IP Based Design 2002

A Parallel TCP/IP Offloading Framework for a TCP/IP Offloading Implementation

Juan M. Solá-Sloan, Isidoro Couvertier Ph.D.

University of Puerto Rico
Mayagüez, Puerto Rico

Abstract

Offloading the Transport Control Protocol/Internet Protocol (TCP/IP) is a task that relieves the kernel from interrupting to process the IP Stack. This article discusses the uses in which a TCP/IP firmware could increase network performance and ease of programming. In addition, this article proposes a parallel framework for a TCP/IP offloading hardware implementation.

1. Introduction

TCP/IP is a protocol suite used worldwide. It is not exclusively used for the Internet. There exist private nets that also used TCP/IP as their protocol suite. TCP/IP has been adopted as the basic pair of Internet protocols. Many hardware implementations have been develop based on the TCP/IP or related protocols pair. Special routers handle UNIX operating system clones to handle TCP/IP on them [9]. A change to the protocol standard would cause a heavy impact in the Internet infrastructure. That's why IPv6 migration is slow [6] [5]. Actually, there is no such need (until the date of this article) of change in the American Continent. Since IPv6 was proposed as a solution for many IPv4 milestones, the migration is not an easy one. Isles of IPv6 are currently tunneling via IPv4. However, changes in TCP/IP standard are not so common. That's why TCP/IP kernel processing load can be extracted from any central processing unit. The IP stack could be process inside a firmware using a common BSD Socket Interface. The firmware could be integrated inside an Intelligent Network Interface Card (INIC), a PDA or cellular phone circuit, a gaming console, an Internet router for home networks etc. Virtually in any place, we want Internet access.

2.0 Problems in processing TCP/IP

TCP/IP load processing faces problems in Multi-homed hosts, low power limited resources devices and Gigabit networks. This problems will be discuss in detail in the following sections

2.1 Multi-homed hosts problems

A Multi-Homed Host is a node inside the Internet infrastructure that is responsible for handling different network related transactions (email service, file service, routing, front-end etc.) and contains multiple data communication media [3] . They depend on Internet communication for their existence. High traffic networks nodes with a single central processing unit interrupt the processor each time a packet is received [3]. Datagram handling is done in software. The processor is constantly interrupted every time a datagram or a datagram fragment is received. Upper layers, since layer 2, of the Internet Layer model are manipulated by the central processing unit of these multi-homed hosts. Usually these multi-homed hosts contained multiple net devices for Internet communications. These increase the interrupts. If TCP/IP were handle by hardware by offloading multi-homed hosts, then applications in the top of the Internet Layer Model would receive more processing time.

2.2 Low resources device

Devices class 1, 2, or 3 of the architecture discussed in [4] are low resources devices. Based on this architecture, devices are handheld/ wireless relying in a slow central processing unit. Implementations in kernel software must include TCP/IP protocol pair for Internet access. Cellular Phones and PDAs with Internet Access can include in its architecture a TCP/IP processing firmware. If TCP/IP can be

process in hardware, then kernel programmers of such devices could rely on cheaper processors. If the functionality of TCP/IP firmware is increase to include protocols like Internet Message Control Protocol (ICMP), Address Resolution Protocol (ARP), User Datagram Protocol (UDP) and some kind of frame carrier handling, then kernels will reduce substantially in size, leaving the Internet firmware all of these tasks.

2.3 Home Networks

An Internet Gateway would need the Internet firmware to access the Internet. As network appliances like [8] evolve they will need access to the World Wide Web. Studies like [7] proposed an end-to-end communication using SIP over an IP layer. A TCP/IP firmware with IPv6 support will accommodate the needs of an Internet Gateway for home networks. Even more, the firmware could support network appliances with Internet Gateway included.

2.4.1 Gigabit networks communication problems

Gigabit Local Area Networks consume main processor cycles by loading it with frame carriers [1]. Performance is wasted on packet handling. IP stacks are handle by software. By offloading, the CPU load is reduced leaving more processing time for the application and socket layers. ISCSI devices would benefit of a TCP/IP firmware. Remote backup systems could include a TCP/IP firmware in its circuit board. This could minimize embedded kernel size inside these devices.

2.4.2 Increasing the Maximum Transmission Size leads to other problems

Current Gigabit technology is faster than the central processing units used on high-end transaction server today. "A rough estimate of the CPU required to handle a given Ethernet link speed is, for every one bit per second of network data processed, one hertz of CPU processing is required"[11]. Therefore, for a 10Gbps network device a 20GHZ CPU is needed (at full utilization) [11]. On current technology, this problem leads to an increase in interrupts generated by the net device.

If the interrupt amount sky rockets when the net device is fast, then more processing time is required for every payload received by the data link layer. In gigabit speed networks, the solution has been increasing the payload amount. Orthodox Ethernet (10/100Mbps) configurations rely on a frame carrier of 1518 octets leaving 1500 bytes maximum for it's payload. The solution in such experiments is to increase the payload size. High bandwidth networks with high payload capacity would receive fragmented datagrams wasting the Maximum Transmission Size (MTU) if there exists a fragmentation point between and end-to-end communication. As examine in [3] fragmentation could occur anywhere in the path from node A to node B. We know that "Reassembling datagrams at the ultimate destination can lead to inefficiency; even if some of the physical networks encountered after the point of fragmentation have large MTU capability, only small fragments traverse them."[2] As said in [3] there are two main options for handling this situation: Routers between A to B must have an equivalent frame carrier structure that produce a 1 to 1 payload relationship, or de-fragmentation offloading at receiving end. So increasing the payload size will only generate more fragments from any packet transmitted.

3. Processing TCP/IP in hardware

As we seen before, TCP/IP could be process in hardware to handle the problems exposed. Part of the Parallel TCP/IP Offloading Framework will be exposed as a design model for a TCP/IP firmware. In [10] the framework is discussed in detail. These implementation must be fast, modular, easy to interface, addressed the issues presented, up to date and scalable. The parallel implementation must maximize parallelism.

3.1 Frame Carrier and Offloading

As said in [3] the frame carrier has to be offloaded to the maximum for a good TCP/IP offloading implementation. A proposed way to deal with these is to include the frame carrier code inside the firmware. Today, the physical layer and part of the data link use the net device hardware. Some data link layer code (as the Linux Packet Filter layer) is needed by upper layers of the Internet related protocols.

3.2 Parallelism inside and outside the TOE firmware

There are two approaches. Inside the firmware, parallel processing can be achieved. Receiving and Sending modules can work in parallel within the firmware (figure 1). By using references to the payload received by the frame carrier, parallel communication can be achieved via shared memory. Besides, firmware modules can be combined together

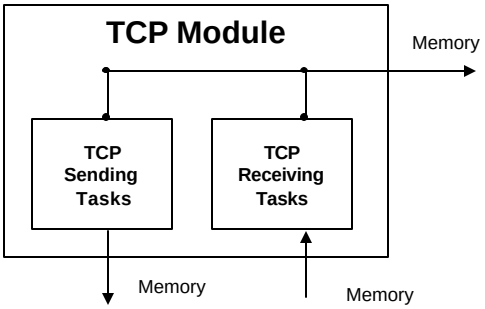

Figure 1 Parallelism within the TCP Module

to work in parallel. The first idea will be discuss in the next sections. The last one is under research by the date this paper was written.

Within the firmware, parallelism can be achieved in various modules. TCP and IP modules can be working separately. Moreover, the receiving and the sending tasks can work in parallel within the modules of the Internet related protocols.

3.4 Layers and Memory Interaction

Modules will be connected to input and output buffers. Upper and lower layers have access to these buffers. They place data in this memory for the firmware modules to handle. Once the data is available, modules begin to work on them. The input and output buffers maintain the flux of full-duplex communication through the entire framework. These shared buffers are depicted in Figure 2.

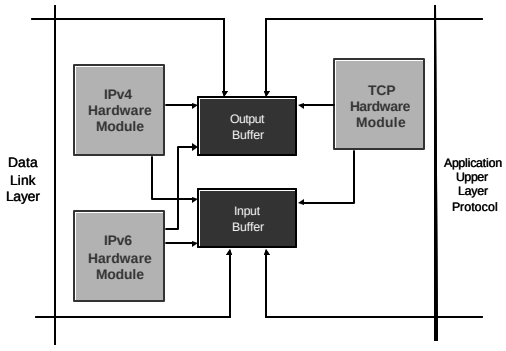

Figure 2 Module and memory interaction diagram

The frame carrier resides in the data link layer of the Internet layer model. It is the modules responsibility to get rid of the frame carrier's header and copy its content to the input buffer. Also, the frame carrier must encapsulate datagrams residing in the output buffer for delivering data to the Internet Gateway. When an IP datagram is ready for delivery the frame carrier must transmit it.

3.5 Memory Management

There are two types of memory: local and shared memory. Local memory resides on every module. These local memories are the registers or cache memory residing inside the modules. Shared memory can be contacted by any module using pointers. Every module uses its local memory for saving references and offsets. This memory will store temporary values needed for the execution of the module. In addition, the registers will store offsets for values relative to the data referenced.

Each module will control indexes that reference the datagram being handled. The index amount that a specific module will control depends directly on the module actions.

The best description for the shared memory area is RAM. Everything resides here, e.g. space for shared variables, indexes and buffers. In the framework constructed, buffers are depicted separately from the memory area for the sake of clarity. In a real implementation, they could also be separated from RAM using some dedicated memory area.

Figure 3 depicts an example of a memory area referenced by indexes used in IPv4 and TCP receiving modules. Pointers are pass within indexes

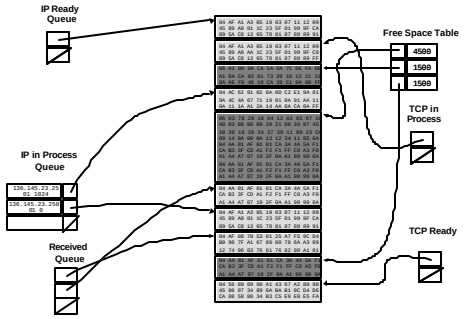

Figure 3 Indexes referencing the share memory area

control by the same module. The input index and the output index have limited control access for every module. This indexes conforms the coordination within the modules. This coordination will be discussed later.

Every time a datagram departs or arrives in the input buffer, it must consult the Memory Manager Module.

This module will manage space in the input/output buffers with an index called a Free Space Table. By consulting this table, the memory manager module can allocate space for incoming payloads. When a datagram depart one of the buffers, a new value must be registered in the Free Space Table. The table is used to manage free space inside the buffers. Its structure contains: a reference to the input or output buffer, a buffer identification, and the size of the available memory area.

3.6 Coordination

The parallel framework is designed to work following a state machine cycle depicted in figure 4. Each process will be blocked if there are no datagram references in its input index. Following an insertion, the hardware starts processing the datagram. The references within the input index could be moved to another index control by the same module. After processing, the resulting datagram reference is placed in the output index. Another module uses this output index as its input index. Indexes to the buffer areas depend directly on each module. In the TCP receiving module, the input index is the IP ready index and that index is the output index from the IP receive module. The output index for TCP will be TCP ready. Then the application layer or another protocol on top of TCP could use this index to reference the already TCP/IP process datagram.

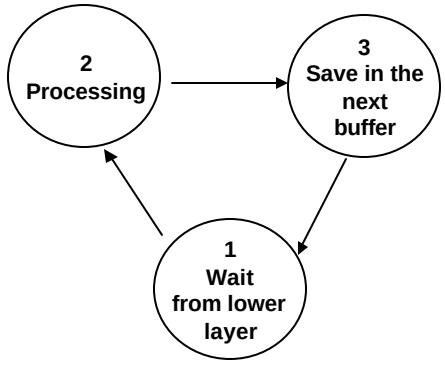

Figure 4 IPv4 state machine module example

Whenever a module places a reference into its output index then it goes back to state one. It will wait if there is no reference in its input index.

3.7 TCP/IP modules

TCP and IP will be handle in separate modules. Within the modules, also, the receiving and sending tasks will be handle separately in order to maximize parallelism.

3.7.1 IPv4 modules

IPv4 receiving tasks will be divided in two stages. First stage is composed of four modules than run in parallel. These modules are: version verifier, checksum handler, and fragmentation handler. If any of these modules found an exemption then a message is send to a discard module. This module will release the memory area where the datagram reside. The second stage will identify what protocol type resides inside the IPv4 data area.

Sending tasks in IPv4 are handle by three stages rather than two. During the first stage, a Pre-Datagram Builder, and the Identifier and Fragmentation Handler work. The Pre-Datagram Builder will handle the following: version number, service type, source address, and time to live (TTL) fields. These values tend to be static values that do not change over orthodox conditions. If fragmentation is inevitable, the Fragmentation Handler identifies every datagram with a unique identifier and a fragment offset number. It will handle the packet received by the upper protocol (transport layer) and fragment it for delivery over low MTU connections. As well, this module sets the flags related to the fragments. IP options, if any, are handled here.

3.7.2 TCP modules

TCP is more complex than IPv4 because it is a connection-oriented protocol. TCP, in addition, is a full-duplex protocol. Inside the framework, as said before, the coordination is done via shared memory. When a datagram arrives, it goes to a first stage composed of three modules: Checksum Verifier, Code Bits Interpreter, and Window Space Verifier. If any of these modules fail, then it will broadcast a message through the discard bus. Any of the modules working should check if they are handling the same datagram broadcasted over the discard bus. If that is the case, these modules will reset their execution with the next datagram in the buffer. Consequently, this datagrams must be handled by their respective sequence numbers. After the first stage, the full-duplex communication is handled. If the current communication is in full-duplex mode, then the acknowledgement number is passed to the Sliding Window Handler. Also, the sequence handler, reports

its octet offset and passes the control to the Port Handler. Finally, the port handler passes its data to the upper layer protocol or the application.

TCP sending tasks are quite different. During the first stage, the Flow Control Handler received data from the upper layer (application or other protocol) for TCP encapsulation. Based on the sliding window algorithm chosen, this module will handle flow control and congestion control. The Port Handler is another module that exists on the TCP Offloaded Framework first stage. This handler is not the same as the one in the receiving tasks. Its tasks are the assignment of the destination and source port fields in the TCP packet. It will plug the socket layer into the appropriate port specified.

During the second stage, a Window Advertiser, Acknowledge Number Generator, Code-Bits Handler and TCP Options modules run in parallel. The Window Advertiser module depends on the TCP receiving tasks. It advertises window size space available in the receiving end of the full-duplex communication. The Acknowledge Number Generator depends on the TCP receiving tasks. It seeks the last sequence number received in order and generates the next number that it expects to receive. Then this number is copied to the corresponding TCP field. The TCP Options will be copied to the packet here if any.

The third stage conforms a Pseudo Header Generator and a Header Length counter. Once the first and second stages have been completed, the Header Length module counts the number of octets that form the TCP header. Then, it copies its value to the TCP header length field. That's all. The Pseudo Header Generator module requires some IP information. It is assumed that this information is already in the shared memory area of the framework. Unlike the other modules, this one is not attached directly to a field. It uses the memory area instead. The pseudo header is created and placed in the shared memory area. The next stage will use it for checksum header calculation.

Finally, the datagram is ready for delivery. Before this, we need to calculate its checksum with a Checksum Calculator module. For the calculation of the TCP header, both the data area and the TCP pseudo header are considered. Then the packet has been process in the TCP sending module.

3.8 The needs of a Socket Interface Layer

All UNIX clones have a common socket interface based all in Berkely Software Distribution (BSD). Also, Microsoft found a similar approach. The socket interface is the same but the source code of all the implementations is kernel dependant. A TCP/IP implementation inside a firmware needs a similar socket interface layer and must accommodate all the options used by the standard BSD socket interface. By this way, any program should run without heavy modifications.

4. Violations to the layer model

In a TCP/IP software implementation, the Internet layer model must be followed. Module modifications and bug fixes in software can be achieved if designer, kernel programmers, hackers, or contributors follow a protocol layer design. Open software is available from different flavors of UNIX/LINUX distributions. Aberrations and violations of the Internet Layer Model are possible if the implementation is done in hardware. The reason for layer design is the ease of software implementation, design and modification. The TCP/IP framework does not violate the orthodox protocol design. After all, these violations are not discarded during the research project. We will not discard the possibility that for a dedicated hardware, violating the layer approach could result in performance increase. That is still on research today.

5. Challenges

We have stated that a TCP/IP firmware must include a socket interface layer. The challenge relies on how to interface the layer coded in hardware with the operating system. A solution can be achieved in open source operating systems. A special kind of driver must be compiled within the kernel in order to achieve maximum performance. The framework is based on protocol handling. Security aspects of this framework are far from the scope of this research project. The framework does not address any security issues at all.

Implementing a TCP/IP protocol stack in hardware is not an easy task. As said in [11] "Implementing TCP/IP completely in hardware poses a significant technical challenge". This challenge leads to another challenge. Does a TCP/IP protocol stack implementation is feasible?

6. Future Work

A Simulation using two PCs with the same configuration is conducted in order to measure: throughput, latency and CPU utilization. No research has been conducted before in TCP/IP offloading

measuring these [11]. This test will use specific re-compiled kernels with and without the IP stack activated. In addition, a hardware candidate is studied for a future research.

References

[1] *Broadcom Gigabit Adapter: Performance Testing Comparison*. Etesting Labs, March 2001. http://www.etestinglabs.com

[2] Comer Douglas. *Computer Networks and* Internets. Prentice Hall. Upper Saddle River NJ, USA 2001.

[3] Couvertier Isidoro, Solá Juan M. TCP/IP Offloading Framework for a TCP/IP Offloading Implementation. *Computing Research Conference 2002. University of Puerto Rico, Mayaguez Campus.* http://mayaweb.upr.clu.edu/crc/crc2002/papers/Sola_Juan.pdf

[4] Eisner Gillet, Sharon., Lehr, William H., Wroclawski, John T, Clark, David D. Do Appliances Threaten Internet Innovation? *IEEE Communication Magazine*. October 2001.

[5] Guardiani Ivano. Migrating from IPv4 to IPv6: Planning and Effective IPv6 Transition. *Telecom Italia Lab*. http://carmen.cselt.it/papers/globalIPsummit-v6trans/home.html

[6] Lawton George. Is IPv6 Finally Gaining Ground? *Computer Innovative Technology Computer Professionals.* IEEE Computer Society. August 2001.

[7] Moyer, Stan., Marples, Dave., Tsang Simon., (Telcordia Technologies, Inc.) A Protocol for Wide-Area Secure Network Appliance Communication. *IEEE Communication Magazine*. October 2001.

[8] Riihijärvi, Janne., Mähönen Petri., Saaranen Mika J. (University of Oulu) Roivanen, Jussi., Soininen, Juha-Pekka. (VTT Electronics), Providing Network Connectivity for Small Appliances: A Functionality Minimized Embedded Web Server. *IEEE Communication Magazine*. October 2001.

[9] Semeria Chuck. Implementing a Flexible Hardware-based Router for the New IP Infrastructure. *Juniper Networks Inc.* http://www.juniper.net/techcenter/techpapers/200015-01.html

[10] Sola Juan M., A Parallel TCP/IP Offloading Framework for a Parallel TCP/IP Offloading Implementation. http://www.geocities.com/Juan_Sola/toe.html

[11] Yeh, Eric., Chao, Herman., Mannem Venu., Gervais, Joe., Bradley, Booth. Introduction to TCP/IP Offload Engine (TOE). *10 Gigabit Ethernet Alliance*. Version 1.0 April 2002 http://www.10gea.org