lwIP : Light Weight IP

The George
Washington University
November 18, 2011

Umang Mistry
umang_m@gwmail.gwu.edu

Abstract

lwIP is a smaller independent implementation of the TCP/IP protocol stack used mainly in limited resources systems such as embedded systems. The aim of the lwIP stack is to primarily reduce the memory usage and code size while still retaining full scale TCP/IP functionality. This is achieved using a special API that omits the need for data copying. The following paper gives a brief overview of the lwIP protocol stack and its design and implementation.

I. Introduction

Over the last few decades, internet technology has become flexible enough to adapt to the many changing network environments with varying link technologies and differing bandwidths and bit error rates.

Here we discuss an implementation of Internet protocols that deals with limited computing resources and memory. This paper shall briefly describe the design and implementation of a smaller TCP/IP stack called "**lwIP**" that is small enough to be used in minimalistic systems like embedded systems. It is an open-source product. It was originally developed by **Adam Dunkels** at the **Swedish Institute of Computer Science** and it is now developed and maintained by worldwide network of developers led by Kieran Mansley.

The focus of lwIP TCP/IP implementation is to reduce resource usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with just tens of kilobytes of free RAM and room enough for around 40 KB of code ROM. It was written in **C** programming language.

lwIP is used by many manufacturers of embedded systems like Altera, Analog Devices, Xilinx and Honeywell.

II. Design & Implementation

A. Overview, Process Model & Protocol Layering in lwIP

In traditional TCP/IP stack, the protocols are structured in a layered fashion, wherein each protocol layer tackles a different part of the communication problem. However, such a design leads to a situation where the communication overheads between the protocol layers will degrade the overall performance. Also, in most TCP/IP implementations, there is a strict division between the application layer and lower protocol layers wherein the lower layer protocols are implemented as part of the operating system kernel. But the operating systems used in minimal systems like lwIP do not generally keep a strict barrier between the kernel and the application layer. Because of this, there is a more relaxed scheme for communication between the OS kernel and the application layer by means of shared memory and awareness of the buffer handling mechanisms used by the lower layers. This increases efficiency since the application layer can reuse the buffers and also read and write directly into those buffers, thus avoiding the expense of performing a copy and saving memory.

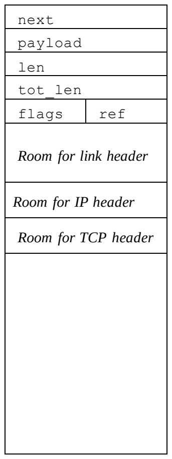
In lwIP, every protocol is implemented as an individual module with a few functions as entry points into every other protocol. lwIP consists of several modules which include those implementing TCP/IP protocols(IP,ICMP,TCP & UDP) plus a number of other support modules. The support modules include the *operating system emulation layer*, the *buffer & memory management subsystems*, *network interface functions* and functions for computing Internet checksum. lwIP

also includes an abstract API which makes it compatible with different platforms and OS environments.
lwIP uses a process model wherein all the protocols reside within a single process. Application programs may either reside in the lwIP process itself or be in separate processes wherein the communication between the TCP/IP stack and the application programs are performed by means of an abstract API. The main advantage of his approach is that it makes lwIP portable across different operating systems.

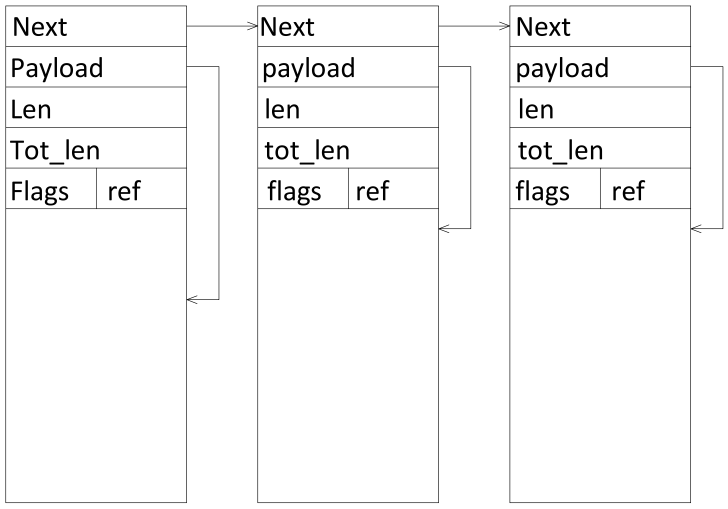
B. OS Emulation Layer & Buffer and Memory Management subsystem.

The OS Emulation Layer provides a uniform interface to operating system services like timers, process synchronization, and message passing mechanisms. In theory, when using lwIP with other operating systems, only an implementation of the OS Emulation Layer for that particular operating system is required. The Emulation layer provides one-shot timers with a granularity of atleast 200ms that calls a predefined function when the time-out happens. This function is used by TCP.
The only process synchronization functionality is provided by semaphores.
Message passing is carried out by a simple mechanism that uses an abstraction called *mailboxes.* Even if the underlying OS does not have native support for this mechanism, they are easily implemented using semaphores.

The buffers in any communication system can contain segments of varying length ranging from full-sized TCP segments with hundreds of bytes worth of data to short ICMP echo requests/replies consisting of only a few bytes. Thus the buffer and memory management system should be capable of accommodating buffers of varying sizes. Also, in order to avoid unnecessary copying of files in minimal systems, it should be possible for the buffer data to reside in memory that isn't managed by the networking subsystem like application memory or ROM.
lwIP uses something called a *pbuf* which is its internal representation of a packet and desigened for minimal stack. The pbuf is designed in such a way as to support the allocation of dynamic memory for packet contents as well as letting packet data reside in static memory. There are 3 types of PBUFs, PBUF_RAM, PBUF_ROM & PBUF_POOL [1].



Shown above is a PBUF_RAM pbuf.
(Copyrights* Design & Implementation of lwIP by Adam Dunkels)



Shown above is a chained PBUF_POOL pbuf.
(Copyrights* Design & Implementation of lwIP by Adam Dunkels)

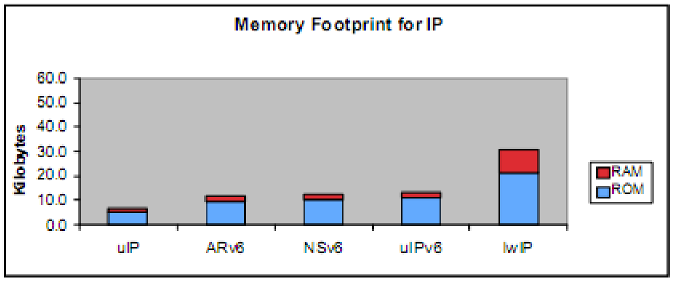
The three types of pbufs have different uses. PBUF_POOL is mainly used by device drivers. PBUF_ROM is used when an application sends data that is located in a memory managed by the application (i.e.

[1] – *Design and Implementation of lwIP , Technical Paper by Adam Dunkels, Swedish Institute of Computer Science, 2001.*

ROM) and hence the name PBUF_ROM. PBUF_RAM is used when an application sends data which is dynamically generated. In essence, incoming pbufs are mainly of the type PBUF_POOL and outgoing ones are either PBUF_ROM or PBUF_RAM type. Also, the internal structure of a pbuf is shown above.

The pbuf module provides the functions for manipulating pbufs. Pbufs are allocated by calling the function *pbuf_alloc()* and are deallocated using function *pbuf_free()*. *Pbuf_ref()* is used to increase the reference count. The function *pbuf_realloc()* shrinks the pbuf such that it occupies just exactly enough memory to cover the size of the data. The function *pbuf_header()* adjusts the payload pointer and the length fields and *pbuf_chain()* & *pbuf_dechain()* are used for chaining pbufs.

The memory manager is responsible for allocating and deallocating contiguous regions of memory. It keeps track of the allocated memory by placing small structures on top of each allocated memory block. Memory is allocated by searching for an unused allocation block that is large enough for the requested allocation.



(Copyrights* IP for Smart Objects, A Dunkels, 2008)

The figure above shows the memory footprint of five embedded TCP/IP stacks wherein lwIP uses around 20 KB of RAM[2] which is slightly higher than the rest but lwIP is open source whereas the rest are commercially available. Such a judicious use of memory enables these embedded TCP/IP stacks to be used in power-constrained devices that are run over sub-milliwatt radio links like 802.15.4. Such low power operation enables years of lifetime on typical AA batteries, even on multi-hop routing nodes.

C. IP, UDP & TCP Processing

1. IP

LwIP implements the most basic functionality of IP, namely send, receive and forward packets. It cannot handle fragmented packets or packets with IP options.

For incoming packets, the *ip_input()* function is called by a network device driver. Here the parity checking as well computing the header checksum is done. If the packet is fragmented, then it is silently discarded. The function then checks the destination address with the IP addresses of the network interfaces on the device. These IP addresses are ordered in a link list and therefore a simple linear search is performed. If there is a match then the protocol field is checked and the packet is passed onto the respective upper layer protocol.

Outgoing packets are handled by using functions *ip_output()* which in turn calls the function *ip_route()* to find the appropriate network interface to transmit the packet on. After that the packet is passes on to the function *ip_output_if()* which uses the outgoing network interface as an argument. Also, since TCP and UDP need to have the destination IP address to compute the transport layer checksum, the outgoing network interface must in some cases be determined before the packet reaches the IP layer. This is done by letting the transport layer functions call the *ip_route()* function directly

Forwarding packets is taken care of by the function *ip_forward()*. This is done when none of the network interfaces have an IP address match with the destination IP address of the packet. Once the function is called the TTL field is decreased by one and if it reaches zero then an appropriate ICMP error message is sent out back to the sender.

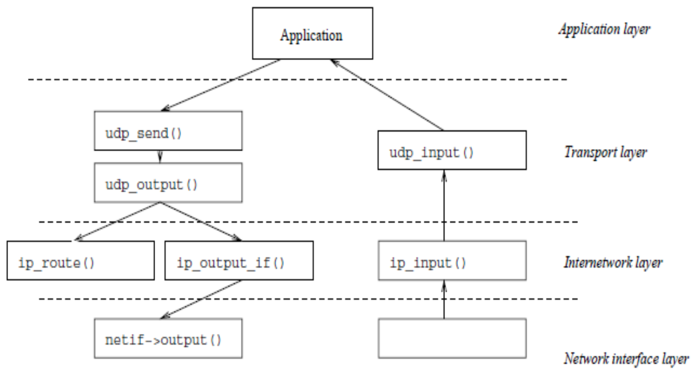
ICMP packets received by the *ip_input()* function are

[2] *IP for Smart Objects- Adam Dunkels, IPSO Allicance, Sept' 2008*

handed over to the function *icmp_input(),* the header is decoded and the appropriate action is taken. *Icmp_dest_unreach()* is the function used for sending out ICMP Destination Unreachable messages.

2. UDP

A UDP session is defined by the IP addresses and port numbers of the end points and these are stored in the local_ip, dest_ip, local_port, dest_port fields[3]. Due to the simplicity of UDP, the input and output processing is also fairly simple. To send data, udp_send() function is called which in turn calls the function udp_output(). After the checksum is calculated and the UDP header fields are filled the packet is turned over to ip_output_if() function for transmission. Shown below is diagram showing the flow chart of UDP processing.

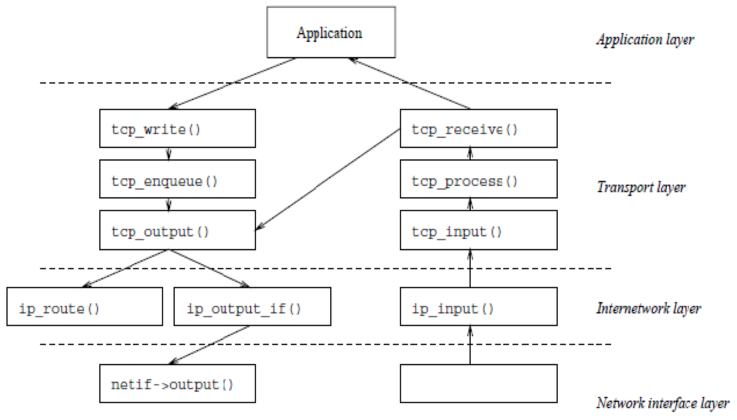


(Copyrights* Design & Implementation of lwIP by Adam Dunkels)

When a UDP datagram arrives, the IP layer calls he udp_input() function.

3. TCP

The TCP code is more complex than all the other protocols described here and it constitutes 50% of the total code size of lwIP. The following is a flowchart of the TCP processing. It shows the working flow of how a TCP segment is processed while sending one packet and receiving one packet and which function are called respectively.



(Copyrights* Design & Implementation of lwIP by Adam Dunkels)

The basic TCP processing is divided into six functions: tcp_input(), tcp_process(), tcp_receive() which are related to TCP input processing, and tcp_write(), tcp_enqueue(), and tcp_output() which deals with output processing.
The rcv_nxt and rcv_wnd fields in the data structures are used for ACKs and advertising Window Size during incoming TCP segments.

III. lwIP stack interface

There are 2 ways of interfacing the TCP/IP stack, either calling the functions in the TCP/UDP modules directly, or using the lwIP API. The 1st approach is based on callbacks and an application program that uses this approach can hence not operate in a sequential manner. This makes it harder to program and the application code is harder to understand. Also, the application program that interfaces the TCP/UDP modules directly has reside in the same process as the TCP/IP stack. This is because a callback function cannot be called across a process boundary. The advantage here is that it eliminates the need for context switching but the disadvantage is that the application program cannot involve itself in any long running computations since TCP/IP processing cannot occur in conjunction with it. This is overcome using the 2nd approach wherein the application is split into two parts, one dealing with the computation and other dealing with the communication.

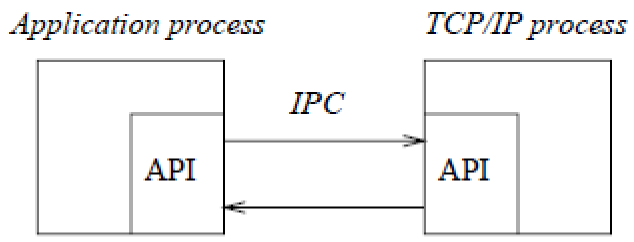
[3] *Modularity and Efficiency in Protocol Implementation, RFC 817, D. D. Clark, July 1982*

IV. lwIP API (Application Program Interface)

The lwIP API was designed for lwIP and it uses the knowledge of the internal structure of lwIP to achieve efficiency. The API does not require that the data be copied between the application program and the TCP/IP stack, since the application program can manipulate the internal buffers.

Network data is received in the form of buffers where data is partitioned into smaller chunks of memory as seen earlier. However, many application need to manipulate data in a continuous stream of memory. Hence a convenience function for copying the data from a fragmented buffer to a continuous memory exists as part of this API. Also, sending data is done differently depending on whether the data has to be sent over a TCP connection or as UDP datagrams. For TCP, data is sent by passing the output function a pointer to a continuous memory region. When sending UDP datagrams, the application program will explicitly allocate a buffer and fill it with data.

The implementation of the API is divided into 2 parts due to the process model of the TCP/IP protocol stack. The two parts communicate using the Inter Process Communication (IPC) mechanisms provided by the OS Emulation Layer. There are three IPC mechanisms: shared memory, message passing and semaphores.



The figure above shows the divisions of the API Implementations
(Copyrights* Design & Implementation of lwIP by Adam Dunkels)

The general design principle used is to let as much work as possible be done within the application process rather than in the TCP/IP process. This is important since all processes use the TCP/IP process for their TCP/IP communication. Keeping down the code footprint of the part of the API that is linked with the applications is not as important. This code can be shared among the processes, and even if shared libraries are not supported by the operating system, the code is stored in ROM. Embedded systems usually carry fairly large amounts of ROM, whereas
processing power is scarce. Buffer management is situated in the library part of the API implementation. They are created, copied and deallocated in the application process. The functions that handle network connections are in the part of the API that resides in the TCP/IP process.

V. Conclusions

The recent progress in low-cost embedded system devices is about to make “The Internet of Things” a reality. For this to happen we must learn how memory efficient implementations of the IP stack are designed and how they perform. We have seen one such example through this report-lwIP. lwIP is scaled-down implementation of the TCP/IP protocol suite that focuses on reducing resource usage and while still having a full scale TCP.

VI. References

B. Ahlgren, M. BjÄorkman, and K. Moldeklev. The performance of a no-copy api for communication (extended abstract). In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Mystic, Connecticut, USA, August 1995.

M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999.

C. Brian, P. Indra, W. Geun, J. Prescott, and T. Sakai. IEEE-802.11 wireless local area networks. *IEEE Communications Magazine*, 35(9):116{126, September 1997.

T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol { HTTP/1.0.RFC 1945, Internet Engineering Task Force, May 1996

D. Adam, Design & Implementation of the LWIP TCP/IP Stack, Swedish Institute of Computer Science, February 2001

D. D. Clark. Modularity and efficiency in protocol implementation. RFC 817, Internet Engineering Task Force, July 1982