HotOS IX Paper [HotOS IX Program Index]

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.

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.

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.

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.

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.

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.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.