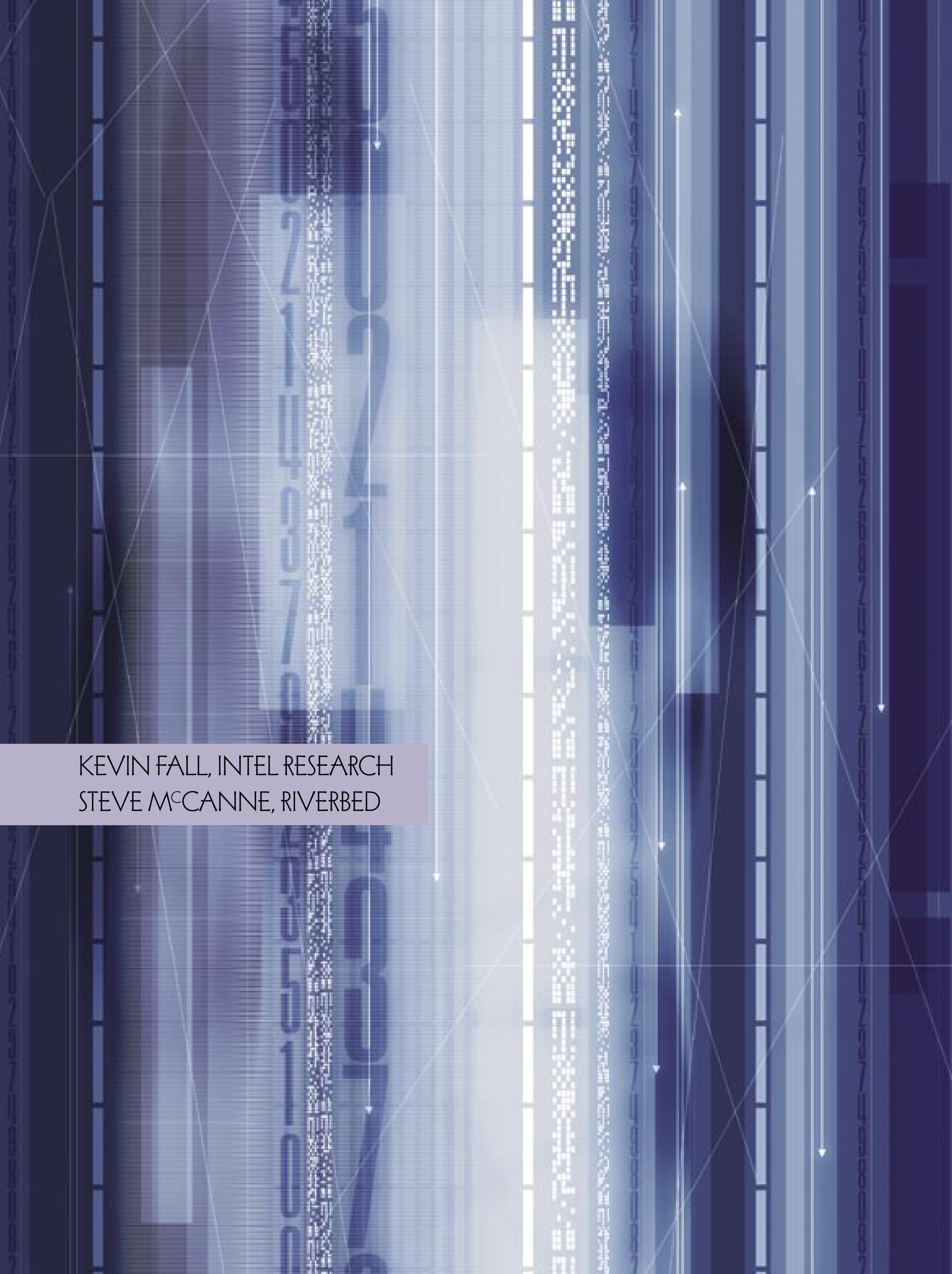

KEVIN FALL, INTEL RESEARCH
STEVE McCANNE, RIVERBED

you don't know jack about Network Performance

Bandwidth is only part of the problem.

Why does an application that works just fine over a LAN come to a grinding halt across the wide-area network? You may have experienced this firsthand when trying to open a document from a remote file share or remotely logging in over a VPN to an application running in headquarters. Why is it that an application that works fine in your office can become virtually useless over the WAN? If you think it's simply because there's not enough bandwidth in the WAN, then *you don't know jack about network performance*.

Consider this real-life example of a large bank with headquarters in Europe and operations in North America. The bank's CIO was getting big-time heat from a business unit with European users trying to access an important application from across the pond. Performance was horrible. Under pressure, the CIO ordered his trusted network operations manager to fix the problem. The network manager dutifully investigated, measuring the transatlantic link utilization and router queue statistics. He reported that there were absolutely no problems with the network, as it was only 3 percent utilized. "I don't care, double the bandwidth!" the CIO ordered. The network manager complied, installing a second OC-3 link. And, guess what? The network went from 3 percent to 1.5 percent utilized, and application performance was still horrible. That CIO *didn't know jack about network performance*.

In this example, and as is all too often the case, IT managers attribute poor application performance over the WAN to inadequate bandwidth, as bandwidth is generally equated with the "speed" of the network. Yet, while bandwidth plays an important role in limiting the overall maximum amount of data that can be moved from place to place in a fixed amount of time, it is only one of several important factors that affect an application's performance. Other key factors—network latency, transport protocol buffer management, congestion control dynamics, and the design of the application's protocol itself—can impact performance so much that they can completely eliminate the useful benefits of upgrading a network to have greater bandwidth or "capacity."

THE SLIDING WINDOW PROTOCOL

To understand why performance is more than a bandwidth problem, let's start by decomposing how the underlying transport protocol works. As you probably know, a typical application on the Internet will somehow employ

you don't know jack about Network Performance

TCP/IP for moving data. TCP is a sort of advanced form of a *sliding window* protocol. It operates by sending one or more packets of information from a sender to a receiver. When packets are successfully delivered, the receiver sends a reply ACK (acknowledgment packet) back to the sender to indicate that it successfully received what the sender sent. In a sliding window protocol, the *window size* is the amount of data a sender is allowed to have "in" the network without having yet received an acknowledgment for it.

Given that most data transfers over the Internet or across an enterprise WAN use a sliding window protocol, what can we reason about the performance of such transfers? Assume for a moment that a packet is always a fixed size, *p* bytes. We can express the window size in terms of some number of packets, *k*. The window size, *w,* is simply the product of these two: $w=kp$. Now, if we are worried about how much data we can move in a certain amount of time over a network, we are really worried about how many windows can be moved in some amount of time.

Thus, to determine the rate of a transfer, we need to compute the rate at which windows of data can be transmitted across the network—that is, we need to understand how much time it takes to move *w* bits. This, in turn, requires knowledge of the time required to move *any* information (even one bit) between the sender and receiver, which is called the network *latency,* or delay. Although the latency can be measured as either one-way (source to destination) or two-way (source to destination and back), it is generally the two-way delay we are concerned with. This value is called the RTT (round-trip time). Note that the RTT, which can vary over time, is rarely twice the one-way delay time, but it is frequently approximated this way.

Latency arises from several distinct sources. First, there is the *propagation delay*, generally controlled by nature. For coaxial cable, bits propagate at between 60 and 90 percent of the speed of light, while for RF and optical, they propagate at effectively the speed of light.

Another component of latency is *transmission delay*, which derives directly from the bandwidth of the underlying communication link. For example, sending a 100-byte packet over a 100-byte-per-second network link requires one second to inject the entire packet into the network. Note that the transmission delay is just the packet injection time; it says nothing about the additional time required for a packet to reach its destination.

The final component of latency is the *queuing delay*, which represents the time a packet must wait in a holding area (e.g., a queue in a router) while other packets are transmitted, until the first bit of that packet makes its way onto the communication link. In many cases, including the Internet, the queuing delay is not easily measured and varies rapidly.

The RTT is the sum of all these latency components—the propagation delay, the queuing delay, and the transmission delay—for each communication link along both the forward and reverse path between sender and receiver.

Now that we understand all the components of the RTT, we can look at how it impacts the performance of a data transfer. Given that TCP provides *reliable* communications over the potentially lossy IP network layer, there must be some way to recover from unexpected packet loss. To handle such losses, TCP uses packet *retransmissions*, where the sender retransmits packets that have been lost in the network. To implement retransmissions, TCP must keep a copy of any data it has injected into the network that it doesn't yet know has been received properly by the receiver. Said another way, TCP must buffer a copy of a window's worth of data (*w* in our terminology) until it has received an acknowledgment for it. Thus, the window size is never permitted to grow larger than the amount of buffer space (i.e., memory) available at the sender. In real systems, there is usually a fixed pool of memory reserved for each TCP connection (frequently called the *socket buffer*, a value that applications can modify).

TCP PERFORMANCE

Given the concepts of the retransmission window and the RTT, we can now reason about the performance of TCP's sliding window communication scheme. Let's suppose the sender uses a window size of 100 bytes. As such, the sender is able to inject 100 bytes into the network but then must wait until it receives the corresponding ACK for that data. At an absolute minimum, the sender must wait one RTT for this to happen, as there is no way it could receive an ACK for a message headed to the destination earlier than the time it takes to reach the destination and return. At that point, the sender can inject another 100 bytes into the network.

How does this behavior translate into throughput? In steady state, the scheme moves *w* bytes of information every RTT amount of time, which in simple mathematical terms means the throughput is *w* / RTT. You probably remember from high school algebra that the function $y=1/x$ defines a hyperbola, implying that throughput decays *hyperbolically* with increasing RTT. In other words, as the RTT gets large, the throughput degrades rapidly.

So, why not just set the window size (*w*) to a correspondingly larger value by increasing packet size (*p*) or number of packets (*k*) to achieve better throughput? Protocol designers long ago addressed this question and concluded that there was, in fact, an "optimal" window size for a given network environment that maximizes network throughput. The best window size turns out to be the one that causes the sender to entirely "fill the pipe." If we think of the path between sender and receiver as a pipe, we want to compute its volume by taking its "width" (its throughput, which is really more like its cross-sectional area, in bits per second) times its "length" (the one-way delay, in seconds). This yields the BDP (*bandwidth-delay product*) and is measured in bits. In other words, the pipe size represents the number of bits that are physically in transit "on the wire" between the sender and receiver.

The optimal window size allows the sender to keep transmitting such that the network is busy and full, carrying data all of the time (hence, never idle and wasting time). This turns out to be a pipe full of bits in the forward direction, then another pipe full while waiting for the ACK for the first packet in the window to come back. Once that first ACK arrives, the sender slides its window forward and can thus send another packet. If the window is just big enough, ACKs continue to arrive back at the sender just in time to continually send just the right number of packets to sustain optimal throughput. You can think of this as packets riding on the top of a full conveyor belt, with the ACKs returning on the bottom.

Thus, achieving optimal throughput merely requires that the sender set its window to the RTT times the network bandwidth. This should be easy, right? Unfortunately, there are many reasons the sender often ends up using a suboptimal window. For example, as mentioned previously, the sender's buffer may be less than this optimal size, imposing an operational limit.

Another reason for a suboptimal window is that the sender may impose *congestion control* upon itself. This is a long-standing area of research in TCP, and new methods appear regularly. Suffice it to say that the congestion control procedure is an algorithm that restricts the TCP window size (by reducing *k* in our equation) to a smaller value than it would otherwise use to avoid overloading, or congesting, the network. When the network becomes overloaded and drops packets, TCP reduces its window size in response, resulting in an effective reduction in its overall sending rate. For networks where packet loss is a strong indicator of congestion, this procedure works well and causes each sender to adjust its window so it receives some share of the network bandwidth. For other networks (e.g., wireless or satellite), where losses may be a result of data corruption instead of congestion, this technique can artificially limit the throughput of TCP. This issue has also been an area of intense research interest.

Another potential bottleneck can arise on the receiver end. Even if the sender were capable of using the optimal window size, the receiver might not have enough memory available to hold and process all of that data at once. To deal with this flow control problem, TCP includes

The best window size
turns out to be the one that causes the sender to entirely "fill the pipe."

in each packet a value called the *window advertisement*, which essentially signals to the sender how much additional data the receiver is willing to accept. If the receiver's buffer becomes full or is too small, the receiver reduces the value signaled in the window advertisement to a manageable level. This value can end up being less than the optimal window size, thereby degrading performance.

In addition, a peculiarity of the original TCP design can cause the advertised window to be smaller than desired. Because only 16 bits were allocated to the window advertisement field in the TCP header, the maximum possible window was limited to 65,535 bytes, a significant impairment to performance in so-called "large, fat networks" (those with large bandwidth-delay products). Fortunately, in 1992, the issue was addressed and solved in an interesting way by RFC 1323. The technique involves scaling the window value carried in the TCP header by multiplying it by 2^n for some value n, called the *window scale*. The value of n is exchanged between the two TCP endpoints at connection establishment time. As n is allowed a maximum value of 14, the largest possible window that TCP can represent when window scaling is used is 2^{30} bytes (1 GB), considerably larger than the original 65,535-byte maximum. This capability is often called TCP with "large windows" and is now automatically negotiated by modern TCP implementations.

APPLICATION PERFORMANCE

All of the limitations to window size discussed so far are a result of the transport protocol implemented in the end systems. More issues arise when we look at application performance. For example, although applications are generally free to choose the sending and receiving buffer sizes, they often don't and simply rely on the default sizes provided by the operating system. The "knobs" controlling the buffer sizes are often hidden behind layers of software or middleware that the application programmer has no control over. Even if the application deliberately configures the buffers, the programmer must choose some *a priori* value, but the optimal size cannot be known at development time since different end hosts communicating over different network paths each need a different optimal value. Moreover, esoteric details about how and when an application sets these buffer sizes compared with other connection setup functions can cause the large window negotiation to fail in subtle ways that the programmer may not catch. Finally, depending on the particular application, making buffers unnecessarily large can increase the overall end-to-end delay and hurt, instead of help, application performance. Doing so can also increase the memory pressure on a busy server, discouraging application programmers from using such large buffers.

Even when all the buffers are set optimally, when the network has plenty of bandwidth, and when TCP congestion control works flawlessly, application performance over the wide area can still suffer significantly because of the application protocols themselves. Every TCP-based application must implement some form of a higher-level messaging protocol on top of the reliable TCP connection. Imagine for a moment how TCP acts when such an application protocol ceases to supply data for transmission on the network. Naturally, the TCP itself stops sending.

Although TCP has various techniques and options to overcome window limitation, it is powerless to overcome a similar problem in the application. If an application's protocol involves requests and responses, and if it fails to implement any way of "keeping the network full" (e.g., by allowing multiple outstanding requests), it can be driven into a condition where it is able to process only one request per RTT. This sort of "chatty" application behavior results in many inefficient back-and-forth exchanges between the end hosts and causes performance to degrade hyperbolically with increasing RTT, just like the throughput of a sliding window protocol. This can be a serious consequence, indeed, for users forced to use applications never designed for large RTT or, more generally, large BDP environments.

Systems may appear to run fine in a LAN environment, yet be **unbearably slow in a WAN.**

It's easy to see how these applications have become commonplace. In a nutshell, it's hard to build an application protocol that *doesn't* work well over a LAN. Consider a LAN based on 100-Mbps Ethernet. LANs generally span only a limited area and incur limited overall delay. Assume the RTT on an Ethernet network is 0.1 ms (.0001 second). The BDP is then about 0.01 MB = 1.3 KB. For an Ethernet packet of 1,500 bytes, the 1.3 KB represents about one packet. It is easily represented by TCP without window scaling and is almost certainly adequately provided for by default buffer size allocations. In fact, because the optimal window size is only about one packet for this small RTT, even a poorly engineered application may perform well. Applications of this kind are in some ways the most troubling, because they will perform significantly worse when moved from a LAN to a WAN, where the RTTs are larger.

LAN VS. WAN

If we compare the LAN scenario with a high-speed WAN scenario, the situation looks somewhat different. Assuming that the RTT is now 80 ms and the bandwidth is 1 Gbps, we have a BDP of about 10 MB. With Ethernet 1,500-byte packet framing, that's about 7,100 packets. In this case it could be challenging to make full use of the network unless care is taken at each layer (application through transport) to ensure full utilization of the network's throughput capacity.

You can check out all these concepts using Ethereal or your favorite packet capture and analysis tool. Set it up to look for traffic on TCP port 445, which allows you to watch the Windows file sharing protocol in action. Then, do something as simple as opening a document from a network file share using Microsoft Word. Go back to Ethereal, which will decode the file system protocol commands, and eyeball the trace. What you'll see will probably surprise you: hundreds, if not thousands, of commands going back and forth between your Word application and the file server, just to open and load a file. Depending on the document, you might see Word open and close it several times, read different parts of the file in nonsequential order, read the same data more than once, copy the data to a temporary file, retrieve the same meta-information about the file and the directory multiple times, and so forth.

Every one of these operations is executed sequentially, requiring a round trip across the network. On your LAN this is no big deal: 1,000 round trips times 0.1 ms per trip is a tenth of a second. Over an 80-ms WAN link, however, 1,000 round trips is more than a minute. Even if you upgrade the WAN circuit to a 45-Mbps DS3 or a 155-Mbps OC-3, it will still take more than a minute to open that document.

If you take away just one concept from this article, remember that network performance is more than bandwidth. The seemingly simple problem of developing applications with good throughput performance may not be so simple, after all. Bad performance creeps up on us for many different reasons: physical factors (RTT, packet loss), transport protocol factors (limited buffers, limited ability to encode a large window, limited receiving application run rate), and application layer protocol dynamics. A poor implementation of either the transport or application layers can easily lead to poor performance. To further frustrate us, entire systems may appear to run fine in a LAN environment, yet be unbearably slow in a WAN or other high-delay environment.

This article aims to help shed some light on these sometimes-complex interactions, so that end users can benefit from deeper understandings of these issues among application and middleware software developers. If anything, you can now say you *do* know jack about network performance. Q

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

KEVIN FALL has held numerous research and teaching positions in networking at the University of California in San Diego, Berkeley, and Santa Cruz; at MIT; and at the Lawrence Berkeley National Laboratory. He was co-founder of NetBoost Corporation (now an Intel company). Since 2000, he has led the Delay Tolerant Networking (DTN) research effort at Intel and served as chair of the Delay Tolerant Networking Research Group (DTNRG), part of the Internet Research Task Force, a companion organization of the Internet Engineering Task Force. As chair of DTNRG, he has been involved in the formulation of DARPA's Disruption Tolerant Networking program. He received his B.A. in computer science from the University of California, Berkeley, and his M.Sc. and Ph.D. in computer science from UC, San Diego.
STEVE McCANNE co-founded Riverbed Technology in 2002 and serves as its CTO. Prior to Riverbed, he co-founded FastForward Networks, which he later sold to Inktomi Corporation. Before embarking on his business career, McCanne served on the faculty in electrical engineering and computer science at the University of California, Berkeley, where he taught and conducted research in Internet protocols and systems. He received the 1997 ACM Doctoral Dissertation Award for his Ph.D. work at U.C. Berkeley on layered multicast video compression and transmission. In 2002 MIT's *Technology Review* named him one of the top 100 young technology innovators for his Internet-related contributions. From 1988 to 1996 he was a member of the Network Research Group at the Lawrence Berkeley National Laboratory, where he developed a number of widely used technologies. This work included protocol development that now forms the foundation of today's Internet standard for streaming media.