TCP/IP for 8-Bit Architectures*

Adam Dunkels
Swedish Institute of Computer Science
adam@sics.se, http://www.sics.se/~adam/

Abstract

We describe two small and portable TCP/IP implementations fulfilling the subset of RFC1122 requirements needed for full host-to-host interoperability. Our TCP/IP implementations do not sacrifice any of TCP's mechanisms such as urgent data or congestion control. They support IP fragment reassembly and the number of multiple simultaneous connections is limited only by the available RAM. Despite being small and simple, our implementations do not require their peers to have complex, full-size stacks, but can communicate with peers running a similarly light-weight stack. The code size is on the order of 10 kilobytes and RAM usage can be configured to be as low as a few hundred bytes.

1 Introduction

With the success of the Internet, the TCP/IP protocol suite has become a global standard for communication. TCP/IP is the underlying protocol used for web page transfers, e-mail transmissions, file transfers, and peer-to-peer networking over the Internet. For embedded systems, being able to run native TCP/IP makes it possible to connect the system directly to an intranet or even the global Internet. Embedded devices with full TCP/IP support will be first-class network citizens, thus being able to fully communicate with other hosts in the network.

Traditional TCP/IP implementations have required far too much resources both in terms of code size and memory usage to be useful in small 8 or 16-bit systems. Code size of a few hundred kilobytes and RAM requirements of several hundreds of kilobytes have made it impossible to fit the full TCP/IP stack into systems with a few tens of kilobytes of RAM and room for less than 100 kilobytes of code.

With this paper we show that, contrary to conventional wisdom, it is possible to implement a full TCP/IP stack that is small enough in terms of code size and memory usage to be useful even in limited 8-bit systems, without having to sacrifice interoperability or protocol functionality required by the protocol specifications.

We have implemented two small generic and portable TCP/IP implementations, *lwIP* (lightweight IP) and *uIP* (micro IP), both fully written in the C programming language. We have made the source code available for both lwIP [3] and uIP [4]. Our implementations have been ported to numerous 8- and 16-bit platforms such as the AVR, H8S/300, 8051, Z80, ARM, M16c, and the x86 CPUs. Devices running our implementations have been used in numerous places throughout the Internet.

2 RFC compliance

The formal requirements for the protocols in the TCP/IP stack is specified in a number of RFC documents published by the Internet Engineering Task Force, IETF. Each of the protocols in the stack is defined in one more RFC documents and RFC1122 [1] collects all requirements and updates the previous RFCs.

In our implementations, we have implemented all RFC requirements that affect host-to-host communication. However, in order to reduce code size, we have removed certain mechanisms in the interface between the application and the stack, such as the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections. Since there are only very few applications that make use of those features, we believe that they can be removed without loss of generality. Table 1 lists the features that uIP and lwIP implements.

3 Memory management

In our target architecture, RAM is the most scarce resource. With only a few kilobytes of RAM available for the TCP/IP stack to use, mechanisms used in traditional TCP/IP cannot be directly applied.

Because of the different design goals for the lwIP and the uIP implementations, we have chosen two different memory management solutions. The lwIP implementation has dynamic buffer and memory allocation mechanisms where memory for holding connection state and packets is dynamically allocated from a global pool of

*This is a shortened version of [5]

Table 1: TCP/IP features implemented by uIP and lwIP

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control	Not needed	x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

available memory blocks. Packets are contained in one or more dynamically allocated buffers of fixed size. The size of the packet buffers is determined by a configuration option at compile time. Buffers are allocated by the network device driver when an incoming packet arrives.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state. The global packet buffer is large enough to contain one packet of maximum size. When a packet arrives from the network, the device driver places it in the global buffer and calls the TCP/IP stack. If the packet contains data, the TCP/IP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into a secondary buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data. Packets that arrive when the application is processing the data must be queued, either by the network device or by the device driver.

Outgoing data is also handled differently because of the different buffer schemes. In lwIP, an application that wishes to send data passes the length and a pointer to the data to the TCP/IP stack as well as a flag which indicates whether the data is volatile or not. The TCP/IP stack allocates buffers of suitable size and either copies the data into the buffers or references the data through pointers. The allocated buffers contain space for the TCP/IP stack to prepend the TCP/IP and link layer headers.

In uIP, the same global packet buffer that is used for incoming packets is also used for the TCP/IP headers of outgoing data. If the application sends dynamic data, it may use the parts of the global packet buffer that are not used for headers as a temporary storage buffer. To send the data, the application passes a pointer to the data as well as the length of the data to the stack. The TCP/IP headers are written into the global buffer and once the headers have been produced, the device driver sends the headers and the application data out on the network.

4 Application program interface

The Application Program Interface (API) defines the way the application program interacts with the TCP/IP stack. The most commonly used API for TCP/IP is the BSD socket API which is used in most Unix systems and has heavily influenced the Microsoft Windows WinSock API. Because the socket API uses stop-and-wait semantics, it requires support from an underlying multitasking operating system. Since the overhead of task management, context switching and allocation of stack space for the tasks might be too high in our target architecture, the BSD socket interface is not suitable for our purposes.

Instead, we have chosen an event driven interface where the application is invoked in response to certain events. Examples of such events are data arriving on a connection, an incoming connection request, or a poll request from the stack. The event based interface fits well in the event based structure used by operating systems such as TinyOS [6]. Furthermore, because the application is able to act on incoming data and connection requests as soon as the TCP/IP stack receives the packet, low response times can be achieved even in low-end systems.

5 Delayed ACK performance hit

Most TCP receivers implement the delayed acknowledgment algorithm [2] for reducing the number of pure acknowledgment packets sent. A TCP receiver using this algorithm will only send acknowledgments for every other received segment. If no segment is received within a specific time-frame, an acknowledgment is sent. The time-frame can be as high as 500 ms but typically is 200 ms.

A TCP sender such as uIP that only handles a single outstanding TCP segment will interact poorly with the delayed acknowledgment algorithm. Because the receiver only receives a single segment at a time, it will wait as much as 500 ms before an acknowledgment is sent. This means that the maximum possible throughput is severely limited by the 500 ms idle time.

It should be noted, however, that since small systems

Table 2: Code size for uIP (AVR)

Function	Code size (bytes)
Checksumming	712
IP, ICMP and TCP	4452
Total	5164

Table 3: Code size for lwIP (AVR)

Function	Code size (bytes)
Memory management	3142
Checksumming	1116
Network interfaces	458
IP	2216
ICMP	594
TCP	14230
Total	21756

running uIP are not very likely to have large amounts of data to send, the delayed acknowledgment throughput degradation of uIP need not be very severe. Small amounts of data sent by such a system will not span more than a single TCP segment, and would therefore not be affected by the throughput degradation anyway. The maximum throughput when uIP acts as a receiver is not affected by the delayed acknowledgment throughput degradation.

6 Code size

The code was compiled for the 8-bit Atmel AVR platforms using gcc [7] version 3.3 respectively, with code size optimization turned on. The resulting size of the compiled code can be seen in Tables 2 and 3. Even though both implementations support ARP and SLIP and lwIP includes UDP, only the protocols discussed in this paper are presented. Because the protocol implementations in uIP are tightly coupled, the individual sizes of the implementations are not reported.

There are several reasons for the dramatic difference in code size between lwIP and uIP. In order to support the more complex and configurable TCP implementation, lwIP has significantly more complex buffer and memory management than uIP. Since lwIP can handle packets that span several buffers, the checksum calculation functions in lwIP are more complex than those in uIP. The support for dynamically changing network interfaces in lwIP also contributes to the size increase of the IP layer because the IP layer has to manage multiple local IP addresses. The IP layer in lwIP is further made larger by the fact that lwIP has support for UDP, which requires that the IP layer is able handle broadcast and multicast packets. Likewise, the ICMP implementation in lwIP has support for UDP error messages which have not been implemented in uIP.

The TCP implementation is lwIP is nearly twice as large as the full IP, ICMP and TCP implementation in uIP. The main reason for this is that lwIP implements the sliding window mechanism which requires a large amount of buffer and queue management functionality that is not required in uIP.

7 Summary and conclusions

We have shown that it is possible to fit a full scale TCP/IP implementation well within the limits of an 8-bit microcontroller, without having to sacrifice the protocol functionality specified by the RFC standards. The price we have to pay is that the throughput of such a small implementation will be lower than for a full scale implementation.

References

[1] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force, October 1989.

[2] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.

[3] A. Dunkels. lwIP - a lightweight TCP/IP stack. Web page. 2002-10-14.
URL: *http://www.sics.se/~adam/lwip/*

[4] A. Dunkels. uIP - a TCP/IP stack for 8- and 16-bit microcontrollers. Web page. 2002-10-14.
URL: *http://dunkels.com/adam/uip/*

[5] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services (MOBISYS '03)*, May 2003.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[7] The GCC Team. The GNU compiler collection. Web page. 2002-10-14.
URL: *http://gcc.gnu.org/*