Architectural Implications of Performing Network Protocol Processing Closer to the Application

Matthew Faulkner, Matthew Jakeman and Stephen Pink
Computing Department
Lancaster University
{**faulknem, jakeman,pink**}**@comp.lancs.ac.uk**

Keywords: Architecture, User Space TCP/IP, Kernel, Optimisation. Operating Systems

Abstract

It has been a while since high-speed host protocol processing has been a central concern of networking research. This paper expands on the ideas presented recently by Van Jacobson[1], who, among others[2], [3], [4], [5] played a large role in the development of efficient host processing almost two decades ago. Recently Jacobson has suggested moving the TCP/IP implementation out of the kernel of the operating system in multi-core CPU-based systems. This reduces the number of context switches and locks used, so that more concurrency, among other performance enhancements, are achieved. We do not attempt to modify this technique as the improvements are significant. Instead we attempt to identify and solve some of the problems inherent in changing the software architecture to achieve such a move. Taking the TCP/IP stack out of the kernel can have significant impact on the overall architectural aspects of Internetworking. This paper discusses this optimisation in relation to addressing, routing and the end-to-end principle of Internetworking. We also show that these changes can have a significant impact on applications. This paper raises a number of questions concerning userspace TCP/IP and makes an attempt at some of their solutions.

1 INTRODUCTION

Attention once again has been paid to high-speed protocol processing in end systems. This is because extremely high speed optical networks are now being attached directly to Internet hosts and these hosts are now routinely equipped with multi-core CPUs. Multi-gigabit per second Ethernet is now beginning to appear on end systems with the prospect of terabit per second network interfaces coming in the near future. New applications are being developed to use these high speed networks and to saturate the new optical network interfaces. Once again, as it did in the late 1980s and early 1990s, the protocol processing in the host has become the bottleneck in the end-to-end transfer of data between two applications on a high speed network.

At the same time as the appearance of very high speed network interfaces, modern PCs are now gaining multiple CPUs with multiple cores driving operating systems that are capable of executing multiple threads in parallel. Van Jacobson has recently pointed out[1] that the traditional software architecture, with the protocol stack implemented in the kernel, even though highly optimised for the case of the single-core, single-CPU system, becomes a performance bottleneck in the face of high speed networks and multiple-cores and multiple CPUs. Jacobson shows that implementing network protocols in user or application space opens up the hardware/software architecture to an increase in performance because of the decrease of software locking and a general exploitation of processing concurrency. According to Jacobson, this speedup in performance should be no surprise as user space protocol implementations for multi-processing is more consistent with an important end-to-end principle in system design that protocol processing is better done closer to the furthest endpoints on the communication channel[6].

This paper expands on the work presented by Jacobson et al[1] by discussing some of the architectural implications of user space TCP/IP protocols implementation. Specifically the effect on addressing, routing, mobility, stack diversity and the end-to-end argument are discussed.

1.1 Organisation

The rest of this paper is organised as follows. The next section introduces and motivates the use of user space network protocol implementations. Section 3 is a discussion of the implications of user space TCP/IP. Section 4 details a number of other advantages. More specifically Section 4 gives several concrete examples of how deploying TCP/IP in this way can help applications. Potential problems of user space network stacks are introduced in Section 5. Following a discussion of the potential problem, related work is discussed in Section 6. Finally this paper concludes in Section 7.

2 USER SPACE NETWORK PROTOCOL IMPLEMENTATIONS

User or application-space implementations of network protocols have been suggested in previous work [7] [8], but few of these studies (except, of course, Jacobson's recent Linux study) has examined the performance benefits that can be gained in modern communication system. Modern computer

systems increasingly include multiple processors. These processors may contain multiple cores. This growth in the number of processors and/or cores is an attempt to increase the concurrent processing within the system. Moving the TCP/IP stack into user space (as opposed to being in a monolithic kernel) means that network processing can take place with fewer cross-core critical sections. Jacobson presents a worst case scenario [1] where the network interrupt processing and the process are configured to be executed on different cores. A comparison to a single core processor shows that a 27% increase in utilisation occurs when the core that performs the network processing is not the same core as the application process.

The main reasons for this reduced performance are spin locks which are used to wait for others cores to exit critical sections and cache misses that occur when the network processing and application processing are performed on different cores. When the core that does network processing is different from the process that consumes the resulting data some form of shared memory is required. Whenever memory is shared between two processes, a lock is needed to ensure consistency of common data structures. Locking is an expensive operation [1]. The cost of locking is increased further when the lock spans multiple cores. Locks can be reduced if per-core data structures are used [9]. This is shown on single-core processor machines where the core that does the network processing is the same core as the the one executing the application process.

Other factors that affect the performance of network processing are [1]

- Context switches - Network processing within the kernel is done using a number of different interrupts. The hardware interrupt occurs when a packet arrives and places the packet on an input queue. A software interrupt is called later to facilitate the packet processing within the kernel. The result of these interrupts is a forced context switch. Another context switch is used to pass to the packet to the application process. Context switching is an expensive operation.

- Extra data copies - Kernel implementations often have extra data copies. For example, copying the packet to the socket buffer and then to an application's buffer in user space.

- Difference in speed between the processing power and memory performance. Any memory options such as loading an instruction from cache will be a principal cost in terms of performance within a single node.

Jacobson [1] suggests that in order to improve performance and to increase concurrency, the processor which does the application processing should also do the network processing. This limits the amount of work done by the hardware and software interrupt service routines. A process or processor (which of these two is used is discussed in Section 3) simply creates an instance of the stack within the application's runtime image rather than communicating with a single instance in the kernel. Thus, protocol processing gets pushed back to the application, i.e., closer to the end-point of the the communication.

The principle benefit from this rearrangement of the software protocol architecture is the increased performance that can be achieved due to the reduced number of spin locks and cache misses. We suggest that moving the network stack out of the kernel has additional benefits for the network architecture. The remainder of this paper explores these benefits, costs and implications.

3 EFFECT ON ADDRESSING

In most IP protocol stack implementations, the IP address is assigned to a network interface. On these systems, there is one, usually kernelised, TCP/IP protocol stack implemented architecturally just above the network interface. It is possible, however, to implement either the transport protocols (TCP and UDP) in user or application space, or, even more radically, to implement the full Internet stack including IP, in application space. This raises the question, however, whether it is desirable to view the IP address as attached to the network interface because there are, in this more radical case, potentially many processors in this system. In this paper, we explore three issues that arise in tackling this change of software protocol architecture. We discuss IP address assignment to network interfaces, to processor-cores, and finally to operating system processes themselves. A discussion of the possible large increase in Internet routing table size due to processor/process IP address assignment is left to Section 5.

3.1 Network Interface Address Assignment

IP addresses assigned to a network interface is the current status quo. This makes the CPU which services the network interrupt a performance bottleneck. In a single processor machine, the same processor is used to run the process as to service the interrupt, with the interrupt, in the case of most systems, having priority. In a multi-processor machine there is at least one other processor which could continue doing work while the interrupt is processed. If the interrupt is serviced in the OS kernel, then all the processors may end up waiting on the kernel to finish servicing the interrupt when there is useful work that could be done in parallel. If the protocol processing for a particular application were pushed out to the processor that was running that application, the other processors could continue to execute code on behalf of other applications instead of stalling and waiting for kernel protocol processing

to complete. This is another way of enforcing the end-to-end argument in system design[6].

3.2 Core Address Assignment

If IP addresses are assigned per core, the bottleneck is pushed further towards the end of the communication channel. Since each core is addressed individually the network interface card becomes another hop within the larger Internet. The network interface card would then do in essence, a lookup similar to what a router does and forward the packet to the destination core process.

Allowing each core to have its own networking stack enables faster processing than is currently possible with a single kernel implementation executing on behalf of all cores. Each application protocol-stack runtime would have its own input and output queues. When a core processor becomes available, the packets can be taken from the queue (for the case of input packet processing) and be processed before being passed to the application code proper. The ability for each core to have its own queue solves locking contention problems and the kernel is no longer the protocol processing bottleneck.

3.3 Process Address Assignment

A user space implementation of the TCP/IP stack could be structured in a number of ways, vis a vis the application. One way would be for the TCP/IP code to be implemented in a run-time library that is linked to the application. Another way could be that the protocol stack was implemented in a running daemon or server communicating with the application through shared memory or the file system or some other kind of interprocess communication. In either case, some other entity, the router or perhaps the code for the network interface card, assigns each instance of process on a processor its own IP address.

However, IP address assignment on a per process/processor basis would increase the number of IP addresses used and would have a significant effect on the size of the routing and forwarding tables. This impact is discussed in Section 5. We also want to say, though, that this problem of increasing the number of IP addresses in the routing and forwarding tables may be avoided by taking the less-radical approach of implementing the transport protocols (TCP and UDP) only in application or user space, and leaving the IP code in the kernel. It remains to be seen, in our research, whether this hybrid approach frees up enough concurrency and gets rid of enough processor locking to substantially increase performance. One needs to remark here that this approach, while relieving the size issue, would mean that some of the other points of flexibility we discuss below could not be taken advantage of.

3.4 Multi-processors: The End-to-End Principle Revisited

A user or application space network protocol implementation scheme provides more end-to-end properties to the communication on a multiprocessor-based host than current kernel-based network protocol stacks do. Checksumming, reliable ordering of data, end-to-end congestion control, and other end-to-end properties are closer to the application in this new implementation architecture than in the traditional monolithic kernel run-time environment. Having these properties of the traditional reliable transport protocol such as TCP run in the same memory image as the application provides the application better and more reliable guarantees of their service. For example, the end-to-end checksum in TCP, if run in the same process space as the application, is able to detect any errors in the data that might be produced by the host hardware itself, as it grows more complex supporting a large number of processors and busses. And the same is true for the cases of reordering of data end-to-end. In addition, as routing moves from just wide area over long distances to the actual routing of packets inside of a multiprocessor host, end-to-end congestion control from application to application may mean process to process, even inside of the same host. As Van Jacobson states:[1]

> "The end of the wire isn't the end of the net".

4 OTHER ADVANTAGES OF USER SPACE VS KERNEL-BASED TCP/IP

Having protocol processing in user space rather than in the kernel opens up a new set of opportunities for flexibility. This section looks at a few examples of these that go over and above the simple performance gains discussed above.

4.1 Process migration

Support for mobility is an important goal for many communication systems. The traditional view of mobility is users of devices such as cell phones and PDAs moving from one location to another. However, the concept of mobility can be defined more broadly to encompass the migration of processes. Process migration [10],can be an advantage to many server-based and other applications. Servers have traditionally been in a single fixed location, but with the increasing demand for 100% uptime, server processes may need to be moved for hardware and software upgrades, service provider changes and general need for physical location changes.

Long uptime requirements, even during maintenance periods, can be achieved by moving a process from one computational location to another. Supporting process migration with current implementation architectures can be a difficult task if the process needs to have persistent TCP connections across the move. This is because a TCP connection gets its

identity from the IP address of the node it is running on. If the process migrates to a new node, and that new node has a different IP address (because the IP address is assigned to the network interface of that node) then the TCP connection cannot survive. However, if the process itself contains its own IP protocol stack with its own IP address, the TCP connection can survive the migration to another processing element. With a user space implementation of the TCP/IP stack linked into the application, e.g., by a run-time library, assigning an IP address to the process and migrating that process becomes simpler from the point of view of the protocol architecture.

4.2 Virtual Private Networks

It is often the case that users wish to connect virtually to a different network using a Virtual Private Network (VPN) [11]. At present a VPN connection is established by the user on behalf of the process. The user then forces all network processing for this process via this network connection. This can often have undesirable effects. For example, connecting to a certain VPN might mean that certain ports are blocked and the user cannot use all the applications they wish. By having multiple network stacks a process could choose to load the VPN network stack. This permits different processes to be on different VPNs as well as certain processes being on the machines standard network.

4.3 Peer to Peer

A large amount of traffic on the Internet has become peer to peer [12]. New suggestions on how to use overlay networks to transfer peer to peer traffic efficiently have recently been made [13]. Some of these suggestions use an additional protocol layer to provide application layer routing. For example, distributed hash tables (DHTs) provide a lookup service based upon an identifier. Currently each application is required to implement its own DHT. However it may be possible to incorporate a DHT within a network stack as a kind of generic service to applications that were linked to it. This would reduce the complexity of the application and potentially provide a faster lookup time.

4.4 Stack Diversity

Using a single network protocol implementation for all application processes provides a single point of failure. If this single point of failure is within the kernel of the operating system, as it is for kernel implementations of network protocols, then the whole system is vulnerable. For example, if a bug in the kernel network code produces an unanticipated buffer overflow, all processes running on that node could be compromised rather than just the network processes linked into the user space run time library in an architecture where application space network stacks are deployed.

4.5 Stack Development and Updates

Network protocol development in the kernel is a difficult task. The edit/compile/debug cycle for kernel code is much longer than it is for code in application space. This is mainly due to the much better development and debugging tools and environments available for application code and the fact that debugging in kernel code means having to deal often with code running at the hardware interrupt level that can result in race conditions.

It is much faster and more reliable to develop, debug, test and deploy new networking code in user or application space. Prototype development of new, optimised versions of network and transport protocols would be quicker and more efficient. An example of such an optimised protocol implementation is TCP Vegas[14]. TCP Vegas is fully compliant with TCP including its congestion control algorithms, but provides performance improvements using a number of different novel methods including more accurate measurements of round trip times. A user space network stack would allow TCP Vegas to be more easily deployed in network stacks, thus improving the performance of applications. In addition, when updates are necessary to make protocol implementations conform to rapidly changing standards due to security and other reasons, updating a user space implementation can be much easier than a kernel. Updating a user space implementation of a protocol stack may be as easy as changing a shared library, whereas updating a kernel implementation often means at least a partial recompilation or loading of a new operating system kernel version.

5 SOME POTENTIAL PROBLEMS

Although not totally foolproof, kernel-based TCP/IP implementations provide a level of security among users that could be lacking in user space implementations. A kernel-based network stack can be more inaccessible to malicious users and applications. For example, the congestion control algorithms in TCP work to everyone's advantage only if every TCP backs off in response to various signs of congestion signals and ramps up in a way that is controlled by its internal algorithms. A malicious user, however, can create an unfair situation by modifying the code in his TCP to be greedy in the face of other users' back-offs. A user or application space TCP is usually easier to modify, especially on a per process basis, to act in this greedy way than is a monolithic kernel implementation. It's not that the kernel implementation is always secure, it's more a question of ease of modification that makes the user space code more vulnerable.

Another potential problem is that for some operating systems, having a TCP/IP stack per process increases the total memory usage among the many users and processors on a single machine, thus increasing the memory footprint of this architecture over the traditional kernel-based network stack

approach. Although some shared library implementations do make an attempt to share running code among instances of the same program, buffer space and certain data structures will probably have to be replicated on a per process basis with the user space approach making the kernel-based approach often more memory friendly. The decrease in the cost of memory resulting in larger host memory sizes, however, could mitigate this potential problem somewhat.

We mentioned earlier that process migration as a form of mobility within and outside of hosts could be done in a straightforward manner with user or application space protocol stacks with an IP address assigned per process. A problem with doing process migration by simply moving a process with its IP address to a different host is that Address Resolution Protocol (ARP) tables would have to be updated on demand in the sending host on the local network in order to deliver IP packets correctly to the host where the process has been moved.[1] We are currently investigating mechanisms, such as those available in Mobile IPv6 that allow users to move to new locations, for their feasibility in the process migration scenario.

Perhaps the most serious of the potential problems, however, is the possibly large growth in Internet routing and forwarding table size if IP addresses were to be assigned to each process in a multi-core or multiprocessor host. Although memory cache size in routers is also growing, this may not be able to keep up with the explosive growth in the size of forwarding tables. We plan to do experiments to understand this potential increase in routing and forwarding table size when individual processes all have their own IP addresses and also to explore several alternative measures to confront the problem.

6 RELATED WORK

Networking performance improvements have been the subject of research since the 1970s. This section discusses some of this related work. Much previous work on user space implementations of TCP/IP has focused on specific problems. For example Apline [7] aims to aid the development of new network protocols by simplifying the development process. Debugging network code within the kernel is a difficult task. Apline simplifies this by allowing a user space implementation of the network protocol to be developed first. Once this user space implementation has been developed, Apline encourages the network protocol to be reintroduced into the kernel.

TCP offload engines move some network processing tasks away from the main CPU. Instead, the processing is done on a processor embedded on the Network Interface Card. Previous research [15] has shown that these techniques do not provide considerable performance improvements. Seong Ang [15] suggests that TCP offload engines have potential, but the cost versus performance benefits gained from such techniques is questionable. Using TCP offload engines does not move the bottleneck away from the end-point of the network path. Instead the bottleneck becomes the NIC. This bottleneck is the very issue that Jacobson tries to address [1]. The bottleneck is created because NIC processors tend to be slower than their CPU counterparts. Companies have started to market offload engines which are aimed for network gaming [16].

Finally, there are a number of different kernel implementation optimisations for network stacks for keeping the number of memory copies to a minimum because it is an expensive task [4]. Traditionally an interface would receive a packet fully before copying this packet to the protocol input function. When the application calls the *socket*() receive function, another copy is invoked. This copy moves the data from kernel to user space. Optimisations such as copy-on-write, page re-mapping (i.e., zero-copy) and single-copy aim to reduce the impact of the copy on performance. Of these solutions, the only method which only alters the networking code, and does not force an application re-write, is single-copy.

In the single-copy [4] paradigm, memory is shared between the network interface and the kernel. The only copy is between kernel space and user space. Single-copy requires that the data is checksummed and copied in the same loop in the kernel code. Reducing the number of memory copies increases the performance of any network processing system whether it is within the kernel or user space.

Changing the data structure used to process packets is another improvement suggested by Jacobson [1]. Instead of the widely used linked-list structure, a carefully designed circular buffer could be used. If correctly designed, such buffers require no locks and more importantly share no writable cache lines between the producer and the consumer (network card / kernel combination).

7 CONCLUSION

Traditionally network protocol implementations have been a part of an operating system kernel. This is an ideal solution for single-core processor machines. However, it is not ideal for multiprocessor or multi-core systems especially when attached to very high speed networks. These multiprocessor systems can increase performance because multiple instructions can be executed concurrently. In kernel-based network implementations, concurrency can be reduced because some processors can idle waiting for the kernel's processor to finish. Work done by Jacobson [1] shows how a network stack pushed closer to the application can increase performance because of reduced context switches, locks and copies. The work presented in this paper expands on the work done by Jacobson [1] by examining the architectural implications of

[1] We thank Paul Tipper at Lancaster University for pointing out this potential problem.

pushing the protocol processing closer to the user application. Issues such as addressing, routing, forwarding and mobility have been discussed in light of this new software protocol architecture and some suggestions as to their solutions have been offered.

REFERENCES

[1] Van Jacobson and B. Felderman. Gumstix embedded computing platform specifications. A modest proposal to help speed up and scale up the linux networking stack. In linux.conf.au, Jan 2006.

[2] John Romkey, David D. Clark, Van Jacobson, and Howard Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, 27(6), June 1989.

[3] Craig Partridge and Stephen Pink. A faster udp. *IEEE/ACM Transactions on Networking*, 1(4):429–440, 1993.

[4] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network Magazine*, 7(4), July 1993.

[5] Eric C. Cooper, Peter A. Steenkiste, Robert D. Sansom, and Brian D. Zill. Protocol Implementation on the Nectar Communication Processor. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 135–144, Philadelphia, PA, 1990.

[6] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), 1984.

[7] David Ely, Stefan Savage, and David Wetherall. Alpine: A user-level infrastructure for network protocol development. In *USITS*. USENIX, 2001.

[8] Aled Edwards and Steve Muir. Experiences implementing a high performance tcp in user-space. *SIGCOMM Comput. Commun. Rev.*, 25(4):196–205, 1995.

[9] Matthew Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. linux.conf.au, Jan 2003.

[10] Jonathan M. Smith. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev.*, 22(3):28–40, 1988.

[11] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. A Framework for IP Based Virtual Private Networks. RFC 2764 (Informational), February 2000.

[12] Alok Madhukar and Carey Williamson. A longitudinal study of p2p traffic classification. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 179–188, Washington, DC, USA, 2006. IEEE Computer Society.

[13] D. Andersen, H. Balakrishnan, M. Frans Kaashoek, and R. Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[14] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[15] Boon Seong Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. Technical Report HPL-2001-8, Hewlett Packard Laboratories, January 17 2001.

[16] Bigfootnetworks http://www.killernic.com/KillerNic/. Killer network interface card.

Biography

Matthew Faulkner is a PhD student in Networking at Lancaster University. He received his MSc in Advanced Computer Science from Lancaster, where he graduated with a distinction and in the top three of the class. His MSc thesis was performed on the potential use of 802.11 in remote areas. His current research interests are network architecture, network protocol implementations and network optimisations.

Matthew Jakeman is currently a PhD student at Lancaster University where he also gained his Batchelor and Masters degrees. His current research focusses on network architecture, network protocol implementation and sensor network communications protocols.

Stephen Pink is a Professor of Computing at Lancaster University. Prior to that he has been a Professor of Computer Science at the University of Arizona and Professor of Computer Communication at Lulea University in Sweden. He has also been Director of Research at the Swedish Institute of Computer Science as well as having been a founder and Chief Technology Officer of two commercial companies. Active in Internet standardisation, he is an author of two RFCs in the IETF. He has published in ACM, IEEE, IFIP and other journals, conference and workshop proceedings.