Design of a Lightweight TCP/IP Protocol Stack with an Event-Driven Scheduler

JOONHYOUK JANG1, JINMAN JUNG1, YOOKUN CHO1, SANGHOON CHOI2 AND SUNG Y. SHIN3

1School of Computer Science and Engineering
Seoul National University
Seoul, 151-744 Korea
2School of Computing
Soongsil University
Seoul, 156-743 Korea
3Electric Engineering and Computer Science Department
South Dakota State University
Brookings, SD57007 USA

The traditional TCP/IP protocol stack is associated with shortcomings related to the context-switching overhead and redundant data copying. The software-based TOE (TCP/IP Offload Engine), also known as lightweight TCP/IP, was developed to optimize the TCP/IP protocol stack to run on an embedded system. In this paper, we propose the design of a lightweight TCP/IP protocol stack that runs on an event-driven scheduler. An event-driven scheduler is one of the main components of a real-time operating system that provides essential functionalities for an embedded system in network communication. We discuss the problems involved in designing a lightweight TCP/IP with an event-driven scheduler, especially for the issues of TCP transmission and TCP retransmission. We implemented and evaluated the proposed TCP/IP stack on an embedded networking device and verified that the proposed TCP/IP stack is well suited for high-performance networking in embedded systems.

Keywords: TCP/IP, TCP/IP offload engine, embedded system

1. INTRODUCTION

With the rapid growth of wired/wireless networks, embedded systems are required to have a high capability for network communications. Several studies have been done to accelerate the speed of packet processing in embedded devices that are uniquely applicable to network communications. Considering the communication environments of the embedded devices, these studies focus on enhancing the TCP protocol [1] and optimizing the TCP/IP implementation [2].

TOE (TCP/IP Offload Engine) [3, 4] refers to a type of implementation of a protocol stack that is optimized for embedded devices. Although general TOEs are implemented in hardware [2, 5], in some cases TOEs are implemented in software [6, 7]. Software-based TOEs have high degrees of flexibility and a low cost compared to hardware-based TOEs. A protocol stack implemented in a TOE is designed to support more efficient packet processing by overcoming the shortcomings of the traditional TCP/IP protocol.

Received May 31, 2011; accepted March 31, 2012.
Communicated by Jinyoung Heo and Tei-Wei Kuo.
* This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0027454) and by Ministry of Culture, Sports and Tourism (MCST) and from Korea Copyright Commission in 2011.
Software-based TOEs, sometimes referred to as lightweight TCP/IPs, run on real-time operating systems. Because real-time operating systems do not support the full functionality of general operating systems, the software-based TOE architecture is customized to the real-time operating system in which runs. This paper proposes the design of a lightweight TCP/IP protocol, which runs on an event-driven scheduler in a real-time operating system. We also define the problems involved in the TCP transmission and retransmission on an event-driven scheduler.

The rest of this paper is organized as follows. In section 2, we introduce previous works on software-based TOEs or the lightweight TCP/IP protocol. In section 3, we define the execution environments associated with the problem. In section 4, we discuss the problems and their solutions as regards TCP transmission and retransmission in the execution environments described in section 3. In section 5, we evaluate our implementation, and in section 6, we conclude this paper.

2. SOFTWARE-BASED TCP/IP OFFLOAD ENGINE

The traditional TCP/IP protocol is designed as a layered architecture. This design concept is not suitable for running on an embedded device because the resource limitations of an embedded device-size of binary executable and memory space-are not considered. In addition, the overhead of context-switching slows down the packet processing, as the layers or the network protocols are executed in separate processes. Each of the software layers handles a data unit of its own form as used in the specific protocol and manages the data structure to maintain the states of the data unit. For example, a TCP connection processes a bit stream, which is divided into segments, while the IP protocol processes the datagram. In an actual implementation case, there are many more software layers, and they all maintain a specific form of data structure, consuming memory space and processing time while dealing with data packets. In particular, data copying between the layers is the most time-consuming process in the network protocol stack. When transmitting data packets, each layer copies data from the packet buffer in the upper layer to its packet buffer and transfers the data to the lower layer after processing the data packets. This occurs in reverse order when receiving data packets.

The most remarkable feature of a software-based TOE, aka lightweight TCP/IP, is the elimination of the overhead of managing complex data structures and copying redundant data packets between software layers in the traditional design of the network protocol stack [8-11]. It provides the layers with unified methods of accessing a shared buffer space to allocate and de-allocate packet buffers. In this way, data structures can be simplified and the number of redundant, time-consuming data copying events is reduced to zero, one, or two. Zero-copy packet transmission and reception can be achieved by integrating applications and network drivers together into a unified buffer management scheme. Otherwise, one or two instances of copying can be allowed to preserve the independence of the applications and network drivers. Another feature of lightweight TCP/IP is its software architecture, which is designed to integrate software layers. Because the layers are not separated strictly, they can stick together or flexibly take charge of the functionalities of the network protocols. With this feature, the program binaries occupy less storage space and the communication between the layers can be simplified. More-
over, the software can be executed in only one or two processes, resulting in more efficient network communication than in the traditional design of the protocol stack.

Previous works in this research area introduced micro-IP [12], lwIP [13], tinyTCP [14], NexGenIP [15] and NETX [16]. Particularly, lwIP, open source software developed by SICS, is one of the most well-known implementations of lightweight TCP/IP. It provides IP, ICMP, UDP, TCP, and DHCP and is designed to support various real-time operating systems. lwIP implements the pBuf, structure which dynamically allocates and de-allocates packet buffers to increase the packet processing performance. Several studies have been done to port lwIP onto real-time operating systems or to optimize a lightweight TCP/IP for an embedded platform [17-19].

3. OVERALL SOFTWARE ARCHITECTURE

3.1 Execution Environments

The embedded system described in this paper is assumed to be a device dedicated to network communication. The program is compiled in one binary file which is embedded in the system. This implies that the software components included in the binary file are executed in one control flow with a main function which calls them. The software embedded in the device consists of the main program, applications, a TCP/IP protocol stack, and a real-time operating system (Fig. 1). The operating system is simplified, including an event-driven scheduler, a network device driver, and a timer. Upon execution, the main program calls an initialization routine and the scheduler. The scheduler executes tasks which are in the ready state. Putting a task into the wait queue in the schedule is implemented as registering a function pointer in it. The initialization routine registers the main functions of applications and the TCP/IP protocol stack and when the scheduler executes them, the applications and TCP/IP protocol stack process their jobs by registering their functions as a task with the scheduler.

![Fig. 1. Software architecture of the system with a representation of network layers and transmission/reception flows.](image-url)
The event-driven scheduler uses a non-preemptive prioritized scheduling policy. It contains two wait queues, and one of two priorities is assigned to each. A task is assigned a priority at the time of creation and is put into the corresponding wait queue. Task creation is done in three ways. A task can be created by an application task, by a network protocol task, or by the timer in the operating system (Fig. 2). The timer provides four functions. It notifies current time, creates a task at a reserved time, creates tasks periodically, and deletes a reserved task.

![Diagram of task creation and execution through the scheduler and timer.](image)

### 3.2 TCP/IP Protocol Stack

We implemented TCP, UDP, IP, ICMP, ARP, and the Ethernet protocol in the TCP/IP protocol stack. These protocols are the minimum requirements necessary to communicate with other entities in an IP network [20]. In addition, the TCP/IP protocol stack does not provide full functionality of the network protocols. Only the essential parts the network protocols are implemented in the TCP/IP protocol stack. We briefly describe the functional specification of the lightweight TCP/IP protocol below.

- Packet transmission and reception
- Multiple applications (port)
- TCP window management for flow control
- TCP connection establishment, management, and destruction
- Checksum calculation
- No TCP options except the MSS (Maximum Segment Size) configuration
- Reordering of out-of-order segments in packet reception is not supported
- Congestion control is not supported
- Calculations like RTO (Retransmission Time Out) is replaced with constant values
4. PACKET TRANSMISSION AND RECEPTION

4.1 Task Granularity

Tasks that are executed on an event-driven scheduler are classified as application tasks, TCP/IP tasks, packet transmission tasks, or packet reception tasks. In these tasks, it is important to define the granularity of the packet transmission task, because this directly affects the implementation methods and system performance. For an event-driven scheduler, it is difficult to guarantee fairness in distributing the execution time in the manner of schedulers in general operating systems do. Hence, it is necessary to balance the execution time of application tasks, packet transmission tasks, and packet reception tasks. The granularity of the packet transmission tasks is the most important factor in balancing the tasks to improve performance of the system.

Fig. 3 shows three levels of task granularity. At the first level of task granularity, a packet transmission task can transmit only one packet. In this case, every time packet transmission is required, a packet transmission task is created and put into the wait queue in the scheduler. This is simple to implement, and the synchronization problem (discussed later) is not serious. Because the first-level granularity maximizes the response time of the system, it is the most appropriate design for interactive communications. However, as the size of data increases, number of tasks waiting in the queue increases...
rapidly. If the size of the wait queues is not sufficiently large, task creation fails repeatedly and the failures reduce the system performance greatly. With the second-level granularity, a packet transmission task transmits multiple packets when executed. In this case, the number of tasks increases much more slowly than the system with first-level granularity. A system with second-level granularity can cope with the problem of task creation failure. Moreover, by controlling the number of packets to transmit assigned to a single task, the system can transmit a large number of data packets without degrading the performance. A disadvantage of second-level granularity is synchronization. Given that the event-driven scheduler is non-preemptive, the execution time of a packet transmission task increases linearly as the number of packets assigned to it increases. Packet reception tasks cannot be executed in a short time if a transmission task is running; thus, the response time of the system increases. Though the effect of this problem is not great without packet losses, the system performance is greatly decreased when a packet is lost.

With the first and the second types of granularity, tasks are created when an application task requests transmission or when data packets remain in the packet buffer. With third-level granularity, a task managing packet transmission runs continuously, such as a daemon process, by creating a task that is identical to itself before it terminates. With the third type of granularity, a packet transmission task can be run as a process in a system with a general operating system. However, in the event-driven scheduler described above, a packet transmission task is executed too frequently to break the balance with other types of tasks.

4.2 Fast Retransmission and Synchronization

In TCP, retransmission is described to be triggered by retransmission timeout (RTO). In reality, most retransmissions are triggered by a fast retransmission mechanism when three duplicated acknowledgements are received from a recipient [20]. To support fast retransmission, a TCP connection is required to have additional states. The state transmission occurs when duplicated acknowledgements are received. However, as described in section 4.1, it is challenging to execute packet transmission tasks and packet reception tasks at the exact time on an event-driven scheduler.

During the time between the arrival of a packet and the execution of the packet reception task processing the packet, there can be multiple packet transmission tasks that are created (Fig. 4). In other words, when a TCP connection decides to retransmit a data packet.
packet by fast retransmission, out-of-order packets are waiting to be sent in the wait queue. If the transmission speed is low enough only a few packets are sent in this way, but progressively more out-of-order tasks remain in the wait queue as the transmission speed increases. These tasks significantly disturb the synchronization and consequently degrade the system performance.

To receive the acknowledgement of the retransmitted packet from a recipient, a mechanism is required to cancel or delay the execution of out-of-order tasks. There are two methods of task cancelation. The first method is to store the IDs of the packet transmission tasks in the connection state, and destroy them at the time fast retransmission is triggered. This method reduces the response time of the system. However, additional states increase the size of the data structure for the connection and the scheduler is required to support the operation so as to destroy numerous tasks at one time. The second method is to determine whether or not a task is out-of-order during the execution of the task itself. If a currently running task is out-of-order, it terminates immediately. This method does not require much structural modification, but the improvement in the response time is smaller than in the first method because the decision is made after the task has been executed by the scheduler.

4.3 Retransmission Timer

According to the TCP standard [20], a TCP connection maintains a retransmission timer for each packet and retransmits it when a timeout occurs. However, an absolute majority of detections and retransmissions of lost packets is performed by the fast retransmission mechanism, not by the retransmission timeout. Thus, it is not cost-effective to manage a timer for each data packet for timeout events that rarely occur in terms of time and memory space. We manage only one retransmission timer in a TCP connection, and the timer stores the transmission time of the first packet in a sliding window that is not acknowledged by the recipient, i.e., the data packet with the lowest sequence number among all unacknowledged data packets. When a timeout occurs, the first packet in the sliding window is retransmitted and the subsequent data packets are processed in the same manner as fast retransmission.

4.4 Dynamic Adaptation

Because the event-driven scheduler does not guarantee fairness of the execution time among tasks, much of the system performance depends on how we balance the execution frequency of the three types of tasks, these being application tasks, packet transmission tasks, and packet reception tasks. Packet reception tasks are created periodically with a default period of 1 millisecond. Considering the purpose of a target device, this period and the task granularity described in section 4.1 can be controlled to maximize the performance of the system. If we increase the frequency of packet reception tasks, the packet reception throughput and the interactivity of the communication increases. Otherwise, the packet transmission throughput can be maximized if we decrease the frequency. In addition, we dynamically adapted the execution frequency of packet transmission tasks and packet reception tasks at runtime by measuring the number of transmitted packets and received packets and observing the state transition of the TCP connection.
5. PERFORMANCE EVALUATION

5.1 Throughput

We implemented the proposed lightweight TCP/IP protocol stack on a board embedded with TI’s TMS320C6455 DSP. Table 1 shows the experimental environments. To ensure the correctness of the measurement, we used three different tools for network analysis: Wire Shark, iPerf, and our own code inserted into the program.

<table>
<thead>
<tr>
<th>Table 1. Experimental environments.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host</strong></td>
</tr>
<tr>
<td>Operating system</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Ethernet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Embedded device</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating system</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Ethernet</td>
</tr>
</tbody>
</table>

First, we measured the maximum throughput at the EMAC driver level without considering the receiver’s reaction. This represented the upper bound in this experimental environment (Fig. 5). We changed the packet size from 128 to 1500 bytes with 100,000 packets and changed the number of packets from 1,000 to 20,000. The maximum throughput at the EMAC driver level was 426 Mbps. In an identical environment, we compared the transmission throughput of TCP/IP with that of UDP/IP. Each packet transmission task transmits five packets, and UDP/IP transmission uses a protocol stack identical to that of TCP/IP transmission apart from the TCP layer. Transmitting 1,000,000 packets, TCP/IP transmission reached 296 Mbps and UDP/IP transmission reached 326 Mbps.
The transmission speed of the proposed lightweight TCP/IP protocol stack reached 69% of the upper bound using EMAC and 90% of the UDP/IP transmission (Fig. 6).

To compare the proposed lightweight TCP/IP protocol stack with the traditional TCP/IP protocol stack, we ported our implementation to a Linux machine using a raw socket. As the size of the data increased, the transmission throughput using the socket interface reached about 80 Mbps and our implementation reached 95 Mbps. The transmission throughput of the proposed lightweight TCP/IP protocol stack was 1.2 to 1.5 times faster than that of TCP/IP protocol stack with Linux.

5.2 Task Granularity and Task Cancelation

We measured the impact of the task granularity described in section 4.1 on the transmission throughput of the system (Fig. 7). With the first and the second levels of granularity, the transmission throughput changed from 10 to 296 Mbps. When the number of data packets for a task was less than 5, the transmission throughput was less than 250 Mbps. In this case, the transmission throughput changed according to the size of the wait queue in the scheduler as too many tasks were created. The size of the wait queue
was varied from 256 to 4096. When the number of data packets for a task exceeded 10, the transmission throughput reached its maximum throughput level. When it exceeded 50, transmission throughput was stable, but buffer overflows frequently occurred and the throughput decreased sharply in the event of a packet loss or when the transmission was delayed. With repetitive experiments, we found that the transmittance of 5 to 20 packets was the best for a task. However, the optimal point can differ according to the execution environment because the available size of the packet buffer and the wait queues are affected by the memory space of the device.

With the third level of granularity, which a task creates itself before termination, the system did not show the expected level of performance. Because the packet transmission tasks were executed too frequently, the packet reception tasks and application tasks could not easily be executed. Finally, we measured the difference in the delay between the execution with task cancelation and that without task cancelation. As it was not easy to measure this at runtime, we captured the packet sequence and traced the retransmission process after a packet loss occurred. When we used the task cancelation mechanism, the recovery was 5 to 10 times faster than it was without.

6. CONCLUSION

With the software-based TCP/IP Offload Engine (TOE), or lightweight TCP/IP, researchers have improved the design of the traditional TCP/IP protocol stack to optimize the system performance for use in embedded systems. In this paper, we proposed the design of a lightweight TCP/IP protocol stack that runs it on an event-driven scheduler of the type used in real-time operating systems and analyzed its characteristics. We introduced three levels of task granularity for an event-driven scheduler and discussed the synchronization problem associated with packet transmission and reception which can occur in the TCP retransmission process. We implemented the proposed lightweight TCP/IP protocol stack on an embedded system and evaluated it to confirm that the proposed design can suitably process network communication efficiently on an embedded system. Lightweight TCP/IPs tend to be designed to achieve platform independence so as to be operable on various platforms. However, designs and implementations that are integrated with the software architecture considering its platform and operating system are required to optimize the network communication performance on an embedded system.
REFERENCES


**Joonhyouk Jang** received his B.S. degree in Computer Science from Seoul National University, Seoul, Korea, in 2009. He is currently a Ph.D. student of School of Computer Science and Engineering, Seoul National University, Seoul, Korea. His current research interests include operating system, embedded systems, and computer security.

**Jinman Jung** received his B.S. degree in Computer Science from Seoul National University, Seoul, Korea, in 2008. He is currently a Ph.D. student of School of Computer Science and Engineering, Seoul National University, Seoul, Korea. His current research interests include operating system, embedded systems, mobile communications, and fault-tolerant computing systems.

**Yookun Cho** received the B.E. degree from Seoul National University, Seoul, Korea, in 1971 and the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, in 1978. Since 1979, he has been with the School of Computer Science and Engineering, Seoul National University, where he is currently a Professor. In 1985, he was a Visiting Assistant Professor with the University of Minnesota, and from 2001 to 2002, he was the President of the Korea Information Science Society. He also served as the honorary conference chair for ACM SAC 2007. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems.
Sanghoon Choi received his B.S. degree in Computer Science from Soongsil University, Seoul, Korea, in 2011. He is currently a Ph.D. student of School of Computing, Soongsil University, Seoul, Korea. His current research interests include operating system, embedded systems, and system software.

Sung Y. Shin received the M.S. and Ph.D. degree in Computer Science from University of Wyoming, Laramie, WY in 1984 and 1991 respectively. He is a professor and Graduate Coordinator of Computer Science at South Dakota State University since 1991. He has authored/coauthored over 130 technical peer reviewed papers in Software Engineering, Software Fault Tolerance, Telemedicine, and Medical Image Processing and GIS area. He worked as a visiting scientist for the Space and Life Science Division at NASA Johnson Space Center in Houston, TX from 1999 to 2002. He was a Conference Chair of ACM SAC 2007, 2009 and 2010. He had served as a Vice Chair of ACM SIGAPP from 2005 to 2009, and he is currently serving as a chair of ACM SIGAPP.