Full 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.

TCP [21] is both the most complex and the most widely used of the transport protocols in the TCP/IP stack. TCP provides reliable full-duplex byte stream transmission on top of the best-effort IP [20] layer. Because IP may reorder or drop packets between the sender and the receiver, TCP has to implement sequence numbering and retransmissions in order to achieve reliable, ordered data transfer.

We have implemented two small generic and portable TCP/IP implementations, *lwIP* (lightweight IP) and *uIP* (micro IP), with slightly different design goals. The lwIP implementation is a full-scale but simplified TCP/IP implementation that includes implementations of IP, ICMP, UDP and TCP and is modular enough to be easily extended with additional protocols. lwIP has support for multiple local network interfaces and has flexible configuration options which makes it suitable for a wide variety of devices.

The uIP implementation is designed to have only the absolute minimal set of features needed for a full TCP/IP stack. It can only handle a single network interface and does not implement UDP, but focuses on the IP, ICMP and TCP protocols.

Both implementations are fully written in the C programming language. We have made the source code available for both lwIP [7] and uIP [8]. 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.

We have studied how the code size and RAM usage of a TCP/IP implementation affect the features of the TCP/IP implementation and the performance of the communication. We have limited our work to studying the implementation of TCP and IP protocols and the interaction between the TCP/IP stack and the application programs. Aspects such as address configuration, security, and energy consumption are out of the scope of this work.

The main contribution of our work is that we have shown

that is it 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.

Recently, other small implementations of the TCP/IP stack have made it possible to run TCP/IP in small 8-bit systems. Those implementations are often heavily specialized for a particular application, usually an embedded web server, and are not suited for handling generic TCP/IP protocols. Future embedded networking applications such as peer-to-peer networking require that the embedded devices are able to act as first-class network citizens and run a TCP/IP implementation that is not tailored for any specific application.

Furthermore, existing TCP/IP implementations for small systems assume that the embedded device always will communicate with a full-scale TCP/IP implementation running on a workstation-class machine. Under this assumption, it is possible to remove certain TCP/IP mechanisms that are very rarely used in such situations. Many of those mechanisms are essential, however, if the embedded device is to communicate with another equally limited device, e.g., when running distributed peer-to-peer services and protocols.

This paper is organized as follows. After a short introduction to TCP/IP in Section 2, related work is presented in Section 3. Section 4 discusses RFC standards compliance. How memory and buffer management is done in our implementations is presented in Section 5 and the application program interface is discussed in Section 6. Details of the protocol implementations is given in Section 7 and Section 8 comments on the performance and maximum throughput of our implementations, presents throughput measurements from experiments and reports on the code size of our implementations. Section 9 gives ideas for future work. Finally, the paper is summarized and concluded in Section 10.

2 TCP/IP overview

From a high level viewpoint, the TCP/IP stack can be seen as a black box that takes incoming packets, and demultiplexes them between the currently active connections. Before the data is delivered to the application, TCP sorts the packets so that they appear in the order they were sent. The TCP/IP stack will also send acknowledgments for the received packets.

Figure 1 shows how packets come from the network device, pass through the TCP/IP stack, and are delivered to the actual applications. In this example there are five active connections, three that are handled by a web server application, one that is handled by the e-mail sender application, and one that is handled by a data logger application.

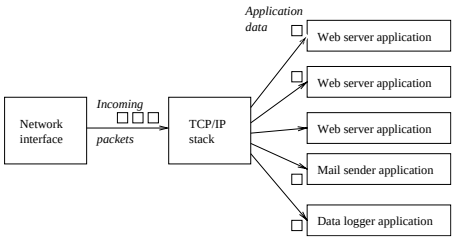

Figure 1: TCP/IP input processing.

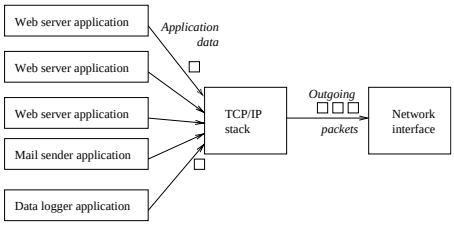

Figure 2: TCP/IP output processing.

A high level view of the output processing can be seen in Figure 2. The TCP/IP stack collects the data sent by the applications before it is actually sent onto the network. TCP has mechanisms for limiting the amount of data that is sent over the network, and each connection has a queue on which the data is held while waiting to be transmitted. The data is not removed from the queue until the receiver has acknowledged the reception of the data. If no acknowledgment is received within a specific time, the data is retransmitted.

Data arrives asynchronously from both the network and the application, and the TCP/IP stack maintains queues in which packets are kept waiting for service. Because packets might be dropped or reordered by the network, incoming packets may arrive out of order. Such packets have to be queued by the TCP/IP stack until a packet that fills the gap arrives. Furthermore, because TCP limits the rate at which data that can be transmitted over each TCP connection, application data might not be immediately sent out onto the network.

The full TCP/IP suite consists of numerous protocols, ranging from low level protocols such as ARP which translates IP addresses to MAC addresses, to application

level protocols such as SMTP that is used to transfer e-mail. We have concentrated our work on the TCP and IP protocols and will refer to upper layer protocols as "the application". Lower layer protocols are often implemented in hardware or firmware and will be referred to as "the network device" that are controlled by the network device driver.

TCP provides a reliable byte stream to the upper layer protocols. It breaks the byte stream into appropriately sized segments and each segment is sent in its own IP packet. The IP packets are sent out on the network by the network device driver. If the destination is not on the physically connected network, the IP packet is forwarded onto another network by a router that is situated between the two networks. If the maximum packet size of the other network is smaller than the size of the IP packet, the packet is fragmented into smaller packets by the router. If possible, the size of the TCP segments are chosen so that fragmentation is minimized. The final recipient of the packet will have to reassemble any fragmented IP packets before they can be passed to higher layers.

3 Related work

There are numerous small TCP/IP implementations for embedded systems. The target architectures range from small 8-bit microcontrollers to 32-bit RISC architectures. Code size varies from a few kilobytes to hundreds of kilobytes. RAM requirements can be as low as 10 bytes up to several megabytes.

Existing TCP/IP implementations can roughly be divided into two categories; those that are adaptations of the Berkeley BSD TCP/IP implementation [18], and those that are written independently from the BSD code. The BSD implementation was originally written for workstation-class machines and was not designed for the limitations of small embedded systems. Because of that, implementations that are derived from the BSD code base are usually suited for larger architectures than our target. An example of a BSD-derived implementation is the InterNiche NicheStack [11], which needs around 50 kilobytes of code space on a 32-bit ARM system.

Many of the independent TCP/IP implementations for embedded processors use a simplified model of the TCP/IP stack which makes several assumptions about the communication environment. The most common assumption is that the embedded system always will communicate with a system such as a PC that runs a full scale, standards compliant TCP/IP implementation. By relying on the standards compliance of the remote host, even an extremely simplified, uncompliant, TCP/IP implementation will be able to communicate. The communication may very well fail, however, once the system is to communicate with another simplified TCP/IP implementation such as another embedded system of the same kind. We will briefly cover a number of such simplifications that are used by existing implementations.

One usual simplification is to tailor the TCP/IP stack for a specific application such as a web server. By doing this, only the parts of the TCP/IP protocols that are required by the application need to be implemented. For instance, a web server application does not need support for urgent data and does not need to actively open TCP connections to other hosts. By removing those mechanisms from the implementation, the complexity is reduced.

The smallest TCP/IP implementations in terms of RAM and code space requirements are heavily specialized for serving web pages and use an approach where the web server does not hold any connection state at all. For example, the iPic match-head sized server [26] and Jeremy Bentham's PICmicro stack [1] require only a few tens of bytes of RAM to serve simple web pages. In such an implementation, retransmissions cannot be made by the TCP module in the embedded system because nothing is known about the active connections. In order to achieve reliable transfers, the system has to rely on the remote host to perform retransmissions. It is possible to run a very simple web server with such an implementation, but there are serious limitations such as not being able to serve web pages that are larger than the size of a single TCP segment, which typically is about one kilobyte.

Other TCP/IP implementations such as the Atmel TCP/IP stack [5] save code space by leaving out certain vital TCP mechanisms. In particular, they often leave out TCP's congestion control mechanisms, which are used to reduce the sending rate when the network is overloaded. While an implementation with no congestion control might work well when connected to a single Ethernet segment, problems can arise when communication spans several networks. In such cases, the intermediate nodes such as switches and routers may be overloaded. Because congestion primarily is caused by the amount of packets in the network, and not the size of these packets, even small 8-bit systems are able to produce enough traffic to cause congestion. A TCP/IP implementation lacking congestion control mechanisms should not be used over the global Internet as it might

contribute to congestion collapse [9].

Texas Instrument's MSP430 TCP/IP stack [6] and the TinyTCP code [4] use another common simplification in that they can handle only one TCP connection at a time. While this is a sensible simplification for many applications, it seriously limits the usefulness of the TCP/IP implementation. For example, it is not possible to communicate with two simultaneous peers with such an implementation. The CMX Micronet stack [27] uses a similar simplification in that it sets a hard limit of 16 on the maximum number of connections.

Yet another simplification that is used by LiveDevices Embedinet implementation [12] and others is to disregard the maximum segment size that a receiver is prepared to handle. Instead, the implementation will send segments that fit into an Ethernet frame of 1500 bytes. This works in a lot of cases due to the fact that many hosts are able to receive packets that are 1500 bytes or larger. Communication will fail, however, if the receiver is a system with limited memory resources that is not able to handle packets of that size.

Finally, the most common simplification is to leave out support for reassembling fragmented IP packets. Even though fragmented IP packets are quite infrequent [25], there are situations in which they may occur. If packets travel over a path which fragments the packets, communication is impossible if the TCP/IP implementation is unable to correctly reassemble them. TCP/IP implementations that are able to correctly reassemble fragmented IP packets, such as the Kadak KwikNET stack [22], are usually too large in terms of code size and RAM requirements to be practical for 8-bit systems.

4 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 [2] collects all requirements and updates the previous RFCs.

The RFC1122 requirements can be divided into two categories; those that deal with the host to host communication and those that deal with communication between the application and the networking stack. An example of the first kind is *"A TCP MUST be able to receive a TCP option in any segment"* and an example of the second kind is *"There MUST be a mechanism for reporting soft TCP error conditions to the application."* A TCP/IP implementation that violates requirements of the first kind may not be able to communicate with other TCP/IP implementations and may even lead to network failures. Violation of the second kind of requirements will only affect the communication within the system and will not affect host-to-host communication.

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

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.

5 Memory and buffer 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 mecha-

nisms where memory for holding connection state and packets is dynamically allocated from a global pool of 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. If the packet is larger than one buffer, more buffers are allocated and the packet is split into the buffers. If the incoming packet is queued by higher layers of the stack or the application, a reference counter in the buffer is incremented. The buffer will not be deallocated until the reference count is zero.

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. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames. Devices that are handled by the processor, such as RS-232 ports, can copy incoming bytes to a separate buffer during application processing. If the buffers are full, the incoming packet is dropped. This will cause performance degradation, but only when multiple connections are running in parallel. This is because uIP advertises a very small receiver window, which means that only a single TCP segment will be in the network per connection.

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, depending on the volatile flag, 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. After the headers are written, the stack passes the buffers to the network device driver. The buffers are not deallocated when the device driver is finished sending the data, but held on a retransmission queue. If the data is lost in the network and have to be retransmitted, the buffers on retransmission queue will be retransmitted. The buffers are not deallocated until the data is known to be received by the peer. If the connection is aborted because of an explicit request from the local application or a reset segment from the peer, the connection's buffers are deallocated.

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. The data is not queued for retransmissions. Instead, the application will have to reproduce the data if a retransmission is necessary.

The total amount of memory usage for our implementations depends heavily on the applications of the particular device in which the implementations are to be run. The memory configuration determines both the amount of traffic the system should be able to handle and the maximum amount of simultaneous connections. A device that will be sending large e-mails while at the same time running a web server with highly dynamic web pages and multiple simultaneous clients, will require more RAM than a simple Telnet server. It is possible to run the uIP implementation with as little as 200 bytes of RAM, but such a configuration will provide extremely low throughput and will only allow a small number of simultaneous connections.

6 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 [10]. 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.

7 Protocol implementations

The protocols in the TCP/IP protocol suite are designed in a layered fashion where each protocol performs a specific function and the interactions between the protocol layers are strictly defined. While the layered approach is a good way to design protocols, it is not always the best way to implement them. For the lwIP implementation, we have chosen a fully modular approach where each protocol implementation is kept fairly separate from the others. In the smaller uIP implementation, the protocol implementations are tightly coupled in order to save code space.

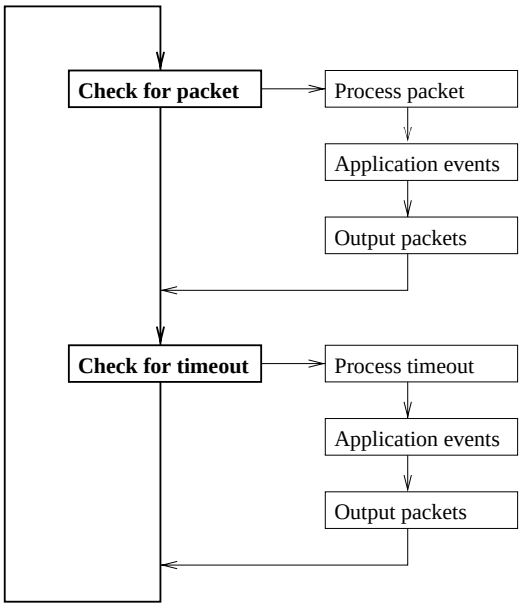

Figure 3: The main control loop.

7.1 Main control loop

The lwIP and uIP stacks can be run either as a task in a multitasking system, or as the main program in a singletasking system. In both cases, the main control loop (Figure 3) does two things repeatedly:

1. Check if a packet has arrived from the network.

2. Check if a periodic timeout has occurred.

If a packet has arrived, the input handler of the TCP/IP stack is invoked. The input handler function will never block, but will return at once. When it returns, the stack or the application for which the incoming packet was intended may have produced one or more reply packets which should be sent out. If so, the network device driver is called to send out these packets.

Periodic timeouts are used to drive TCP mechanisms that depend on timers, such as delayed acknowledgments, retransmissions and round-trip time estimations. When the main control loop infers that the periodic timer should fire, it invokes the timer handler of the TCP/IP stack. Because the TCP/IP stack may perform retransmissions when dealing with a timer event, the network device driver is called to send out the packets that may have been produced.

This is similar to how the BSD implementations drive the TCP/IP stack, but BSD uses software interrupts and a task scheduler to initiate input handlers and timers. In our limited system, we do not depend on such mechanisms being available.

7.2 IP — Internet Protocol

When incoming packets are processed by lwIP and uIP, the IP layer is the first protocol that examines the packet. The IP layer does a few simple checks such as if the destination IP address of the incoming packet matches any of the local IP address and verifies the IP header checksum. Since there are no IP options that are strictly required and because they are very uncommon, both lwIP and uIP drop any IP options in received packets.

7.2.1 IP fragment reassembly

In both lwIP and uIP, IP fragment reassembly is implemented using a separate buffer that holds the packet to be reassembled. An incoming fragment is copied into the right place in the buffer and a bit map is used to keep track of which fragments have been received. Because the first byte of an IP fragment is aligned on an 8-byte boundary, the bit map requires a small amount of memory. When all fragments have been reassembled, the resulting IP packet is passed to the transport layer. If all fragments have not been received within a specified time frame, the packet is dropped.

The current implementation only has a single buffer for holding packets to be reassembled, and therefore does not support simultaneous reassembly of more than one packet. Since fragmented packets are uncommon, we belive this to be a reasonable decision. Extending our implementation to support multiple buffers would be straightforward, however.

7.2.2 Broadcasts and multicasts

IP has the ability to broadcast and multicast packets on the local network. Such packets are addressed to special broadcast and multicast addresses. Broadcast is used heavily in many UDP based protocols such as the Microsoft Windows file-sharing SMB protocol. Multicast is primarily used in protocols used for multimedia distribution such as RTP. TCP is a point-to-point protocol and does not use broadcast or multicast packets.

Because lwIP supports applications using UDP, it has support for both sending and receiving broadcast and multicast packets. In contrast, uIP does not have UDP support and therefore handling of such packets has not been implemented.

7.3 ICMP — Internet Control Message Protocol

The ICMP protocol is used for reporting soft error conditions and for querying host parameters. Its main use is, however, the echo mechanism which is used by the `ping` program.

The ICMP implementations in lwIP and uIP are very simple as we have restricted them to only implement ICMP echo messages. Replies to echo messages are constructed by simply swapping the source and destination IP addresses of incoming echo requests and rewriting the ICMP header with the Echo-Reply message type. The ICMP checksum is adjusted using standard techniques [23].

Since only the ICMP echo message is implemented, there is no support for Path MTU discovery or ICMP redirect messages. Neither of these is strictly required for interoperability; they are performance enhancement mechanisms.

7.4 TCP — Transmission Control Protocol

The TCP implementations in lwIP and uIP are driven by incoming packets and timer events. IP calls TCP when a TCP packet arrives and the main control loop calls TCP periodically.

Incoming packets are parsed by TCP and if the packet contains data that is to be delivered to the application, the application is invoked by the means of a function call. If the incoming packet acknowledges previously sent data, the connection state is updated and the application is informed, allowing it to send out new data.

7.4.1 Listening connections

TCP allows a connection to listen for incoming connection requests. In our implementations, a listening connection is identified by the 16-bit port number and incoming connection requests are checked against the list of listening connections. This list of listening connections is dynamic and can be altered by the applications in the system.

7.4.2 Sending data

When sending data, an application will have to check the number of available bytes in the send window and adjust the number of bytes to send accordingly. The size of the send window is dictated by the memory configuration as well as the buffer space announced by the receiver of the data. If no buffer space is available, the application has to defer the send and wait until later.

Buffer space becomes available when an acknowledgment from the receiver of the data has been received.

The stack informs the application of this event, and the application may then repeat the sending procedure.

7.4.3 Sliding window

Most TCP implementations use a sliding window mechanism for sending data. Multiple data segments are sent in succession without waiting for an acknowledgment for each segment.

The sliding window algorithm uses a lot of 32-bit operations and because 32-bit arithmetic is fairly expensive on most 8-bit CPUs, uIP does not implement it. Also, uIP does not buffer sent packets and a sliding window implementation that does not buffer sent packets will have to be supported by a complex application layer. Instead, uIP allows only a single TCP segment per connection to be unacknowledged at any given time. lwIP, on the other hand, implements TCP's sliding window mechanism using output buffer queues and therefore does not add additional complexity to the application layer.

It is important to note that even though most TCP implementations use the sliding window algorithm, it is not required by the TCP specifications. Removing the sliding window mechanism does not affect interoperability in any way.

7.4.4 Round-trip time estimation

TCP continuously estimates the current Round-Trip Time (RTT) of every active connection in order to find a suitable value for the retransmission time-out.

We have implemented the RTT estimation using TCP's periodic timer. Each time the periodic timer fires, it increments a counter for each connection that has unacknowledged data in the network. When an acknowledgment is received, the current value of the counter is used as a sample of the RTT. The sample is used together with the standard TCP RTT estimation function [13] to calculate an estimate of the RTT. Karn's algorithm [14] is used to ensure that retransmissions do not skew the estimates.

7.4.5 Retransmissions

Retransmissions are driven by the periodic TCP timer. Every time the periodic timer is invoked, the retransmission timer for each connection is decremented. If the timer reaches zero, a retransmission should be made.

The actual retransmission operation is handled differently in uIP and in lwIP. lwIP maintains two output queues: one holds segments that have not yet been sent, the other holds segments that have been sent but not yet been acknowledged by the peer. When a retransmission is required, the first segment on the queue of segments that has not been acknowledged is sent. All other segments in the queue are moved to the queue with unsent segments.

As uIP does not keep track of packet contents after they have been sent by the device driver, uIP requires that the application takes an active part in performing the retransmission. When uIP decides that a segment should be retransmitted, it calls the application with a flag set indicating that a retransmission is required. The application checks the retransmission flag and produces the same data that was previously sent. From the application's standpoint, performing a retransmission is not different from how the data originally was sent. Therefore the application can be written in such a way that the same code is used both for sending data and retransmitting data. Also, it is important to note that even though the actual retransmission operation is carried out by the application, it is the responsibility of the stack to know when the retransmission should be made. Thus the complexity of the application does not necessarily increase because it takes an active part in doing retransmissions.

7.4.6 Flow control

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In each TCP segment, the sender of the segment indicates its available buffer space. A TCP sender must not send more data than the buffer space indicated by the receiver.

In our implementations, the application cannot send more data than the receiving host can buffer. Before sending data, the application checks how many bytes it is allowed to send and does not send more data than the other host can accept. If the remote host cannot accept any data at all, the stack initiates the zero window probing mechanism.

The application is responsible for controlling the size of the window size indicated in sent segments. If the application must wait or buffer data, it can explicitly close

the window so that the sender will not send data until the application is able to handle it.

7.4.7 Congestion control

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. The algorithms used for congestion control [13] are designed to be simple to implement and require only a few lines of code.

Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed. lwIP has the ability to have multiple in-flight segments and therefore implements all of TCP's congestion control mechanisms.

7.4.8 Urgent data

TCP's urgent data mechanism provides an application-to-application notification mechanism, which can be used by an application to mark parts of the data stream as being more urgent than the normal stream. It is up to the receiving application to interpret the meaning of the urgent data.

In many TCP implementations, including the BSD implementation, the urgent data feature increases the complexity of the implementation because it requires an asynchronous notification mechanism in an otherwise synchronous API. As our implementations already use an asynchronous event based API, the implementation of the urgent data feature does not lead to increased complexity.

7.4.9 Connection state

Each TCP connection requires a certain amount of state information in the embedded device. Because the state information uses RAM, we have aimed towards minimizing the amount of state needed for each connection in our implementations.

The uIP implementation, which does not use the sliding window mechanism, requires far less state information than the lwIP implementation. The sliding window implementation requires that the connection state includes several 32-bit sequence numbers, not only for keeping track of the current sequence numbers of the connection, but also for remembering the sequence numbers of the last window updates. Furthermore, because lwIP is able to handle multiple local IP addresses, the connection state must include the local IP address. Finally, as lwIP maintains queues for outgoing segments, the memory for the queues is included in the connection state. This makes the state information needed for lwIP nearly 60 bytes larger than that of uIP which requires 30 bytes per connection.

8 Results

8.1 Performance limits

In TCP/IP implementations for high-end systems, processing time is dominated by the checksum calculation loop, the operation of copying packet data and context switching [15]. Operating systems for high-end systems often have multiple protection domains for protecting kernel data from user processes and user processes from each other. Because the TCP/IP stack is run in the kernel, data has to be copied between the kernel space and the address space of the user processes and a context switch has to be performed once the data has been copied. Performance can be enhanced by combining the copy operation with the checksum calculation [19]. Because high-end systems usually have numerous active connections, packet demultiplexing is also an expensive operation [17].

A small embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the TCP/IP stack and the application program. With an event based API there is no context switch between the TCP/IP stack and the applications.

In such limited systems, the TCP/IP processing overhead is dominated by the copying of packet data from the network device to host memory, and checksum calculation. Apart from the checksum calculation and copying, the TCP processing done for an incoming packet involves only updating a few counters and flags before handing the data over to the application. Thus an estimate of the CPU overhead of our TCP/IP implementations can be obtained by calculating the amount of CPU cycles needed for the checksum calculation and copying of a maximum sized packet.

8.2 The impact of delayed acknowledgments

Most TCP receivers implement the delayed acknowledgment algorithm [3] 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.

Thus the maximum throughput equation when sending data from uIP will be $p = s/(t + t_d)$ where s is the segment size and t_d is the delayed acknowledgment timeout, which typically is between 200 and 500 ms. With a segment size of 1000 bytes, a round-trip time of 40 ms and a delayed acknowledgment timeout of 200 ms, the maximum throughput will be 4166 bytes per second. With the delayed acknowledgment algorithm disabled at the receiver, the maximum throughput would be 25000 bytes per second.

It should be noted, however, that since small systems 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.

8.3 Measurements

For our experiments we connected a 450 MHz Pentium III PC running FreeBSD 4.7 to an Ethernut board [16] through a dedicated 10 megabit/second Ethernet network. The Ethernut board is a commercially available embedded system equipped with a RealTek RTL8019AS Ethernet controller, an Atmel Atmega128 AVR microcontroller running at 14.7456 MHz with 128 kilobytes of flash ROM for code storage and 32 kilobytes of RAM. The FreeBSD host was configured to run the Dummynet delay emulator software [24] in order to facilitate controlled delays for the communication between the PC and the embedded system.

In the embedded system, a simple web server was run on top of the uIP and lwIP stacks. Using the `fetch` file retrieval utility, a file consisting of null bytes was downloaded ten times from the embedded system. The reported throughput was logged, and the mean throughput of the ten downloads was calculated. By redirecting file output to `/dev/null`, the file was immediately discarded by the FreeBSD host. The file size was 200 kilobytes for the uIP tests, and 200 megabytes for the lwIP tests. The size of the file made it impossible to keep it all in the memory of the embedded system. Instead, the file was generated by the web server as it was sent out on the network.

The total TCP/IP memory consumption in the embedded system was varied by changing the send window size. For uIP, the send window was varied between 50 bytes and the maximum possible value of 1450 bytes in steps of 50 bytes. The send window configuration translates into a total RAM usage of between 400 bytes and 3 kilobytes. The lwIP send window was varied between 500 and 11000 bytes in steps of 500 bytes, leading to a total RAM consumption of between 5 and 16 kilobytes.

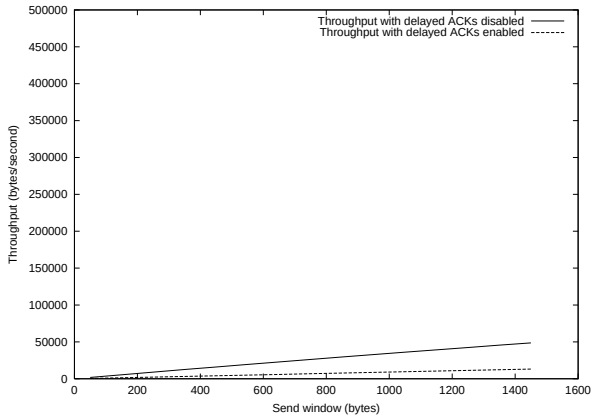

Figure 4: uIP sending data with 10 ms emulated delay.

Figure 4 shows the mean throughput of the ten file downloads from the web server running on top of uIP, with an additional 10 ms delay created by the Dummynet delay emulator. The two curves show the measured throughput with the delayed acknowledgment algorithm disabled and enabled at the receiving FreeBSD host, respectively. The performance degradation caused by the delayed ac-

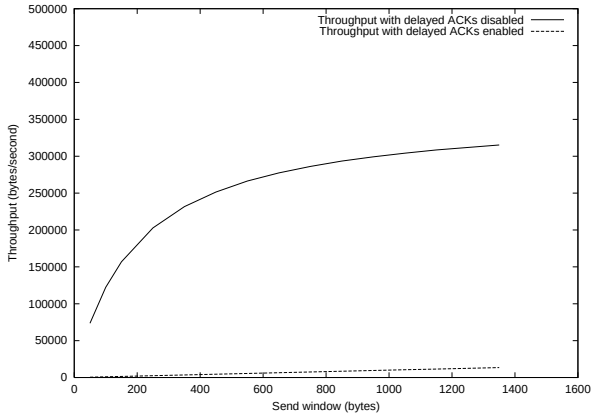

Figure 5: uIP sending data without emulated delay.

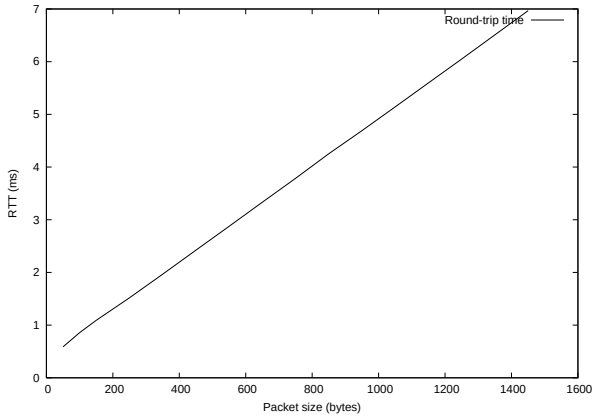

Figure 6: Round-trip time as a function of packet size.

knowledgments is evident.

Figure 5 shows the same setup, but without the 10 ms emulated delay. The lower curve, showing the throughput with delayed acknowledgments enabled, is very similar to the lower one in Figure 4. The upper curve, however, does not show the same linear relation as the previous figure, but shows an increasing throughput where the increase declines with increasing send window size. One explanation for the declining increase of throughput is that the round-trip time increases with the send window size because of the increased per-packet processing time. Figure 6 shows the round-trip time as a function of packet size. These measurements were taken using the `ping` program and therefore include the cost for the packet copying operation twice; once for packet input and once for packet output.

The throughput of lwIP shows slightly different characteristics. Figure 7 shows three measured throughput curves, without emulated delay, and with emulated delays of 10 ms and 20 ms. For all measurements, the delayed acknowledgment algorithm is enabled at the FreeBSD receiver. We see that for small send window sizes, lwIP also suffers from the delayed acknowledgment throughput degradation. With a send window larger than two maximum TCP segment sizes (3000 bytes), lwIP is able to send out two TCP segments per round-trip time and thereby avoids the delayed acknowledgments throughput degradation. Without emulated delay, the throughput quickly reaches a maximum of about 415 kilobytes per second. This limit is likely to be the processing limit of the lwIP code in the embedded system and therefore is the maximum possible throughput for lwIP in this particular system.

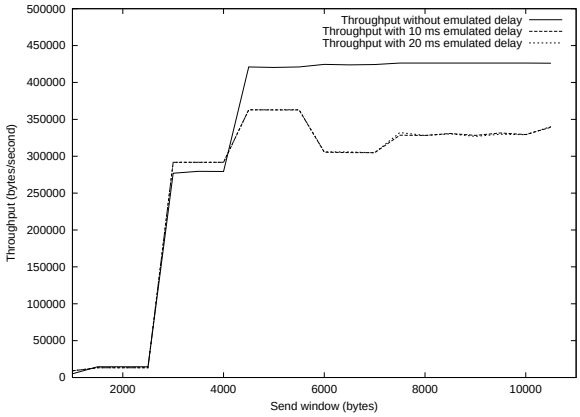

Figure 7: lwIP sending data with and without emulated delays.

The maximum throughput with emulated delays is lower than without delay emulation, and the similarity of the two curves suggests that the throughput degradation could be caused by interaction with the Dummynet software.

8.4 Code size

The code was compiled for the 32-bit Intel x86 and the 8-bit Atmel AVR platforms using gcc [28] versions 2.95.3 and 3.3 respectively, with code size optimization turned on. The resulting size of the compiled code can be seen in Tables 2 to 5. 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

Table 2: Code size for uIP (x86)

Function	Code size (bytes)
Checksumming	464
IP, ICMP and TCP	4724
Total	5188

Table 3: Code size for uIP (AVR)

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

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.

The different memory and buffer management schemes used by lwIP and uIP have implications on code size, mainly in 8-bit systems. Because uIP uses a global buffer for all incoming packets, the absolute memory addresses of the protocol header fields are known at compile time. Using this information, the compiler is able to generate code that uses absolute addressing, which on many 8-bit processors requires less code than indirect addressing.

Is it interesting to note that the size of the compiled lwIP code is larger on the AVR than on the x86, while the uIP code is of about the same size on the two platforms. The main reason for this is that lwIP uses 32-bit arithmetic to a much larger degree than uIP and each 32-bit operation is compiled into a large number of machine code instructions.

Table 4: Code size for lwIP (x86)

Function	Code size (bytes)
Memory management	2512
Checksumming	504
Network interfaces	364
IP	1624
ICMP	392
TCP	9192
Total	14588

Table 5: 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

9 Future work

Prioritized connections. It is advantageous to be able to prioritize certain connections such as Telnet connections for manual configuration of the device. Even in a system that is under heavy load from numerous clients, it should be possible to remotely control and configure the device. In order to do provide this, different connection types could be given different priority. For efficiency, such differentiation should be done as far down in the system as possible, preferably in the device driver.

Security aspects. When connecting systems to a network, or even to the global Internet, the security of the system is very important. Identifying levels of security and mechanisms for implementing security for embedded devices is crucial for connecting systems to the global Internet.

Address auto-configuration. If hundreds or even thou-

sands of small embedded devices should be deployed, auto-configuration of IP addresses is advantageous. Such mechanisms already exist in IPv6, the next version of the Internet Protocol, and are currently being standardized for IPv4.

Improving throughput. The throughput degradation problem caused by the poor interaction with the delayed acknowledgment algorithm should be fixed. By increasing the maximum number of in-flight segments from one to two, the problem will not appear. When increasing the amount of in-flight segments, congestion control mechanisms will have to be employed. Those mechanisms are trivial, however, when the upper limit is two simultaneous segments.

Performance enhancing proxy. It might be possible to increase the performance of communication with the embedded devices through the use of a proxy situated near the devices. Such a proxy would have more memory than the devices and could assume responsibility for buffering data.

10 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, but that the throughput of such a small implementation will suffer. We have not removed any TCP/IP mechanisms in our implementations, but have full support for reassembly of IP fragments and urgent TCP data. Instead, we have minimized the interface between the TCP/IP stack and the application.

The maximum achievable throughput for our implementations is determined by the send window size that the TCP/IP stack has been configured to use. When sending data with uIP, the delayed ACK mechanism at the receiver lowers the maximum achievable throughput considerably. In many situations however, a limited system running uIP will not produce so much data that this will cause problems. lwIP is not affected by the delayed ACK throughput degradation when using a large enough send window.

11 Acknowledgments

Many thanks go to Martin Nilsson, who has provided encouragement and been a source of inspiration throughout the preparation of this paper. Thanks also go to Deborah Wallach for comments and suggestions, the anonymous reviewers whose comments were highly appreciated, and to all who have contributed bugfixes, patches and suggestions to the lwIP and uIP implementations.

References

[1] J. Bentham. *TCP/IP Lean: Web servers for embedded systems*. CMP Books, October 2000.

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

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

[4] G. H. Cooper. TinyTCP. Web page. 2002-10-14. **URL:** *http://www.csonline.net/bpaddock/tinytcp/*

[5] Atmel Corporation. Embedded web server. AVR 460, January 2001. Avalible from www.atmel.com.

[6] A. Dannenberg. MSP430 internet connectivity. SLAA 137, November 2001. Avalible from www.ti.com.

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

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

[9] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, August 1999.

[10] 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.

[11] InterNiche Technologies Inc. NicheStack portable TCP/IP stack. Web page. 2002-10-14. **URL:** *http://www.iniche.com/products/tcpip.htm*

[12] LiveDevices Inc. Embedinet - embedded internet software products. Web page. 2002-10-14.
URL: *http://www.livedevices.com/net_products/embedinet.shtml*

[13] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.

[14] P. Karn and C. Partridge. Improving round-trip time estimates in reliablie transport protocols. In *Proceedings of the SIGCOMM '87 Conference*, Stowe, Vermont, August 1987.

[15] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In *Proceedings of the ACM SIGCOMM '93 Symposium*, pages 259–268, September 1993.

[16] H. Kipp. Ethernut embedded ethernet. Web page. 2002-10-14.
URL: *http://www.ethernut.de/en/*

[17] P. E. McKenney and K. F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[19] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.

[20] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[21] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[22] Kadak Products. Kadak KwikNET TCP/IP stack. Web page. 2002-10-14.
URL: *http://www.kadak.com/html/kdkp1030.htm*

[23] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.

[24] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[25] C. Shannon, D. Moore, and K. Claffy. Beyond folklore: Observations on fragmented traffic. *IEEE/ACM Transactions on Networking*, 10(6), December 2002.

[26] H. Shrikumar. IPic - a match head sized webserver. Web page. 2002-10-14.
URL: *http://www-ccs.cs.umass.edu/~shri/iPic.html*

[27] CMX Systems. CMX-MicroNet true TCP/IP networking. Web page. 2002-10-14.
URL: *http://www.cmx.com/micronet.htm*

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