A TCP Offload Engine Emulator for Estimating the Impact of Removing Protocol Processing From a Host Running Apache HTTP Server

Juan M. Solá-Sloan and Isidoro Couvertier-Reyes
University of Puerto Rico, Mayagüez Campus
email: jsola@ece.uprm.edu, icouver@ece.uprm.edu

Keywords: TOE, Protocol Offload, TCP Offload, hacking, Apache, Web Servers

Abstract
This article focuses on an emulator used for validating a model for TCP offload. The TCP Offload Engine Emulator (TOE-Em) is a full TCP offload program that emulates the behavior of a TOE device using another PC as a front-end. Its purpose is to study the impact of full offload in a real environment. We confronted Apache 2.2 with and without the support of the TOE-Em by requesting different file sets. The CPU utilization and the delay of the system was obtained and analyzed. In our context, full TCP offload was beneficial for files greater than 10 KB.

1. INTRODUCTION
Traditionally, protocol processing is implemented as part of the operating system (OS) of a host. The need for faster responses encourages their implementation into a specialized hardware. Hardware implementations have a tendency of achieving higher performance than software ones, but with a significant increase in the implementation effort due to their complexity and cost. It is common to see how vendors deviate from hardware implementations when faster CPUs are available. Novel methods for understanding the specific features of protocol processing are needed for deciding if a hardware or software implementation suffices.

A TCP offload engine (TOE) is an entity outside the main CPU of the host that handles TCP/IP in an end-to-end communication [8]. A TOE device alleviates the problem that arises when application and protocol processing compete for a share of CPU time. There is no guarantee that the inclusion of a TOE within the system results in performance increases. A poorly designed interface between the CPU and the TOE is likely to ruin most of the performance benefit brought by offloading [20]. The result is a system that is unable to provide its clients with better or similar response times than the non-TOE host. However, the inclusion of an OE could be beneficial in some cases [16].

Studying the benefits of full offload is cumbersome and costly. The performance benefits of full offload have been studied using SMPs and high performance computers but scarcely on off-the-shelf hardware [32]. Studies have been performed on similar hardware but are focused on some parts of TCP offload [10, 12]. We created the TOE-Em for two main reasons: A hardware implementation is complex and extremely costly, and finding hardware that suited our needs was difficult. Also, we did not want to fall on the same pitfalls that others fell as a result of hardware constraints [3].

On the advent of multi-core multi-threaded processors and new technologies on outboard processors a new debate related to where protocol processing must be performed arises. There exist the possibility of processing the protocols in another core of a multi-core CPU. The TOE-Em was used to hit the tip of the iceberg on this possible track.

2. RELATED WORK
Processing the TCP protocol outside the OS have been suggested as early as 1988. In [5], the authors suggested that some of the operations needed to handle the packets could be outboard onto a special controller. This idea has arisen today with the development of a faster MAC layer protocol (10 Gbps Ethernet). High overhead for packet processing can leave few CPU cycles available for actual application processing [6, 8, 12, 29].

TOE have not yet demonstrated significant performance benefits [20, 25] possibly because the current scarcity of NIC hardware resources that limits scalability [4]. Full offload is not new, however, it has never succeeded for complex general-purpose protocols such as TCP and IP, as vendors claim [32]. In [4], Brecht, et.al. stated that while future TOE designs will likely show improvements, the processing elements in TOE are likely to always be behind the performance curve of mainstream processors. The resource limitations of a peripheral device limit the maximum processing capability and memory capacity of a TOE device [17].

As multi-core processors provide mechanisms for fast inter-core communication, it becomes more attractive to dedicate some cores to protocol processing as done in [2, 4, 14, 23, 24]. There exist investigations that are focused on optimizing TCP processing itself for reducing the resources that TCP consumes. The goal in [11] is to lessen memory usage and to increase the scalability of TCP. Other works focuses on a mixture of hardware and software techniques that allowed proper processing of the protocols [10].
4. ANALYTICAL MODEL OVERVIEW

Although, this article emphasizes on the TOE-Em implementation we are presenting a brief summary of our probabilistic model in this section.

The performance metrics obtained from the analytical model are utilization and delay. The CPU utilization is abstracted by multiplying the arrival rate of requests ($\lambda$) by the expected time used to process it by the application ($E[X_o]$) plus the expected time used to convert the file into packets of protocol $P_x(E[X_o])$. Therefore: $U_{cpu} = \lambda(E[X_o] + E[X_p])$.

When using an OE $E[X_p]$ is eliminated from the equation but a new random variable emerges. Let $X_o$ be a random variable that describes the processing of the overhead when communicating with the entity. Then, the utilization is $U_{cpu}' = \lambda(E[X_o] + E[X_o])$. Notice that if $E[X_o] > E[X_p]$ then inclusion of an OE degrades the system.

The analytical model is composed of two tandem queues. The first one models the CPU and the second one models the TOE. Using the P-K formula of the M/G/1 queue the delay within each queue has been estimated. The delay of the system is the maximum delay among the queues. For further information about the analytical model please refer to [28].

5. EXPERIMENTAL SETUP

In order to validate our model, a test bed was setup. The test bed was composed of two PCs. One of them acted as the Web application server and the other as the TOE device (see Figure 1). The PC that acted as the TOE device was called the front-end PC (FEPC). These two PCs are connected together via a 1 Gbps connection using a crossover cable. The Web server (WS) communicates with the outside world via the FEPC. The FEPC runs the TOE-Em. Two more nodes were used to generate the workload and the requests that were sent to the test bed. The specific architecture of this test bed is presented on Section 5.3.

5.1. The TCP Offload Engine Emulator

The TCP Offload Engine Emulator (TOE-Em) is a multiprocess program written in ANSI C that have all the functionality of our simulated TOE device. The TOE-Em provides the socket interface to the WS. This program is the gate in which packets come in and go out of the system. The following sections discuss the TOE-Em in detail.

5.1.1. TOE-Em Composition

The TOE-Em is a program that spawns processes and maps them to similar processes running at the WS. This is achieved by using dedicated processes that interact with each other via UNIX System V IPC [30]. The TOE-Em is composed of three types of processes: the Commander and Reader process (CR), the Forker Process (FP), and the Slave Processes (SP). The following paragraphs describe each of them in detail.

The CR process is the most important process of the TOE-Em. Upon start up, the CR enters into a critical loop. If there is a command waiting to be read, the CR reads it and stores it. If not, the CR blocks (see Figure 2a). However, if the read is successful, then, three conditions are tested. After examining a command, if it is one that is not destined to a particular SP, then, the CR process assigns the command to an available SP to process it. If the command is destined to a specific SP, the CR checks if the SP is available to process the command. If available, the CR signals it. The SP receives this signal and
starts working (see Figure 2a). Finally, if the command is for the FP, the CR checks for its availability and, if available, signals it. If any of these conditions is not met, an error occurs and the TOE-Em ends.

The function of the slave process is executing the commands assigned by the CR. Immediately, after it is spawned, the SP enters into a critical loop (see Figure 2b). Each and every SP, including the FP, checks if there is a notification from the CR indicating that a command is awaiting to be processed. If there is one, the SP identifies the command and if valid, issues the command. Finally, the SP releases its resources and loops back to the beginning. The SP blocks if there is nothing to do.

The FP is a specialized SP that spawns SPs and allocate resources for them. Also, the FP removes any slaves and its related resources. The FP is created when the first child process is spawned.

5.1.2. Process Mapping

A socket descriptor is an identifier returned by the OS used to reference an instance that is associated with a socket. The descriptors created by each SP are only available for that particular SP. This happens since the kernel at the FEPC assigns the socket descriptor to a specific process identifier (PID). All the state variables used to handle the TCP module exist in the context of that particular PID. However, two different PID could assign the same socket descriptor number, but reside in two different contexts. A way of identifying this peculiarity was devised. A new descriptor was created by the TOE-Em and was sent back to the WS that is a combination of the PID and the actual socket descriptor. This was done via the build_key function for mapping the socket descriptor within the right context (see Figure 3). The function creates a new descriptor by multiplying the actual descriptor number by 10^6 and adding the PID as a number. This works for us since the maximum length for a PID in the LINUX version we used was six digits. This new socket descriptor is given to the WS as a valid one. Also, Figure 3 presents a function that breaks the new descriptor into the PID and descriptor number pair.

```
#define PID_KEY_ 1000000
typedef unsigned int u_int;
int build_key(u_int mypid, int skt)
    { return (mypid + skt * PID_KEY_); }

void break_key(int npid, u_int *mypid, int *skt)
    { (*skt) = (int) npid / PID_KEY_;  
      (*mypid) = (u_int) npid % PID_KEY_; }
```

Figure 3. Source Code to Build and Break the Key
5.1.3. Process Coordination

In order to coexist, the CR and SP need to communicate with each other. This is done via the Process Control Table (PCT) that resides in shared memory. The CR uses the PCT for consulting if the SP is available for accepting commands and to assign the command to that particular SP. Then, the SP consults the table and executes the command. This is done by examining the data_entry field of the PCT. If this field is empty there is nothing to do and the process blocks. The process remains blocked until a signal is received from the CR.

One advantage of this approach is that the SP always points to its appropriate control slot. Upon creation, a slot to this table is assigned to every single PID. The address to this slot references the PCT record for each PID. The SP uses this address to consult the PCT.

5.1.4. TOE-Em Memory Management

During the design and implementation phase of the TOE-Em, memory management was a key issue that was not overlooked. The design had a top priority of avoiding heavy memory allocations. Stacks were used for reducing memory allocations. The TOE-Em initializes a share memory area of a size that is around 370 KB upon startup. This size is the result of multiplying the maximum number of processes that the TOE-Em can handle (250) by the size of the Ethernet frame (1514). This is done since every socket command that the WS generates travels within a single frame. Consequently, the 370 KB area can be partitioned in 250 slots of 1514 bytes each. This makes addressing an easy task that can be carried by offsetting. This simple calculation is faster than searching over the entire memory area for a slot. The stack is used to store all the addresses that are free. Therefore, every time a command arrives from the WS, the TOE-Em, pops an address from this stack, and stores the incoming frame. After identifying the target SP, the CR, assigns the slot to an SP and then signals it. This slot is not pushed into the stack again until the SP finishes with it (recall Figure 2b). This slot becomes a free slot. The CR, then, without the need to allocate more memory, could reuse this buffer again by “popping” its address from the stack. Notice that race conditions could surface by using this design, however, semaphores were used to avoid this type of errors.

5.1.5. TOE-Em Commands and Raw Sockets

Raw Sockets were used to bypass TCP/IP on the WS. LINUX raw sockets provides the capability for handling Ethernet frames directly from user space (also called packet sockets). In order to send commands to and from the TOE, a special control protocol was used. The protocol was called the raw protocol and is presented in Figure 4. This is used to simulate the communication between the CPU and the TOE. The raw protocol’s header is composed of four main fields.

The first one is the command field, which identifies the command to be executed either at the WS or at the TOE. The second field is the optional parameters length. The third field is the data length and is used on some commands that send data directly via the payload of the frame. The fourth field is the destination PID. This is used sometimes to indicate which PID is going to handle the command. The last field is optional and is used to send additional control structures. The length of this field varies and is indicated by the second field.

![Raw Protocol](image)

Raw Protocol Header

- Command ID
- Parameters Length
- Data Length
- Destination PID
- (Optional)

Ethernet Header Raw Protocol Payload Trailer

Every POSIX socket command is mapped into the TOE-Em by identifying the command field on the raw protocol’s header. Therefore, every time a socket function is invoked at the WS, a frame is created containing the raw protocol header. The TOE-Em reads this frame and executes the command. If the TOE-Em needs to send a reply to the WS, another frame is created.

In [22], the sendfile() system call was proposed for reducing the number of memory copies that result from passing the data through multiple OS contexts. The TOE-Em provides a way of simulating this system call. Notice that, whenever files are going to be sent via sendfile(), the file resides on both PCs. The file is open and processed normally as Apache does it at the server. However, all files required for our test also reside on the FEPC. A command is sent to the FEPC with the filename to transfer whenever sendfile() is invoked at the server. This simulates a TOE with the capabilities proposed in [20, 22].

Apache does not transfer all its files using sendfile(). Small files are transmitted using write() or writev(). The raw protocol carries the data into its payload when sendfile() is used. The TOE-Em is supported for reading the data directly from the raw protocol’s payload. If dynamic content is going to be transferred from the server to the FEPC it must not use sendfile(). However, this aspect was not tested.

5.2. Hacked Version of Apache 2.2

The WS application selected was Apache version 2.2. We observed this for files less than 8KB
accounts for approximately 65% of all Web domains on the Internet [21]. Another important aspect is that the WS is open source and can be modified for suiting our needs.

A hacked\textsuperscript{2} version of Apache was used to interface it with our TOE-Em. This version includes a modified socket interface that uses raw sockets to communicate to the TOE-Em. The entire socket interface of Apache 2.2 was changed in order to adapt it to our needs. Four of the Apache 2.2 sources were modified: sockets.c, sockaddr.c,sockopt.c, sendrecv.c.

The Apache 2.2 core application was mostly maintained intact. Altering Apache’s core substantially could result in diverging from our scope. The hacked version stays as a multi-process program with a slight modification that is described as follows. Upon startup, Apache forks into several child processes by default. Each child listens to TCP port 80 and waits for a connection. In order to alert the TOE-Em of this event a new function was called from prefork.c that issues a notification that a new process has been spawned. Then, the TOE-Em could spawn a new SP and maps it to this newly created child. This is the only intrusion we made to Apache’s core application.

5.3. System Architecture

The hosts have the following configuration: The FEPC was an Athlon XP 2600+ 1.9 Ghz with a cache size of 512 KB and 756MB of RAM, the PC acting as our server was an Intel Pentium 1.8 Ghz with a cache size of 128KB with 512MB of RAM. Both PCs run LINUX with kernel version 2.6-18.

Several nodes were used to act as users requesting pages from the Web server (WS). The main machine was a Genuine Intel Pentium 1.6 Ghz with a cache size 512MB of RAM and running kernel version 2.6-18. Another PC was used for generating the file set and this was an AMD K6-2 600Mhz with kernel version 2.4.8.

5.4. Workload Generation

We combine notions from different benchmarks to stress our testbed in different ways. Our target is to study how an offloaded host behaves for different types of file sizes. Webstone 2.5 was used for testing our system [31]. Webstone was chosen since it is open source, and previous works are available for validating our results. Since Webstone is bundled with a limited file system, SURGE was used for generating the set of files for our test bed [1]. We did not use only SURGE since it could not be configured easily for our purposes.

The 2000 files generated by SURGE has been classified into four different classes similar to the way it was done in [13]. Table 1 presents these classes. Notice that the files are grouped based on their sizes. Files with sizes ranging from 0 to 1 KB has been classified as Class 0. Class 1 contains files with length greater than 1KB and so on.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 - 1KB</td>
</tr>
<tr>
<td>1</td>
<td>1KB - 10KB</td>
</tr>
<tr>
<td>2</td>
<td>10KB - 100KB</td>
</tr>
<tr>
<td>3</td>
<td>100KB - 1 MB</td>
</tr>
<tr>
<td>4</td>
<td>Over 1MB</td>
</tr>
</tbody>
</table>

Table 1. File Classes

6. ANALYSIS OF RESULTS

Tests were run for all the classes three times for 10 minutes each. Our test bed was confronted with a default Apache 2.2 configuration for a single processor system with no protocol offload capabilities. The /proc file system was used to acquire our results. Webstone 2.5 was supported with inter-arrival times following an exponential distribution for confronting our analytical model with our implementation.

Figure 5 presents the CPU utilization of the application server estimated by the analytical model and obtained from the test bed. The results from the analytical model are presented with lines and the results obtained from the test bed as points. The NON-TOE configuration corresponds to a host running the unhacked Apache while the TOE configuration corresponds to the same host running the hacked Apache. Notice that for Class 1, the CPU of the server increments its load when the TOE is included. This also happens for Class 0 (not illustrated). Therefore, based in our measurements, offload is not beneficial for files with sizes less or equal to 10 KB. However, for files with sizes greater than 10 KB the CPU utilization of the host is reduced. The most substantial improvement is achieved for files greater than 1024 KB (1 MB).

Table 2 presents the delay estimated by our analytical model versus the ones obtained by the emulator for Class 0 (file size < 1KB) and Class 4 (file size > 1MB). This is a summary of the preliminary results we are currently validating at the time this article was written. These classes were chosen since both represent totally different event sets. The application processing and the overhead used to communicate with the TOE dominate over the time it takes to handle the actual data transfer at the TOE when the server is processing files for Class 0. Therefore, the bottleneck of the system is the CPU. However, as files became larger and larger the bottleneck of the system is shifted to the TOE.

7. CONCLUSIONS

The first version of the TOE-Em presented in this article has been a success for our purposes. We were able to validate
Figure 5. Utilization Estimated Versus Obtained

<table>
<thead>
<tr>
<th>Class 0 Estimated</th>
<th>Class 0 TOE-Em</th>
<th>Class 4 Estimated</th>
<th>Class 4 TOE-Em</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00353</td>
<td>0.00337</td>
<td>0.03161</td>
<td>0.03088</td>
</tr>
<tr>
<td>0.00380</td>
<td>0.00371</td>
<td>0.03288</td>
<td>0.03119</td>
</tr>
<tr>
<td>0.00481</td>
<td>0.00446</td>
<td>0.03450</td>
<td>0.03292</td>
</tr>
<tr>
<td>0.00617</td>
<td>0.00535</td>
<td>0.04370</td>
<td>0.04372</td>
</tr>
</tbody>
</table>

Table 2. Delay Estimated by The Model and Obtained from the Emulator

Our analytical model since it has estimated the utilization of the system with a $p$-value greater than 0.75 on a K-S test.

Our way of classifying files based on their sizes gave us a new perspective of the problem. Offloading protocol processing is beneficial for some files only in our context. Performance benefits could be concealed when running a test with the full file set. It is known that small files are accessed more than large ones [1]. Therefore, we need to be very careful in deciding what is beneficial or not. The OS should discriminate which files are the ones who will be offloaded to the TOE and which not. Researchers should follow our approach of classifying files in order to get a better picture of the problem. This is true since it is known that a system is stressed differently when handling different file sizes [13].

Our approach, simulates the exchange of control messages between the OE and the WS. The raw protocol creates an abstraction of the commands involved in the communication between the TOE and the WS. In a real implementation, this happens inside the host. We assumed that when the TOE is part of the host, the communication between the TOE and the CPU, is faster than in our current test bed. However, the contribution of this research is using the simulation to validate our analytical model. Also, our approach is apt for analyzing the performance of the system.

8. FUTURE WORK AND FINAL REMARKS

Our TOE-Em could be optimized even further. The TOE-Em has a design flaw that needs to be improved on a second version of the emulator. This is an issue that involves the CR process. This process stops reading frames from the NIC whenever it finds that the SP it needs is busy. The CR, blocks until the SP is available. All other incoming com-
mands must wait for this SP to be ready. Incoming frames containing the commands are buffered and are not available until the next recvfrom(). Adding a queue for storing the commands seems to unravel this issue. This was analyzed under the design phase of the TOE-Em, however, the idea of implementing a queue conflicted with the memory management constraints imposed by us. Two approaches can be used; either handling the memory in a shared area, or using dedicated memory for every process and passing the incoming frames via pipes. Consequently, this approach impacts the original design heavily. The designer of a real offload engine must maintain memory copies at minimum for achieving better performance. This issue has to be solve in order to improve performance of the TOE-Em.

Obtaining measurements when the host utilization was near 100% was sometimes cumbersome. Ethernet frames containing the commands were dropped when the FEPC was saturated. Frames containing commands began to be dropped at the FEPC when the NIC’s buffer overflowed. The raw protocol used by the TOE-Em did not provide a way of handling flow control. Consequently, sometimes the whole system hanged near the saturation point of the FEPC. This has to be solved if the TOE-Em is going to be used in a real scenario as a front-end.

ACKNOWLEDGEMENTS

The authors would like to thank all colleagues who contributed to this study. We want to thank Dr. Nayda Santes of the ECE Department of the University of Puerto Rico at Mayagüez for her collaboration and comments. Also, we want to thank Dr. José Vega Vilca of the Institute of Statistics of the University of Puerto Rico at Rio Piedras for his collaboration during the design of the analytical model.

REFERENCES


