Article development led by queue.acm.org

The history of NFE processors sheds light on the trade-offs involved in designing network stack software.

BY MIKE O'DELL

Network Front-end Processors, Yet Again

"This time for sure, Rocky!"

—Bullwinkle J. Moose

THE HISTORY OF the network front-end (NFE) processor, best known as a TCP offload engine (or TOE), extends back to the Arpanet interface message processor and possibly before. The notion is beguilingly simple: partition the work of executing communications protocols from the work of executing the applications that require the services of those protocols. That way, the applications and the network machinery can achieve maximum performance and efficiency, possibly taking advantage of special hardware performance assistance. While this looks utterly compelling on the whiteboard, architectural

and implementation realities intrude, often with considerable force.

This article will not attempt to discern whether the NFE is a heavenly gift or a manifestation of evil incarnate. Rather, it will follow its evolution starting from a pure host-based implementation of a network stack and then moving the network stack farther from that initial position, observing the issues that arise. The goal is to offer insight into the trade-offs that influence the location choice for network stack software in a larger systems context. As such, it is an attempt to prevent old mistakes from being reinvented while harvesting as much clean grain as possible.

As a starting point, consider the canonical structure of a common workstation or server before the advent of multicore processors. Ignoring the provenance of the operating-system code, this model springs directly from the quintessential early to mid-1980s computer science department computer, the DEC VAX 11/780 with a 10Mb Ethernet interface with single-cycle direct memory access (DMA) ability and connected to a relatively slow 16-bit bus (the DEC Unibus).

Since there is only one processor, the network stack vies for the attention of the CPU with everything else running on the machine, albeit probably with the aid of a software priority mechanism that makes the network code "more equal than others."

When a packet arrives, the Ethernet interface validates the Ethernet frame cyclic redundancy check (CRC) and then uses DMA to transfer the packet into buffers used by the network code for protocol processing. The DMA transfers require only one local bus cycle for each 16-bit word, and on the VAX 11/780 the processor controller for the Unibus buffers 16-bit words into a single 32-bit transfer into main memory.

The TCP checksum is then calculated by the network code, the protocol state machinery conducts its business, and the TCP payload data is copied into "socket buffers" to await consumption

A VAX-11/780 from 1983 with 16MB of RAM, and the Ethernet interface containing a Motorola 68000 processor to handle the network traffic.

by the application program. When the read for the payload data happens, it is copied from the socket buffer into application process memory to be digested as required. That makes a total of four passes over the data in a single packet before the application gets a shot at using it. When networks were slow compared with memory bandwidth and processor speed, the data-copy inefficiency was considered minor compared with the joy of a working network stack, so it failed to provoke immediate improvement.

This base-case platform appears to be the origin of the folk theorem that “TCP needs one (VAX-)MIPS per

10 megabits/second of network performance.” The 10Mbps Ethernet can deliver about a megabyte/second of payload, so this is consistent with the other folk theorem of “one megabyte of memory per MIPS per megabyte of I/O.” Where this came from is difficult to pin down, but it is frequently credited to Gene Amdahl.

Now, let’s move this same model to PC hardware. For a long time, one of the principal distinctions between PCs and minicomputers was I/O performance. To be brutal, compared with its minicomputer forebears, the PC platform started life with almost no I/O capabilities. Over the life of the

PC platform, that conspicuous lack prompted major renovations of the PC’s I/O architecture. For the period of our interest, that progressed from the 16-bit ISA bus, to 32-bit PCI, and now PCI Express. For reasons too boring to explore here, for a very long time, packets moved from PC Ethernet cards into protocol processing buffers with a byte-copy operation performed by the CPU, upping the data-handling pass count to five.

The first significant improvement came when the raw-packet copy operation and TCP checksum were combined. Some network code tried to do this in software. As PCI Ethernet

cards developed efficient DMA hardware, some combined the TCP checksum generation with the copy operation, reducing the pass count to three. This clearly reduced CPU use for a given amount of TCP throughput and started the march to “protocol assist” services performed by network interfaces. (“If a little help is good, a lot of help should be better!”) Adapting the network stack code to exploit this new checksum capability was not trivial, but the handwriting on the wall made it clear that such evolution was likely to continue. Significant redesign of the network code had to be done to allow functions to move between hardware and software with greater ease in the future. This was genuine architectural progress, although it did not happen overnight.

A Success Disaster

With the explosion of the Web, performance demands on network servers skyrocketed. Processors and network interfaces were getting faster, and memory bandwidth strangulation was being solved. Gigabit Ethernet quickly became commonplace on server motherboards (and gamer desktop motherboards!). By this time, the cost of all those data copies was clearly unacceptable. Simply halving the number of copies would come close to doubling the sustainable transaction rate for many Web workloads.

This gave rise to the Holy Grail of what became known as *zero-copy TCP*. The idea was that programs written to exploit this new capability could have data delivered right into application buffers without any intervening copies (ignoring the possible exception of one efficient DMA transfer from the hardware). Clearly this would require some cooperation (or at least reduced antagonism) from designers of Ethernet interface hardware, but a working solution would win many hearts and minds.

The step from a zero-copy TCP network stack to a full-blown TCP offload engine looks pretty obvious at this point. It seems even more attractive given that many PC-based platforms were slow to exploit the multiprocessor abilities the PC was developing. (Whether it is multiple chips or multiple cores on one chip is largely irrelevant.) The

Simply moving data directly off the network wire into application buffers is not sufficient. The delivery of packets must be coordinated with all the other things the application is doing and all the other operating-system machinery behind the scenes.

ability to add a fast processor that can be applied entirely to protocol processing is certainly an attractive idea. It is, however, much more difficult to do in real life than it first appears on the whiteboard.

Simply moving data directly off the network wire into application buffers is not sufficient. The delivery of packets must be coordinated with all the other things the application is doing and all the other operating-system machinery behind the scenes. As a result, the network protocol stack interacts with the rest of the operating system in exquisitely delicate ways. Truth be told, this coordination machinery is the lion's share of the code in most stack implementations. The actual TCP state machine fits on a half page, once divorced of all the glue and scaffolding needed to integrate it with the rest of the system environment. It is precisely this subtle and complex control coupling that makes it surprisingly difficult to isolate a network protocol stack fully from its host operating system. There are multiple reasons why this interaction is such a rich breeding ground for implementation bugs, but one vast category is “abstraction mismatch.”

Because communications protocols inherently deal with multiple communicating entities, some assumptions must be made about the behavior of those entities. The degree to which those assumptions match between a host system and protocol code determines how difficult it will be to map to existing semantics and how much new structure and machinery will be required. When networking first went into Berkeley Unix, subtleties on both sides required considerable effort to reconcile. There was a critical desire to make network connections appear to be natural extensions of existing Unix machinery: file descriptors, pipes, and the other ideas that make Unix conceptually compact. But because of radical differences in behavior, especially delay, it is impossible to completely disguise reading 1,000 bytes from a round-the-world network connection so that it appears indistinguishable from reading that same 1,000 bytes from a file on a local file system. Networks have new behaviors that require new interfaces to capture and manage, but those new interfaces must make sense with exist-

ing interfaces. This was difficult work, and the modifications left few pieces of the system untouched; a few changed in profound ways.

The fundamental capabilities provided by a network protocol stack are data transfer, multiplexing, flow control, and error management. All of these functions are required for the coordinated delivery of data between endpoints across the Internet. Indeed, the purpose of all the structure in the packet headers: to carry the control coordination information, as well as the payload data.

The critical observation is that the exact same operations are required to coordinate the interaction of a network protocol stack and the host operating system within a single system. When all the code is in the same place (that is, running on the same processor), this signaling is easily done with simple procedure calls. If, however, the network protocol stack executes on a remote processor such as a TOE, this signaling must be done with an explicit protocol carried across whatever connects the front-end processor to the host operating system. This protocol is called a host-front end protocol (HFEP).

Designing an HFEP is not trivial, especially if the goal is that it be materially simpler than the protocol being offloaded to the remote processor. Historically, the HFEP has been the Achilles' heel of NFE processors. The HFEP ends up being asymptotically as complex as the "primary" protocol being offloaded, so there is very little to gain in offloading it. In addition, the HFEP must be implemented twice: once in the host and once in the front-end processor, each one of those being a different host platform as far as the HFEP is concerned. Two implementations, two integrations with host operating systems—this means twice as many sources of subtle race conditions, deadlocks, buffer starvations, and other nasty bugs. This cost requires a huge payoff to cover it.

But Wait a Minute...

About now some readers may be eager to throw a penalty flag for "unconvincing hand waving" because even in the base case, there is a protocol between the Ethernet interface and the host

computer device driver. "Doesn't that count?" you rightfully ask. Yes, indeed, it does.

There is a long history of peripheral chips being designed with absolutely dreadful interfaces. Such chips have been known to make device-driver writers contemplate slow, painful violence if they ever meet the chip designer in a dark alley. The very early Ethernet chips from one famous semiconductor company were absolute masterpieces of egregious overdesign. Not only did they contain many complex functions of dubious utility, but also the functions that were genuinely required suffered from the same virulent infestation of bugs that plagued the useless bits. Tom Lyon wrote a famous Usenix paper in 1985, "All the Chips that Fit," delivering an epic rant on this expansive topic. (It should be required reading for anyone contemplating hardware design.)

If the goal is efficiency and performance of network code, all of the "mini-protocols" in the entire network protocol subsystem must be examined carefully. Both internal complexity and integration complexity can be serious bottlenecks. Ultimately, the question is how hard is it to glue this piece onto the other pieces it must interact with frequently? If it is very difficult, it is likely not fast (in an absolute sense), nor is it likely robust from a bug standpoint.

Remember the protocol state machines are generally not the principal source of complexity or performance issues. One extra data copy can make a huge difference in the maximum achievable performance. Therefore, implementations must focus on avoiding data motion: put it where it goes the first time it is touched, then leave it alone. If some other operation on packet payload is required, such as checksum computation, bury it inside an unavoidable operation such as the single transfer into memory. In line with those suggestions, streamline the operating-system interface to maximize concurrency. Once all those issues have been addressed aggressively, there's not a lot of work left to avoid.

What Does All this Mean for NFEs?

Many times, but not every time, an NFE is likely to be an overly complex solution to the wrong part of the problem. It is possibly an expedient short-term

measure (and there's certainly a place in the world for those), but as a long-term architectural approach, the commoditization of processor cores makes specialized hardware very difficult to justify.

Lacking NFEs, what is required for maximizing host-based network performance? Here are some guidelines:

- Wire interfaces should be designed to be fast and brilliantly simple. Do the bit-speed work and then get the data into memory as quickly as possible, doing any additional work such as checksums that can readily be buried in the unavoidable transfer. Streamline the device as seen by the driver so as to avoid playing "Twenty Questions" with the hardware to determine what just happened.

- Interconnects should have sufficient capacity to carry the network traffic without strangling other I/O operations. From the standpoint of a network interface, PCI Express appears to have adequate performance for 10Gbps Ethernet as does HyperTransport 3.0.

- The system must have sufficient memory bandwidth to get the network payload in and out without strangling the rest of the system, especially the processors. Historically, the PC platform has been chronically starved for memory bandwidth.

- Processors should have enough cores able to exploit the sufficient memory bandwidth.

- Network protocol stacks should be designed to maximize parallelism and minimize blocking, while never copying data.

- A set of network APIs should be designed to maximize performance as opposed to mandatory similarity with existing system calls. Backward compatibility is important to support, but some applications may wish to pay more to get more.

Historical Perspective

NFEs have been rediscovered in at least four or five different periods. In the spirit of full and fair disclosure, I must admit to having directly contributed to two of those efforts and having purchased and integrated yet another. So why does this idea keep recurring if it turns out to be much more difficult than it first appears?

The capacities and economics of computer systems do not advance smoothly, nor are the rates of improvement of various components synchronized. The resulting interactions produce dramatically different trade-offs in system partitioning that evolve over time. What is correct today may not be right after the next technology improvement. An example will illustrate the point.

Once upon a time, disk storage was expensive—really expensive—but it also exhibited significant economy of scale. At that time, LAN connectivity and processor performance were sufficient to make it desirable to share large disks among multiple workstations, giving rise to the *diskless workstation*. This lasted for a number of years, but as disks slid down the learning curve, the decreasing cost per megabyte of disk space overwhelmed the operational complexity of diskless workstations so they became diskfull, and they have been ever since—until relatively recently. Today the typical large organization averages the better part of one PC per employee, so the operational grief of administering all those desktop PCs is substantial. This cost is now high enough that the diskless workstation has been rediscovered, this time named *thin clients*. All the storage is elsewhere; nothing permanent exists on the desktop unit. History is busily repeating itself. Why? Because the various cost curves have moved enough, relative to each other, to the point where centralization makes sense.

The same thing happens with NFEs. At a point in time, systems don't have enough network "go-fast" to deliver the performance required, so just add a dedicated processor to the network interface to make up for it. The economics of that are fleeting at best, however. Between chip design and system-integration complexity, an NFE will need to be an economically attractive solution for quite some time to recoup the development costs. Unfortunately, the relentless improvements in processor, memory system, and system interconnect in the base PC platform make that window of advantage a shrinking, fast-moving target. Does anyone else remember the HiFN file compression processor chip? It was built into PC systems for a very short time. Proces-

sors quickly improved enough to do compression/decompression on the fly, however, and that was the end of HiFN's dream—well before the dropping cost of disk storage would have killed it.

Any effort to question the efficacy of NFEs should include a caveat for one particular case that merits a special mention because it indeed makes a compelling case for a particular style of NFE.

The proliferation of microcontrollers in devices such as thermostats, light switches, toasters, and almost everything else with more than a simple on/off switch has created a real opportunity for NFEs. Almost all of these microcontroller applications are typified by intense cost pressure, which usually translates into extreme limitations on available computing resources. It is simply out of the question to put a network stack in the vast majority of these systems, but the desirability of remote management of these devices increases daily.

This has created a new breed of NFE: the network communications adapter (NCA) that specializes in the simplicity of the protocol between the microcontroller host and the NFE—serial ASCII. Most microcontrollers have some serial port ability, so by looking like a terminal, the NCA can play the role of translator, speaking serial out one side and TCP/IP out the other. The NCA appears as a host on the TCP network, often containing a simple Web server that vends state information and may provide certain other management functions that get translated into simple ASCII exchanges with the microcontroller system.

An NCA is usually implemented in one of the more powerful microcontrollers that have been designed to provide an Ethernet interface and support enough RAM and ROM to contain a simplified network stack. The NCA is now available as an off-the-shelf module designed for easy integration no more difficult than a modem on a serial port.

The question of which is the tail and which is the dog comes to mind in many of these applications. From the TCP network's point of view, the NCA is the host and the microcontroller is being managed. From the point of view of

the lighting controller, the NCA looks like just one more switch, albeit a chaty one. This distinction is usually irrelevant—it just makes hash of pedantic layering diagrams. There's something quite satisfying about that.

Conclusion

Rather than debate the religious propriety of NFEs, particularly the TOE variety, I have examined the architectural issues that have produced their recurring rise and fall. The TOE-style NFE is best viewed as a tactical tool with a limited expected lifetime of economic viability, not an enduring architectural approach. This is just another example of the recurring ebb and flow of functions between specialized peripherals and the system CPU(s), as the economics slosh back and forth interacting with system requirements. The limited lifetime of the NFE's advantages makes it difficult to justify the significant development costs for any but the highest-value applications.

That said, the inexpensive NCA is likely to be an approach that does endure. It literally transforms network communication into an inexpensive, pluggable physical component. By doing so, it provides an avenue for dealing with the extreme cost pressure inherent in microcontroller applications while providing an incremental option of genuine network citizenship when the customer will pay for it. □

Related articles on queue.acm.org

TCP Offload to the Rescue

Andy Currid

<http://queue.acm.org/detail.cfm?id=1005069>

Network Virtualization

Scott Rixner

<http://queue.acm.org/detail.cfm?id=1348592>

DAFS: A New High-Performance Networked File System

Steve Kleiman

<http://queue.acm.org/detail.cfm?id=1388770>

Mike O'Dell is a venture partner at New Enterprise Associates (NEA), Chevy Chase, MD, where he works to identify early-stage IT, communications, and energy opportunities. Prior to this position, Odell was chief scientist at UUNET Technologies, responsible for network and product architecture during the emergence of the commercial Internet. He has also held positions at Bellcore (now Telcordia), a GaAs Sparc supercomputer startup, and a U.S. government contractor. He was founding editor of *Computing Systems*, an international refereed scholarly journal.

© 2009 ACM 0001-0782/09/0600 \$10.00