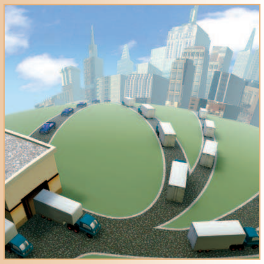

Publish–Subscribe for High-Performance Computing

High-performance computing could significantly benefit from publish–subscribe communication, but current systems don't deliver the kind of performance required by applications in that domain. In response, the authors developed Echo, a high-performance event-delivery middleware designed to scale to the data rates typically found in grid environments. This article provides an overview of Echo, the infrastructure on which it's built, and the techniques used to implement it.

Greg Eisenhauer and Karsten Schwan
Georgia Institute of Technology

Fabián E. Bustamante
Northwestern University

Event-based communication is an important component of many distributed applications and services. The publish–subscribe paradigm it supports is well-suited to the reactive nature of many novel applications (including collaborative-environment, mobile, and pervasive computing), allowing subscribers to state their interests and receive notification of any publisher-issued event that meets that interest. This *decoupled* approach to communication aids system adaptability, scalability, and fault tolerance[1] because it enables the rapid and dynamic integration of legacy software into distributed systems, supports software reuse, facilitates software evolution, and fits with the component-based approaches that have become increasingly popular in wide-area high-performance computing.

Unfortunately, most existing publish–subscribe systems also impose substantial overhead, delivering significantly less than the bandwidth and latency available from the raw network. This situation has limited the application of such systems in high-performance computing, in which computational progress often depends directly on delivered network throughput.

In response to these challenges, we developed Echo at Georgia Tech as a high-performance event-delivery middleware designed to scale to the data rates found in grid-style computing environments (www.cc.gatech.edu/systems/projects/ECho). Echo lets applications reap the maximum benefit from available bandwidth by allowing receivers to customize delivery through *derived event channels*, mechanisms that can operate at network transmission speeds.

 Published by the IEEE Computer Society 1089-7801/06/$20.00

Echo Functionality

Echo supports semantics common to both *channel*- and *type*-based publish–subscribe systems.[1] With channel-based subscription, an *event channel* is the mechanism through which event sinks and sources are matched: source clients publish events to a specific channel, and the sink clients subscribed to that channel receive notification of the event. In addition, Echo supports typed channels, which transmit and handle fully typed events.

Efficient Event Notification

Many event service implementations are centralized in some way, either with an overall event server or with specific objects representing each channel. To avoid the potential reliability and performance problems associated with centralized approaches, Echo event channels are lightweight, fully distributed, virtual entities.

Figure 1a depicts a set of processes communicating via event channels. In the figure, the channels exist in the space between processes, but in practice, they're distributed, with bookkeeping data residing in each process in which they're referenced (see Figure 1b). The process that creates the event channel is distinguished because it's the contact point for other processes wishing to use the channel. The distribution of event notifications, however, is completely decentralized and makes no distinctions among processes. A source sends event messages directly to all sinks, and network traffic for individual channels is multiplexed over shared communication links. Sharing the communication links helps ensure that Echo event channels are lightweight entities, thus allowing many to coexist in a single process. (Some high-performance systems with multi-endpoint delivery needs can benefit from overlay networks that use intermediate nodes to help multiplex and relay events. We describe Echo's support for those systems later.)

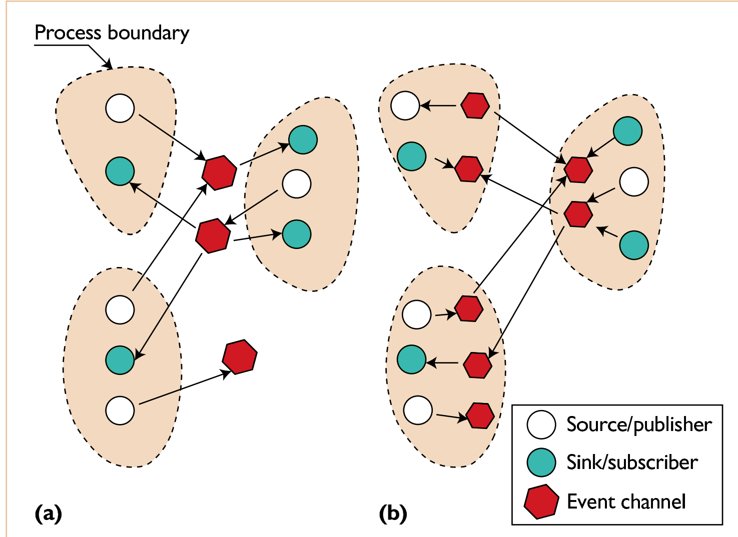

Figure 1. Using event channels for communication. In this (a) abstract view of event channels and (b) an Echo realization of event channels, we see the decentralized structure of Echo's realization.

Event Notification Types and Typed Channels

One of Echo's differentiating characteristics is its support for efficient transmission and handling of fully typed events. Some event delivery systems leave event data marshalling to the application; others support only generalized name–value pairs to represent all or part of the event data. In contrast, Echo allows structured types to be associated with event channels, sinks, and sources, and automatically handles heterogeneous data transfer issues.

Echo is implemented on top of PBIO (Portable Binary I/O),[2] a package developed at Georgia Tech to simplify heterogeneous binary data transfer. Building marshalling functionality into Echo using PBIO allows for layering that nearly eliminates data copies during marshalling and unmarshalling. As other researchers have noted,[3] minimizing these data copies is critical to delivering full network bandwidth to higher levels of software abstraction. Layering with PBIO makes Echo suitable for applications that demand high-performance communication of large amounts of data. In particular, because PBIO and Echo can directly transport structured types, memory-resident data in a source program can be published, sent to subscribers, and recreated as memory-resident data at the destination with minimal transformation.

Base type handling and optimization. In the context of high-performance messaging, Echo event types are most functionally similar to the user-defined types found in the message-passing interface (MPI), a widely used standard in high-performance systems. The main differences are in expressive power and implementation. Like MPI's user-defined types, Echo event types describe C-style structures made up of atomic data types. Both systems support nested structures and statically sized arrays, but Echo's type system extends this to support null-terminated strings and dynamically sized arrays. (Dynamic array sizes are given by an integer-typed field in the record. Full information about the types Echo and PBIO support appears elsewhere.[2])

The full declaration of message types to the underlying communication systems, as supported by Echo and MPI, makes several performance optimizations possible: it allows the underlying communication system to optimize buffer usage, minimize copying, and take advantage of the sending and receiving systems' characteristics. Unlike many MPI implementations, Echo and PBIO exploit this by combining native data representation (NDR) with the dynamic generation of unmarshalling routines. The reliance on NDR lets Echo and PBIO avoid the overhead associated with intermediate *wire* formats by using the sender's NDR as the wire format. A wire format is the representation of data during transmission, such as the external data representation (XDR) or the Internet Inter-Orb Protocol (IIOP); it is typically fixed and agreed upon in advance. On the receiving side, we must convert from the wire format (that is, the sender's NDR) to the receiver's native format – a process commonly referred to as unmarshalling. If necessary, PBIO converts the receiver's native representation upon receipt by dynamically generating conversion routines. As we've previously shown,[4] PBIO encode times don't vary with data size, and its decode times are much faster than MPI's. Given that as much as two thirds of the latency in a heterogeneous message exchange is software conversion overhead,[4] PBIO's NDR approach yields round-trip message latencies as low as 40 percent compared to that of MPI.

In PBIO and Echo, each marshalled data package contains a *format cookie* (similar to the markup in an XML document) that identifies the meta-information associated with the data. The interested party can then retrieve the meta-information required to decode and process an event, thus allowing intervening hosts to filter or transform events without a priori knowledge of the event's contents. Once retrieved, PBIO caches the meta-information for reuse. Because data streams in high-performance computing typically consist of data that can be described with only a few structured types, meta-information retrieval has a minimal impact on steady-state performance.

Type extension. Echo supports the robust evolution of sets of programs communicating with events by allowing variation in the data types associated with a single channel. In particular, a source can submit an event whose type is a superset of the type associated with its channel. Conversely, an event sink can have a type that is a subset of the event type associated with its channel. This enables the independent and unsynchronized evolution of event types at either end without disrupting previously set communications. Echo even allows type variation in intraprocess communication, imposing no conversions when source and sink use identical types, but performing the necessary transformations when source and sink types differ in content or layout.

In terms of flexibility, Echo's features most closely resemble those systems that support the marshalling of objects as messages. In such systems, the support that subclassing and type extension provide for robust system evolution is substantively similar to that provided by Echo's type variation, but object-based marshalling often suffers from prohibitively poor performance. XML-based messaging systems naturally offer a type-flexible coupling between event sources and sinks because of XML parser characteristics. However, the performance overhead of XML-based messaging (due to the required binary-encoded/text/binary-encoded conversions and the transport of significantly larger messages) makes it unsuitable for the needs of high-performance computing applications. Echo's strength is that it maintains the application integration advantages of object- and XML-based systems while offering communication performance that, in many cases, outperforms more traditional (and rigid) message-passing systems.

Basic Data Exchange Performance

Echo's delivered bandwidth and latency are near what the raw network offers when it transports a similar number of bytes between user-level application endpoints. Its performance advantage is due mostly to its adoption of NDR[4] as a wire format with out-of-band access to the message's meta-information. This, together with careful software layering, allows event data to be written directly to the wire from memory without any copies or transformation. At the destination, the receiving process can often use the data directly from the receive buffer without further transformation. When transformation is necessary due to different native data formats, Echo relies on dynamically generated subroutines. Figure 2 provides relative performance measures; a detailed discussion of Echo's performance characteristics and the sources of its performance advantages appear elsewhere.[4–6]

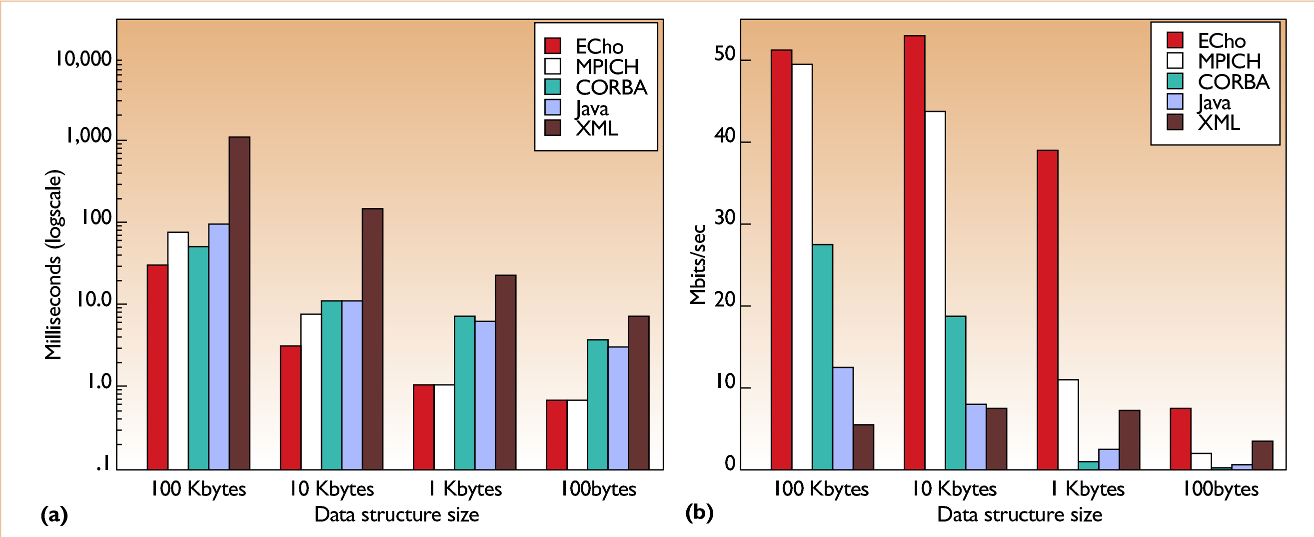

Figure 2. Relative performance. Echo's performance with respect to (a) round-trip latency and (b) delivered bandwidth shows that Echo delivers more bandwidth at a lower latency across a range of data structure sizes.

Event Filtering and Transformation

Event subscription schemes differ in the ways in which users specifiy their events of interest. Beyond subscription schemes, a few event systems provide some form of filtering mechanisms to allow for more specific, customization of the event stream. Echo's principal contribution to specializing data flows is the concept and realization of derived event channels.

Echo's derived event channel bears some similarity to prior work on content-based filtering and pattern-based filtering and transformation.[7] However, Echo allows general computations over event data and ensures their efficient execution through the use of dynamic code generation (DCG) and the reliance on decentralized event distribution.The Java-based approach of Distributed Asynchronous Collections[8] offers broad generality in content-based subscriptions, but it lacks the transformation capacity of derived event channels and offers significantly lower throughput and higher latency than Echo.

Derived Event Channels

Echo's approach to event channel customization is to extend the channels via *derivation*. Applications that want to customize their event data can create a new channel whose contents are derived from those of a preexisting channel through an application-provided derivation function, *F*. The event channel implementation moves this function to all event sources in the original channel, executes it locally whenever events are submitted, and transmits any resulting event via the derived channel. If the derived event channel's sinks are local to any of the sources in the original setup, network traffic between them is avoided entirely. This has the advantage of eliminating unwanted event traffic and the associated waste of computational and network resources.

Mobile Functions and the E-Code Language

A critical issue when implementing derived event channels is the nature of the function *F* and its specification. Because the client specifies *F* to be evaluated at the (possibly remote) source, a simple function pointer is obviously insufficient. Corba and Siena use a relatively restricted filter language (for example, a combination of Boolean operators),[9] an approach that enables efficient interpretation, but can limit the system's applicability. Ideally, we could express *F* through a more general programming language. Although this is relatively easy to support in a homogeneous system using dynamic linked libraries (DLLs), it becomes particularly difficult in heterogeneous settings, especially if type safety is to be maintained. Opting for an interpreted language (such as, Tcl or Java) avoids problems with heterogeneity, but at the cost of performance.

Given Echo's target domain of high-performance computing, and based on the observation that most commonly used or required filters are relatively simple, we made Echo's derived event channel rely on a small language, E-Code,[10] as well as DCG. We

```
{
  if ((input.trade_price < 75.5) ||
      (input.trade_price > 78.5)) {
       return 1; /* submit to derived */
  }
  return 0; /* do not submit to derived */
}
```
(a)

```
{
  int i;
  int j;
  double sum = 0.0;
  for(i = 0; i<37; i= i+1) {
    for(j = 0; j<253; j=j+1) {
      sum = sum + input.wind_velocity[j][i];
    }
  }
  output.av_velocity = sum / (37 * 253);
  return 1;
}
```
(b)

Figure 3. Example ECL event filter/transformation functions. A specialization filter (a) passes only those stock trades outside a predefined range and (b) computes the average of an input array and passes the average to its output.

express the function F in E-Code and use dynamic code generation to create a native version of F on the source host. E-code is currently a subset of C, supporting C operators, `for` loops, if statements, and return statements.

E-Code's dynamic code generation capabilities are based on a Georgia Tech DCG package that provides a virtual reduced instruction set computing (RISC). E-Code consists primarily of a lexer, parser, semanticizer, and code generator and is the equivalent of a just-in-time compiler. As such, the E-Code/RISC system generates native machine code directly into the application's memory without reference to an external compiler. Because E-Code is designed to operate on the fully described and array-bounded structured data types that PBIO supports, E-Code can ensure that the generated code doesn't reference memory other than what the event provided to it. Thus, Echo doesn't have to rely on a virtual machine or other sandboxing techniques, and its generated filters and transformations can run at roughly the speed of unoptimized native code. Generating native code for an E-Code subroutine is considerably faster than forking an external compiler — for example, for a 66-line-integer gray-scaling code, ECL requires only 4 ms to generate native code, whereas forking the GNU C compiler (GCC) requires 700 ms.

Echo currently supports two uses of derived event channels. In its simplest mode, the derived channel's event type is the same as that of the original channel. In this case, the E-Code required is a Boolean filter function that accepts a single parameter, which is the input event. If the function operating on an event returns a nonzero value, Echo submits the event to the derived event channel, otherwise it is filtered out. Event filters can be quite simple, such as the stock-trading example in Figure 3a, which passes trade information into the derived event channel only when the stock is trading outside of a specified range. When used to derive a channel, Echo transports this code in string form to the event sources associated with the parent channel, which is where Echo parses it and generates its native code. Echo evaluates the code string in the context of a function declaration of the form:

`input f(`⟨*input event type*⟩ `input)`

where ⟨*input event type*⟩ is the type associated with the parent channel. Once derived, the created channel behaves as a normal channel with respect to sinks. It has all the parent channel's sources as implicit sources, but any new sources providing unfiltered events could also be associated with it. Because the derived event channel is a full-fledged channel, its content is also subject to *chained* derivations that further customize the data stream.

Beyond this basic mode, Echo also supports derived event channels in which the event type associated with the derived channel is different from that of the parent channel. In this case, Echo adds an output parameter to the function declaration described earlier: the generated code is responsible for initializing all the output data structure's elements, as in the example in Figure 3b, which is taken from a global climate simulation. The relative performance gains from event filtering depend directly on the proportion of filtered-out events or the size reduction achieved by event transformation, and are thus highly application-specific.

Underlying System

The facilities described so far give a simple view of Echo's external interfaces, but Echo typically targets complex, large-scale environments and settings in which massive data streams must be deliv-

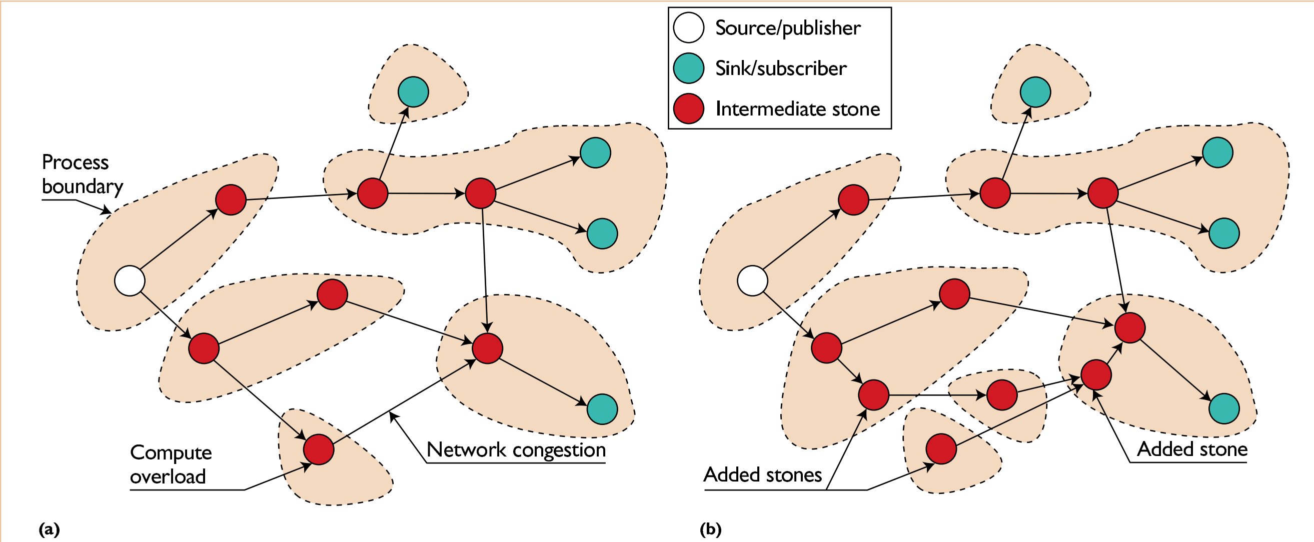

Figure 4. Changes in an overlay network. (a) An overlay with localized congestion or computational overload, and (b) the same overlay with additional processing stones.

ered to multiple clients with some degree of application-level, quality-of-service requirements.

We built early versions of Echo on a PBIO-based point-to-point messaging system that always performed direct source-to-sink event delivery. This approach delivers the targeted results in cluster-based high-performance computing environments, but in enterprise or wide-area environments, point-to-point messaging can limit application scalability with respect to the number of sinks or subscribers per notification.

To address this issue, as well as to provide an infrastructure for continuing research in adaptive delivery, our recent work builds on EVPath, a package for facilitating the construction of event notification overlay networks. Work on EVPath began in late 2004, and although it isn't available for distribution yet, you can find preliminary information at www.cc.gatech.edu/systems/projects/EVPath.

Overlays: EVPath

EVPath is a middleware package designed to facilitate the dynamic composition of overlay networks for message passing. Instead of imposing a particular overlay, EVPath focuses on link's characteristic monitoring (such as available link bandwidth, processing load, and event rates), overlay control, and actual data handling. For global decisions on the suitability of overlay paths, service placement, and so on, EVPath relies on higher-level controls.

Although EVPath is designed to implement the paths over which data can travel, it doesn't implement a path abstraction itself. Instead, the principal abstractions in EVPath are *stones* (similar to stepping stones), which, when linked together, compose a data path. Message flows between stones can be both intra- and interprocess, with interprocess flows managed by special *output stones*. Other types of stones include *terminal stones*, which implement data sinks; *filter stones*, which transform data; and *split stones*, which implement data-based routing decisions and can redirect incoming data to one or more other stones for further processing.

All stones have associated queues that hold incoming data before it's processed. For output stones, data can accumulate in the queue if an outgoing network link is congested. Output stones have congestion handlers that can discard events or perform other application-specified data reductions. For other types of stones, the queue serves only as a temporary stopping point during normal operation because immediate processing and transfer to the next stone is almost always possible. However, the ability to freeze portions of the message flow for a period of time is essential for dynamically reconfiguring the path without disrupting the current data-processing activity. Figure 4a shows an overlay experiencing difficulty caused by either a compute overload on one of the processing nodes or undue congestion in one of the associated network links. Figure 4b shows the same logical overlay, but the problems have been relieved by adding an extra stone, to which some of the data from the problematic stone is offloaded.

In this case, an upstream stone responsible for splitting the data between the two processing nodes and a downstream node that reassembles the stream have also been added.

To support the enactment of this kind of dynamic overlay change, EVPath lets higher software layers suspend processing in stones, relocate queued events, create and destroy stones, and modify those stones' linkages. It also has built-in mechanisms for monitoring and collecting suitable information relating to system performance and capabilities, making such information available as attributes to higher-level decision-making layers (which manage the overlays to ensure desired end-to-end quality characteristics). Echo's current use of EVPath is in its infancy, but it's developing into a system that constructs optimized overlay networks on the fly and includes the ability to migrate processing (such as derived event channel filters) into interior network nodes.

Advanced Topics

We designed EVPath and Echo to be extensible research tools. As such, they must be flexible enough to perform and support investigation in a variety of situations — EVPath, for example, is designed as an enactment layer that supports message passing on overlay networks. Because the construction, optimization, and management of overlay networks is an active research area,[11] EVPath isn't customized for a particular overlay mechanism, but relies instead on an external overlay management layer.

Support for congestion handlers and network monitoring in EVPath is designed to support the creation of applications that can adapt and respond appropriately to changes in network-level conditions, customizing their own behavior and bandwidth demands as well as potentially adapting the network protocols themselves.[12] Network monitoring also informs overlay creation and management. Our Active Streams[13] project demonstrates early work in these directions. With Active Streams, we explored a novel approach that, by leveraging much of this infrastructure, aims to facilitate the task of building large-scale distributed systems for heterogeneous, highly dynamic environments through the online composition, mapping, and steering of filters on data streams.

Most component-based systems rely on publish-subscribe infrastructures for integration. To take full advantage of the component-based approach, our work on Echo extends the applicability of publish-subscribe to encompass applications' main, performance-intensive, data flows.

Echo has been in continuous use for over five years, in applications ranging from corporate information flow models to high-performance scientific computations. In our ongoing work, we're investigating application-level event filters that can customize information flow based on dynamic network resource availability such as bandwidth and end-to-end latency. We envision Echo extensions that allow the messaging system to play a more active role in resolving the versioning differences resulting from application evolution, including techniques such as message morphing.[14] Because Echo filters are simple and transportable, we might be able to migrate them to the most appropriate location in both a single's node's network stack and over the multiple nodes in the data path. We're currently integrating new overlay management schemes to enable filter migration over the network. We're working toward moving such functionality into "smart" network routers. In a local node, we've been able to place them at different levels in the network stack, into the kernel, and even attached into network interface cards.

References

1. P. Eugster et al., "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, 2003, pp. 114–131.
2. G. Eisenhauer and L.K. Daley, "Fast Heterogeneous Binary Data Interchange," *Proc. Heterogeneous Computing Workshop* (HCW 2000), IEEE CS Press, 2000, pp. 90–101.
3. M. Lauria, S. Pakin, and A.A. Chien, "Efficient Layering for High Speed Communication: Fast Messages 2.x," *Proc. 7th High Performance Distributed Computing* (HPDC-7), IEEE CS Press, 1998, pp. 10–20.
4. F.E. Bustamante et al., "Efficient Wire Formats for High Performance Computing," *Proc. Supercomputing00* (SC 2000), IEEE CS Press, 2000, article 39 on the CD-ROM version.
5. G. Eisenhauer, F.E. Bustamante, and K. Schwan, "Event Services in High Performance Systems," *Cluster Computing: The J. of Networks, Software Tools, and Applications*, vol. 4, no. 3, 2001, pp. 243–252.
6. P. Widener, G. Eisenhauer, and K. Schwan, "Open Metadata Formats: Efficient XML-Based Communication for High Performance Computing," *Proc. 10th High Performance Distributed Computing* (HPDC-10), IEEE CS Press, 2001, pp. 739–742.
7. R. Strom et al., "Gryphon: An Information Flow-Based Approach to Message Brokering," *Int'l Symp. Software Reliability Eng.*, IEEE CS Press, 1998; www.chillarege.com/

fastabstracts/issre98/98423.html.
8. P. Eugster, R. Guerraoui, and Joe Sventek, “Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction,” *Proc. European Conf. Object-Oriented Programming* (ECOOP’00), Springer-Verlag, 2000, pp. 252–276.
9. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, “Challenges for Distributed Event Services: Scalability vs. Expressiveness,” *Proc. Engineering Distributed Objects* (EDO 99), IEEE CS Press, 1999, pp. 72–78.
10. G. Eisenhauer, *Dynamic Code Generation with the E-Code Language*, tech. report GIT-CC-02-42, College of Computing, Georgia Inst. of Technology, July 2002; ftp://ftp.cc.gatech.edu/tech reports.
11. S. Birrer and F.E. Bustamante, “Resilient Peer-to-Peer Multicast without the Cost,” *Proc. Multimedia Computing and Networking* (MMCN), ACM Press, 2005; www.aqualab.cs.northwestern.edu/projects/Nemo.html.
12. Q. He and K. Schwan, “IQ-RUDP: Coordinating Application Adaptation with Network Transport,” *Proc. 11th High-Performance Distributed Computing* (HPDC-11), IEEE CS Press, 2002, pp. 369–378.
13. F.E. Bustamante, *The Active Streams Approach to Adaptive Distributed Applications and Services*, PhD thesis, College of Computing, Georgia Inst. of Technology, 2001.
14. S. Agarwala, G. Eisenhauer, and K. Schwan, “Lightweight Morphing Support for Evolving Data Exchanges in Distributed Applications,” *Proc. 25th Int’l Conf. Distributed Computer Systems* (ICDCS-25), IEEE CS Press, 2005; www.cc.gatech.edu/~sandip/research/morph_icdcs05.pdf.

Greg Eisenhauer is a research scientist in the College of Computing at the Georgia Institute of Technology. His research interests include interactive computational steering, novel communication infrastructures, dynamic adaptation and autonomic systems, performance evaluation, and scientific computing. Eisenhauer has an MS in computer science from the University of Illinois and a PhD in computer science from Georgia Tech. He is a member of the IEEE Computer Society and the ACM. Contact him at eisen@cc.gatech.edu.

Fabián E. Bustamante is an assistant professor of computer science in the Department of Electrical Engineering and Computer Science at Northwestern University. His research interests include distributed systems and networks, and operating systems support for massively distributed systems, including Internet and vehicular network services. Bustamante has an MS and PhD in computer science from the Georgia Institute of Technology. He is a member of the IEEE Computer Society, the ACM, and Usenix. Contact him at fabianb@cs.northwestern.edu.

Karsten Schwan is a professor in the College of Computing at the Georgia Institute of Technology, where he also directs the Center for Experimental Research in Computer Systems (CERCS). His research interests include middleware and systems support for distributed systems running critical applications, focusing on the real-time behavior of distributed and parallel applications, the adaptive nature of pervasive applications, and the dynamic configuration and adaptation of distributed application or system components. Schwan has an MSc and a PhD in computer science from Carnegie-Mellon University. Contact him at schwan@cc.gatech.edu.