Editors: Munindar P. Singh • singh@ncsu.edu
Krithi Ramamritham • krithi@cse.iitb.ac.in

Live Distributed Objects

Enabling the Active Web

Krzysztof Ostrowski and Ken Birman • *Cornell University*
Danny Dolev • *Hebrew University*

Distributed computing has lagged behind the productivity revolution that has transformed the desktop in recent years. Programmers still generally treat the Web as a separate technology space and develop network applications using low-level message-passing primitives or unreliable Web services method invocations. Live distributed objects are designed to offer developers a scalable multicast infrastructure that's tightly integrated with a runtime environment.

Developers who work with modern component-integration and productivity tools can create desktop applications faster and easier than ever before. These tools promote a development style in which the language, runtime environment, debugger, and profiler create a seamless whole. Architects of *distributed systems* face a more difficult challenge, especially when those applications need to replicate data or services to achieve scalability, guarantee fast response times, or put content close to the end user. Here, support has severely lagged the desktop options. The available technologies, such as publish–subscribe message buses, Web services remote method invocation, and multicast toolkits, are often implemented as free-standing proprietary libraries that are only superficially integrated with operating systems and modern component-oriented development tools. Moreover, the most common technologies provide such weak guarantees in the event of failure that the developer is typically forced to implement complex application-level "damage repair" mechanisms. This creates a world in which only experts can feel comfortable — and even the experts disagree about the best way to recover from common kinds of failures such as timeouts and components that crash and then reboot.

In exploring ways to bridge this gap, our team at Cornell created *live distributed objects*. Live objects have the look and feel of ordinary objects in managed environments such as .NET or J2EE, except that they needn't reside at a single location. A live object can be understood as a distributed mechanism through which a group of software components communicate with each other, share data, exchange events, and coordinate actions in a decentralized, peer-to-peer fashion (see Figure 1). A live object can represent, for example, a streaming video, a news channel, a collaboratively edited document, a replicated variable, or a fault-tolerant service.

The existing kinds of live objects are customizable; we're hoping that a small set of objects could suffice for a great variety of applications. On the other hand, the set is extensible; our platform makes it surprisingly easy to build new kinds of objects. The approach is intended to scale, and although our current system targets enterprise LAN settings, we plan to eventually support Internet deployments that might have, for example, tens of thousands of IPTV channels, new forms of collaboration and gaming environments, and new forms of self-managed applications that could literally span the globe. All the technology described in this article is working today in the lab and will be available for free public download from www.vcs.cornell.edu/projects/quicksilver/ early in 2008.

Potential Applications

From the programmer's perspective, live objects could replace earlier technologies such as multi-

 1089-7801/07/$25.00 © 2007 IEEE

cast, group communication, publish–subscribe, and state-machine replication. These often-proprietary technologies fit poorly into the modern component-oriented development style. In contrast, live objects fit easily into environments such as .NET and J2EE and can leverage their powerful type systems and management features. Development and debugging tools work in a natural way.

Live objects are also easy to use. Just as they browse for clip art in local folders, end users can store live objects in live folders and then build live documents and other applications by "cut and paste" or "drag and drop." Applications can access live objects much as they access other kinds of applications and files today (live objects adhere to the popular Web services standards), but we think the biggest potential is associated with this new kind of "no coding needed" style of application development. In effect, a developer without programming skills could create sophisticated collaboration, workflow, gaming, or other distributed applications by following the same steps used to create a presentation or a Web page. Moreover, these live applications inherit sophisticated reliability and scalability properties from our platform. If live objects were to take off, they could be the gateway to an active, trustworthy Web.

The Active Web

To understand the concept of an active Web, consider the popular Second Life virtual reality environment (www.secondlife.com), which has gained millions of users in just a few years. In the words of its inventors, "Second Life is a 3D virtual world entirely built and owned by its residents ... a vast digital continent, teeming with people, entertainment, experiences, and opportunity." A typical Second Life scenario might start with a user's avatar walking into a smoke-filled bar where a seedy collection of creatures is playing poker around a stained oak table. The avatar pulls up a chair, joins the game, and perhaps the user makes (or loses) a fortune — in real money.

Today, virtual worlds such as Second Life need to be hosted by big data centers. This has obvious disadvantages, particularly if the data center is halfway across the country (see Figure 2). Not only is performance limited by high latencies, but privacy could be a concern because everything an end user "sees" must flow through the data center.

In contrast, we envision a future in which data flows directly from the applications that generate it (for example, rendering an avatar) to those that consume it (for example, by displaying the room). Such an approach can support much higher data rates with lower latency. A data center might still play a supporting role, but such a solution would scale better and improve security and privacy because most of the data flows directly from the producer to the consumer. An active Web based on live objects could be a world with millions of IPTV streams, new forms of interactive art, live electronic health records that integrate regional medical providers, or banking systems that could trade "live" financial instruments. Moreover, live objects can interoperate with traditional documents to create live memos, spreadsheets, databases, and so forth.

Lacking platform support, such applications would be tricky to build. An electronic health-records system, for example, would need to achieve high levels of availability and consistency, be largely self-configuring, and maintain privacy and security. A typical deployment scenario would involve decentralized systems linked over networks integrating subsystems running at hospitals, other care providers, laboratories, insurance companies, pharmacies, and so on. Electronic monitoring devices and other sensors

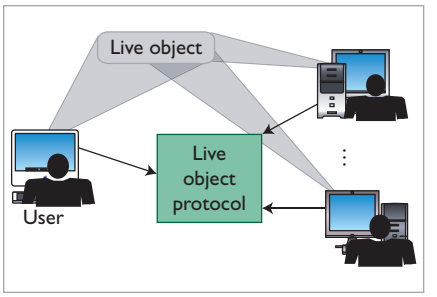

Figure 1. The live objects space. When multiple users across the Internet share a live distributed object, such as a room in an online game, their workstations instantiate local representatives that cooperate to implement whatever abstraction makes the object "live." Our live distributed objects platform automatically propagates updates and handles failures.

running in the hospital and at patients' homes would contribute time-sensitive data, and some therapeutic and drug-delivery devices would be remotely controlled. This highlights a second challenge: we need not only to enable a new style for developing such applications but also to ensure that the underlying platform can enforce the needed properties.

The Quicksilver System

Cornell's Quicksilver system offers a glimpse of the live object concept in action. As Figure 3 illustrates, Quicksilver is built in two layers. One extends a system such as .NET to support live objects by embedding them in the .NET common language runtime, as well as focusing on the hooks connecting the objects to the .NET type system and the Windows shell (the GUI that interprets mouse actions). Quicksilver's second layer provides the scalable and extensible communication infrastructure needed to make the objects "live" and "distributed."

Briefly, live objects provide the following abstract functionality:

- Each object has an Internet-wide unique identity, which serves a role

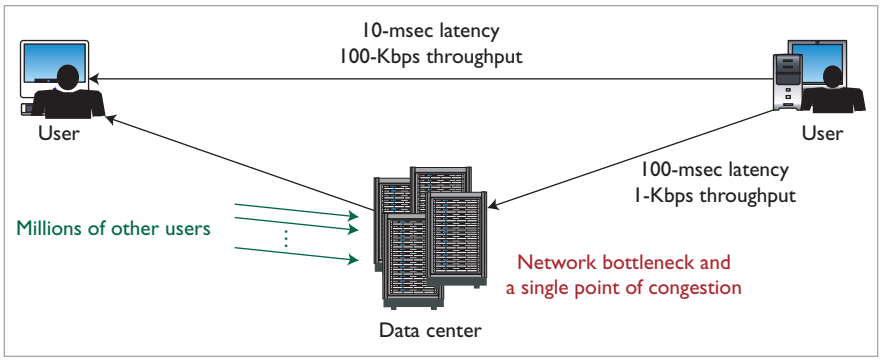

Figure 2. Benefits of direct communication. Existing massively multiplayer online games and virtual worlds rely on server farms to maintain system state. Letting users communicate directly in a peer-to-peer fashion, rather than indirectly through shared servers, slashes server loads, potentially improves latency and throughput, and better protects privacy.

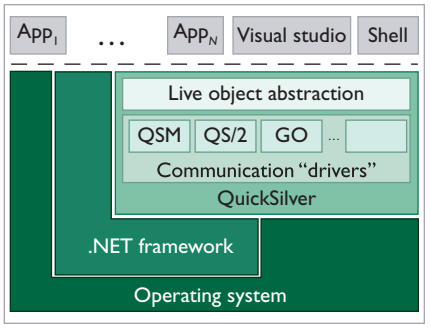

Figure 3. Quicksilver architecture. The system is composed of two layers: one exposes virtual objects to applications as a part of the .NET type system, and the other is an extensible communication engine with "drivers" for different object types. Live objects are accessible programmatically from applications, Windows shell-like files, and development environments such as Visual Studio.

similar to an ordinary file shortcut. Such shortcuts to live objects can be stored in various network directories, much like the Internet DNS, as well as dragged to users' desktops, exchanged over email, placed in file-system folders or inside documents, and so on.

- When a component tries to access a live object by clicking on the object's shortcut or programmatically accessing its reference, an authentication and join protocol executes to enforce security, fetch the object's components from distributed live object directories, obtain a snapshot of the object's state, initialize it, and so on. The object's code maintains its state in response to event notifications.
- Live objects have *distributed types*. Examples of types might include an IPTV feed, a chat window, or a dynamically tracked product inventory count; new types are easy to create by customizing existing objects or developing new ones. When applications interact with live objects, type-checking helps verify both that the object interface is used correctly (for example, if a live object were used in a spreadsheet, we'd want the spreadsheet to understand the associated data type) and that the communication type beneath the object matches the developer's intent. For example, an IPTV viewer object would need to run over a communications channel supporting the IPTV protocol standard.
- A live object provides a natural interface to its clients. For example, an object representing a video stream would provide methods to send or receive video frames, an object representing a replicated variable would provide `get/set` methods, and so on. Moreover, objects can be composed so that a replicated variable or a video stream could connect to a display object that showed new data in a window on the end user's display.
- Each object has object-specific logic implementing some abstraction, such as a room, medical record, or gossip-based overlay network. This logic would usually be very simple because it typically runs over a powerful communications infrastructure that offers support to groups of live objects running on different machines. For example, a chat object doesn't need to establish one-to-one connections between its various representatives: the object can disseminate each chat "event" by asking the runtime environment we provide to replicate it to the endpoints associated with other currently active instances of the same object. Moreover, although the interface from a live object's logic to our platform is simple, it supports a powerful form of extensibility: type-specific underlying *communication drivers* provide the protocols for replicating the object's data among components and for multicasting events among them. As we'll explain, this creates an opportunity to introduce reliability and trust properties.

The underlying communications layer consists of a set of pluggable communication substrates, which play roles analogous to those of device drivers. Each live object needs a communication driver to replicate its state and propagate events and updates; different objects might use drivers specialized for different settings (wireless, WAN, or LAN, for example) or properties (secure, low latency, and so on). QuickSilver lets developers adapt existing multicast and group communication toolkits for use as communication drivers. It also comes with the

following suite of built-in scalable, high-performance communication drivers designed to complement each other for common application types:

- Quicksilver Scalable Multicast (QSM) is optimized to support large numbers of live objects that might represent streams of events, such as video channels, file backup folders, or stock-price update notifications in a trading system. Designed for enterprise LANs and data centers, the current version of QSM can support tens of thousands of event streams, hundreds of users, and transmissions at network speeds. The system is robust under stress and reliably delivers events. (More about our experimental evaluation methodology and findings is available in several technical reports available from www.cs.cornell.edu/projects/quicksilver/).
- QuickSilver 2 (QS/2) is a second version of QSM in which the emphasis shifts to flexibility rather than raw performance. The QS/2 design has the flexibility to accommodate Internet WAN scenarios, mobile or wireless applications, and networks of low-power sensors. QS/2 will be available for download in 2008.
- The QuickSilver Properties Framework (PF) is a prototype of a communication driver that can work in tandem with QSM or QS/2 and is optimized to support live objects with strong reliability properties, such as replicated variables in a banking system or collaboratively edited documents, which need highly consistent updates to preserve data integrity. PF supports a wide range of reliability models, including custom, user-defined reliability types. We expect to make this available from our download site in mid-2008.
- Gossip Objects (GO) uses a kind of peer-to-peer communication that the research community refers to as a "gossip protocol" because it mimics the way that information spreads when people gossip about a hot rumor.[1] GO is intended to support live objects that can help systems manage themselves, set their own configuration parameters, and even diagnose and self-repair when failures occur. The GO communication driver is currently under development in the As Scalable as Possible project (ASAP; www.irisa.fr/asap), headed by Anne-Marie Kermarrec at the French national computer-science research laboratories (INRIA/IRISA) at the University of Rennes. GO should also be available in 2008.

We expect this list to grow over time, with protocols to support wireless communication, IPTV, security protocols, and protocols with real-time properties.

How Can It Scale?

Although the need for brevity precludes a highly technical discussion of scalability, some discussion of how we've tackled the problem in QSM should provide a rough intuition. QSM's role is to reliably multicast data in support of higher-level events related to multiple live objects, such as updates to their state and commands that cause actions to occur. In a typical data-center scenario, some computers would use large numbers of live objects, shared with other computers throughout the data center. For example, a group of computers might share some live object representing a particular IPTV channel, while another group of computers tracks a particular stock's price by sharing a stock-quote object, and yet other groups are engaged in a role-playing game in which live objects represent the game state. Because the granularity of objects could be rather small (a single stock, IPTV channel, or user's avatar, for example), the world of QSM could include vast numbers of overlapping groups of computers, each corresponding to some live object and having as its members the computers on which the live object is being used.

When we set out to build QSM, we realized that it would be helpful if we could assume that groups of live objects overlapped in a simple, hierarchical pattern. Before looking at the way QSM makes use of this property, let's develop some intuition into what a clean, hierarchical overlap among groups of objects might look like. For example, objects A, B, and C might be directly *superimposed*, such that the groups of computers using them overlap perfectly — this might happen if those computers were all using some live document containing A, B, and C. A hierarchy could also include subsets, such that object D might live on half of the computers and E on the other half. Let's define the term *region* to denote a set of computers that all use the same live objects. In this example, one region would correspond to computers using objects A, B, C, and D, and another region would include the computers using objects A, B, C, and E.

Given a hierarchy of overlapping groups, QSM starts by decomposing it into a set of regions and then constructs a data-dissemination overlay for each, using IP multicast if possible (for example, it might assign each region a unique IP multicast address). If IP multicast isn't available, QSM can also run on some form of software-based end-to-end network overlay. Note that dissemination is *unreliable*: like a UDP transmission in the Internet, a message might reach none of its destinations, some of them, or (if we're really lucky) all.

We'll need to automate the handling of the cases in which a message doesn't reach all the members of the group of objects for which it's intended. For this purpose, QSM builds a peer-to-peer repair structure within each region. This

involves many subtle issues, but the basic approach starts by constructing a logical token ring that links the computers in the region (if a region gets larger than roughly 25 computers, we break it into a tree of smaller rings linked by a higher-level ring). As the token travels around the ring a few times per second, we encode each computer's received message set. A machine that has a copy of a message some other machine lacks forwards a copy. If an entire region lacks a multicast, the sender remulticasts it. Our experiments indicate that this is very rare: most packet loss involves individual machines dropping single packets or a few in a row, and that can be repaired locally with the help of nearby peers.

Clustering computers into regions lets QSM handle large numbers of live objects efficiently by amortizing overhead. Whereas tens of thousands of live objects would involve tens of thousands of multicast protocols in traditional systems, QSM can use a smaller number of token rings, each of which combines the work of ensuring reliability for all the groups of objects that map to that token ring. We often see 10 or even 100 groups per region, so this amortization can work extremely well. In effect, a token ring provides reliability for many objects at a time, but because it operates at the regional level where all computers are members of the same groups of objects, this information has a very compact representation: a typical token is between 400 and 800 bytes long.

To handle computer crashes, QSM incorporates a hierarchical status-monitoring service structure that plays a role analogous to the Internet DNS but explicitly tracks each component's status (`live` or `failed`), as well as some additional information used within our protocols. We employ a consensus protocol to ensure that membership decisions are consistently reported.

Obviously, we're skipping a lot of details, such as the flow-control mechanism and data aggregation when multiple small messages are sent to the same group, but the upshot is that QSM seems to break every performance record we're aware of.[2] (Details are available in a collection of technical papers at www.cs.cornell.edu/projects/quicksilver.) We've scaled it to many thousands of groups (live objects) per computer, supported groups with hundreds of members, and saturated 100-Mbit Ethernet interconnects with inexpensive PCs on the endpoints. The system is stable under stress and tolerates load fluctuations, broadcast storms, and other degenerate behaviors very well. Moreover, even at the highest loads, overheads are low: in the scenarios just described, QSM CPU loads were less than 10 percent on receivers, and peaked at 40 percent on senders at the very highest data rate, leaving ample CPU capacity for other tasks, such as rendering the live objects' content.

Now, all of this reflects a simplifying assumption — namely, that groups overlap to form a simple hierarchy. In general, however, that won't be the case; if users share documents that contain live objects, we'll certainly see groups that overlap, but the overlap could be highly irregular. Fortunately, it turns out that we can solve this problem by superimposing multiple QSM hierarchies. We're finding that it's possible to cover even very irregular sets of overlapping groups with a surprisingly small number of hierarchies, provided the groups have Zipf-like popularity and traffic levels[3] — and there's a very good reason to believe that popularity would indeed be Zipf-like.[4] For example, studies of financial instruments show that the ith most popular stock or bond tends to be popular in proportion to $1/i^{\alpha}$, where α can be as large as 2.5 to 3.5. Trading volumes are also Zipf-like. We're betting that the same property will hold in the active Web if applications use large numbers of live objects.

What's in a Type?

Live objects open the possibility of extending normal type systems to encompass distributed behavioral patterns. A replicated variable that represents an account balance in a banking system might need strong reliability and fault-tolerance properties such as virtual synchrony, whereas weaker reliability properties and gossip scalability might be a better match for a similar variable in a monitoring application. To make live objects truly useful, we thus need a way to describe such behaviors as a part of their types.

In QuickSilver, a live object's type is a tuple, the elements of which specify different aspects of the type (much like in aspect-oriented programming), including the object's interface (in the usual sense) and its "category" (replicated service, replicated variable, event stream, gossip object, and so on), which determines the proper communication protocol driver to use. Other aspects configure the underlying dissemination substrate and specify the object's reliability, fault-tolerance, and security properties.

By expressing distributed types in this way, we enable a next step in which developers could use type information as part of the application design and implementation process. Development environments such as .NET's Visual Studio would "understand" the possible distributed behaviors of such typed objects and could guide developers through the process of implementing code that would run correctly under the assumption that the communication drivers underlying the live object implemented the specified behaviors. The current live objects system implements some simple forms of distributed type checking, but the idea could be carried much further. The runtime system could throw exceptions in response to mistakes — if, for example, a component designed to work correctly only with live virtually synchronous multicast streams tried to

access a live multicast stream with a weaker QSM or best-effort reliability property.

We intend to pursue this direction of research because type checking can dramatically reduce bug rates while also providing a "hook" for other purposes. Type-based programming and debugging tools have transformed the experience of building applications for desktop environments. Service-oriented architectures (which also revolve around type systems, albeit simple ones) are having a similar impact in networked applications that interact with services hosted in data centers. A natural step in that direction, live objects bring the benefits of strong typing to the realm of decentralized peer-to-peer applications. The active Web could be far more than just a veneer over the same old Internet technologies.

Next Steps

Live distributed objects, layered over the QSM technology, are working today. However, we've also made substantial progress with two technologies that aren't quite ready for external users.

Quicksilver Properties Framework

The basic goal of the PF is to support stronger reliability models than are available in QSM. Earlier, we described how QSM provides what might be called best-effort reliability: as long as the system believes that some computer is healthy, it keeps trying to deliver messages to it. This is a natural form of reliability, but distributed systems sometimes need much stronger guarantees. Two important examples of these are *virtual synchrony* and *transactions*.

Virtual synchrony is a powerful distributed computing model in which active programs join *process groups*, within which multicasts disseminate updates and other events.[5] The moment a process joins, it can initialize itself using a *state transfer* from some active member. If a member fails, the group members are immediately informed, so that they can initiate corrective action. This model's power is that it can support consistency guarantees — all the users of a virtual synchrony group see the same thing in the same order. This can be important when multiple users are concurrently taking actions that the physical world needs to somehow order. In some card games, for example, the first user to slap a card onto the stack wins. Using traditional event-notification solutions, different users might see the same events in different orders — it's easy to imagine (virtual) fights erupting in a Second Life gambling saloon in such cases. With virtual synchrony, the Quicksilver platform would enforce a single, system-wide event ordering, so that all participants would see the same events in the same sequence.

Transactions offer an even stronger execution model. Suppose that a power failure were to crash a few machines simultaneously. The virtual synchrony model provides consistency only among live objects that remain operational. If an object crashes and then restarts, saved data from its previous life is discarded. Yet, suppose that the object represented the bank, and real money were changing hands. Transactional live objects support what the database community refers to as the ACID model (atomicity, concurrency, independence and durability). With this guarantee, crashed objects can reconstruct consistent, agreed-on state after they recover. However, these stronger guarantees come at a steep price in terms of performance and scalability.[5]

When we decided to support live objects with stronger reliability guarantees than QSM offers, we didn't want to limit ourselves to just one of these models. Accordingly, PF is configurable so that each object can specify the appropriate kind of consistency guarantee. PF implements different models by executing a script to control the delivery of messages and other events and to trigger actions such as forwarding messages to repair losses or sending ordering information when batches of messages must be placed into a total order. The script is coded in a new programming language we've developed, which can represent models such as virtual synchrony or transactions in just 30 or 40 lines of very high-level code that PF automatically translates into a scalable, hierarchical protocol. The advantage of this approach is that a single system can support live objects that require any of a range of reliability models. In contrast, the more traditional approach involves building a separate system for each — a transactional system, a virtual synchrony systems, and so forth.

We don't expect PF to subsume QSM's weaker reliability property because it isn't clear how to support strong models such as virtual synchrony or transactions in settings that might have hundreds or thousands of overlapping groups. Instead, we're assuming that live objects layered over QSM should suffice for most cases. In the rare cases in which stronger properties are required, PF working in tandem with QSM could provide them.

It's too early to tell, but at Cornell, we're betting that the active Web will be the next big thing for the Internet.

Thus, QSM would do most of the work in large systems, but PF might "step in" here and there as needed to support a small number of live objects with much stronger properties.

Gossip Objects

As mentioned earlier, the GO project is under way at INRIA/IRISA.[1] To understand the underlying concept, consider the following real-world scenario: someone sees Mary getting into John's car in the company parking lot and starts a rumor that the two are going out together. The information spreads exponentially quickly, and in no time at all, everyone in the company has heard the story. GO implements protocols that spread information using a computer analog of this sort of rumor mongering (researchers refer to it as an *epidemic model*).

When finished, GO will let live objects share information by replicating it using this gossip-based communication layer. Gossip turns out to be ideally suited for tasks such as tracking overall system state, assisting in autoconfiguration or repair, and constructing overlay networks (including those needed by QSM, QS/2, and PF). Although gossip protocols can be slow, they need little configuration, are stable under stress, never require much bandwidth, and remain "healthy" despite disruptions that can cripple other kinds of protocols.

Thus, we have a fast and scalable but limited-reliability option in QSM, a robust and self-configuring but relatively slow technology in GO, and a way to support strong properties through PF. This list will grow to include additional options, such as IPTV or BitTorrent protocols, in the future. We're also looking at security issues with the goal of securing the live objects infrastructure and preserving the privacy of sensitive data.

Live objects enable a completely new kind of distributed programming, inspired by the Web, but in which much of the content is dynamic and capable of evolving rapidly in real time. End users who lack programming skills can combine live objects to create sophisticated, fault-tolerant applications, often with little more than a few mouse clicks. Live objects could represent video feeds, streams of media or other content generated by participating computers, telemetry from sensors, and so on.

By integrating such content into systems such as Windows or J2EE in a clean and natural way that leverages the power of type systems and component-integration technologies while offering a portal to distributed computing, we're hoping to enable a revolution. Live objects could open the door to a new and disruptive generation of active Web applications that combine high data rates with strong properties, including fault-tolerance, consistency, and security.

It's too early to tell, but at Cornell, we're betting that the active Web will be the next big thing for the Internet — and that live objects will make it a reality.

References

1. K. Birman et al., "Exploiting Gossip for Self-Management in Scalable Event Notification Systems," *Proc. IEEE Distributed Event Processing Systems and Architecture Workshop* (DEPSA 2007), IEEE CS Press, 2007; www.cs.uga.edu/~laks/depsa/.
2. K. Ostrowski and K. Birman, *Implementing High-Performance Multicast in a Managed Environment*, tech. report TR2007-2087, Cornell Univ., Mar. 2007; www.cs.cornell.edu/projects/quicksilver/QSM/.
3. Y. Vigfusson et al., "Tiling a Distributed System for Efficient Multicast," submitted for publication to 5th Usenix Symp. Networked Systems Design and Implementation (NSDI 08), 2008; preprints available on request.
4. M.E.J. Newman, "The Structure and Function of Complex Networks," *SIAM Rev.*, vol. 45, no. 2, Mar. 2003, pp. 167–256; http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&tid=SIREAD00004500000200016700000 1&tidtype=cvips&tgifs=yes.
5. K. Birman, *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Springer-Verlag, 2005.

Acknowledgments

Our research was supported by grants from the US Air Force Office of Sponsored Research, the Air Force Research Laboratories in Rome, NY, the US National Science Foundation (NSF) Cybertrust program, the TRUST NSF Science and Technology Center, and Intel.

Krzysztof Ostrowski is a PhD candidate at Cornell University in the Department of Computer Science. He is the primary developer of the live distributed objects and Quicksilver technologies and has published several papers on these systems, as well as the broader topic of architectural standards for Web services event notification. Contact him at krzys@cs.cornell.edu.

Ken Birman is a professor of computer science at Cornell University. His research focuses on reliability issues in distributed computing systems. Birman has a PhD in computer science from the University of California, Berkeley, and has published more than 150 conference and journal papers and edited three textbooks in the area. He was the principle developer of the Isis toolkit, which was used to implement the core communications technology of the New York and Swiss Stock Exchanges, the French Air Traffic Control System, and the US Navy AEGIS warship. The associated virtual synchrony model became a Corba standard. He is a fellow of the ACM. Contact him at ken@cs.cornell.edu.

Danny Dolev is a professor of computer science at Hebrew University in Jerusalem. His research explores the theory and practice of distributed computing, and over the course of his career, he has published nearly 200 papers on a wide range of topics in the field. Dolev has a PhD in computer science from the Weizmann Institute of Science. His group developed the Transis group communication system, which explored availability during partitioning failures and resulted in a theory of optimal availability in fault-tolerant systems. Contact him at dolev@cs.huji.ac.il.