIs should be designed from the perspective of the caller. When a programmer is given the job of creating an API, he or she is usually immediately in problem-solving mode: What data structures and algorithms do I need for the job, and what input and output parameters are necessary to get it done? It's all downhill from there: the implementer is focused on solving the problem, and the concerns of the caller are quickly forgotten. Here is a typical example of this:

```
makeTV(false, true);
```

This evidently is a function call that creates a TV. But what is the meaning of the parameters? Compare with the following:

```
makeTV(Color, FlatScreen);
```

The second version is much more readable *to the caller*: even without reading the manual, it is obvious that the call creates a color flat-screen TV. To the implementer, however, the first version is just as usable:

```
void makeTV(bool isBlackAndWhite,
            bool isFlatScreen)
{ /* ... */ }
```

The implementer gets nicely named variables that indicate whether the TV is black and white or color, and whether it has a flat screen or a conventional one, but that information is lost to the caller. The second version requires the implementer to do more work—namely, to add enum definitions and change the function signature:

```
enum ColorType { Color, BlackAndWhite };
enum ScreenType { CRT, FlatScreen };
void makeTV(ColorType col, ScreenType st);
```

This alternative definition requires the implementer to think about the problem in terms of the caller. However, the implementer is preoccupied with getting the TV created, so there is little room in the implementer's mind for worrying about somebody else's problems.

A great way to get usable APIs is to let the *customer* (namely, the caller) write the function signature, and to give that signature to a programmer to implement. This step alone eliminates at least half of poor APIs: too often, the implementers of APIs never use their own creations, with disastrous consequences for usability. Moreover, an API is not about programming, data structures, or algorithms—an API is a *user* interface, just as much as a GUI is. The user at the using end of the API is a programmer—that is, a human being. Even though we tend to think of APIs as machine interfaces, they are not: they are *human*–machine interfaces.

What should drive the design of APIs is not the needs of the implementer. After all, the implementer needs to implement the API only once, but the callers of the API need to call it hundreds or thousands of times. This means that good APIs are designed with the needs of the caller in mind, even if that makes the implementer's job more complicated.

Good APIs don't pass the buck. There are many ways to "pass the buck" when designing an API. A favorite way is to be afraid of setting policy: "Well, the caller might want to do this or that, and I can't be sure which, so I'll make it configurable." The typical outcome of this approach is functions that take five or ten parameters. Because the designer does not have the spine to set policy

and be clear about what the API should and should not do, the API ends up with far more complexity than necessary. This approach also violates minimalism and the principle of "I should not pay for what I don't use": if a function has ten parameters, five of which are irrelevant for the majority of use cases, callers pay the price of supplying ten parameters every time they make a call, even when they could not care less about the functionality provided by the extra five parameters. A good API is clear about what it wants to achieve and what it does not want to achieve, and is not afraid to be up-front about it. The resulting simplicity usually amply repays the minor loss of functionality, especially if the API has well-chosen fundamental operations that can easily be composed into more complex ones.

Another way of passing the buck is to sacrifice usability on the altar of efficiency. For example, the CORBA C++ mapping requires callers to fastidiously keep track of memory allocation and deallocation responsibilities; the result is an API that makes it incredibly easy to corrupt memory. When benchmarking the mapping, it turns out to be quite fast because it avoids many memory allocations and deallocations. The performance gain, however, is an illusion because, instead of the API doing the dirty work, it makes the caller responsible for doing the dirty work—overall, the same number of memory allocations takes place regardless. In other words, a safer API could be provided with zero runtime overhead. By benchmarking only the work done inside the API (instead of the overall work done by both caller and API), the designers can claim to have created a better-performing API, even though the performance advantage is due only to selective accounting.

The original C version of `select()` exhibits the same approach:

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Like the .NET version, the C version also overwrites its arguments. This again reflects the needs of the implementer rather than the caller: it is easier and more efficient to clobber the arguments than to allocate separate output arrays of file descriptors, and it avoids the problems of how to deallocate the output arrays again. All this really does, however, is shift the burden from implementer to caller—at a net efficiency gain of zero.

The Unix kernel also is not without blemish and passes the buck occasionally: many a programmer has cursed the decision to allow some system calls to be interrupted, forcing programmers to deal explicitly with `EINTR` and restart interrupted system calls manually, instead of having the kernel do this transparently.

Passing the buck can take many different forms, the details of which vary greatly from API to API. The key questions for the designer are: Is there anything I could reasonably do for the caller I am not doing? If so, do I have valid reasons for not doing it? Explicitly asking these questions makes design the result of a conscious process and discourages "design by accident."

APIs should be documented before they are implemented. A big problem with API documentation is that it is usually written after the API is implemented, and often written by the implementer. The implementer, however, is mentally contaminated by the implementation and will have a tendency simply to write down what he or she has done. This often leads to incomplete documentation because the implementer is too familiar with the API and assumes that some things are obvious when they are not. Worse, it often leads to APIs that miss important use cases entirely. On the other hand, if the caller (not the implementer) writes the documentation, the caller can approach the problem from a "this is what I need" perspective, unburdened by implementation concerns. This makes it more likely that the API addresses the needs of the caller and prevents many design flaws from arising in the first place.

Of course, the caller may ask for something that turns out to be unreasonable from an implementation perspective. Caller and implementer can then iterate over the design until they reach agreement. That way, neither caller nor implementation concerns are neglected.

Once documented and implemented, the API should be tried out by someone unfamiliar with it. Initially, that person should check how much of the API can be understood without looking at the documentation. If an API can be used without documentation, chances are that it is good: a self-documenting API is the best kind of API there is. While test driving the API and its documentation, the user is likely to ask important "what if" questions: What if the third parameter is `null`? Is that legal? What if I want to wait indefinitely for a socket to become ready? Can I do that? These questions often pinpoint design flaws, and a cross-check with the documentation will confirm whether the questions have answers and whether the answers are reasonable.

Make sure that documentation is *complete*, particularly with respect to error behavior. The behavior of an API when things go wrong is as much a part of the formal

API Design Matters

contract as when things go right. Does the documentation say whether the API maintains the strong exception guarantee? Does it detail the state of out and in-out parameters in case of an error? Does it detail any side effects that may linger after an error has occurred? Does it provide enough information for the caller to make sense of an error? (Throwing a `DidntWork` exception from all socket operations just doesn't cut it!) Programmers *do* need to know how an API behaves when something goes wrong, and they *do* need to get detailed error information they can process programmatically. (Human-readable error messages are nice for diagnostics and debugging, but not nice if they are the only things available—there is nothing worse than having to write a parser for error strings just so I can control the flow of my program.)

Unit and system testing also have an impact on APIs because they can expose things that no one thought of earlier. Test results can help improve the documentation and, therefore, the API. (Yes, the documentation *is* part of the API.)

The worst person to write documentation is the implementer, and the worst time to write documentation is after implementation. Doing so greatly increases the chance that interface, implementation, and documentation will *all* have problems.

Good APIs are ergonomic. Ergonomics is a major field of study in its own right, and probably one of the hardest parts of API design to pin down. Much has been written about this topic in the form of style guides that define naming conventions, code layout, documentation style, and so on. Beyond mere style issues though, achieving good ergonomics is hard because it raises complex cognitive and psychological issues. Programmers are humans and are not created with cookie cutters, so an API that seems fine to one programmer can be perceived as only so-so by another.

Especially for large and complex APIs, a major part of ergonomics relates to consistency. For example, an API is easier to use if its functions always place parameters of a particular type in the same order. Similarly, APIs are easier to use if they establish naming themes that group related functions together with a particular naming style. The same is true for APIs that establish simple and uniform conventions for related tasks and that use uniform error handling.

Consistency is important because not only does it make things easier to use and memorize, but it also enables transference of learning: having learned a part of an API, the caller also has learned much of the remainder of the API and so experiences minimal friction. Transference is important not only within APIs but also across APIs—the more concepts APIs can adopt from each other, the easier it becomes to master all of them. (The Unix standard I/O library violates this idea in a number of places. For example, the `read()` and `write()` system calls place the file descriptor first, but the standard library I/O calls, such as `fgets()` and `fputs()`, place the stream pointer last, except for `fscanf()` and `fprintf()`, which place it first. This lack of parallelism is jarring to many people.)

Good ergonomics and getting an API to "feel" right require a lot of expertise because the designer has to juggle numerous and often conflicting demands. Finding the correct tradeoff among these demands is the hallmark of good design.

API CHANGE REQUIRES CULTURAL CHANGE

I am convinced that it is possible to do better when it comes to API design. Apart from the nitty-gritty technical issues, I believe that we need to address a number of cultural issues to get on top of the API problem. What we need is not only technical wisdom, but also a change in the way we teach and practice software engineering.

EDUCATION

Back in the late '70s and early '80s, when I was cutting my teeth as a programmer and getting my degree, much of the emphasis in a budding programmer's education was on data structures and algorithms. They were the bread and butter of programming, and a good understanding of data structures such as lists, balanced trees, and hash tables was essential, as was a good understanding of common algorithms and their performance tradeoffs. These were also the days when system libraries provided only the most basic functions, such as simple I/O and string manipulation; higher-level functions such as `bsearch()` and `qsort()` were the exception rather than the rule. This meant that it was *de rigueur* for a competent

programmer to know how to write various data structures and manipulate them efficiently.

We have moved on considerably since then. Virtually every major development platform today comes with libraries full of pre-canned data structures and algorithms. In fact, these days if I catch a programmer writing a linked list, that person had better have a very good reason for doing so instead of using an implementation provided by a system library.

Similarly, in the '70s and '80s, if I wanted to create software, I had to write pretty much everything from scratch: if I needed encryption, I wrote it from scratch; if I needed compression, I wrote it from scratch; if I needed inter-process communication, I wrote it from scratch. All this has changed dramatically with the open source movement. Today, open source is available for almost every imaginable kind of reusable functionality. As a result, the process of creating software has changed considerably: instead of *creating* functionality, much of today's software engineering is about *integrating* existing functionality or about repackaging it in some way. To put it differently: API design today is much more important than it was 20 years ago, not only because we are designing more APIs, but also because these APIs tend to provide access to much richer and more complex functionality than they used to.

Looking at the curriculum of many universities, it seems that this shift in emphasis has gone largely unnoticed. In my days as an undergraduate, no one ever bothered to explain how to decide whether something should be a return value or an out parameter, how to choose between raising an exception and returning an error code, or how to decide if it might be appropriate for a function to modify its arguments. Little seems to have changed since then: my son, who is currently working toward a software engineering degree at the same university where I earned my degree, tells me that still no one bothers to explain these things. Little wonder then that we see so many poorly designed APIs: it is not reasonable to expect programmers to be good at something they have never been taught.

Yet, good API design, even though complex, is something that *can* be taught. If undergraduates can learn how to write hash tables, they can also learn when it is appropriate to throw an exception as opposed to returning an error code, and they can learn to distinguish a poor API from a good one. What is needed is recognition of the importance of the topic; much of the research and wisdom are available already—all we need to do is pass them on.

CAREER PATH

I am 47, and I write code. Looking around me, I realize how unusual this is: in my company, all of my programming colleagues are younger than I and, when I look at former programming colleagues, most of them no longer write code; instead, they have moved on to different positions (such as project manager) or have left the industry entirely. I see this trend everywhere in the software industry: older programmers are rare, quite often because no career path exists for them beyond a certain point. I recall how much effort it took me to resist a forced "promotion" into a management position at a former company—I ended up staying a programmer, but was told that future pay increases were pretty much out of the question if I was unwilling to move into management.

There is also a belief that older programmers "lose the edge" and don't cut it anymore. That belief is mistaken, in my opinion: older programmers may not burn as much midnight oil as younger ones, but that's not because they are old, but because they get the job done without having to stay up past midnight.

This loss of older programmers is unfortunate, particularly when it comes to API design. While good API design can be learned, there is no substitute for experience. Many good APIs were created by programmers who had to suffer under a bad one and then decided to redo the job, but properly this time. It takes time and a healthy dose of "once burned, twice shy" to gather the expertise that is necessary to do better. Unfortunately, the industry trend is to promote precisely its most experienced people away from programming, just when they could put their accumulated expertise to good use.

Another trend is for companies to promote their best programmers to designer or system architect. Typically, these programmers are farmed out to various projects as consultants, with the aim of ensuring that the project takes off on the right track and avoids mistakes it might make without the wisdom of the consultants. The intent of this practice is laudable, but the outcome is usually sobering: because the consultants are so valuable, having given their advice, they are moved to the next project long before implementation is finished, let alone testing and delivery. By the time the consultants have moved on, any problems with their earlier sage advice are no longer their problems, but the problems of a project they have long since left behind. In other words, the consultants never get to live through the consequences of their own design decisions, which is a perfect way to breed them into incompetence. The way to keep designers sharp and honest is to make them eat their own dog food. Any pro-

API Design Matters

cess that deprives designers of that feedback is ultimately doomed to failure.

EXTERNAL CONTROLS

Years ago, I was working on a large development project that, for contractual reasons, was forced into an operating-system upgrade during a critical phase shortly before a delivery deadline. After the upgrade, the previously working system started behaving strangely and occasionally produced random and inexplicable failures. The process of tracking down the problem took nearly two days, during which a large team of programmers was mostly twiddling its thumbs. Ultimately, the cause turned out to be a change in the behavior of awk's index() function. Once we identified the problem, the fix was trivial—we simply installed the previous version of awk. The point is that a minor change in the semantics of a minor part of an API had cost the project thousands of dollars, and the change was the result of a side effect of a programmer fixing an unrelated bug.

This anecdote hints at a problem we will increasingly have to face in the future. With the ever-growing importance of computing, there are APIs whose correct functioning is important almost beyond description. For example, consider the importance of APIs such as the Unix system call interface, the C library, Win32, or OpenSSL. Any change in interface or semantics of these APIs incurs an enormous economic cost and can introduce vulnerabilities. It is irresponsible to allow a single company (let alone a single developer) to make changes to such critical APIs without external controls.

As an analogy, a building contractor cannot simply try out a new concrete mixture to see how well it performs. To use a new concrete mixture, a lengthy testing and approval process must be followed, and failure to follow that process incurs criminal penalties. At least for mission-critical APIs, a similar process is necessary, as a matter of self-defense: if a substantial fraction of the world's economy depends on the safety and correct functioning of certain APIs, it stands to reason that any changes to these APIs should be carefully monitored.

Whether such controls should take the form of legislation and criminal penalties is debatable. Legislation would likely introduce an entirely new set of problems, such as stifling innovation and making software more expensive. (The ongoing legal battle between Microsoft and the European Union is a case in point.) I see a real danger of just such a scenario occurring. Up to now, we have been lucky, and the damage caused by malware such as worms has been relatively minor. We won't be lucky forever: the first worm to exploit an API flaw to wipe out more than 10 percent of the world's PCs would cause economic and human damage on such a scale that legislators *would* be kicked into action. If that were to happen, we would likely swap one set of problems for another one that is worse.

What are the alternatives to legislation? The open source community has shown the way for many years: open peer review of APIs and implementations has proven an extremely effective way to ferret out design flaws, inefficiencies, and security holes. This process avoids the problems associated with legislation, catches many flaws before an API is widely used, and makes it more likely that, when a zero-day defect is discovered, it is fixed and a patch distributed promptly.

In the future, we will likely see a combination of both tighter legislative controls and more open peer review. Finding the right balance between the two is crucial to the future of computing and our economy. API design truly matters—but we had better realize it before events run away with things and remove any choice. 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 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, 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.

introducing...

ACM's *Newly Expanded* Online Books & Courses Programs!

Helping Members Meet Today's Career Challenges

Over 2,200 FREE Online Courses from SkillSoft

The ACM online course program features **free and unlimited access to over 2,200 online courses** from SkillSoft, a leading provider of e-learning solutions. This new collection of courses offers a host of valuable resources that will help to maximize your learning experience. Available on a wide range of information technology and business subjects, these courses are open to ACM Professional and Student Members.

SkillSoft courses offer a number of valuable features, including:

- ***Job Aids***, tools and forms that complement and support course content
- ***Skillbriefs***, condensed summaries of the instructional content of a course topic
- ***Mentoring*** via email, online chats, threaded discussions - 24/7
- ***Exercises***, offering a thorough interactive practice session appropriate to the learning points covered previously in the course
- ***Downloadable content*** for easy and convenient access

600 FREE Online Books from Safari

The ACM online books program includes **free and unlimited access to 600 online books** from Safari® Books Online, featuring leading publishers including O'Reilly. Safari puts a complete IT and business e-reference library right on your desktop. Available to ACM Professional Members, Safari will help you zero in on exactly the information you need, right when you need it.

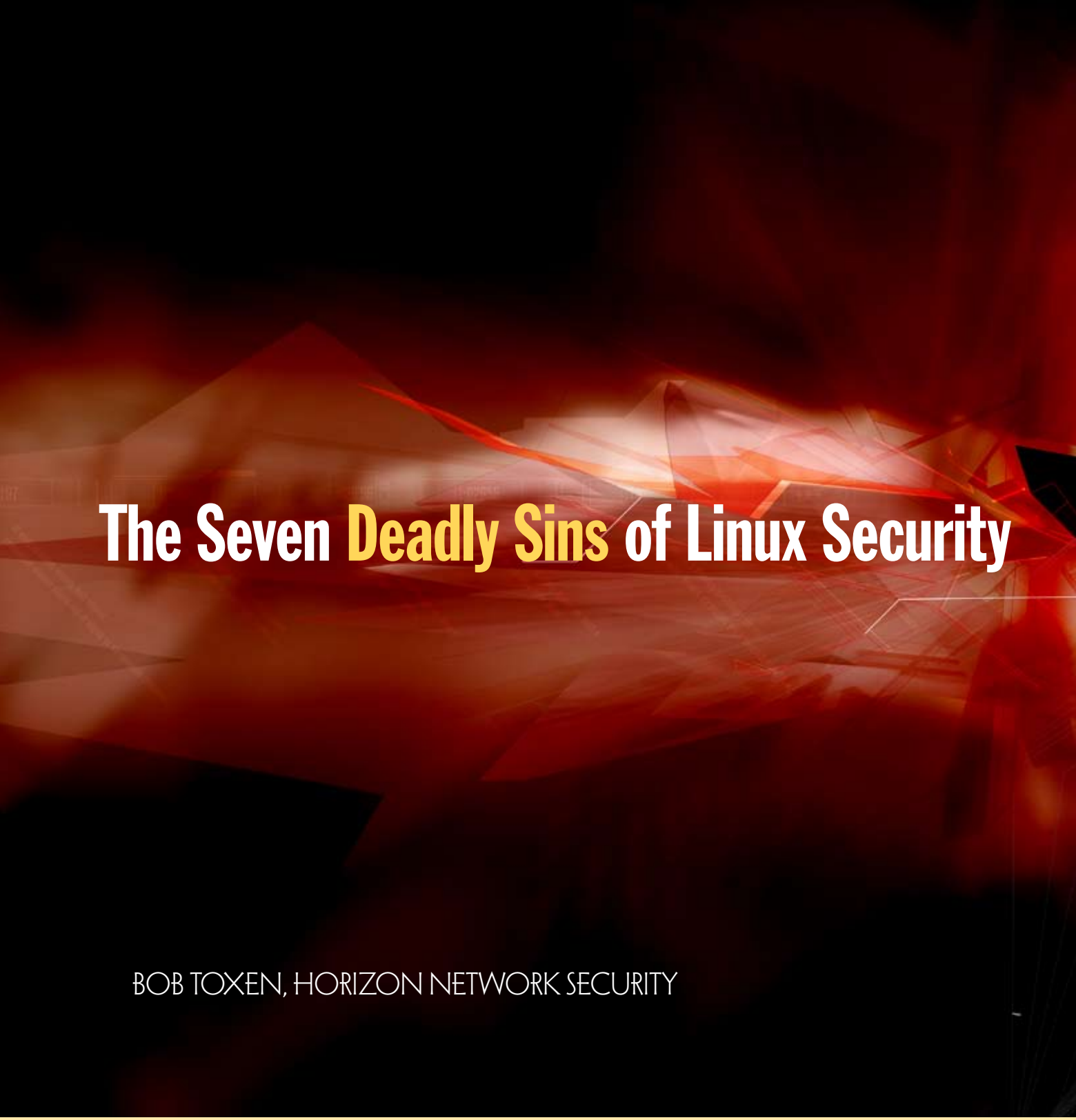
500 FREE Online Books from Books24x7

All Professional and Student Members also have **free and unlimited access to 500 online books** from Books24x7®, a rotating collection of complete unabridged books on the hottest computing topics. This virtual library puts information at your fingertips. Search, bookmark, or read cover-to-cover. Your bookshelf allows for quick retrieval and bookmarks let you easily return to specific places in a book.

Association for Computing Machinery

Advancing Computing as a Science & Profession

pd.acm.org
www.acm.org/join



The Seven Deadly Sins of Linux Security

BOB TOXEN, HORIZON NETWORK SECURITY

The problem with security advice is that there is too much of it and that those responsible for security certainly have too little time to implement all of it. The challenge is to determine what the biggest risks are and to worry about those first and about others as time permits. Presented here are the seven common problems—the seven deadly sins of security—most likely to allow major damage to occur to your system or bank account. If any of these are a problem on any of your systems, you will want to take care of them immediately.

These seven deadly sins are based on my research and experience, which includes too many people who wait until after their Linux or Unix systems have suffered security breaches before they take action to increase system security, and on forensics analysis and discussions with systems administrators. Most of these sins and their solu-

tions also apply to Macs, Windows, and other platforms.

They are not ordered by risk level because committing any one of them will likely allow your system to be compromised if it is accessible from the Internet. Even if you are behind a firewall, if you receive any untrusted data from the Internet, such as Web pages, e-mail, or instant messages, your system is at great risk. Avoid these sins like the devil.

Without further ado, here are the seven deadly sins and what to do about them.

SIN ONE: WEAK PASSWORDS

As a systems administrator, you are aware of the system breaches possible on your Linux or Unix machine. You have taken the time and effort to devise a difficult-to-guess root password that uses at least 12 characters that

The Seven Deadly Sins of Linux Security

include at least two words or no words from the dictionary, uses both letters and digits, and has upper- and lowercase letters and some punctuation characters.

I still run into clients with passwords so simple that any hacker could break them in a few minutes with a tweaked version of ssh that guesses different passwords. Such hacker tools can be found on the Web easily with Google or built by any C or C++ programmer. On Internet-accessible systems, I have seen root passwords consisting of a word followed by a small number, where that word is related to the company, what it does, who is in it, or where it is. A good hacker will go to your Web site and see all of this information, then feed it into a password-cracking program.

Another common mistake is to use the same password or very similar passwords for root accounts (or other important accounts) on different systems. Thus, a cracker who breaches one system through a means other than password guessing will then be able to install a Trojaned server for ssh, FTP, or IMAP, or a Trojaned CGI program on that system, see what passwords you use, and try them on the other systems. I have seen this happen many times.

A variation is to use ssh public keys to allow an account on one system to ssh into another system without supplying any password. At the very least, pick a moderately hard-to-crack password for your ssh keys. If you must have an automatic program use ssh without a password to ssh into another system, then create either a separate nonroot account on the target system or an alternate account with UID 0 but a login "shell" that does just what is needed, such as doing a backup.

An even better solution, say for a remote backup, would be for the system needing to be backed up to ssh into the system receiving the backups as a unique unprivileged account for this purpose and copy an encrypted version of the backup. Thus, if the backup server is compromised, no confidential data will be obtained.

Let's hope your root password is awesome and that no one could guess it in 100 years. OK, some obsessive with a program such as Crack could destroy it in a few days except that you use shadow passwords, but that's another story. It is critically important to select good passwords.

How are your users doing? Choke, cough, gag, hack. Every account is a possible entry point. Have your users followed your advice, company policy, or threats to devise good passwords? Are they being as careful as you are? Probably not. Now it is your turn to don the black hat and think like your enemy.

Can you break into your users' accounts by using a password-cracking program? You definitely will need to get written management approval to conduct this level of security audit. There are notable cases of unauthorized audits landing people in jail or at least on the unemployment rolls. (Randal Schwartz is one. The software consultant and author was brought to trial for accessing a password file at Intel in what he says was an attempt to show lapses in security.)

You might even install a module in the passwd program that automatically tries to break a user's proposed new password. Though the standard passwd program

TIP

Consider how severe the consequences would be if one account or one system gets hacked. Can the hacker then get into other accounts or other systems? If so, change passwords, ssh usage, etc. so that the hacker cannot spread the damage to other accounts and systems.

This illustrates the concept of containment. Accept that some account, possibly root, on some system will get compromised. Ensure that the compromise will not spread by doing careful failure analysis now, before you suffer a compromise.

TIP

Protecting every account is critical because of local root vulnerabilities in various programs and the Linux kernel itself. These vulnerabilities allow a hacker who gets shell access as any user to make himself or herself root.

```
passwd password requisite /usr/lib/security/pam_cracklib.so retry=3
passwd password required /usr/lib/security/pam_pwdb.so use_authtok
```

FIG 1

TIP

Avoid default passwords as if your job depended on it.

makes very simple tests, there are more sophisticated routines that include much of Crack's capability. One way to do this is to make use of the cracklib capability in the PAM (pluggable authentication modules) enhancements to the passwd program. The cracklib library analyzes passwords to determine if they can be easily cracked. PAM offers additional security for Linux and Unix systems.

Edit the /etc/pam.d/passwd file to include the code in figure 1. This will cause the PAM-enabled passwd program to load these dynamically loadable program libraries. PAM now is standard with Red Hat. On some systems these are in /lib instead of /usr/lib. (Another good source for PAM information is http://www.sun.com/software/solaris/pam/.)

On Slackware this capability will be enabled if the following line is present in /etc/login.defs (and the dictionary is installed):

```
CRACKLIB_DICTPATH   /var/cache/cracklib/cracklib_dict
```

Consider restricting which remote systems can ssh into your systems' various accounts either through IP tables firewall rules or by editing your ssh server's configuration file, /etc/ssh/sshd_config, to limit which remote systems can ssh in and which accounts they can ssh into, or use both methods for additional security. Make this list very short for root (in sshd_config).

SIN TWO: OPEN NETWORK PORTS

Just as every account on your system is a potential path for a password cracker, every network service is a road to it. Disable and uninstall services you do not need. Most Linux distributions and Unix vendors install tons of software and services by default. They deliberately prefer easy over secure. Many of these are neither necessary nor wanted. Take the time to remove software and services you do not need. Better yet, do not install them to begin with.

To find out which services are being run, use the netstat -atuv command. Even a home system can have dozens of different ports open. A large Web server could have more.

If there are services listed that you do not want to be provided by this box, disable them. Many distributions offer a control panel to do this easily, including Red Hat and Mandriva. You may want to remove the binaries from the disk or chmod them to 0, especially any that are set-UID or set-GID.

NFS, finger, the shell, exec, login r* services (rsh, rexec, and rlogin), FTP, telnet, sendmail, DNS, and linuxconf are some of the more popular services that get installed by default on many Linux distributions; at least some of these should not be enabled for most systems. Most are controlled by the daemon xinetd; these can be disabled by editing the /etc/xinetd.d/* scripts.

You do not need the FTP or telnet daemons to use the respective clients to connect to other systems. You do not need the sendmail daemon listening on port 25 to send mail out, to send mail to local users, or to download mail via POP or IMAP. (You do need to invoke sendmail periodically to de-spool delayed outgoing mail.) You need DNS (named, the name daemon) only if other systems will be querying yours for this data. Most programs running on your own system will be very happy to read /etc/resolv.conf and query the main DNS server of your ISP or organization instead of contacting a named process running on your system. Coincidentally, named's ports are some of the most popular ports that crackers use to break into systems. If you do need to run named, use the recently added facilities that allow it to chroot itself and switch to a nonroot user.

All of these services, except the normal installations of NFS,[1] DNS, and sendmail, are started on demand by xinetd. They may be turned off by commenting out their entries under /etc/xinetd.d. Many distributions offer a control panel or Linuxconf to do this easily, including Red Hat and Mandriva.

The Seven Deadly Sins of Linux Security

The stand-alone services are turned off by altering their entries under /etc/rc.d or in configuration files there.

On Red Hat-based systems, issue the following commands to shut down portmap and prevent it from being restarted on reboot.

```
/etc/rc.d/init.d/portmap stop
chkconfig --del portmap
```

An alternative tool is the ASCII menu-based ntsysv program. Like chkconfig, ntsysv manipulates the symbolic links only under /etc/rc.d/rc[0-6].d, so you also will need to explicitly shut down the service. To do both of these, issue the commands

```
/etc/rc.d/init.d/portmap stop
ntsysv
```

On other distributions that use System V-style startup scripts (/etc/rc.d/rc[0-6].d directories for Red Hat derivations and /etc/rc.[0-b].d for Debian), rename the appropriate script under rcX.d (X usually is 3) that starts with a capital *S* and has the service name in it. For example,

```
cd /etc/rc.d/rc3.d
mv S11portmap K11portmap
```

Just as only scripts starting with *S* are invoked when entering the respective run level, scripts starting with *K* are invoked when exiting that run level. This is to turn off daemons that should run only in that run level. For example, this mechanism will turn off sshd, the ssh daemon, when switching from run level 3 (multiuser with networking) to run level s (single-user mode). Just as a selected *Ssomething* script can be disabled by renaming to *ssomething*, one of these latter scripts can be renamed from *Ksomething* to *ksomething* to disable it.

On Slackware and similar systems, simply comment out the lines starting them in /etc/rc.d/*. The grep program may be used to find these. Be sure to terminate any of these services that are running on your system after altering the configuration files.

If you do not want to bother with kill, a simple reboot will do this and verify that the configuration files were correctly altered. (Having a set of available rescue disks before this reboot would be a fine idea.)

To remove these services from your system, you can use your distribution's package manager. Red Hat-based installations use RPM; Debian-based distributions use dpkg; SuSE uses YAST; and Slackware uses pkgtool.

Linux and Unix are like the Swiss army knife of networking: they have one or two tools that get used all the time, others that are used less often, and some that are never used. Unlike the Swiss army knife, you can slim down Linux or Unix to just the services you need and discard those you do not. I will never use the awl or scissors on my knife just as I will never use rsh or the set-UID to root features of mount or umount.

Decide which ports you wish to have open (such as www and ftp) and close the rest. Closing unnecessary ports makes your system more secure and perform better.

> The most careful sysadmins will reboot their systems several times after making changes to startup scripts, other configuration files, and the kernel, and after installing security patches to ensure correct and reliable startup and operation.
>
> TIP

SIN THREE: OLD SOFTWARE VERSIONS

Linux and Unix are not perfect. People find new vulnerabilities every month.[2] Do not despair, though. The speed with which problems are found and fixed in Linux is the fastest on the planet. Your challenge as an administrator is to keep up with the changes.

Each distribution has a mailing list through which security bulletins are issued, and an FTP or Web site where the fix will be available. There are also excellent independent security mailing lists, such as Bugtraq and X-Force's Alert. You can (and should) subscribe to these lists.[3]

Other good sources of Linux security information are http://www.lwn.net/ and http://www.linuxtoday.com/. These sites are distribution-neutral and carry all of the major distributions' security advisories.

One of the advantages of Linux is that when a fix is issued, it is very quick to install. Furthermore, unless it is in the kernel, your downtime for that service is on the order of seconds or minutes. Rarely, if ever, is a reboot necessary.

SIN FOUR: INSECURE AND BADLY CONFIGURED PROGRAMS

The use of insecure programs (such as PHP, FTP, rsh, NFS, and portmap) in other than carefully controlled situations and failure to configure other programs properly continues to be a major security sin.

Most sysadmins know that POP and IMAP (unless wrapped in SSL), telnet, and FTP[4] send passwords and data in the clear (unencrypted). They know that PHP, NFS, and portmap have a history of security problems, as well as design defects in their authentication. Many use them anyway, and then are surprised when they get broken into. Instead, use spop, simap, ssh, and ssh's scp or sftp, or put a good firewall in front of that subnet, or set up a restricted VPN between your facilities. If you absolutely must use PHP, keep it patched and carefully audit your code for problems.

Many programs are secure only if properly configured. It is common for sysadmins to configure them improperly, sometimes because of a lack of training and understanding of the risks; other times use of an insecure feature is deliberate, because "I just gotta have it." A recent case in point is Apache's PHP capability, which has had a history of security problems. These problems have been well publicized, and still some people cannot seem to use it securely or find an alternative. Security and convenience are often contradictory, and you have to make a choice between the two.

Before deciding to deploy a service (or changing which capabilities will be used or how the service will be deployed), do some research. Check the security history and understand how the service may be deployed securely. If it cannot be deployed securely, what are secure alternatives? I still encounter people using FTP, not realizing that sftp is an excellent alternative. Putting an insecure service such as NFS behind a firewall may be the solution for some. For others, putting their insecure Windows networks behind firewalls, with their different offices linked via a VPN between these same Linux firewalls, offers excellent security. Configure a firewall with separate subnets on separate interfaces for different categories of users, such as students and faculty or sales, human resources, and engineering.

Absolutely prohibit wireless networks inside of the firewall or to any system with confidential information unless all wireless traffic first is encrypted with IPsec or equivalent. Do not rely on WEP (Wired Equivalent Privacy) or its successors.

Web servers and CGI programs are the bane of Linux and Unix computer security. Simply speaking, a CGI program is one of the easiest ways that a hacker can get into your system. It is essentially a program that runs on your computer at the request of anyone and everyone without passwords and has the access to do powerful things (for example, shipping valuable merchandise, revealing confidential data such as your customers' credit card numbers, and moving money between accounts).

A CGI allows anyone to access your Web site, good intentions or not. While other "accepted" servers such as sendmail and named also will talk with anyone, the scope of what a client may request is far smaller. Although these latter servers have had their share of serious security bugs, those that keep their security patches up to date have minimal risk.

Here are a few hard and fast rules that will help make your Web site secure.

Know your data (supplied by Web clients).

- Establish maximums and minimums for data-entry values and lengths of fields.
- Decide which characters are acceptable in each field. Expect the malicious to send you control characters and non-ASCII bytes. Expect that crackers will use the % encoding or alternate character sets to generate these evil characters. Thus, you need to check for illegal characters both before and after % conversion and in different character sets.
- Double-check each entered value. A surprising number of shopping-cart packages put the price of items in the form and believe the price in the filled-out form sent by the user. All a user needs to do to give himself or herself a discount is to alter this form.
- If possible enumerate the allowed values instead of using ranges (except for listing ranges of letters and digits).
- Understand, too, that an evil Web client can send bytes back to your server. The hacker may copy and alter your Web form to change your "fixed" fields, etc.
- Use a secure language. Client-supplied data never should be handed directly to a shell script; there are too many opportunities for a cracker to get a shell or to exploit a buffer overflow vulnerability. For many that secure language will be C, C++, Perl, Java, or Python. If that language offers checking for tainted data, use

it. One language does not fit all. Perl has a number of features to enable safer CGI programs.[5] These include the "tainted data" feature, the -w flag to warn you about things that you are creating but not using, the strict capability, and perlsec. These features are discussed in http://perldoc.perl.org/perlsec.html.

- If you have many CGI programs—with a few being carefully written so that they manipulate confidential data, and some that are more casually written because they do not handle critical data—consider the following. Use the suEXEC program that comes with Apache to run these different classes of CGIs as different Linux or Unix users. This allows you to use operating system file permissions to prevent the less-trusted CGIs from accessing more confidential data. Documentation on suEXEC is available at http://apache.org/docs/suexec.html.

Analyze and audit CGIs for vulnerabilities.
When writing CGI programs, look at them the way a cracker would and try to break them. Stop buffer overflows by using good programming techniques. An easy way to determine if the line is larger than the buffer is to see that it does not end with a newline character, as this example illustrates:

```
#include <stdio.h>
#include <string.h>

int   c;
char  buf[200];

if (!fgets(buf, sizeof buf, stdin))
        error();
else if (!strchr(buf, '\n')) {
                /* Read rest of long line. */
        while ((c = getchar()) != EOF
          && c != '\n')
                ;
        overflow();
}
```

Do not use the gets() routine because it does not do any checking for buffer overflows; use fgets() instead. Many of the other popular C string functions have similar weaknesses. The strcpy() function, for example, "lets" you copy a large buffer into a small buffer, overwriting unrelated memory. The strncpy() function is an excellent alternative. A safe way to copy strings is:

```
strncpy(dest_buf, source_buf,
  sizeof dest_buf);
dest_buf[sizeof dest_buf - 1] = '\0';
```

To detect a problem, one possibility is:

```
if (strlen(source_buf)
  >= sizeof dest_buf)
  error();
else
  strcpy(dest_buf, source_buf);
```

Check for escape sequences, the possibility of a client issuing Linux or Unix commands (by inserting spaces, quotes, or semicolons), binary data, calls to other programs, etc. Often it is safer to have a list of allowed characters rather than determining each unsafe character.

The following C code may be used to process a field in which the client should supply his or her name. In this example, the calling process supplies a NUL-terminated string; this routine returns 0 if the string is a legal name, and -1 otherwise. The second argument specifies the maximum legal string allowed, including the terminating NUL byte. Note that the calling routine must be careful to ensure that its buffer did not overflow. I chose clear code over slightly more efficient code.

```
#include <string.h>

char  okname[] = " .'-,abcdefghijklmnopqrstuvwxyz"
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

/* Return 0 on legal names, -1 otherwise. */
legal(char *name, int maxlen)
{
        if (!name || !*name
          || strlen(name) >= maxlen)
                return -1;
        while (*name)
                if (!strchr(okname, *name++))
                        return -1;
        return 0;
}
```

Many system break-ins relating to Linux and Unix Web servers happen via insecure CGIs.

Implement rings of security in CGIs.
Try to design your application so that even if it finds a CGI vulnerability, the system is protected from major damage. One solution is to have CGIs just be front ends for a solidly written server running on a different machine. The more hurdles a hacker must jump to reach the goal, the more likely it is that he or she will stumble.

Watch for bug reports in third-party CGIs and inspect their code.
If you use third party-supplied CGI scripts (such as shopping carts), you should subscribe to the appropriate mailing lists and watch for security bulletins. If possible, get the source code and review it. If you do not know the language, then get someone who does to review it.

Many CGIs, both commercial and open source, have severe security holes that are well known to the hacker community. Many locally written CGIs have security vulnerabilities because the programmers who write them typically have no training in writing secure code and such code is rarely audited.

Avoid creating and using set-UID and set-GID programs to the maximum extent possible, especially programs set-UID to root (and try real hard).
Many system programs run as root. Frequently all these programs need to be set-UID to run as some user to gain access to data that should not be world accessible. Other programs need to be set-UID to root only when starting to open a low network port for listening or to change its privileges to that of a particular user. In this case, the program then should give up root privileges. Apache, named, and ftpd were enhanced several years ago to do this for better security. Different programs may need to be set-UID to different users to protect them from each other.

Do not keep clients' confidential data on the Web server.
Avoid storing users' privileged data (credit card numbers, financial details, mailing addresses and phone numbers, etc.) on the same machine as the Web server. This separation will force a hacker to crack two systems instead of just one to get this data.

Do not include users' confidential data (credit card numbers, financial details, mailing addresses and phone numbers, session ID, etc.) in an URL or cookie.[6]
Frequently this is done (insecurely) as arguments to a CGI program. Consider the following example:

```
www.abroker.com/cgi-bin/address_change?account=666
?passwd=secret&addr=1+Maple+St.&phone=301-688-6524
```

Some browsers may store this URL (containing confidential data) in a history file. If someone is browsing from a public terminal, such as a school or library, you could be liable for careless handling of the data. Similar issues are present for cookies.

Be very sure that the privileged data that a user supplies on a form does not show up as the default data for the next person to "pull down" that form and see.
Yes, this has actually happened.

Always protect the user who types in a password.
Take the user to a secured area prior to this information being entered and ensure that the password or credit card number will be encrypted on the system (with https) before transmission to your server.

SIN FIVE: INSUFFICIENT RESOURCES AND MISPLACED PRIORITIES

At many organizations, management simply will not approve sufficient resources to allow sysadmins to provide good security. It takes many things to achieve a truly comprehensive security solution. Education, design, proper implementation, user training, maintenance, and continual vigilance all are required for an organization to be secure. Frequently, security is limited to what a sysadmin is willing to do on his or her own time. Yet, a sysadmin who is unwilling to spend the time will certainly be blamed for any violations. This deadly sin concerns problems that are not the sysadmin's direct responsibility. In other words, management will not allow the sysadmin to make the changes necessary for good security.

This may not be a "technical" problem, but it has been the cause of break-ins at numerous organizations. Lack of resources commonly is a result of misplaced priorities. For example, the following is a common misconception of those whose organizations have not been broken into: "The media exaggerates every danger well beyond the true risk." Show your manager media accounts of large companies that have suffered security breaches. If you shopped at T.J. Maxx or Marshalls in 2006, you probably received a new credit card number thanks to TJX Cos., the parent company, which suffered a security breach in December. Circuit City suffered a similar breach. Consider making a present of Bruce Schneier's excellent

The Seven Deadly Sins of Linux Security

book, *Secrets and Lies: Digital Security in a Networked World* (Wiley, 2004), to your boss. *Secrets and Lies* is aimed at management and limits the tech-speak.

On a number of occasions, I have warned clients about major security problems only to have them decide that security was not as important as getting that next release out or making nonsecurity-related computer improvements. Later, they learned the sad reality—recovering from a security breach commonly costs 10 times as much as having implemented good security before the break-in—and only then did they spend the money to implement the security.

Furthermore, the estimate of the cost of recovering from a security breach being 10 times the cost of prevention is only the direct cost. It does not account for the lost market opportunities for delayed products, the loss of customers who heard about the security breach and went elsewhere, and the costs to customers and employees who could not access your Web site and e-mail during recovery. It does not account for lost investors and other consequences of bad publicity, and it most certainly does not account for the damage done to an IT professional's career.

What can be done to resolve insufficient resources and misplaced priorities? Spend an hour or two a week working on security as a skunk-works project.[7] Demonstrate a Linux firewall, Web server, or VPN. Show how easy it is to update Linux software when patches come in, to use ssh and gpg, to crack most passwords, or attack a Wi-Fi wireless network. Do scans of your network from your home system (using nmap with the -O flag) to show how open your network is. Install Snort and PortSentry outside of your firewall (if any) to show how often your network is attacked.

Make a point of talking with your colleagues to get detailed accounts of problems that you can then relay to your management. Have a good consultant or other trusted outside source do a security audit of your company and recommend improvements. Giving up leads to procrastination, and procrastination results in compromised systems. That is the dark side of The Force. Never give up. Never surrender.[8]

Misplaced priorities can also mean using Microsoft because "We are a Microsoft shop," disregarding that it may not have sufficient security for servers accessible from the Internet.

SIN SIX: STALE AND UNNECESSARY ACCOUNTS

As discussed before, each account is a possible entry point into the system. A stale account's password will not be changed, thereby leaving a hole. If the account has data that needs to be reassigned, disable the account by putting a * or !! in the ex-user's password field (after the first colon) in the /etc/passwd file. This disables logging in via that account because no password encrypts into either of these values and shadow password-enabled code understands these sequences. Get things cleaned up as soon as possible. Make sure that no set-UID or set-GID programs or publicly readable or writable files containing confidential data remain in that account.

Issuing the following code

```
chmod 0 /home/someone
find / -user someone -ls
```

is a good start. Note that the user may have a mailbox, files in the print spool directory, accounts in various applications, etc. that will need to be attended to.

Some of the services you removed (while correcting an earlier sin) have accounts in the /etc/passwd file. When you remove that service, make sure that the /etc/passwd account also is removed or disabled. Some of the notables are FTP, NFS, uucp, mail, gopher, and news. If you do not need them, get rid of them.

SIN SEVEN: PROCRASTINATION

In many reports of intrusions the sysadmins say, "I meant to install... IP Tables... TCP Wrappers... a newer version of... a firewall... turn off NFS and portmap... stop using PHP..." Clearly they knew, at least vaguely, what had to be done but delayed until it was too late.

Sure, you have more responsibilities than time, but consider setting aside an hour twice a week to upgrade security. Those hours may come with bag lunches at

TIP

When a user will no longer be using the system, be sure to remove his or her account from the system quickly.

your desk, but that beats a cot in your office so that you can work around the clock for a week recovering from a compromise. Sadly, I know of one company where they did bring in those cots for a number of engineers during a weeks-long recovery project following a breach. Worse, they procrastinated on deciding to build a firewall until after this event. Q

ACKNOWLEDGMENTS

This article is based on *RealWorld Linux Security: Intrusion, Detection, Prevention, and Recovery*, second edition, by Bob Toxen (Prentice Hall PTR, 2003, ISBN 0130464562); chapter 2, section 2, "The Seven Most Deadly Sins."

Thanks to Prentice Hall PTR for granting permission to use material from the book in this article. Thanks to Larry Gee, a very talented programmer, for co-authoring this section of the book.

REFERENCES

1. NFS consists of these daemons and a few more, including: rpc.nfsd, rpc.mountd, portmap, rpc.lockd, rpc.statd, rpc.rquotad, and automounter, scattered among a number of startup scripts. A cracker process can lie to portmap and masquerade as a legitimate server. NFS has had plenty of security bugs in the past, and its design prevents it from being made secure in many configurations.
2. Most recent vulnerabilities are not directly exploitable remotely on most systems. This means that most systems are not at risk for remote attack from the Internet. Many of the vulnerabilities may be taken advantage of by someone with a regular shell account on the system. Others are in programs that most people do not use and that are not set-UID or set-GID and thus are not a threat.

   This is different from most Windows vulnerabilities where almost every client system or server using that major version of Windows is vulnerable to remote attack over the Internet and thus to complete control by crackers. We observe that most Windows vulnerabilities affect all Windows versions released in the past four years, including Vista. We have recently seen Vista included with past versions of Windows for several remote "root" vulnerabilities.
3. Subscribe to Bugtraq by sending e-mail to bugtraq-digest-subscribe@securityfocus.com with empty subject and content. Subscribe to X-Force's Alert by logging on to https://atla-mm1.iss.net/mailman/listinfo/alert.
4. If you are doing only anonymous FTP, your password is normally your e-mail address. Unless you are a government researcher at Groom Lake (Area 51) and you do not want to acknowledge the existence of such a facility, then generally you have nothing to worry about.
5. Most of the information on Perl presented here is from Kurt Seifried's writings.
6. Fidelity Investments, which manages $900 billion of its customers' money, did not follow this advice. In May 2002, it was reported that by changing the digits in the URL of the page displaying his statement—a three-digit number—a client saw other clients' statements.
7. A skunk-works project is one done in secret without management approval or knowledge.
8. Thanks, *Galaxy Quest*.

TIP

The Linux 2.6 kernel prior to 2.6.17.4 has a nasty local root vulnerability where anyone with a shell account, possibly via ssh or abusing a Web server CGI program, can make himself or herself root. See CVE-2006-2451.

Are any of your systems vulnerable to this right now? I thought so.

A partial fix is to issue the command:

```
chmod 700 /etc/cron*/.
```

A better solution is to write a kernel-loadable module to prevent use of the prctl() system call by other than root.

Of course, the only full solution is to upgrade your kernel. If the system is at a remote office or colocation facility where there are no experienced sysadmins, then good luck if the new kernel does not boot.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BOB TOXEN is a computer and network security consultant with 33 years of Unix experience and 12 years of Linux experience. He was one of the 162 developers of Berkeley Unix and one of the four creators of Silicon Graphics' Unix. He has been an advisor to the George. W. Bush administration on computer issues at the five principal intelligence agencies.

Toward a Commodity Enterprise Middleware

JOHN O'HARA, JPMORGAN

AMQP (Advanced Message Queuing Protocol) was born out of my own experience and frustrations in developing front- and back-office processing systems at investment banks. It seemed to me that we were living in integration Groundhog Day—the same problems of connecting systems together would crop up with depressing regularity. Each time the same discussions about which products to use would happen, and each time the architecture of some system would be curtailed to allow for the fact that the chosen middleware was reassuringly expensive.

From 1996 through to 2003 I was waiting for the solution to this obvious requirement to materialize as a standard, and thereby become a commodity. But that failed to happen, and I grew tired of waiting.

Consequently, AMQP was created and had its first mission-critical deployment to production in mid-2006. That project paid for itself with its first deployment, serves 2,000 users, and processes 300 million messages per day.

This article sets out the motivations, capabilities, and credentials of AMQP and offers it as a practical solution for a standards-based messaging infrastructure.

AMQP is a binary wire protocol and well-defined set of behaviors for transmitting application messages between systems using a combination of store-and-forward, publish-and-subscribe, and other techniques.[1] I use the term *application messages* to distinguish AMQP from instant messaging or other forms of end-user messaging. AMQP addresses the scenario where there is likely to be some economic impact if a message is lost, does not arrive in a timely manner, or is improperly processed.

The protocol is designed to be usable from different programming environments, operating systems, and hardware devices, as well as making high-performance implementations possible on various network transports including TCP, SCTP (Stream Control Transmission Protocol), and InfiniBand.

THE NEED FOR A STANDARD

Every major investment bank on Wall Street has at some point built its own messaging middleware. Many have either faded away or spun off to become commercial proprietary solutions.

Why do they build their own middleware? The financial services industry has some of the most demanding needs for messaging both in guaranteed delivery and in publish-subscribe. Demands often exceed the capabilities of currently available software, and there is no shortage of technology expertise in banks. Building one's own middleware is therefore a credible approach.

Banks are looking for high-performance service buses from which to hang their system architectures. Web services are not fitting the bill because they are too compute- and bandwidth-intensive per unit of work.

The growth of automated trading is also igniting inter-


Can AMQP enable a new era in messaging middleware?

Toward a Commodity Enterprise Middleware

est in improving middleware. Banks are still pushing the envelope with market data events exceeding 500,000 per second at source, according to the Options Price Reporting Authority. Processing that flood of information and executing timely business transactions off the back of it is challenging. Market data volumes exacerbate transaction-processing volumes.

Given the clear need, why do many internal efforts not endure? Despite their technical abilities, banks are not software houses; messaging middleware is complex software, and it is difficult for a bank to focus the level of intellect and talent on the problem for a long time.

Banks have managed to work together in creating open technical standards where such standards are absolutely necessary to doing business; the FIX (Financial Information Exchange) protocol, FAST (FIX Adapted for Streaming), FpML (Financial products Markup Language), and SWIFT (Society for Worldwide Interbank Financial Telecommunication) are all good examples.

In 2003 I embarked on a quest to standardize MOM (message-oriented middleware) technology *outside* my firm, so we could pick it up inside the firm and use it between firms.

MAKING IT HAPPEN

This had to be an industry initiative. Home-grown middleware could not thrive in the small market available within a host organization, even the largest host.

It is also notable that pervasive networking standards such as Ethernet, the Internet Protocol, e-mail, and the Web share some traits. They are all royalty-free and unencumbered by patents, they are all publicly specified, and they all shipped with a useful early implementation for free. The combination of freedom and usefulness drives their adoption when predicated on fitness for purpose.

To succeed, AMQP needed to adopt these same characteristics:

- It needed to be a fully defined, open, royalty-free, unpatented specification to enable anyone to implement a compatible service. Furthermore, the standard specification had to be clearly separate from the implementations; otherwise, it would not be a fair market for commercial entities to enter. AMQP had to be appealing for commercial implementation and exploitation or it would not succeed.
- AMQP needed to have real implementations of the specification; otherwise, the specification would not be immediately useful or interesting to front-line developers with pressing needs. Ideally, it should have more than one implementation to qualify as a potential IETF draft standard.[2] So, there are real implementations you can run today (as detailed later in the article).
- AMQP software had to be proven in live systems. Middleware is a critical piece of any system and must be trusted. That trust has to be earned. To this extent, it was clear we would have to deploy an implementation in a high-profile, mission-critical application to assuage the fears of other early adopters. So, a combination of OpenAMQ and Qpid are live at JPMorgan, supporting 2,000 users on five continents and processing 300 million messages per day.
- Finally, and most importantly, AMQP needed to be a collective effort. Openness to partnership and the ideas of others had to be there from the beginning. To this end, we carefully selected a partner to co-develop the specification and implement the software. We chose iMatix, a boutique European development house that had clearly demonstrated a commitment to open source and sound ethics, and had a strong engineering background and excellent writing abilities.

Because the project was sponsored by a bank, it also had to "wash its own face," as they say. This was *not* a research project. Through sheer good luck, there was a need to refresh some large systems with very specific requirements. This provided a tangible return for AMQP investment, so I was able to convince a forward-looking CIO that AMQP was the way to go.

THE AMQP WORKING GROUP

When the shape of AMQP had been worked out between JPMorgan and iMatix, and the basics of the specification forged in the heat of an initial implementation, the time was right to extend the partnership and encourage others to bring their talents to the specification and share ownership of AMQP's future.

STRONG GOVERNANCE

The heart of openness and trust in any group effort is effective governance. Expanding the group required a new contractual framework and a plan for the end game where AMQP could become a standard. Red Hat took the

lead in establishing the legal framework for the standard; it, too, understood the issues in managing open intellectual property. The key part of doing this is to ensure that everyone contributing has the authority to do so and that there is a paper trail from every potential owner of IP through to the group effort, and that the intent to share is clear even in draft revisions of specifications. The result was a contract that clearly committed the members of the working group to promote unrestricted open middleware through AMQP.

The members of the working group have granted licenses to the necessary parts of their patent portfolios to anyone who wants to implement AMQP. You can see the license grant in the specification itself.

This level of contribution shows the commitment of the group to open middleware. The AMQP Working Group's Web site is http://www.amqp.org.

USER DRIVEN

The AMQP Working Group is quite unique in technology standards work because of the heavy involvement of users. JPMorgan, Credit Suisse, TWIST, and to some degree Cisco are more end users than developers. This balance leads to a group of people interested in solving the problem, not pandering to technology agendas or product agendas.

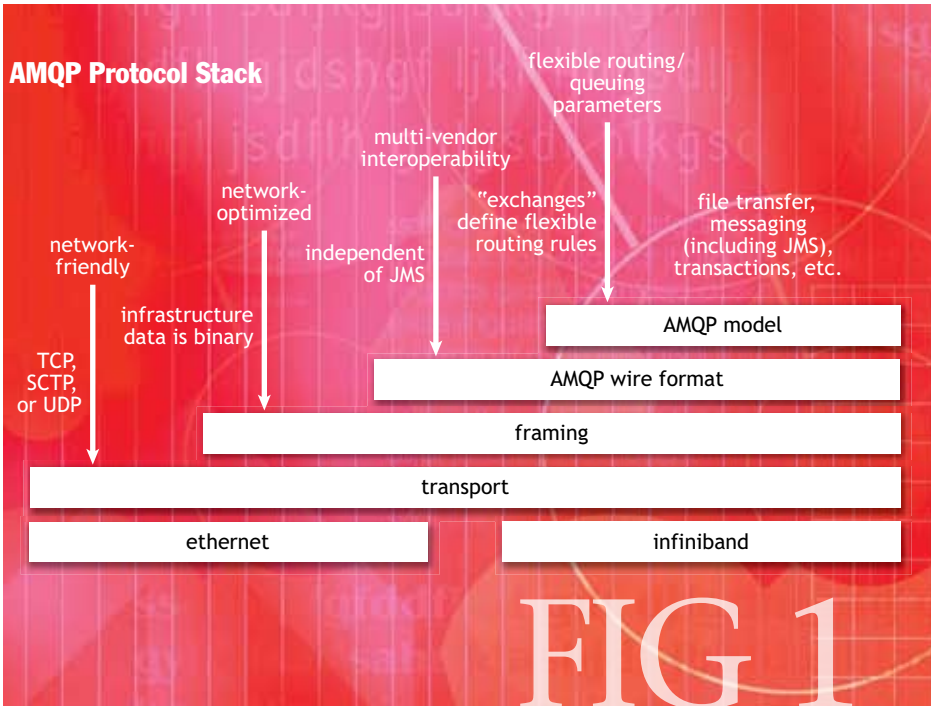
ARCHITECTURE

From the beginning, AMQP's design objective was to define enough MOM semantics (see figure 1) to meet the needs of most commercial computing systems and to do so in an efficient manner that could ultimately be embedded into the network infrastructure. It's not just for banks.

AMQP encompasses the domains of store-and-forward messaging, publish-and-subscribe messaging, and file transfer. It incorporates common patterns to ease the traversal of firewalls while retaining security, and to permit network QoS. To ease adoption and migration, AMQP is also designed to encompass JMS (Java Message Service) semantics. JMS is a hugely popular API for Java programmers and cannot be ignored. AMQP goes further, however, and includes additional semantics not found in JMS that members of the working group have found useful in delivering large, robust systems over the decades. Interestingly, AMQP does not itself specify the API a developer uses, though it is likely that will happen in the future.

An example feature not found in JMS is AMQP's Mandatory Delivery Mode, in which a client can use AMQP to request services from a pool of servers connected to AMQP broker queues. The AMQP broker can load-balance requests among the services subscribed to a request queue, and the number of processes providing a service can dynamically grow and shrink with no impact on the client(s). If the service pool shrinks to zero, however, the client can be informed by AMQP using the Mandatory Delivery Mode since that may be an operational error for the application.

AMQP also specifies a small wire-level type system for message properties, enabling them to be read efficiently by many programming languages, as well as by the MOM servers themselves for filtering and routing purposes. Thus, not only can a Python client read headers set in Java servers, but different vendors can relay messages between their implementations seamlessly. The type system, however, suffers the usual problem in that object



Toward a Commodity Enterprise Middleware

types cannot be transported in the headers; what would a Cobol program do with a Smalltalk object?

Finally, AMQP draws heavily on the heritage of IETF open standards. It tries not to reinvent existing concepts. Early versions of the AMQP wire protocol were influenced by SMTP,[3] MIME,[4] HTTP-NG,[5] NFSv4,[6] SCTP,[7] BEEP,[8] and the writings of Marshal Rose,[9] an IETF veteran.

MAIN FEATURES

AMQP is split into two main areas: transport model and queuing model. AMQP is unusual in that it thoroughly specifies the semantics of the services it provides within the queuing model; since applications have a very intimate relationship with their middleware, this needs to be well defined or interoperability cannot be achieved. In this respect, AMQP semantics are more tightly defined than JMS semantics.

As stated, AMQP's transport is a binary protocol using network byte ordering. We wanted to make it easy to embed AMQP inside the ASICs (application-specific integrated circuits) of network elements, by design. With free tools such as Wireshark, it is not necessary to use XML for technical infrastructure layers that only specialists will see. XML has been used in the likes of BEEP and XMPP: in the case of BEEP it complicates the protocol; in the case of XMPP it is limited to being carried on a stream-oriented transport. AMQP aims to be high performance and flexible, to be hardware friendly rather than human friendly. The protocol *specification* itself, however, is written in XML so implementers can code-generate large portions of their implementations; this makes it easier for vendors to support the technology.

The transport model itself can reuse different underlying transports. The first is TCP/IP, but by adopting SCTP, we can obtain better parallelism for messages (SCTP removes the byte-stream head-of-line blocking problem imposed by TCP). There are also planned mappings to UDP to support AMQP over multicast, and bindings to InfiniBand are planned. InfiniBand's performance is generating a lot of interest at banks, and AMQP would make it very accessible to developers.

TCP/IP, however, is expected to be the default choice for most end users for best interoperability. With these options emerging, it is important for AMQP to establish a useful functional default set of capabilities that all implementations must adhere to or suffer the lowest-common-denominator problem that plagued protocols such as early versions of NFS (many servers did not implement file locking). Hopefully, a compliance testing kit will address this issue.

MESSAGES

Messages in AMQP are self-contained and long-lived, and their content is immutable and opaque. The content of messages is essentially unlimited in size; 4GB messages are supported just as easily as 4KB messages. Messages have headers that AMQP can read and use to help in routing.

You can liken this to a postal service: a message is the envelope, the headers are information written on the envelope and visible to the mail carrier, who may add various postmarks to the envelope to help deliver the message. The valuable content is within the envelope, hidden from and not modified by the carrier. The analogy holds quite well, except that it is possible for AMQP to make unlimited copies of the messages to deliver if required.

QUEUES

Queues are the core concept in AMQP. Every message *always* ends up in a queue, even if it is an in-memory private queue feeding a client directly. To extend the postal analogy, queues are mailboxes at the final destination or intermediate holding areas in the sorting office.

Queues can store messages in memory or on disk. They can search and reorder messages, and they may participate in transactions. The administrator can configure the service levels they expect from the queues with regard to latency, durability, availability, etc. These are all aspects of implementation and not defined by AMQP. This is one way commercial implementations can differentiate themselves while remaining AMQP-compliant and interoperable.

EXCHANGES

Exchanges are the delivery service for messages. In the postal analogy, exchanges provide sorting and delivery services. In the AMQP model, selecting a different carrier is how *different ways of delivering the message* are selected. The exchange used by a publish operation determines if the delivery will be direct or publish-and-subscribe, for

example. The exchange concept is how AMQP brings together and abstracts different middleware delivery models. It is also the main extension point in the protocol.

A client chooses the exchange used to deliver each message as it is published. The exchange looks at the information in the headers of a message and selects where they should be transferred to. This is how AMQP brings the various messaging idioms together—clients can select which exchange should route their messages.

Several exchanges must be supported by a compliant AMQP implementation:

- The direct exchange will queue a message directly at a single queue, choosing the queue on the basis of the "routing key" header in the message and matching it by name. This is how a letter carrier delivers a message to a postal address.
- The topic exchange will copy and queue the message to all clients that have expressed an interest based on a rapid pattern match with the routing key header. You can think of the routing key as an address, but it is a more abstract concept useful to several types of routing.
- The headers exchange will examine all the headers in a message, evaluating them against query predicates provided by interested clients using those predicates to select the final queues, copying the message as necessary.

Throughout this process, exchanges never store messages, but they do retain binding parameters supplied to them by the clients using them. These bindings are the arguments to the exchange routing functions that enable the selection of one or more queues.

BINDINGS

The arguments supplied to exchanges to enable the routing of messages are known as bindings (see figure 2). Bindings vary depending on the nature of the exchange; the direct exchange requires less binding information than the headers exchange. Notably, it is not always clear which entity should provide the binding information for a particular messaging interaction. In the direct exchange, the sender is providing the association between a routing key and the desired destination queue. This is the origin of the "destination" addressing idiom so common to JMS and other queuing products.

In the topic exchange, it is the receiving client that provides the binding information, specifying that when the topic exchange sees a message that matches any given client(s') binding(s), the message should be delivered to all of them.

AMQP has no concept of a "destination," since it does not make sense for consumer-driven messaging. It would limit its abstract routing capabilities. The concept of bindings and the convention of using a routing key as the default addressing information overcome the artificial divisions that have existed in many messaging products.

IMPLEMENTATIONS

A standard is nothing without implementations. AMQP has several available now. The first implementation is iMatix OpenAMQ (http://www.openamq.org), which is a C implementation of the server in production use at JPMorgan.

Apache's Qpid project has entered incubation (http://incubator.apache.org/projects/qpid.html). It will be Apache's multilanguage implementation of AMQP, with servers available in C++ and Java, and clients available in C, C++, Java JMS, Python, and Ruby on Rails, with more to follow. Qpid has a very active development community and is making rapid progress toward its M2 release. Also, Qpid is being used as the basis for several commercial offerings, notably from IONA and Red Hat.

Most recently, and intriguingly, Rabbit Technologies has developed an implementation on the Erlang/OTP (Open Telecom Platform), building on that language's strong support for parallelism and networking.

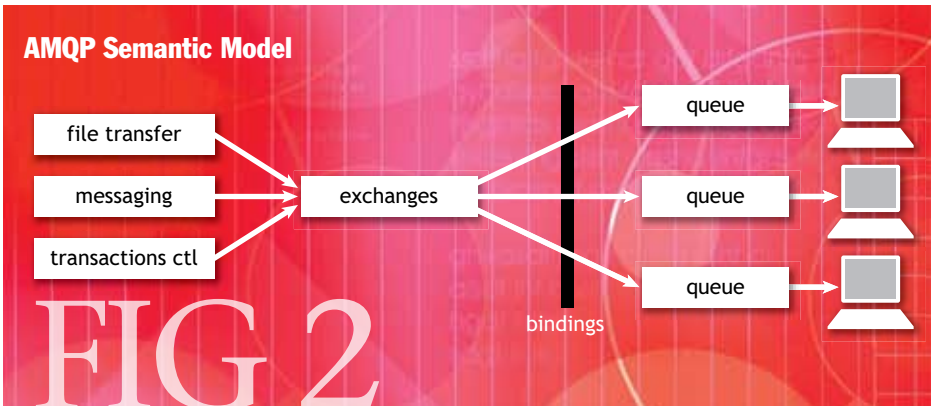


FIG 2

Of course, you should be able to mix and match client and server components from any vendor product—Qpid's Java client talking to RabbitMQ's Erlang server, for example.

The best way to learn more about AMQP is to visit http://www.amqp.org, where you can download the specification, try one of the free implementations,

Toward a Commodity Enterprise Middleware

or even build your own. Both the protocol working group and the implementation groups are open to review and feedback; we want this protocol to be useful, successful, and inclusive.

A TRANSPORT FOR OTHER STANDARDS

In addition to being used to support organizations' internal messaging needs, AMQP is useful as a standard transport for other standards; after all, open standards should be built using other open standards.

Many business messaging standards describe business transactions, but there are only a relatively small number of transport options. HTTP is often used simply because it is perceived to be generally available, and firewalls are often configured to let HTTP traffic pass. HTTP offers only basic push messaging semantics, however; it does not enable any kind of call-back or event notification, and it requires the applications that use it to build their own message store, reliability, and access control facilities on top of it. EDI AS2 and WS-RM are examples of standards layered on HTTP; but these describe only message transport, not the semantics of how a message is delivered to an application. This leaves the developer with a partial solution and more problems to solve.

By offering AMQP as a standard for business messaging, we can make the full richness of messaging middleware available to other standards. Using AMQP servers, we can remove the burden of providing availability and reliability from end-user applications, making them simpler, cheaper, and more functional. The only requirement is that an AMQP server is run and that TCP/IP port 5672 be opened in the firewall. Given that any commercial activity involves lawyers and contracts, opening a port is a small price to pay to gain rich messaging functionality.

The TWIST standards organization, which promotes standards around financial supply chain management, is a member of the AMQP working group and promotes AMQP as its preferred standard for transporting business messages.

The FIX community is also looking at AMQP as a possible transport layer for FIX 5. FIX is a popular trading protocol that has recently been extended to support efficient delivery of market data; the current session layer, however, cannot provide publish-subscribe or scalable guaranteed delivery. AMQP offers FIX users the chance to get the features they need, while remaining open.

SOA WITH AMQP

SOA (service-oriented architecture) is a technique for building large systems out of highly cohesive services loosely coupled to provide business processes. SOA is not a new concept. It was well known in the mid-'90s and earlier, but in its latest incarnation it is being pitched as *Web service*-oriented architecture. The architectural pattern needs neither HTTP nor XML.

One of the most fundamental parts of SOA is the communications mechanism that links the services. The term EMB (enterprise message bus) is closely linked to SOA.

Traditional deployments of EMBs have used proprietary technology, but enterprises would rather have standards-based solutions that are open, as well as the ability to choose between and switch suppliers, and the improvements that competition brings. AMQP, therefore, represents an ideal choice for an EMB. It allows a clean migration away from proprietary protocols and provides an avenue into other standards, including Web services.

Web services has four basic parts: service description, XML message content, service discovery, and transport. The transport is commonly presumed to be HTTP, but it does not have to be. Enterprises often use XML over messaging middleware as the transport for all the benefits that brings. Having done this, enterprises find they have created the problem they wanted to avoid: running an open architecture over a proprietary transport. Combining Web services with AMQP as a transport gives the richness an enterprise needs with the openness it craves in its core architecture.

Use of AMQP in SOA is just beginning, and you don't need anything other than AMQP to do it. I am already involved in a project to migrate an existing mission-critical EMB from a proprietary middleware to AMQP, so we can cost-effectively scale the bus to many more systems.

CONNECTING TO LEGACY MIDDLEWARE

AMQP is a complete middleware protocol. It is not a lowest-common-denominator solution, and the only political design constraint is its explicit support for JMS semantics. Obviously, software that implements the AMQP specification will be able to interoperate, even where that software is from different suppliers. That means that the JMS client from product A would be able to talk to the C++ server

from product B and send messages to a Ruby client in product C. Of course, in the real world there will be some teething difficulties, and the AMQP working group is starting to focus on how to create and manage a protocol compliance test suite to mitigate that risk.

There are many middleware products, however, each with its internal architecture. The AMQP Working Group encourages middleware vendors to implement AMQP in their products, but it will be nontrivial to get a good semantic match, and some vendors may be reluctant to support interoperability for commercial reasons.

In the meantime, there will be a need for bridging from the installed base of proprietary products. The easiest way to do this may be to use one of the many commercial or open source EAI (enterprise application integration) packages, but we expect several AMQP products to include native bridges to the most common proprietary middleware soon.

ADOPTING AMQP

The most natural way for an organization to adopt AMQP is to deploy it opportunistically or as part of a strategy to move to standards-based EMB. This approach can enable an organization to benefit from competitively priced or open source solutions and gain experience with the protocol and products that support it. This is the approach we took: using AMQP in isolated systems and then branching into core EMB systems.

Over the course of a few years, much of your messaging may become AMQP-enabled by following this approach. Your current middleware supplier may adopt AMQP and enable your move to a standards-based model in that way. It is likely that AMQP will eventually be provided as a core service by your network infrastructure, in hardware.

On the other hand, if your company is embarking on SOA (or other bus-based architecture), we suggest that you seriously consider an en-masse deployment of AMQP as the backbone of SOA. Doing so may position you to benefit from competition among suppliers of AMQP-compatible middleware so you may achieve the levels of support and qualities of service you need from a suitable supplier. Deploying AMQP may also mitigate the worrisome issue of vendor lock-in or supplier failure for the critical bus component of an SOA, which is a long-term investment for your company. This same thinking applies where open protocols for e-business are being deployed between trading partners over the Internet or leased lines. Of course, whatever strategy you choose is yours alone and must be determined by your circumstances.

CONCLUSION

After two decades of proprietary messaging middleware, a credible standards-based alternative exists at last. The AMQP Working Group is rapidly evolving the protocol and hopes to reach version 1.0 during 2008, but implementations are available today that are both useful and proven in real-world deployments.

AMQP lets more applications be enterprise-grade without the costs associated with that label. It provides a capable messaging backbone that can be a springboard for new innovations and new kinds of applications. Q

REFERENCES

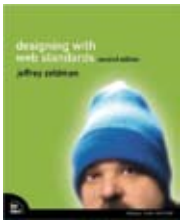
1. O'Hara, R.J., Hintjens, P., Greig, R., Sim, G., et al. 2006. AMQP Advanced Message Queuing Protocol Version 0.8; http://www.amqp.org/tikiwiki/tiki-index.php?page=Download.
2. Bradner, S. 1996. IETF Standards Process, RFC 2026.
3. Postel, J., Klensin, J. 1982/2001. Simple Message Transfer Protocol, RFC 821 / 2821.
4. Freed, N., Borenstein, N. 1996. Multipurpose Internet Mail Extensions (MIME), RFC 2045.
5. Janssen, B. 1998. Binary Wire Protocol for HTTP-ng; http://www.w3.org/Protocols/HTTP-NG/.
6. Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., Noveck, D. 2003. Network File System (NFS) version 4 Protocol, RFC 3530.
7. Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer. H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., Paxson, V. 2000. Stream Control Transmission Protocol, RFC 2960.
8. Rose, M. 2001. The Blocks Extensible Exchange Protocol Core, RFC 3080.
9. Rose, M. 2002. "Introduction: Application Protocol Design." *BEEP: The Definitive Guide*. O'Reilly.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

R. JOHN O'HARA is a senior architect and distinguished engineer at JPMorgan. He drove the creation of AMQP and was an early contributor to the FpML financial markets messaging standard. He is chairman of the AMQP Working Group. His interests include data center virtualization, creative uses of open source for commercial data processing, event and data replication middleware, and making software architecture "simple and intuitive." O'Hara holds an honors degree in electronic engineering and computer science from Aston University, England.

Designing with Web Standards (second edition)
Jeffrey Zeldman, Peachpit Press, 2006, $44.99,
ISBN: 0321385551.



Jeffrey Zeldman has written an excellent update of his critically acclaimed book. This second edition covers the changes to Web browsers, Web development techniques, and the Web community's acceptance of Web standards in the four years since the first edition. One of the most significant changes has been improved support for CSS (cascading style sheets) layout among all browsers.

Part 1 of the book addresses the importance of Web standards. Anyone who is already convinced of the value of using Web standards, and who doesn't need the information to convince others, can skip part 1 and go directly to part 2, the how-to of designing with Web standards.

Part 2 begins with a discussion about modern markup. XHTML is a reformulation of HTML using XML. The next chapter covers simple rules for converting from HTML to XHTML. The following chapter begins by walking the reader through an example of building a Web page using a hybrid layout. The example demonstrates how to use CSS to incorporate accessibility into the page, and by extension into the Web site as a whole.

Development of the example Web page is interrupted to cover CSS basics, after which Zeldman picks up with the example, using CSS to display the Web page without having to make changes to the page code.

A discussion of typography follows; it controls how text looks on the screen. Zeldman debunks many of the myths surrounding Web accessibility and provides tips for making Web sites more accessible. The final chapter brings together concepts learned earlier in the book and adds a few new techniques to create a CSS design. In the first edition of the book, many of the techniques were cutting edge. In the years between editions, many of these techniques have become part of Web development best practices.

I highly recommend this book for all Web professionals. Those just beginning their careers can learn the right way to build standards-compliant Web sites. Those who have been in the field for decades can learn current best practices that will make their jobs easier, while still meeting the requirements of their clients. —*Will Wallace*

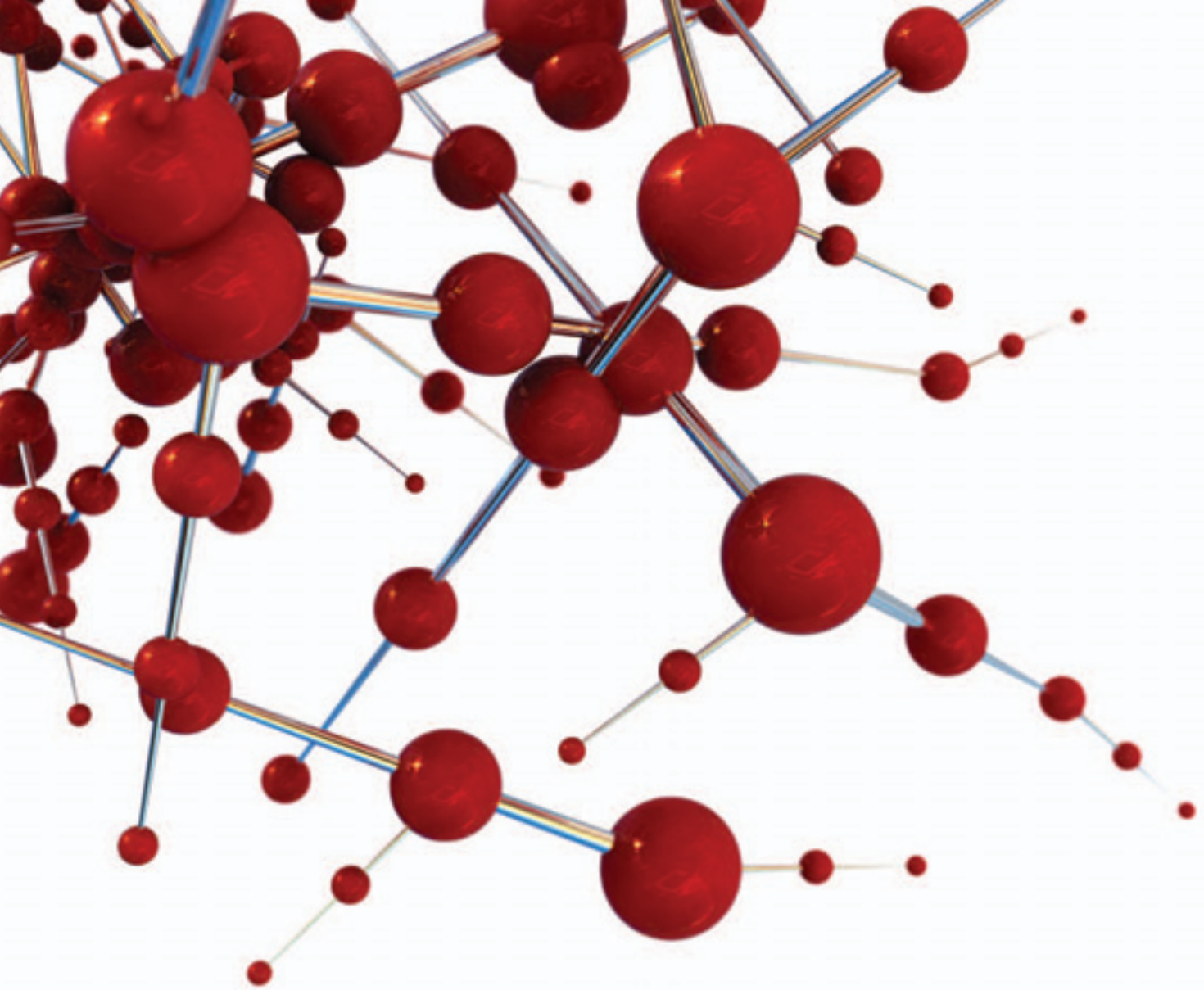
Expert VB 2005 Business Objects (second edition)
Rockford Lhotka, Apress, 2006, $59.99,
ISBN: 1590596315.

If you are looking for a good .NET companion framework, you should seriously consider CSLA (component-based scalable logical architecture). Rockford Lhotka designed it to ease the development of business objects that must be reused and deployed in a variety of distributed scenarios—for example, two-tier architectures with desktop interfaces or three-tier architectures with Web interfaces. The result is a framework that provides built-in support for multilevel undo/redo, business rules, two-way data binding for both Windows and Web forms, object persistence, custom authentication, and integrated authorization.

This book reports on the CSLA framework. Its 12 chapters are well organized and easy to follow for the average .NET programmer; a few sections delve into some .NET intricacies that are necessary for implementing two-way data binding, for example, but Lhotka has managed to take the reader from the essential concepts to the intricacies so that they are easy to understand. Furthermore, readers who are not interested in the details may skip these sections safely.

The first chapter is an essay on distributed architectures in which the emphasis is on the distinction between logical and physical models and the mappings between them; this chapter explains the motivation for a framework such as CSLA, whose design goals and main features are presented in chapter 2. Chapters 3 through 5 deal with the implementation of the framework itself. The rest of the book reports on using the framework to implement a small, but not trivial, project management system to which the user can have access through a typical desktop application, Web page, or SOAP.

I must confess that I enjoyed evaluating this book, and I definitely recommend it to programmers who develop typical business applications and wish to take the .NET framework a step further. I also think that it is a valuable resource for information technology students since Lhotka's style of writing is didactic and the design of the framework is quite clean. For readers who prefer C#, another version is available. —*Rafael Corchuelo*



MAY

OSBC (Open Source Business Conference)
May 22-23, 2007
San Francisco, California
http://www.osbc.com/live/13/

O'Reilly Where 2.0 Conference
May 29-30, 2007
San Jose, California
http://conferences.oreillynet.com/where2007/

JUNE

Tech·Ed
June 4-8, 2007
Orlando, Florida
http://www.microsoft.com/events/teched2007/default.mspx

Apple WWDC (Worldwide Developers Conference)
June 11-15, 2007
San Francisco, California
http://developer.apple.com/wwdc/

Workshop on Experimental Computer Science
June 13-14, 2007
San Diego, California
http://www.cs.huji.ac.il/~feit/exp/

Usenix Annual Technical Conference
June 17-22, 2007
Santa Clara, California
http://www.usenix.org/events/usenix07/

Better Software Conference and Expo
June 18-21, 2007
Las Vegas, Nevada
http://www.sqe.com/bettersoftwareconf/

BREW
June 20-22, 2007
San Diego, California
http://brew.qualcomm.com/brew/brew_2007/

JULY

Web Design World
July 8-11, 2007
Seattle, Washington
http://www.ftponline.com/conferences/webdesignworld/2007/seattle/

IEEE International Conference on Web Services
July 9-13, 2007
Salt Lake City, Utah
http://conferences.computer.org/icws/2007/

CIO & CSO Business Continuity Forum
July 17-18, 2007
New York, New York
http://public.cxo.com/conferences/index.html?conferenceID=6

Networkers at Cisco Live
July 22-26, 2007
Anaheim, California
http://www.cisco.com/web/learning/le21/le34/networkers/nw07

OMG's BPM Think Tank
July 23-25, 2007
Burlingame, California
http://www.omg.org/news/meetings/ThinkTank/

O'Reilly Open Source Convention
July 23-27, 2007
Portland, Oregon
http://conferences.oreillynet.com/os2007/

AUGUST

SIGGRAPH
August 5-9, 2007
San Diego, California
http://www.siggraph.org/s2007/index.html

LinuxWorld
August 6-9, 2007
San Francisco, California
http://www.linuxworldexpo.com/live/12/

Usenix Security Symposium
August 6-10, 2007
Boston, Massachusetts
http://www.usenix.org/events/sec07/

Agile Conference
August 13-17, 2007
Washington, DC
http://www.agile2007.com/

SEPTEMBER

Embedded Systems Conference
September 18-21, 2007
Boston, Massachusetts
http://www.embedded.com/esc/boston/

Gartner Master Data Management Summit
September 19-21, 2007
Hollywood, Florida
http://www.gartner.com/it/page.jsp?id=501889&tab=overview

The WiMAX World Conference
September 25-27, 2007
Chicago, Illinois
http://www.wimaxworld.com/

To announce an event, E-MAIL QUEUE-ED@ACM.ORG OR FAX +1-212-944-1318

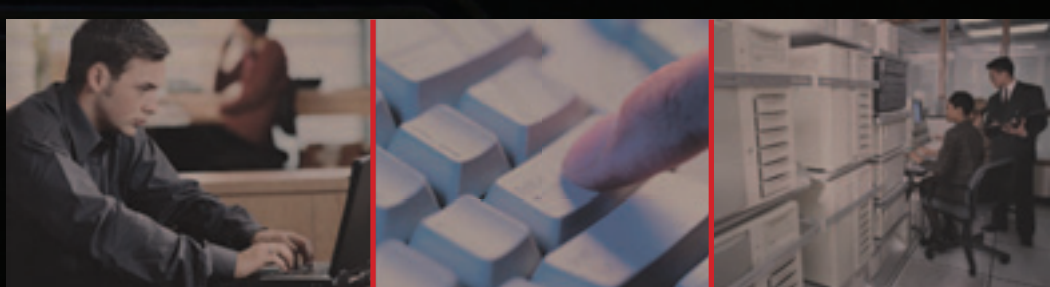
THE ACM A. M. TURING AWARD

THE ASSOCIATION FOR COMPUTING MACHINERY AND INTEL WOULD LIKE TO EXTEND WARM WISHES AND CONGRATULATIONS ON BEHALF OF THE ENTIRE COMPUTING COMMUNITY TO **FRAN ALLEN** ON RECEIVING THIS YEAR'S ACM A.M. TURING AWARD.

BY THE COMMUNITY...

FROM THE COMMUNITY...

FOR THE COMMUNITY...



"Fran Allen's work on the Parallel TRANslation (PTRAN) project built on her earlier work on program optimization. Over the years, this foundation has enabled the advance of programming-productivity based on the co-evolution of higher level programming language and optimization technologies. It is particularly timely that this award comes as parallel computing is becoming an element of the most pervasive of computing platforms—laptop and desktop personal computers—and the opportunities for new and important contributions to parallel programming and efficient implementation abound."

Andrew A. Chien
Vice President, Corporate Technology Group
Director, Intel Research

Financial support for the ACM A. M. Turing Award is provided by Intel Corporation.

CLASSIFIED

Qualcomm CDMA Technologies

Digital ASIC Verification Engineering Director

Direct teams of engineers driven by verification leads, and define chip level verification strategies. Drive projects through all phases until all goals are met/exceeded. Requirements: A minimum of 12-15 years experience in systems, digital ASIC design and verification, with at least 5-7 years focused on functional verification, 3-5 years experience in design, and 5+ years in engineering management. Strong operational experience in managing digital ASIC design verification projects from start to finish. Strong track record of execution.

Epic Systems Corporation

Software Developer

Epic builds multi-tier enterprise software for healthcare organizations using a variety of technologies. Your goal is to manage large amounts of data with sub-second response times and rock-solid stability. Working on a small team, you'll participate in all aspects of the development process, from meeting customers and design through implementation, quality assurance, and delivery. We bring enhancements to the market quickly, so you will see your hard work make a difference. To qualify, you must have a BS or MS in CS, Math or a related field, and a track record of academic excellence.

Business.com

Information Retrieval Engineer

Work as part of an elite team head quartered in Silicon Valley to design and develop next generation search and information retrieval applications.

Responsibilities

- Evaluate and implement new algorithms for information retrieval and extraction, machine learning, and natural language processing
- Design and develop products and systems capable of discovering, extracting and manipulating web data for use on the Business.com site and for network partners
- Partner with counterpart in Product Management to manage the development of all Information Retrieval and Search systems

Skills/Qualifications

- Information retrieval, machine learning, and Web data mining.
- 3-5 yrs software development exp. in Java, C, or C++, RDB's
- BS in computer science (or related) required, MS preferred
- Experience with search engines is a strong plus

Please send resume or CV to: Staffing@Business.com

Continued from page 64

Debate on Wiki's accuracy has been growing since the site launched. That has been the fate of all reference works, as Diderot and Lavoisier will confirm. In any "live" and growing corpus (Wiki now has more than 6 million entries), some errors are inevitable. Facts do change, don't yer know? The problem is how to judge overall reliability from the occasional headline-grabbing "disasters" (usually malevolently planted by the disgruntled), which are uncovered and, claim the pro-Wikimites, promptly corrected.

As with our vast suppositories [sic] of software, however, the notion of "unknown bugs" and "undetected bugs" looms as a Zen demon (if I can mix my creeds). Is there an acceptable level of reliability? If so, can we assess it from sampling? Do a thousand minor Wiki typos count more than, say, an entry on Islam or a map of Africa that may unintentionally provoke violence or threats of violence? When experts disagree, should both sides be represented with balanced word counts? Can the cranks have their day on Wiki? Cranks may be tomorrow's gurus. The obvious paradox is that in the normal "look-up" situation, we seldom consult Wiki in the domains in which we are reasonably knowledgeable. Maintaining accuracy therefore calls for dedicated specialists with the time to spare for regular and thorough vetting.

The volatility of online data remains a mixed curse/blessing (see this column, April and July/August 2005, ibid.). What was a major manual effort for editors of the *Great Soviet Encyclopedia* as past heroes were "delisted" (or "unentered"?) is now a few deft clicks away. A remarkable case is reported where the Russian subscribers to the encyclopedia were co-opted into helping the State revisionists: in 1953 they were sent a page entry on the Bering Strait and asked to insert it after cutting out the same-size entry on the disgraced Lavrentiy Beria—thus preserving the pagination.[3]

Just as a thought experiment, imagine an order from up high to eliminate the entry at STALIN, Joseph. "We need exact 321-page in-situ replacement—any ideas?" Back comes the nervous editor: "We find that STALLMAN, Richard would fit alphabetically and even culturally. His FSF supports our aims in bringing down Microsoft and the other wicked capitalists who steal and sell People's software. But big problem, Boss—we are having trouble covering Stallman in less than 400 pages."

Back to Wiki's weracity, and enter Larry Sanger. As a disillusioned Wiki co-founder, his decision to set up a rival online resource deserves our careful attention. Enter Citizendium as the Wiki-killer! The choice of name is hardly an encouraging sign, yet, I say, the more references the better. Beware the man browsing one site. Although Sanger expresses concern over the errors in Wikipedia, his main beef is the underlying structure and ethos that throws doubt on its ability to ensure reliability. He calls the Wiki management community dysfunctional, invoking the crushing term "Rigid Egalitarianism." In particular, he dislikes the freedom with which anonymous Wiki contributors with unproved credentials can provide new and edit old information. Citizendium will correct this loophole by applying strict control over who does what and with transparent accountability. Just like a "real" encyclopedia, you may say. Sanger's team has much ground to make up, and I wish them well. One might add a sad note: Are we seeing another "damn good cause" afflicted with a bad dose of the "schisms"?

IF THIS BE ERROR...

Returning to the theme of reader cooperation, I offer a brief, yet apposite example of a newly exposed, half-forgotten mis-forecast in the June 24, 1974, Science section of *Time* magazine in which expert climatologists warned of global *cooling*. The next Ice Age loometh, and one geo-guru (University of Toronto, no less) was even more precise: "I don't believe that the world's present population is sustainable if there are three years like 1972 in a row."

Here are two examples of how higher marks qualify for my Doryphoric Palme d'Or (recall that a *doryphore* is "one who takes excessive delight in spotting small errors," where *excessive* and *small* remain undefined) when mistakes are published in authoritative texts.

First, Bill Bryson's *A Short History of Nearly Everything* (Broadway Books, 2003) is an excellent introduction to the natural sciences for the laid-back laity (I've given it as a prezzie to all my grandchildren), partly because Bryson is a fine writer rather than a trained scientist.

He better appreciates the hurdles for those who failed math and physics not through lack of wit but because of poor presentation and motivation. However, he states that "seven one-thousandths" is "0.007 percent" and repeats this deception by offering "six one-thousandths" as "0.006 percent." I hear the grumpy, unfair reaction: How can we believe anything this Bryson tells us?

Second, because the author/editor/commentator of *God Created the Integers* (Running Press, 2005)[4] is famed cosmologist Stephen Hawking, the reviewer, John Stillwell (*American Mathematical Monthly*, Mathematical Association of America, March 2007) can hardly resist a smirk in finding several "more or less serious errors together with other distinctly misleading statements." Hawking writes, "Riemann recognized that in spaces of nonconstant curvature bodies may move about without stretching" (page 820). This is a topic well within the author's domain of competence, yet my readers will surely spot the mistake noted by Stillwell. The word *nonconstant* should be *constant*! Is this the Orwellian peak of misspeak? For YES read NO? For FALSE read TRUE? As with my love for Nelly Moorcroft. But reversed. My declared constancy proved inconstant.

Proof that Hawking simply wrote carelessly is revealed later on the same page where Riemann himself (genuflect, genuflect) is quoted with the correct proposition (bodies can move without stretching in spaces of constant curvature). Later, Hawking goes wrong again: "Of course, space need not be flat, it need not even be of constant curvature as it must be for the sum of angles of a triangle to be constant." Wrong or very misleading, claims Stillwell. Constant curvature does not imply the sum of angles are invariant! The very sphere (idealized) most of us inhabit is constant curvature, but we all know (wake up at the back) that angle-sums vary with area. What Hawking should have said was that "zero curvature" guarantees angle-sum invariance.

JACK BE AGILE

I hope *agile* is still the in-vogue programmers' paradigmatic predicate. Writing a few months ahead of publication has always been a hazard in our fair but unfairly volatile trade. I see signs of the *nimble* overtaking the *agile*, presumably by changing lanes and ignoring the speed limits.

I know that Joshua E. Smith designed an XML-compatible language called Nimble in 1999, yet this name seemed based on *nimble* as a folksy synonym for *agile*. Nobody sings, "Jack be agile, Jack be quick," do they? But can we expect Nimble programming to become a more widely entrenched general concept crowned with the accolade Methodology? Incidentally, wordsmiths will notice that Nimble is billed not as XML-compatible or XML-conforming but as XML-conformant. Readers are invited to submit their definitions of these three terms and explain how they might differ.

TITLE THEME

The financial consultants Deloitte splash the banner,

Seenogapsinyourbusinessthinking

proving, if proof were needed, that reading undelimited "words" can be a pain. In fact, it can lead to dire ambiguity as in "man's laughter" and "manslaughter." The comic's straight man described his sex life as "infrequent," to which Henny Youngman responded, "Is that one word or two?" And how many see the connection between "atone" and "at one?" Reader prizes for similar examples.

My collapsed, self-referential headline to this column, "Alloneword," is now embedded in computer newspeak, and further borrowed for a rock band. It will be familiar to all those who have ever had to dictate or speech-spell their e-mail or Web addresses. That branch of mankind must include all my readers and, indeed, a large, ever-growing proportion of those who are wired into our Brave New World of Web. (The participle *wired* remains a quaint synonym for *connected*, even when that nirvana is achieved wirelessly.) Thus, we announce, "I'm joethejollyblogger, alloneword, at discountmousepads, alloneword, dot see-oh dot you-kay."

By the way, don't rush to register the confusing domain alloneword. It's been "took!" (http://www.alloneword.org houses the illustrated Figures of Speech by Mervyn Peake. Worth a visit).

Less worthwhile (I'm scarce able to mention it) is http://www.twitter.com. Twitterers (or twits as I prefer to call them) are a global community of underemployed addicts with sub-blog attention spans. Once registered (there don't seem to be any tests for literacy or sanity), twits can submit realtime biographical sound "bytes" describing what they claim to be doing at that very moment. On the bright side, the max allowed burst of narcissism is 140 words per twitter. On the dark side, you are invited to read what other twits are up to. My entry dated "now": "Just got out of bed. Marmite butty as per usual. Logged into twitter.com. Slashing my wrists. Bye-bye all." Q

REFERENCES

1. Shakespeare, W. *Merchant of Venice* (act 1 scene

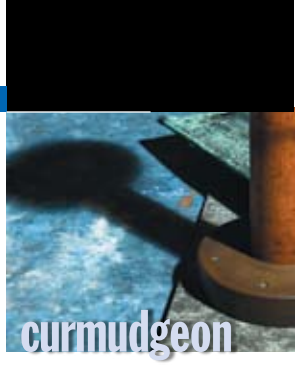
1). Incidentally, this play may cast light on the Bard's sexual preferences: "My ventures are not in one bottom trusted."

2. In my school, Latin *v*'s were pronounced as *w*'s. In German and Polish we were taught to reverse this rule—most confusing. Caesar's declaration has been much parodied: "Veni, vidi, volo in domum redire" (I came, I saw, I wanna go home) and "Veni, vidi, Visa" (I came, I saw, I shopped).
3. I was reminded of this ancient anecdote by an entry in Wiki! Strangely, I now feel less inclined to believe it. It has that too-telling-to-be-true, urban-mythic feel. Why draw attention so openly to Beria's downfall while Russia is going through the motions of mourning Stalin? Reader input will be rewarded.
4. From Leopold Kronecker (1823-1891) and his famous dictum, "God made the integers; all else is the work of Man." This is one of those many slick aphorisms that defy reasonable analysis (if you spot the pun on *analysis*, a branch of mathematics, forgive me). Both theists and atheists have problems with Kronecker. Theists can claim that a Real God, by definition, is perfectly capable of extending the discrete integers to the Real number continuum. Atheists say that "God don't enter into the equation." Early man invented the integers (how else can one impose taxes or count slaves?); later, after much trial and error, came the rationals or fractions (how else to divide the pie?), followed by the badly named irrationals, made fully Real and reasonable by Richard Dedekind and other demigods in the late 19th century.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

STAN KELLY-BOOTLE (http://www.feniks.com/skb/; http://www.sarcheck.com), born in Liverpool, England, read pure mathematics at Cambridge in the 1950s before tackling the impurities of computer science on the pioneering EDSAC I. His many books include *The Devil's DP Dictionary* (McGraw-Hill, 1981), *Understanding Unix* (Sybex, 1994), and the recent e-book *Computer Language—The Stan Kelly-Bootle Reader* (http://tinyurl.com/ab68). *Software Development Magazine* has named him as the first recipient of the new annual Stan Kelly-Bootle Eclectech Award for his "lifetime achievements in technology and letters." Neither Nobel nor Turing achieved such prized eponymous recognition. Under his nom-de-folk, Stan Kelly, he has enjoyed a parallel career as a singer and songwriter.



Alloneword

Stan Kelly-Bootle, Author

Three years ago, to the very tick, my first Curmudgeon column appeared in *ACM Queue* to the rapturous, one-handed claps of the silent majority. Since then my essays have alternated intermittently with those of other grumpy contributors. With this issue (muffled drumroll), I'm proud to announce a Gore-like climate change in the regime that will redefine the shallow roots of ACJ (agile computer journalism, of which more anon). The astute *ACM Queue* Management (yes, there is such—you really must read the opening pages of this magazine!) has offered me the chance to go solo. For the next few *Queues*, at least, I am crowned King Curmudgeon, the Idi Amin of Moaners, nay, Supreme General Secretary of the Complaining Party! "I am Sir Oracle, and when I ope my lips, let no dog bark!"[1] Or rather, under the new dispensation, I command you to bark back via curmudgeon@acmqueue.com with your own pet peeves or counter-moans, which I promise to print if printable (subject to as light an editing as the Law dictates).

I also plan to pose posers and ask FUQs (frequently unanswered questions), as was my wont in the *Unix Review* Devil's Advocate columns of yore (1984-2000). As then, huge, literally invaluable prizes are offered for your answers and selected responses that meet my unpublished "Rules & Regulations." Suffice it to say that the customary bribes are encouraged; friends and relations enjoy traditional nepotistic advantages (in the old days my mother inevitably won the white Rolls-Royce convertible); and tedious accuracy scores lower than cunning disinformation. An ongoing challenge goes out to readers who encounter risible misprints and howlers in the computer literature, not excluding my own usually deliberate mishtakes.

Any errors you detect will be judged against the expected authority and inerrancy of the source. Thus, the many marketeering deviations from the untrampled snow-white truth will seldom rate highly unless, say, Gates or Jobs drops a real whopper. I allow new retrospective findings of false prophecies, but not the well-worn ones: at one end we have the quite plausible 1947 prediction by T. J. Watson (three IBM computers will more than meet the world's needs) and, at the other, the less plausible Bill Gates ("640K ought to be enough for anybody," 1981), which reflected the sad fact that IBM PC designers spurned the larger, linear-address space of the Motorola MC68000 microprocessor in favor of the Intel 8088. Bill later topped this faux pas: "The Internet? We are not interested in it" (1993). He also made several other ill-timed predictions about OS/2 (optimism unjustified) and Java (pessimism unjustified), but I'm loath to cast bricks: back in 1942 I swore undying love to a certain Nelly Moorcroft in a Liverpool jigger (back alley) while the Nazi bombs were falling...but I digress.

Errors, deceptions, AND AMBIGUITY

A particular source from which mistakes are sought is the much-cited Wikipedia. Wiki, as in Caesar's Weni, Widi, Wiki,[2] has arrived, looked around, and conquered. It has reached the top 10 in the most-visited site list, a remarkable achievement for a noncommercial project started in 2001.

Wikipedia, and the Web/Internet generally, received glowing praise from UK Education Secretary Alan Johnson as "an incredible source for good in education" for both teachers and pupils. "Wikipedia," he told a Schoolteachers' Union conference in April, "enables anybody to access information which was once the preserve only of those who could afford the subscription to *Encyclopaedia Britannica* and could spend the time necessary to navigate its maze of indexes and content pages." He's correct about the cost but rather out of date on the "maze," since the Britannica is now available online with the usual search and hyperlink features to replace the chore of heavy page turning. Predictably, some teachers groaned at the Wiki endorsement, having suffered from the increasingly blatant plagiarism by students innocently unable to distinguish fact from opinion and deliberate distortion. Cartoons show children boasting A levels in new subjects called "Cut and Paste" and "Drag and Drop." Well, I suppose they are modern skills to be honed and rewarded. Forget the content, dig the layout!

Continued on page 61

Get the complete picture of your next project.

- Manage features, defects and tasks in a highly-organized fashion with development work item drilldown
- See the entire history of work items, features and defects with fully configurable workflows
- Intelligent resource planning and allocations with integrated meeting requests
- Full visibility into the development effort with intuitive planning interfaces
- Real-time development task rollup for better project planning

TechExcel **DevSuite**

Whether you adopt just one DevSuite product, or implement the entire integrated suite, you will benefit from a powerful, knowledge-centric solution that is easily tailored to your organization's needs and development processes.

- Supports agile, iterative, waterfall and other methodologies
- Scalable from small teams to thousands of users
- Facilitates close collaboration of globally distributed teams
- Windows Smart Client and Web interface

Find out how TechExcel DevSuite empowers knowledge-centric ALM for your organization.
Visit www.techexcel.com/alm and view on-demand demonstrations today.

TechExcel

www.techexcel.com | 1-800-439-7782