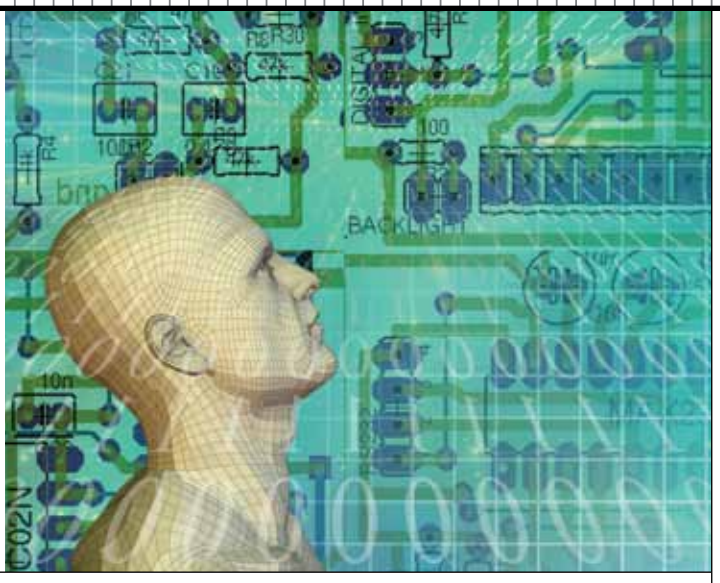
Van Jacobson: Getting NSFNet off the Ground



Charles Severance
University of Michigan

Approximately 20 years ago, TCP/IP got a big boost from Van Jacobson and his team in the form of the slow-start algorithm.

For those of us closely watching the earliest implementation of the National Science Foundation Network (NSFNet) in the mid to late 1980s, it felt like witnessing a baby sea turtle hatch far from the water and then sprint toward the safety of the ocean, all the while dodging hungry seagulls.

Telephone companies wanted to maintain their monopoly on long-distance data transport, using leased-line technology in the US and X.25 data networks in Europe and elsewhere. Well-developed proprietary networks were already available from commercial vendors, including IBM's SNA and DECnet from Digital Equipment (DEC).

The stakes were high. Through patents, licenses, and usage fees, telephone companies could extract profit from every aspect of distributed computing, and if NSFNet succeeded, it would prevent those companies from providing a full-stack solution to wide area data networking. Companies that owned the core network technology also owned the edges of that network and could ultimately set the agenda for all hardware and software connected to it.

If NSFNet had selected DECnet as its underlying protocol instead of TCP/IP, in all likelihood, instead of carrying an Apple or Android phone right now, you would have a DEC phone in your pocket running the VMS operating system. If this seems fanciful or unlikely, look up the "AT&T You Will" advertisements from 1993 on YouTube. The future would have been very different if proprietary forces found a way to own the Internet's core in the late 1980s.

The seagulls were completely aware of this situation, and they knew what was at stake. It's why such a large flock gathered over this lone baby sea turtle sprinting for the safety of the ocean way back then.

When the NSF decided that its first national network would adopt the TCP/IP protocol, we all breathed a sigh of relief. At least with TCP/IP, university computer scientists and other open-minded technologists would collectively own the network technology. We could make the network a neutral ground and move innovation forward under our own control without the sluggishness of proprietary companies that worried most about reducing their market share. Our baby sea turtle had successfully dodged the seagulls and made it into the ocean.

But soon after NSFNet first deployed, the previously solid TCP/IP protocol started to experience extended outages. I talked to Van Jacobson of PARC, a Xerox company, about the causes of those early problems and what it took to get NSFNet running smoothly again (see www.computer.org/computingconversations for our full discussion).

TCP/IP FAILS TO SCALE

One reason for our excitement about TCP/IP was that it allowed us to write a limitless number of applications for the network. Researchers could develop low-latency applications such as instant messaging or remote log in, medium-bandwidth applications such as e-mail, or latency-tolerant high-bandwidth applications such as bulk file transfer and mix them all up on the same network infrastructure.

 Published by the IEEE Computer Society

Proprietary networks like SNA and DECnet supported limited use cases via carefully crafted implementations that made sure each application got the network access it needed. By deeply understanding each particular type of traffic, these proprietary networks could maintain balance between the needs of competing applications.

Although this functioned well, it meant that each new use case on these proprietary networks required careful engineering to fit it in with the rest to avoid destabilizing the overall network. This approach kept network performance consistent but at the cost of extremely slow innovation.

TCP/IP networks were truly layered in that the lower levels didn't discriminate based on the traffic-generation application. The hope was to support a wide range of applications without requiring special case tuning for each new application.

As NSFNet rolled out, it proved to be an immediate success, with the growth of traffic and number of connected hosts increasing at 15 percent per month—in fact, NSFNet's size doubled every five months. This led to some problems, as described by Jacobson:

> We were tying together 10-megabit campus infrastructure with 56-kilobit wires, and it was wildly popular because people who couldn't talk suddenly could. They're sending e-mails, moving huge files, and everybody is really excited. But anyone on any of those campuses [connected to NSFNet] could oversubscribe the [backbone] network by a factor of 1,000, so we had a lot of packets piling up and getting dropped.

Jacobson confronted the failure of using TCP/IP over slow leased lines to extend campus networks nearly every day:

> At the time, I was a researcher at Lawrence Berkley Lab, which is in the hills above the Berkeley campus, and I was also teaching on the Berkeley campus. Even in the mid-1980s, for every class, there was a message [netnews] group that was set up, all the assignments would be put online. I was trying to get course materials from my office in LBL to a machine in Evans Hall at Berkeley, and there was zero throughput in the network. It was one packet every 10 minutes or so.

The same problem was happening all across NSFNet and anywhere else where TCP/IP was used for wide area networking. It seemed that almost as soon as a new connection was added, it worked for a few days until more users joined, then everything ground to a halt as the link became completely clogged with traffic. And no one knew why.

This failure to scale rekindled the debate as to whether TCP/IP was a suitable protocol for such a complex and difficult task.

This failure to scale rekindled the debate as to whether TCP/IP was a suitable protocol for such a complex and difficult task. Many suggested scrapping TCP/IP and switching to DECnet. The physics community had already built HEPNet, a wide area network based on DECnet protocols. It seemed to work quite well and could smoothly balance low-latency access such as remote login with high-bandwidth access file transfer activity. Perhaps TCP/IP just wasn't up to the task. Perhaps academics couldn't build robust networking protocols, and using networking software from a commercial vendor was a better option.

Jacobson and the teams that built the TCP/IP implementation in Berkeley Unix weren't about to give up. This was simply a small flaw in TCP/IP that needed fixing. They knew that TCP/IP was well engineered and had performed exceedingly well in other networks with diverse types of application traffic but with fewer users. Their first hypothesis was that there must be a bug in the software:

> I went down and talked to Mike Karels, who was heading the BSD group, the people that developed Berkeley Unix. He was getting reports of these problems from all over the country. We talked for a long time about what was going wrong. Is there a mistake in the protocol, a mistake in the protocol implementation? [TCP/IP] was working on smaller-scale tests, and then it suddenly fell apart. We struggled for three or four months, just going through the code, writing tools to capture packet traces, and looking at the packet traces, trying to sort out what was breaking.

They spent months searching for an elusive bug in the design or implementation that they would ultimately never find. In a moment of desperation, they decided that perhaps their initial assumption was simply wrong:

> I remember the two of us were sitting in Mike's [Karels] office after we had been pounding our head against the wall for literally months and one of us said, You know, the reason that I can't figure out why it is breaking is that I don't understand how it ever worked. We're sending these bits out at 10 megabits—they're zipping across campus—and they're running into this 56-kilobit wire. We expect them to go through that wire and pop out the other side. How could that function? That turned out to be the crucial starting point. At that point, we started saying, What is it about this protocol that makes it work? How does it deal with all those bandwidth changes? How does it deal with the multiple hops?

They switched their focus from searching for a bug to measuring TCP/IP's behavior when it functioned properly across a wide area network with a combination of fast and slow network links. TCP/IP wants to presend more than one packet, to fill the pipeline with packets and maximize its use of available bandwidth. The sending system starts by sending several packets (initial window size) and then waits to send more until it receives acknowledgments (ACKs) from the remote system.

If the initial number of packets is too small, it isn't possible to efficiently use high-speed connections, and if the initial number of packets is too large, the packets pile up at the slowest connection, and the system drops them. At some point, the sending system detects a timeout and resends the packets, which only makes the problem worse. According to Jacobson:

> If you turn them on, suddenly you get in this repetitive failure mode where you saturate the buffering that was available at some gateway; when you retransmit, you do the same thing again. So we were always losing packets. But if you turned it on more gradually, you wouldn't overload the buffering: you would get enough of a clock going so that you could control the amount of backlog to fit the available buffer even as the number of packets in flight increases. You would start with a sporadic clock, but you would eventually fill in the detail and get a per-packet clock. ... The hard part in TCP is not in keeping it running, it's in getting it started. Because once you have it running, a clock tells you exactly what to do.

Jacobson called it the "slow-start algorithm." If we could get every TCP/IP implementation to implement the slow-start algorithm we could get our baby turtle back off the beach and back into the ocean for good.

IMPLEMENTING SLOW START

An absolutely critical element of the slow-start algorithm is that every computer in the network needs to implement it in a similar manner. If some operating systems implemented slow start and others didn't, those computers that sent packets more aggressively would get better throughput than the "responsible," slow-start-using computers. There was concern that if some but not all of the TCP/IP implementations used slow start and new market entrants didn't, it would lead to repeated network collapse and endless arguments as to who was at fault.

An absolutely critical element of the slow-start algorithm is that every computer in the network needs to implement it in a similar manner.

Time was of the essence, as new TCP/IP implementations were under active development in many companies and universities. Thankfully, according to Jacobson,

> At that time [in 1988], there were like four implementations on the market. There was Berkeley Unix, there was the MIT PC/TCP [for Windows systems], there was a BBN [Bolt, Beranek, and Newman] one for Butterfly and IMP [Interface Message Processors; an early router] systems, and there was a Multics one.

Once the slow-start design was in place, the team quickly started to develop fixes to the Berkeley Unix operating system to demonstrate the algorithm. Up to that point, they had made lots of changes deep inside the Unix kernel to instrument the network protocols and develop models of what was going wrong. Before they knew what the problem was, these kernel modifications weren't particularly well thought out or elegant:

> I had this horrible driver hack that would let us snarf packets from the kernel. You set what you wanted to snarf was by using adb [a debugger] to patch the kernel [while it was running] with the ports you wanted to look at. The driver would capture those into a circular buffer, and you would read kernel memory to pull those packets out. Craig Leres and Chris Torek, who were working in my group at LBL and were both long-time kernel hackers, were embarrassed at [my kernel hacks] so they put together a really nice, clean driver called the Berkeley Packet Filter [BPF] that would let you pull packets out of the kernel via a very efficient ioctl() interface.

Once the first version of the slow-start algorithm seemed to be working on the Berkeley computers, it was time to share the code with other schools running Berkeley Unix to validate the idea and determine if the patches actually solved the problem. Jacobson and Karels sent the patches to the TCP/IP mailing list, and an eager group of software developers and system administrators started furiously testing because the problem on their networks was so acute. The initial results weren't very promising—installing the patches crashed the system. But working with the other developers, Jacobson and his team quickly improved the code in several subsequent releases over the next 24 hours:

> After about a day, we got a version that didn't immediately panic [crash] and then started working on the actual algorithm with a little bit of tuning to make sure that it actually helped all the time and didn't do any harm. It was completely a community effort, and when the community was saying that this mostly does good and never seems to do harm, that was what

> Mike [Karels] needed to put it into the [Berkley Unix] kernel.

It took about a month between the first release of the slow-start patches and when the code was of sufficient quality to be included in the official Berkley Unix release. It eventually debuted publicly as a core capability of the BSD Unix 4.3 (Tahoe) release in June 1988. The other major TCP/IP implementations quickly followed suit, and in a remarkably short time, the slow-start algorithm was virtually universal.

ALL'S WELL

Although TCP/IP engineering and improvement is nearly continuous, the slow-start algorithm solved the last major engineering issue that caused the entire Internet "to crash." With billions of computers connecting and millions more coming on every month (including several in your pockets or purses), it's comforting to know that they all come from the factory with the slow-start algorithm built in.

The algorithm's very simple concept allows a TCP/IP implementation to gauge the bandwidth for each connection by starting out a little tentatively, and once it gets a sense of the available throughput for the connection, it quickly expands its window of in-flight packets to make best use of that throughput.

Interestingly, as the routers that make up the Internet's fabric become faster and have more memory, they're storing more in-flight packets longer and then forwarding them later, when the TCP/IP protocol would suggest that the packets be dropped. When this happens, the packets that are successfully transmitted after a delay have a slower apparent round-trip time. When your system sees this slower round-trip time, the slow-start algorithm starts backing off because it thinks there's a bottleneck somewhere between the sender and receiver, which leads to an unnecessary reduction in throughput. A router's proper behavior is to discard packets that have been stuck in a router too long to properly communicate the nature of their network communication to the sending and receiving systems.

Van Jacobson continues to research the best way to use the resources in packet-switched networks. His latest thinking is content-centric networking (www.parc.com/work/focus-area/content-centric-networking/), which puts the vast amounts of memory and processing power found in backbone routers to good use instead of causing problems like buffer bloat.

Increasingly, we're streaming content from places like YouTube, Netflix, and live TV over the Internet. IP Multicast has long been a hoped-for solution in this space, but it has proven difficult to completely synchronize all sources and destinations connected to a common stream and adjust to varying network connection speeds and congestion conditions. In addition, Multicast operates at the IP (packet) level and not at the TCP (stream) level, thus it can't take advantage of knowing how packets fit together to form continuous content.

Although it's a gross oversimplification, content-centric networking uses the buffer space already present in routers to provide the ability to efficiently access streams of content from a single source going to multiple destinations. Content-centric networking naturally handles widely varying network connection throughputs as well as relaxes the need to send every single packet to all locations synchronously.

We plan to visit Jacobson again in a future article to explore content-centric networking in more detail.

***Charles Severance**, editor of the Computing Conversations article and* Computer*'s multimedia editor, is a clinical associate professor and teaches in the School of Information at the University of Michigan. You can follow him on Twitter @drchuck or contact him at csev@umich.edu.*