USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

TCP offload is a dumb idea whose time has come

Jeffrey C. Mogul
Hewlett-Packard Laboratories
Palo Alto, CA, 94304
JeffMogul@acm.org

Abstract

Network interface implementors have repeatedly attempted to offload TCP processing from the host CPU. These efforts met with little success, because they were based on faulty premises. TCP offload *per se* is neither of much overall benefit nor free from significant costs and risks. But TCP offload in the service of very specific goals might actually be useful. In the context of the replacement of storage-specific interconnect via commoditized network hardware, TCP offload (and more generally, offloading the transport protocol) appropriately solves an important problem.

1 Introduction

TCP [18] has been the main transport protocol for the Internet Protocol stack for twenty years. During this time, there has been repeated debate over the implementation costs of the TCP layer.

One central question of this debate has been whether it is more appropriate to implement TCP in host CPU software, or in the network interface subsystem. The latter approach is usually called "TCP Offload" (the category is sometimes referred to as a "TCP Offload Engine," or TOE), although it in fact includes all protocol layers below TCP, as well. Typical reasons given for TCP offload include the reduction of host CPU requirements for protocol stack processing and checksumming, fewer interrupts to the host CPU, fewer bytes copied over the system bus, and the potential for offloading computationally expensive features such as encryption.

TCP offload poses some difficulties, including both purely technical challenges (either generic to all transports or specific to TCP), and some more subtle issues of technology deployment.

In some variants of the argument in favor of TCP offload, proponents assert the need for transport-protocol offload but recognize the difficulty of doing this for TCP, and have proposed deploying new transport protocols that support offloading. For example, the XTP protocol [8] was originally designed specifically for efficient implementation in VLSI, although later revisions of the specification [23] omit this rationale.

To this day, TCP offload has never firmly caught on in the commercial world (except sometimes as a stopgap to add TCP support to immature systems [16]), and has been scorned by the academic community and Internet purists. This paper starts by analyzing why TCP offload has repeatedly failed.

The lack of prior success with TCP offload does not, however, necessarily imply that this approach is categorically without merit. Indeed, the analysis of past failures points out that novel applications of TCP might benefit from TCP offload, but for reasons not clearly anticipated by early proponents. TCP offload does appear to be appropriately suited when used in the larger context in which storage-interconnect hardware, such as SCSI or FiberChannel, is on the verge of being replaced by Ethernet-based hardware and specific upper-level protocols (ULPs), such as iSCSI. These protocols can exploit "Remote Direct Memory Access" (RDMA) functionality provided by network interface subsystems. This paper ends by analyzing how TCP offload (and more generally, offloading certain transport protocols) can prove useful, not as a generic protocol implementation strategy, but as a component in an RDMA design.

This paper is *not* a defense of RDMA. Rather, it argues that the choice to use RDMA more clearly justifies offloading the transport protocol than has any previous application.

2 Why TCP offload is a dumb idea

TCP offload has been unsuccessful in the past for two kinds of reasons: fundamental performance issues, and difficulties resulting from the complexities of deploying TCP offload in practice.

2.1 Fundamental performance issues

Although TCP offload is usually justified as a performance improvement, in practice the performance benefits are either minimized or actually negated, for many reasons:

Limited processing requirements: Processing TCP headers simply doesn't (or shouldn't) take many cycles. Jacobson [11] showed how to use "header prediction" to process the common case for a TCP connection in very few instructions. The overhead of the TCP protocol *per se* does not justify offloading. Clark *et al.* [9] showed more generally that TCP should not be expensive to implement.

Moore's Law: Adding a transport protocol implementation to a Network Interface Controller (NIC) requires considerably more hardware complexity than a simple MAC-layer-only NIC. Complexity increases time-to-market, and because Moore's Law rapidly increases the performance of general-purpose CPU chips, complex special-purpose NIC chips can fall behind CPU performance. The TOE can become the bottleneck, especially if the vendor cannot afford to utilize the latest fab. (On the other hand, using a general-purpose CPU as a TOE could lead to a poor tradeoff between cost and performance [1].)

Partridge [17] pointed out that the Moore's Law issue could be irrelevant once each NIC chip is fast enough to handle packets at full line rate; further improvements in NIC performance might not matter (except to reduce power consumption). Sarkar *et al.* [21], however, showed that current protocol-offload NIC system products are not yet fast enough. Their results also imply that any extra latency imposed by protocol offload in the NIC will hurt performance for real applications. Moore's Law considerations may plague even "full-line-rate" NICs until they are fast enough to avoid adding much delay.

Complex interfaces to TOEs: O'Dell [14] has observed that "the problem has always been that the protocol for talking to the front-end processor and gluing it onto the API was just as complex (often more so, in fact) as the protocol being `offloaded'." Similarly, Partridge [16] observed that "The idea was that you passed your data over the bus to an NIC that did all the TCP work for you. However, it didn't give a performance improvement because to a large degree, it recreated TCP over the bus. That is, for each write, you had to add a bus header, including context information (identifying the process and TCP connection IDs) and then ship the packet down to the board. On inbound, you had to pass up the process and TCP connection info and then the kernel had to demux the bus unit of data to the right process (and do all that nasty memory alignment stuff to put it into the process's buffer in the right place)." While better approaches are now known, in general TOE designers had trouble designing an efficient host interface.

Suboptimal buffer management: Although a TOE can deliver a received TCP data segment to a chosen location in memory, this still leaves ULP protocol headers mingled with ULP data, unless complex features are included in the TOE interface.

Connection management: The TOE must maintain connection state for each TCP connection, and must coordinate this state with the host operating system. Especially for short-lived connections, any savings gained from less host involvement in processing data packet is wasted by this extra connection management overhead.

Resource management: If the transport protocol resides in the NIC, the NIC and the host OS must coordinate responsibility for resources such as data buffers, TCP port numbers, etc. The ownership problem for TCP buffers is more complex than the seemingly analogous problem for packet buffers, because outgoing TCP buffers must be held until acknowledged, and received buffers sometimes must be held pending reassembly. Resource management becomes even harder during overload, when host OS policy decisions must be supported. None of these problems are insuperable, but they reduce the benefits of offloading.

Event management: Much of the cost of processing a short TCP connection comes from the overhead of managing application-visible events [2]. Protocol offload does nothing to reduce the frequency of such events, and so fails to solve one of the primary costs of running a busy Web server (for example).

Much simpler NIC extensions can be effective: Numerous projects have demonstrated that instead of offloading the entire transport protocol, a NIC can be more simply extended so as to support extremely efficient TCP implementations. These extensions typically eliminate the need for memory copies, and/or offload the TCP checksum (eliminating the need for the CPU to touch the data in many cases, and thus avoiding data cache pollution). For example, Dalton *et al.* [10] described a NIC supporting a single-copy host OS implementation of TCP. Chase *et al.* [7] summarize several approaches to optimizing end-system TCP performance.

These criticisms of TCP offload apply most clearly when one starts with a well-tuned, highly scalable host OS implementation of TCP. TCP offload might be an expedient solution to the problems caused by second-rate host OS implementations, but this is not itself an architectural justification for TOE.

2.2 Deployment issues

Even if TCP offload were justified by its performance, it creates significant deployment, maintenance, and management problems:

Scaling issues: Some servers must maintain huge numbers of connections [2]. Modern host operating systems now generally place no limits except those based on RAM availability. If the TOE implementation has lower limits (perhaps constrained by on-board RAM), this could limit system scalability. Scaling concerns also apply to the IP routing table.

Bugs: Protocol implementations have bugs. Mature implementations have fewer bugs, but still require patches from time to time. Updating the firmware of a programmable TOE could be more difficult than updating a host OS. Clearly, non-programmable TOEs are even worse in this respect [1].

Quality Assurance (QA): System vendors must test complete systems prior to shipping them. Use of TOE increases the number of complex components to be tested, and (especially if the TOE comes from a different supplier) increases the difficulty of locating bugs.

Finger-pointing: When a TCP-related bug appears in a traditional system, it is not hard to decide whether the NIC is at fault, because non-TOE NICs perform fairly simple functions. With a system using TCP offloading, deciding whether the bug is in the NIC or the host could be much harder.

Subversion of NIC software: O'Dell has argued that the programmability of TOE NICs offers a target for malicious modifications [14]. This argument is somewhat weakened by the reality that many (if not most) high-speed NICs are already reprogrammable, but the extra capabilities of a TOE NIC might increase the options for subversion.

System management interfaces: System administrators prefer to use a consistent set of management interfaces (UIs and commands). Especially if the TOE and OS come from different vendors, it might be hard to provide a consistent, integrated management interface. Also, TOE NICs might not provide as much state visibility to system managers as can be provided by host OS TCP implementations.

Concerns about NIC vendors: NIC vendors have typically been smaller than host OS vendors, with less sophistication about overall system design and fewer resources to apply to support and maintenance. If a TOE NIC vendor fails or exits the market, customers can be left without support.

While none of these concerns are definitive arguments against TOE, they have tended to outweigh the limited performance benefits.

2.3 Analysis: mismatched applications

While it might appear from the preceding discussion that TCP offload is inherently useless, a more accurate statement would be that past attempts to employ TCP offload were mismatched to the applications in question.

Traditionally, TCP has been used either for WAN networking applications (email, FTP, Web) or for relatively low-bandwidth LAN applications (Telnet, X/11). Often, as is the case with email and the Web, the TCP connection lifetimes are quite short, and the connection count at a busy (server) system is high.

Because these are seen as the important applications of TCP, they are often used as the rationale for TCP offload. But these applications are exactly those for which the problems of TCP offload (scalability to large numbers of connections, per-connection overhead, low ratio of protocol processing cost to intrinsic network costs) are most obvious. In other words, in most WAN applications, the end-host TCP-related costs are insignificant, except for the connection-management costs that are either unsolved or worsened by TOE.

The implication of this observation is that the sweet spot for TCP offload is not for traditional TCP applications, but for applications that involve high bandwidth, low-latency, long-duration connections.

3 Why TCP offload's time has come

Computers generate high data rates on three kinds of channels (besides networks): graphics systems, storage systems, and interprocessor interconnects. Historically, these rates have been provided by special-purpose interface hardware, which trades flexibility and price for high bandwidth and high reliability.

For storage especially, the cost and limitations of special-purpose connection hardware is increasingly hard to justify, in the face of much cheaper Gbit/sec (or faster) Ethernet hardware. Replacing fabrics such as SCSI and Fiber Channel with switched Ethernet connections between storage and hosts promises increased configuration flexibility, more interoperability, and lower prices.

However, replicating traditional storage-specific performance using traditional network protocol stacks would be difficult, not because of protocol processing overheads, but because of data copy costs – especially since host busses are now often the main bottleneck. Traditional network implementations require one or more data copies, especially to preserve the semantics of system calls such as read() and write(). These APIs allow applications to choose when and how data buffers appear in their address spaces. Even with in-kernel applications (such as NFS), complete copy avoidance is not easy.

Several OS designs have been proposed to support traditional APIs and kernel structures while avoiding all unnecessary copies. For example, Brustoloni [4, 5] has explored several solutions to these problems.

Nevertheless, copy-avoidance designs have not been widely adopted, due to significant limitations. For example, when network maximum segment size (MSS) values are smaller than VM page sizes, which is often the case, page-remapping techniques are insufficient (and page-remapping often imposes overheads of its own.) Brustoloni also points out that "many copy avoidance techniques for network I/O are not applicable or may even backfire if applied to file I/O." [4]. Other designs that eliminate unnecessary copies, such as I/O Lite [15], require the use of new APIs (and hence force application changes). Dalton *et al.* [10] list some other difficulties with single-copy techniques.

Remote Direct Memory Access (RDMA) offers the possibility of sidestepping the problems with software-based copy-avoidance schemes. The NIC hardware (or at any rate, software resident on the NIC) implements the RDMA protocol. The kernel or application software registers buffer regions via the NIC driver, and obtains protected buffer reference tokens called *region IDs*. The software exchanges these region IDs with its connection peer, via RDMA messages sent over the transport connection. Special RDMA message directives ("verbs") enable a remote system to read or write memory regions named by the region IDs. The receiving NIC recognizes and interprets these directives, validates the region IDs, and performs protected data transfers to or from the named regions.[1]

In effect, RDMA provides the same low-overhead access between storage and memory currently provided by traditional DMA-based disk controllers.

(Some people have proposed factoring an RDMA protocol into two layers. A Direct Data Placement (DDP) protocol simply allows a sender to cause the receiving NIC to place data in the right memory locations. To this DDP functionality, a full RDMA protocol adds a remote-read operation: system *A* sends a message to system *B*, causing the NIC at *B* to transfer data from one of *B*'s buffers to one of *A*'s buffers without waking up the CPU at *B*. David Black [3] argues that a DDP protocol by itself can provide sufficient copy avoidance for many applications. Most of the points I will make about RDMA also apply to a DDP-only approach.)

An RDMA-enabled NIC (RNIC) needs its own implementation of all lower-level protocols, since to rely on the host OS stack would defeat the purpose. Moreover, in order for RDMA to substitute for hardware storage interfaces, it must provide highly reliable data transfer, so RDMA must be layered over a reliable transport such as TCP or SCTP [22]. This forces the RNIC to implement the transport layer.

Therefore, offloading the transport layer becomes valuable not for its own sake, but rather because that allows offloading of the RDMA layer. And offloading the RDMA layer is valuable because, unlike traditional TCP applications, RDMA applications are likely to use a relatively small number of low-latency, high-bandwidth transport connections, precisely the environment where TCP offloading might be beneficial. Also, RDMA allows the RNIC to separate ULP data from ULP control (i.e., headers) and therefore simplifies the received-buffer placement problems of pure TCP offload.

For example, Magoutis *et al.* [13] show that the RDMA-based Direct Access File System can outperform even a zero-copy implementation of NFS, in part because RDMA also helps to enable user-level implementation of the file system client. Also, storage access implies the use of large ULP messages, which amortize offloading's increased per-packet costs while reaping the reduced per-byte costs.

Although much of the work on RDMA has focussed on storage systems, high-bandwidth graphics applications (e.g., streaming HDTV videos) have similar characteristics. A video-on-demand connection might use RDMA both at the server (for access to the stored video) and at the client (for rendering the video).

4 Implications for operating systems

Because RDMA is explicitly a performance optimization, not a source of functional benefits, it can only succeed if its design fits comfortably into many layers of a complete system: networking, I/O, memory architecture, operating system, and upper-level application. A misfit with any of these layers could obviate any benefits.

In particular, an RNIC design done without any consideration for the structures of real operating systems will not deliver good performance and flexibility. Experience from an analogous effort, to offload DES cryptography, showed that overlooking the way that software will use the device can eliminate much of the potential performance gain [12]. Good hardware design is certainly not impossible, but it requires co-development with the operating system support.

RDMA aspects requiring such co-development include:

Getting the semantics right: RDMA introduces many issues related to buffer ownership, operation completion, and errors. Members of the various groups trying to designs RDMA protocols (including the RDMA Consortium [19] and the IETF's RDDP Working Group [20]) have had difficulty resolving many basic issues in these designs. These disagreements might imply the lack of sufficiently mature principles underlying the mixed use of remotely- and locally-managed buffers.

OS-to-RDMA interfaces: These interfaces include, for example, buffer allocation; mapping and protection of buffers; and handling exceptions beyond what the RNIC can deal with (such as routing and ARP information for a new peer address).

Application-to-RDMA interfaces: These interfaces include, for example, buffer ownership; notification of RDMA completion events; and bidirectional interfaces to RDMA verbs.

Network configuration and management: RNICs will require IP addresses, subnet masks, etc., and will have to report statistics for use by network management tools. Ideally, the operating system should provide a "single system image" for network management functions, even though it includes several independent network stack implementations.

Defenses against attacks: an RNIC acts as an extension of the operating system's protection mechanisms, and thus should defend against subversions of these mechanisms. The RNIC could refuse access to certain regions of memory known to store kernel code or data structures, except in narrowly-defined circumstances (e.g., bootstrapping).

Since the RNIC includes a TCP implementation, there will be temptation to use that as a pure TOE path for non-RDMA TCP connections, instead of the kernel's own stack. This temptation must be resisted, because it will lead to over-complex RNICs, interfaces, and host OS modifications. However, an RNIC might easily support certain simple features that have been proposed [5] for copy-avoidance in OS-based network stacks.

5 Difficulties

RDMA introduces several tricky problems, especially in the area of security. Prior storage-networking designs assumed a closed, physically secure network, but IP-based RDMA potentially leaves a host vulnerable to the entire world.

Offloading the transport protocol exacerbates the security problem by adding more opportunities for bugs. Many (if not most) security holes discovered recently are implementation bugs, not specification bugs. Even if an RDMA protocol design can be shown to be secure, this does not imply that all of its implementations would be secure. Hackers actively find and exploit bugs, and an RDMA bug could be much more severe than traditional protocol-stack bugs, because it might allow unbounded and unchecked access to host memory.

RDMA security therefore cannot be provided by sprinkling some IPSec pixie dust over the protocol; it will require attention to all layers of the system.

The use of TCP below RDMA is controversial, because it requires TCP modifications (or a thin intermediate layer whose implementation is entangled with the TCP layer) in order to reliably mark RDMA message boundaries. While SCTP is widely accepted as inherently better than TCP as a transport for RDMA, some vendors believe that TCP is adequate, and intend to ship RDMA/TCP implementations long before offloaded SCTP layers are mature. This paper's main point is not that *TCP* offload is a good idea, but rather that *transport-protocol* offload is appropriate for RNICs. TCP might simply represent the best available choice for several years.

6 Conclusions

TCP offload has been "a solution in search of a problem" for several decades. This paper identifies several inherent reasons why general-purpose TCP offload has repeatedly failed. However, as hardware trends change the feasibility and economics of network-based storage connections, RDMA will become a significant and appropriate justification for TOEs.

RDMA's remotely-managed network buffers could be an innovation analogous to novel memory consistency models: an attempt to focus on necessary features for real applications, giving up the simplicity of a narrow interface for the potential of significant performance scaling. But as in the case of relaxed consistency, we may see a period where variants are proposed, tested, evolved, and sometimes discarded. The principles that must be developed are squarely in the domain of operating systems.

Acknowledgments

I would like to thank David Black, Craig Partridge, and especially Jeff Chase, as well as the anonymous reviewers, for their helpful comments.

References

[1] B. S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. Tech. Rep. HPL-2001-8, HP Labs, Jan. 2001.

[2] G. Banga and J. C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. 1998 USENIX Annual Technical Conf.*, pages 1–12, New Orleans, LA, June 1998. USENIX.

[3] D. Black. Personal communication, 2003.

[4] J. Brustoloni. Interoperation of copy avoidance in network and file I/O. In *Proc. INFOCOM '99*, pages 534–542, New York, NY, Mar. 1999. IEEE.

[5] J. Brustoloni and P. Steenkiste. Effects of buffering semantics on I/O performance. In *Proc. OSDI-II*, pages 277–291, Seattle, WA, Oct. 1996. USENIX.

[6] J. S. Chase. *High Performance TCP/IP Networking* (Mahbub Hassan and Raj Jain, Editors), chapter 13, *TCP Implementation*. Prentice-Hall. In preparation.

[7] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End-system optimizations for high-speed TCP. *IEEE Communications*, 39(4):68–74, Apr. 2001.

[8] G. Chesson. XTP/PE overview. In *Proc. IEEE 13th Conf. on Local Computer Networks*, pages 292–296, Oct. 1988.

[9] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[10] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner: Architectural support for high performance protocols. *IEEE Network Magazine*, 7(4):36–43, 1995.

[11] V. Jacobson. 4BSD TCP header prediction. *Computer Communication Review*, 20(2):13–15, Apr. 1990.

[12] M. Lindemann and S. W. Smith. Improving DES coprocessor throughput for short operations. In *Proc. 10th USENIX Security Symp.*, Washington, DC, Aug. 2001.

[13] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the Direct Access File System. In *Proc. USENIX 2002 Annual Tech. Conf.*, pages 1–14, Monterey, CA, June 2002.

[14] M. O'Dell. Re: how bad an idea is this? Message on TSV mailing list, Nov. 2002.

[15] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Trans. Computer Systems*, 18(1):37–66, Feb. 2000.

[16] C. Partridge. Re: how bad an idea is this? Message on TSV mailing list, Nov. 2002.

[17] C. Partridge. Personal communication, 2003.

[18] J. B. Postel. Transmission Control Protocol. RFC 793, Information Sciences Institute, Sept. 1981.

[19] RDMA Consortium. http://www.rdmaconsortium.org.

[20] Remote Direct Data Placement Working Group. http://www.ietf.org/html.charters/rddp-charter.html.

[21] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: Does hardware support help? In *Proc. 2nd USENIX Conf. on File and Storage Technologies*, pages 231–244, San Francisco, CA, March 2003.

[22] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, Network Working Group, Oct. 2000.

[23] T. Strayer. Xpress Transport Protocol, Rev. 4.0b. XTP Forum, 1998.

Notes

[1]Much of this paragraph was adapted, with permission, from a forthcoming book chapter by Jeff Chase [6].