High bandwidth, low latency, and multihoming challenge the sockets API.

BY GEORGE V. NEVILLE-NEIL

Whither Sockets?

ONE OF THE most pervasive and longest-lasting interfaces in software is the sockets API. Developed by the Computer Systems Research Group at the University of California at Berkeley, the sockets API was first released as part of the 4.1c BSD operating system in 1982. While there are longer-lived APIs—

for example, those dealing with Unix file I/O—it is quite impressive for an API to have remained in use and largely unchanged for 27 years. The only major update to the sockets API has been the extension of ancillary routines to accommodate the larger addresses used by IPv6.²

The Internet and the networking world in general have changed in very significant ways since the sockets API was first developed, but in many ways the API has had the effect of narrowing the way in which developers think about and write networked applications. This article briefly examines some of the conditions present when the sockets API was developed and considers how those conditions shaped the way in which networking code was written. Later, I look at ways in which developers have tried to get around some of the inherent limitations in the API and address the future of sockets in a changing networked world.

The two biggest differences between the networks of 1982 and 2009

are topology and speed. For the most part it is the increase in speed rather than the changes in topology that people notice. The maximum bandwidth of a commercially available long-haul network link in 1982 was 1.5Mbps. The Ethernet LAN, which was being deployed at the same time, had a speed of 10Mbps. A home user—and there were very few of these—was lucky to have a 300bps connection over a phone line to any computing facility. The round-trip time between two machines on a local area network was measured in tens of milliseconds, and between systems over the Internet in hundreds of milliseconds, depending of course on location and the number of hops a packet would be subjected to when being routed between machines. (See page 52 for a look at the early Internet.)

The topology of networks at the time was relatively simple. Most computers had a single connection to a local area network; the LAN was connected to a primitive router that might have a few connections to other LANs and a single

ARPANET GEOGRAPHIC MAP, OCTOBER 1980

(Note: This map does not show ARPA's experimental satellite connections.)
Names shown are IMP names, not (necessarily) host names.
Source: <http://personalpages.manchester.ac.uk/staff/m.dodge/cybergeography/atlas/arpanet4.gif>.

connection to the Internet. For one application to another application, the connection was either across a LAN or transiting one or more routers, called IMPs (Internet message passing).

History of Sockets

The model of distributed programming that came to be most popularized by the sockets API was the client/server model, in which there is a server and a set of clients. The clients send messages to the server to ask it to do work on their behalf, wait for the server to do the work requested, and at some later point receive an answer. This model of computing is now so ubiquitous it is often the only model with which many software engineers are familiar. At the time it was designed, however, it was

seen as a way of extending the Unix file I/O model over a computer network. One other factor that focused the sockets API down to the client/server model was that the most popular protocol it supported was TCP, which has an inherently 1:1 communication model.

The sockets API made the client/server model easy to implement because of the small number of extra system calls that programmers would need to add to their non-networked code so it could take advantage of other computing resources. Although other models are possible, with the sockets API the client/server model is the one that has come to dominate networked computing.

Although the sockets API has more entry points than those shown in Table

1, it is those five shown that are central to the API and that differentiate it from regular file I/O. In reality the `socket()` call could have been dropped and replaced with a variant of `open()`, but this was not done at the time. The `socket()` and `open()` calls actually return the same thing to a program: a process-unique file descriptor that is used in all subsequent operations with the API. It is the simplicity of the API that has led to its ubiquity, but that ubiquity has held back the development of alternative or enhanced APIs that could help programmers develop other types of distributed programs.

Client/server computing had many advantages at the time it was developed. It allowed many users to share resources, such as large storage arrays and expensive printing facilities, while keeping these facilities within the control of the same departments that had once run mainframe computing facilities. With this sharing model, it was possible to increase the utilization of what, at the time, were expensive resources.

Three disparate areas of networking are not well served by the sockets API: low-latency or real-time applications; high-bandwidth applications;

Table 1: Socket API systems calls.

<code>socket()</code>	Create a communication endpoint
<code>bind()</code>	Bind the endpoint to some set of network-layer parameters
<code>listen()</code>	Set a limit on the number of outstanding work requests
<code>accept()</code>	Accept one or more work requests from a client
<code>connect()</code>	Contact a server to submit a work request

and multihomed systems—that is, those with multiple network interfaces. Many people confuse increasing network bandwidth with higher performance, but increasing bandwidth does not necessarily reduce latency. The challenge for the sockets API is giving the application faster access to network data.

The way in which any program using the sockets API sends and receives data is via calls to the operating system. All of these calls have one thing in common: the calling program must repeatedly ask for data to be delivered. In a world of client/server computing these constant requests make perfect sense, because the server cannot do anything without a request from the client. It makes little sense for a print server to call a client unless the client has something it wishes to print. What, however, if the service provided is music or video distribution? In a media distribution service there may be one or more sources of data and many listeners. For as long as the user is listening to or viewing the media, the most likely case is that the application will want whatever data has arrived. Specifically requesting new data is a waste of time and resources for the application. The sockets API does not provide the programmer a way in which to say, “Whenever there is data for me, call me to process it directly.”

Sockets programs are instead written from the viewpoint of a dearth of, rather than a wealth of, data. Network programs are so used to waiting on data that they use a separate system call, `socket()`, so that they can listen to multiple sources of data without blocking on a single request. The typical processing loop of a sockets-based program isn’t simply `read()`, `process()`, `read()`, but instead `select()`, `read()`, `process()`, `select()`. Although the addition of a single system call to a loop would not seem to add much of a burden, this is not the case. Each system call requires arguments to be marshaled and copied into the kernel, as well as causing the system to block the calling process and schedule another. If there were data available to the caller when it invoked `select()`, then all of the work that went into crossing the user/kernel boundary was wasted because a `read()` would have returned data immediately. The

Sockets programs are written from the viewpoint of a dearth of, rather than a wealth of, data.

constant check/read/check is wasteful unless the time between successive requests is quite long.

Solving this problem requires inverting the communication model between an application and the operating system. Various attempts to provide an API that allows the kernel to call directly into a program have been proposed but none has gained wide acceptance—for a few reasons. The operating systems that existed at the time the sockets API was developed were, except in very esoteric circumstances, single threaded and executed on single-processor computers. If the kernel had been fitted with an up-call API, there would have been the problem of which context the call could have executed in. Having all other work on a system pause because the kernel was executing an up-call into an application would have been unacceptable, particularly in timesharing systems with tens to hundreds of users. The only place in which such software architecture did gain currency was in embedded systems and networked routers where there were no users and no virtual memory.

The issue of virtual memory compounds the problems of implementing a kernel up-call mechanism. The memory allocated to a user process is virtual memory, but the memory used by devices such as network interfaces is physical. Having the kernel map physical memory from a device into a user-space program breaks one of the fundamental protections provided by a virtual memory system.

Attempts to Overcome Performance Issues

A couple of different mechanisms have been proposed and sometimes implemented on various operating systems to overcome the performance issues present in the sockets API. One such mechanism is *zero-copy sockets*. Anyone who has worked on a network stack knows that copying data is what kills the performance of networking protocols. Therefore, to improve the speed of networked applications that are more interested in high bandwidth than in low latency, the operating system is modified to remove as many data copies as possible. Traditionally, an operating system performs two copies for each packet received by the system.

Table 2: APIs added by SCTP.

API	Explanation
sctp_bindx()	Bind or unbind an SCTP socket to a list of addresses
sctp_connectx()	Connect an SCTP socket with multiple destination addresses
sctp_generic_recvmmsg()	Receive data from a peer
sctp_generic_sendmsg(), sctp_generic_sendmsg_iov()	Send data to a peer
sctp_getaddrlen()	Return the address length of an address family
sctp_getassocid()	Return an association ID for a specified socket address
sctp_getpaddrs(), sctp_getladdrs()	Return list of addresses to caller
sctp_peeloff()	Detach an association from a one-to-many socket to a separate file descriptor
sctp_sendx()	Send a message from an SCTP socket
sctp_sendmsgx()	Send a message from an SCTP socket

The first copy is performed by the network driver from the network device's memory into the kernel's memory, and the second is performed by the sockets layer in the kernel when the data is read by the user program. Each of these copy operations is expensive because it must occur for each message that the system receives. Similarly, when the program wants to send a message, data must be copied from the user's program into the kernel for each message sent; then that data will be copied into the buffers used by the device to transmit it on the network.

Most operating-system designers and developers know that data copying is anathema to system performance and work to minimize such copies *within the kernel*. The easiest way for the kernel to avoid a data copy is to have device drivers copy data directly into and out of kernel memory. On modern network devices this is a result of how they structure their memory. The driver and kernel share two rings of packet descriptors—one for transmit and one for receive—where each descriptor has a single pointer to memory. The network device driver initially fills these rings with memory from the kernel. When data is received, the device sets a flag in the correct receive descriptor and tells the kernel, usually via an interrupt, that there is data waiting for it. The kernel then removes the filled buffer from the receive descriptor ring and replaces it with a fresh buffer for the device to fill. The packet, in the form of the buffer, then moves through the network stack

until it reaches the socket layer, where it is *copied* out of the kernel when the user's program calls `read()`. Data sent by the program is handled in a similar way by the kernel, in that kernel buffers are eventually added to the transmit descriptor ring and a flag is then set to tell the device that it can place the data in the buffer on the network.

All of this work in the kernel leaves the last copy problem unsolved, and several attempts have been made to extend the sockets API to remove this copy operation.^{1, 3} The problem remains as to how memory can be safely shared across the user/kernel boundary. The kernel cannot give its memory over to the user program, because at that point it loses control over the memory. A user program that crashes may leave the kernel without a significant chunk of usable memory, leading to system performance degradation. There are also security issues inherent in sharing memory buffers across the kernel/user boundary. There is no single answer to how a user program might achieve higher bandwidth using the sockets API.

For programmers who are more concerned with latency than with bandwidth, even less has been done. The only significant improvement for programs that are waiting for a network event has been the addition of a set of kernel events that a program can wait on. Kernel events, or `kevents()`, are an extension of the `select()` mechanism to encompass any possible event that the kernel might be able to tell the program about. Before the advent of `kevents`, a user program could call

`select()` on any file descriptor, which would let the program know when any of a set of file descriptors was readable, writable, or had an error. When programs were written to sit in a loop and wait on a set of file descriptors—for example, reading from the network and writing to disk—the `select()` call was sufficient, but once a program wanted to check for other events, such as timers and signals, `select()` no longer served. The problem for low-latency apps is that `kevents()` do not deliver data; they deliver only a signal that data is ready, just as the `select()` call did. The next logical step would be to have an event-based API that also delivered data. There is no reason to have the application cross the user/kernel boundary twice simply to get the data the kernel knows the application wants.

Lack of Support for Multihoming

The sockets API not only presents performance problems to the application writer, but also narrows the type of communication that can take place. The client/server paradigm is inherently a 1:1 type of communication. Although a server may handle requests from a diverse group of clients, each client has only one connection to a single server for a request or set of requests. In a world in which each computer had only one network interface, that paradigm made perfect sense. A connection between a client and server is identified by a quad of <Source IP, Source Port, Destination IP, Destination Port>. Since services generally have a well-known destination port (for example, 80 for HTTP), the only value that can easily vary is the source port, since the IP addresses are fixed.

In the Internet of 1982 each machine that was not a router had only a single network interface, meaning that to identify a service, such as a remote printer, the client computer needed a single destination address and port and had, itself, only a single source address and port to work with. While it did exist, the idea that a computer might have multiple ways of reaching a service was too complicated and far too expensive to implement. Given these constraints, there was no reason for the sockets API to expose to the programmer the ability to write a multihomed program—one that could manage

which interfaces or connections mattered to it. Such features, when they were implemented, were a part of the routing software within the operating system. The only way programs could get access to them was through an obscure set of nonstandard kernel APIs called a routing socket.

On a system with multiple network interfaces it is not possible, using the standard sockets API, to write an application that can easily be multihomed—that is, take advantage of both interfaces so if one fails, or if the primary route over which the packets were flowing breaks, the application would not lose its connection to the server.

The recently developed Stream Control Transport Protocol (SCTP)⁴ incorporates support for multihoming at the protocol level, but it is impossible to export this support through the sockets API. Several ad-hoc system calls were initially provided and are the only way to access this functionality. At the moment this is the only protocol that has both the capacity and user demand for this feature, so the API has not been standardized across more than a few operating systems. Table 2 shows the APIs that SCTP added.

While the list of functions in Table 2 contains more APIs than are strictly necessary, it is important to note that many are derivatives of preexisting APIs, such as `send()`, which need to be extended to work in a multihoming world. The set of APIs needs to be harmonized to make multihoming a first-class citizen in the sockets world. The problem now is that sockets are so successful and ubiquitous that it is very hard to change the existing API set for fear of confusing its users or the preexisting programs that use it.

As systems come to have more network interfaces built in, providing the ability to write applications that take advantage of multihoming will be an absolute necessity. One can easily imagine the use of such technology in a smartphone, which already has three network interfaces: its primary connection via the cellular network, a WiFi interface, and often a Bluetooth interface as well. There is no reason for an application to lose connectivity if even one of these network interfaces is working properly. The problem for application designers is that they want their code

As systems come to have more network interfaces built in, providing the ability to write applications that take advantage of multihoming will be an absolute necessity.

to work, with few or no changes, across a plethora of devices, from cellphones, to laptops, to desktops, and so on. With properly defined APIs we would remove the artificial barrier that prevents this. It is only because of the history of the sockets API and the fact that it has been “good enough” to date that this need has not yet been addressed.

High bandwidth, low latency, and multihoming are driving the development of alternatives to the sockets API. With LANs now reaching 10Gbps, it is obvious that for many applications client/server style communication is far too inefficient to use the available bandwidth. The communication paradigms supported by the sockets API must be expanded to allow for memory sharing across the kernel boundary, as well as for lower-latency mechanisms to deliver data to applications. Multihoming must become a first-class feature of the sockets API because devices with multiple active interfaces are now becoming the norm for networked systems.

Related articles on queue.acm.org

Code Spelunking: Exploring Cavernous Code Bases

George Neville-Neil

<http://queue.acm.org/detail.cfm?id=945136>

API Design Matters

Michi Henning

<http://queue.acm.org/detail.cfm?id=1255422>

You Don't Know Jack about Network Performance

Kevin Fall and Steve McCanne

<http://queue.acm.org/detail.cfm?id=1066069>

References

1. Balaji, P., Bhagvat, S., Jin, H.-W., and Panda, D.K. Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (sdp) over infiniband journal. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*.
2. Gilligan, R., Thomson, S., Bound, J., McCann, J., and Stevens, W. Basic Socket Interface Extensions for IPv6. RFC 3493 (Feb. 2003); <http://www.rfc-editor.org/rfc/rfc3493.txt>.
3. Romanow, A., Mogul, J., Talpey, T., and Bailey, S. Remote Direct Memory Access (RDMA) over IP Problem Statement. RFC 4297 (Dec. 2005); <http://www.rfc-editor.org/rfc/rfc4297.txt>.
4. Stewart, R., et al. Stream Control Transmission Protocol. RFC 2960 (Oct. 2000); <http://www.ietf.org/rfc/rfc2960.txt>.

George V. Neville-Neil (kv@acm.org) is the proprietor of Neville-Neil Consulting. He works on networking and operating systems code and teaches courses on program-related topics.