

Minimal TCP/IP implementation with proxy support

Adam Dunkels
adam@sics.se

February 2001

Abstract

Over the last years, interest for connecting small devices such as sensors to an existing network infrastructure such as the global Internet has steadily increased. Such devices often has very limited CPU and memory resources and may not be able to run an instance of the TCP/IP protocol suite.

In this thesis, techniques for reducing the resource usage in a TCP/IP implementation is presented. A generic mechanism for offloading the TCP/IP stack in a small device is described. The principle the mechanism is to move much of the resource demanding tasks from the client to an intermediate agent known as a proxy. In particular, this pertains to the buffering needed by TCP. The proxy does not require any modifications to TCP and may be used with any TCP/IP implementation. The proxy works at the transport level and keeps some of the end to end semantics of TCP.

Apart from the proxy mechanism, a TCP/IP stack that is small enough in terms of dynamic memory usage and code footprint to be used in a minimal system has been developed. The TCP/IP stack does not require help from a proxy, but may be configured to take advantage of a supporting proxy.

Keywords: Proxy methods, TCP/IP implementation, embedded systems.

Contents

1	Introduction	1
1.1	Goals	1
1.2	Methodology and limitations	2
1.3	Thesis structure	2
2	Background	3
2.1	The TCP/IP protocol suite	3
2.1.1	The Internet Protocol — IP	4
2.1.2	Internet Control Message Protocol — ICMP	5
2.1.3	The simple datagram protocol — UDP	6
2.1.4	Reliable byte stream — TCP	6
2.2	The BSD implementations	11
2.3	Buffer and memory management	12
2.4	Application Program Interface	12
2.5	Performance bottlenecks	12
2.5.1	Data touching	13
2.6	Small TCP/IP stacks	13
3	The proxy based architecture	14
3.1	Architecture	15
3.2	Per-packet processing	15
3.2.1	IP fragment reassembly	15
3.2.2	Removing IP options	16
3.3	Per-connection processing	16
3.3.1	Caching unacknowledged data	17
3.3.2	Ordering of data	18
3.3.3	Distributed state	20
3.4	Alternative approaches	24
3.5	Reliability	24
3.6	Proxy implementation	24
3.6.1	Interaction with the FreeBSD kernel	25
4	Design and implementation of the TCP/IP stack	26
4.1	Overview	26
4.2	Process model	27
4.3	The operating system emulation layer	27
4.4	Buffer and memory management	28
4.4.1	Packet buffers — pbufs	28
4.4.2	Memory management	30
4.5	Network interfaces	30
4.6	IP processing	31
4.6.1	Receiving packets	31

4.6.2	Sending packets	32
4.6.3	Forwarding packets	32
4.6.4	ICMP processing	33
4.7	UDP processing	33
4.8	TCP processing	34
4.8.1	Overview	34
4.8.2	Data structures	35
4.8.3	Sequence number calculations	37
4.8.4	Queuing and transmitting data	37
4.8.5	Receiving segments	38
4.8.6	Accepting new connections	39
4.8.7	Fast retransmit	39
4.8.8	Timers	39
4.8.9	Round-trip time estimation	40
4.8.10	Congestion control	40
4.9	Interfacing the stack	40
4.10	Application Program Interface	41
4.10.1	Basic concepts	41
4.10.2	Implementation of the API	41
4.11	Statistical code analysis	42
4.11.1	Lines of code	43
4.11.2	Object code size	44
4.12	Performance analysis	45
5	Summary	46
5.1	The small TCP/IP stack	46
5.2	The API	46
5.3	The proxy scheme	46
5.4	Future work	47
A	API reference	48
A.1	Data types	48
A.1.1	Netbufs	48
A.2	Buffer functions	48
A.3	Network connection functions	53
B	BSD socket library	59
B.1	The representation of a socket	59
B.2	Allocating a socket	59
B.2.1	The <code>socket()</code> call	59
B.3	Connection setup	60
B.3.1	The <code>bind()</code> call	60
B.3.2	The <code>connect()</code> call	60
B.3.3	The <code>listen()</code> call	61
B.3.4	The <code>accept()</code> call	61
B.4	Sending and receiving data	62
B.4.1	The <code>send()</code> call	62
B.4.2	The <code>sendto()</code> and <code>sendmsg()</code> calls	63
B.4.3	The <code>write()</code> call	63
B.4.4	The <code>recv()</code> and <code>read()</code> calls	64
B.4.5	The <code>recvfrom()</code> and <code>recvmsg()</code> calls	65

C Code examples	66
C.1 Using the API	66
C.2 Directly interfacing the stack	68
D Glossary	71
Bibliography	73

Chapter 1

Introduction

Over the last few years, the interest for connecting computers and computer supported devices to wireless networks has steadily increased. Computers are becoming more and more seamlessly integrated with everyday equipment and prices are dropping. At the same time wireless networking technologies, such as Bluetooth [HNI+98] and IEEE 802.11b WLAN [BIG+97], are emerging. This gives rise to many new fascinating scenarios in areas such as health care, safety and security, transportation, and processing industry. Small devices such as sensors can be connected to an existing network infrastructure such as the global Internet, and monitored from anywhere.

The Internet technology has proven itself flexible enough to incorporate the changing network environments of the past few decades. While originally developed for low speed networks such as the ARPANET, the Internet technology today runs over a large spectrum of link technologies with vastly different characteristics in terms of bandwidth and bit error rate. It is highly advantageous to use the existing Internet technology in the wireless networks of tomorrow since a large amount of applications using the Internet technology have been developed. Also, the large connectivity of the global Internet is a strong incentive.

Since small devices such as sensors are often required to be physically small and inexpensive, an implementation of the Internet protocols will have to deal with having limited computing resources and memory. Despite the fact that there are numerous TCP/IP implementations for embedded and minimal systems little research has been conducted in the area. Implementing a minimal TCP/IP stack is most often considered to be an engineering activity, and thus has not received research attention.

In this thesis techniques for reducing the resources needed for an implementation of the Internet protocol stack in a small device with scarce computing and memory resources are presented. The principle of the mechanism is to move as much as possible of the resource demanding tasks from the small device to an intermediate agent known as a proxy, while still keeping as much of the end-to-end semantics of TCP as possible. The proxy typically has order of magnitudes more computing and memory resources than the small device.

1.1 Goals

There are two goals with this work:

- Designing and implementing a small TCP/IP stack that uses very little resources. The stack should have support for TCP, UDP, ICMP and IP with rudimentary routing.
- The development of a proxy scheme for offloading the small TCP/IP stack.

In order to minimize the TCP/IP implementation, the proxy should implement parts of the standards. The proxy should also offload the memory and CPU of the small system in which the stack runs. The TCP/IP implementation should be sufficiently small in terms of code size and resource demands to be used in minimal systems.

1.2 Methodology and limitations

The research in this thesis have been conducted in an experimental fashion. After an initial idea is sprung, informal reasoning around the idea gives insights into its soundness. Those ideas that are found to be sound are then implemented and tested to see if they should be further pursued or discarded. If an idea should be further pursued it is refined and further discussed.

The testing has been conducted in an environment of a virtual network of processors running on a single FreeBSD 4.1 host. Network devices and links has been emulated in software. The decision to limit the work this way was taken in order to get the work done within the given time frame, and such testing has been noted in future work.

Testing was done by manually downloading files and web pages from a running instance of LWIP over the virtual network. No formal test programs for verifying were used due to time constraints. This has also been noted as future work.

1.3 Thesis structure

The thesis is organized as follows.

Chapter 2 provides a background of the Internet protocol suite.

Chapter 3 presents the architecture and mechanisms of the proxy scheme.

Chapter 4 describes the design and implementation of LWIP, the TCP/IP stack in the small client system. This does not go into details on the code level, such as which parameters are used in function calls, but rather present data structures used, algorithms and mechanisms, and the general flow of control.

Chapter 5 summarizes the work and suggests future work.

Appendix A is a reference manual for the LWIP API.

Appendix B is an implementation of the BSD socket API using the LWIP API.

Appendix C shows some code examples of how to use the LWIP API for applications.

Appendix D contains a glossary.

Of those, all but Chapter 2 presents own work conducted within the scope of this thesis.

Chapter 2

Background

2.1 The TCP/IP protocol suite

Over the last few decades, the protocols in the TCP/IP suite have evolved from being pure research funded by the US military to a world-wide standard for computer communication through the deployment of the global Internet. The TCP/IP protocols are relatively simplistic by design, and are based on the end-to-end principle [SRC84, Car96]. This means that the complexity is pushed to the network edges. Basically, the intermediate nodes, routers, in a TCP/IP internetwork are relatively simple, and the end-nodes implement complex functionality such as reliable transmission protocols or cryptographic security. Most importantly, the intermediate nodes do not keep state of any connections running over them.

A TCP/IP internetwork, or internet for short, is a best-effort datagram network. Information sent over an internet must be divided into blocks called packets or datagrams before transmission. Best-effort means that a transmitted datagram *may* reach its final destination, but there is no guarantee. Datagrams may also be lost, reordered or delivered in any number of copies to the final destination.

The TCP/IP protocols require very little of the underlying link level technology; the only assumption is that the link level provides some form of addressing, i.e., there should exist a way to transmit packets to the appropriate host. Specifically, there is no requirement that the link level has reliable transmission. Many protocols require that the link level supports broadcasts, and some applications require multicast support from the link level. Broadcasting requires that a packet can be transmitted to all network interfaces on the physical network while multicast requires the capability of transmitting a packet to a group of network interfaces. For the most basic functionality, however, broadcast and multicast capabilities are not needed. This implies that a TCP/IP internetwork can be built upon almost any link layer technology.

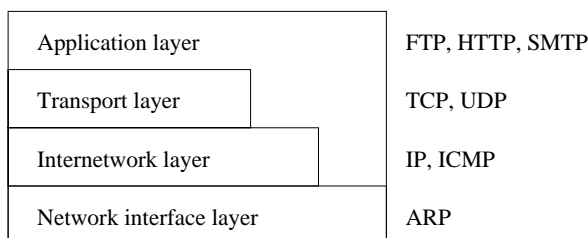


Figure 2.1. The TCP/IP protocol stack with examples of protocols.

The TCP/IP protocol stack consists of four layers, as seen in Figure 2.1. However, the layering is not kept as strict as in other protocol stacks, and it is possible for, e.g., application layer

functions to access the internetwork layer directly. The functionality provided by each layer is (from bottom to top):

The network interface layer is responsible for transporting data over the physical (directly connected) network. It takes care of low-level addressing and address mapping;

The internetwork layer provides abstract addressing and routing of datagrams between different physical networks. It provides an unreliable datagram service to the upper layers;

The transport layer takes care of addressing processes at each host. UDP is used for pure process addressed datagrams, whereas TCP provides a reliable stream transmission for the application layer protocols;

The application layer utilizes the lower layers to provide functionality to the end user. Applications include email (SMTP), world wide web page transfer (HTTP), file transfer (FTP), etc.

Each layer adds a protocol header to the data as shown in Figure 2.2. The figure shows application data encapsulated in a TCP segment, which in turn is included in an IP packet. The IP packet is then encapsulated in a link level frame. Each protocol layer has added a header that keeps protocol specific information. The link layer has also added a trailer.

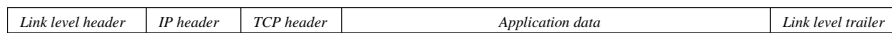


Figure 2.2. A link level frame with TCP/IP headers

2.1.1 The Internet Protocol — IP

The Internet Protocol [Pos81b] IP, is the basic delivery mechanism used in an internet. IP addresses, routes, and optionally fragments packets. Each host can be equipped with any number of network interfaces connected to an internet and each network interface is given atleast one IP address that is unique within the network.

Fragmentation is used when an IP packet is larger than the maximum sized link level frame that can be used. The packet is divided into fragments that fit into link level frames and each fragment is given it's own IP header. Certain fields of the IP header are used to identify to which non-fragmented IP packet the fragments belong. The fragments are treated as ordinary IP packets by the intermediate routers and the final recipient of the packet is responsible for reassembling the fragments into the original packet before the packet is delivered to upper layers.

IP options

The IP options are control information which are appended to the IP header. The IP options may contain time stamps, or information for routers about which forwarding decisions to make. In normal communication IP options are unnecessary but in special cases they can be useful. In today's Internet, packets carrying IP options are very rare [Pax97].

IP routing

The infrastructure of any internet, such as the global Internet, is built up by interconnected *routers*. The routers are responsible for forwarding IP packets in the general direction of the final recipient of the packet. Figure 2.3 shows an example internet with a number of hosts (boxes) connected to a few routers (circles). If host *H* sends data to host *I*, it will send an IP packet towards router *R*, which will inspect the destination address of the IP packet, and conclude that router *S* (as opposed to router *T*) is in the general direction of the final recipient, and will forward

the IP datagram to router *S*. Router *S* will find that the final recipient is directly connected, and will forward the packet on the local network to host *I*.

The IP header contains a field called the time to live (TTL) field for IPv4 and HopLimit for IPv6. Each time an IP packet is forwarded by a router this field is decremented and when it reaches zero, the packet is dropped. This ensures that IP packets eventually will leave the network, and is used to prevent packets circling forever.

In order to gather information about the topology of the network the routers communicate with routing protocols. In the routing protocol messages, the routers report on the reachability of networks and hosts and each router gathers knowledge of the general direction of hosts on the network. In case of a network failure the router can find new working paths through the network by using information exchanged in the routing protocol.

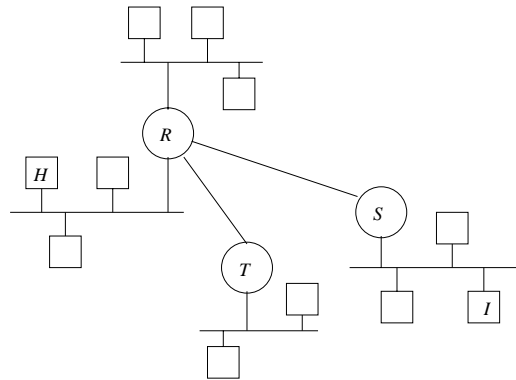


Figure 2.3. An internet with routers and hosts

Congestion

IP routers work with the so called store-and-forward principle, where incoming IP packets are buffered in an internal queue if they cannot be forwarded immediately. The available memory for buffered packets is not unlimited, however, and any packets arriving when the buffer is full are dropped. Most often, no notification to either the sender or the receiver of the packet is given. When the queue in a router is full, and packets are being dropped, the router is said to be congested.

2.1.2 Internet Control Message Protocol — ICMP

The Internet Control Message Protocol [Pos81a] ICMP, provides IP with an unreliable signaling and querying mechanism. ICMP messages are sent in a number of situations, often to report a errors. ICMP messages can be sent in response to IP packets that are destined to a host or network that is unreachable or when an IP packet has timed out, i.e., where the TTL field of the IP header has reached zero. There are two classes of ICMP messages, those which are sent from end hosts and those that source from routers. ICMP messages from routers report on network problems or better routes, whereas ICMP messages from end hosts typically report that a transport layer port was unreachable or are replies for the ECHO mechanism.

The querying ICMP echo mechanism is probably the most commonly used ICMP mechanism. The ICMP echo mechanism is not used to report on errors. Rather, this is used by programs such as ping to check whether a host is reachable over the network. A host that receives an ICMP ECHO message responds by sending an ICMP ECHO-REPLY message back to the sender of the ICMP ECHO message. Any data contained in the ECHO message is copied into the ECHO-REPLY message.

Even though ICMP uses IP as its delivery mechanism, ICMP is considered an integral part of IP and is often implemented as such.

2.1.3 The simple datagram protocol — UDP

The User Datagram Protocol [Pos80] UDP, is the simplest protocol in the TCP/IP suite and the RFC specifying UDP fits on two printed pages. UDP provides an extra layer of multiplexing; where IP provides addressing of a specific host in an internet, UDP provides per-process addressing by the use of ports. The ports are 16 bit values that are used to distinguish between different senders and receivers at each endpoint. Each UDP datagram is addressed to a specific port at the end host and incoming UDP datagrams are demultiplexed between the recipients.

UDP also optionally calculates a checksum over the datagram. The checksum covers the UDP header and data as well as a pseudo header consisting of certain fields of the IP header, including the IP source and destination addresses. The checksum does not make UDP reliable however, since UDP datagrams with a failing checksum are dropped without notifying the application process. Delivery of UDP datagrams is not guaranteed and UDP datagrams may arrive out of order and in any number of copies due to the nature of IP.

UDP is used for applications that requires low latency but not a very reliable transfer, such as applications that send real time video or audio.

UDP Lite

UDP Lite [LDP99] is an extension to UDP which allows the checksum to cover only a part of the UDP datagram, most commonly the UDP header and any application level header directly following it. This is useful for applications which send and receive data that is insensitive to spurious bit errors, such as real time audio or video. Wireless links are prone to errors, and when using UDP Lite, datagrams that otherwise would be discarded due to a failing UDP checksum can still be used. UDP Lite utilizes the fact that the length field in the UDP header is redundant, since the length of the datagram can be obtained from IP. Instead, the length field specifies how much of the datagram is covered by the checksum. In a low end system, checksumming only parts of the datagrams can also be a performance win.

2.1.4 Reliable byte stream — TCP

The Transmission Control Protocol [Pos81c] TCP, provides a reliable byte stream on top of the unreliable datagram service provided by the IP layer. Reliability is achieved by buffering of data combined with positive acknowledgments (ACKs) and retransmissions. TCP hides the datagram oriented IP network behind a virtual circuit abstraction, in which each virtual circuit is called a connection. A connection is identified by the IP addresses and TCP port numbers of the endpoints.

TCP options

TCP options provide additional control information other than that in the TCP header. TCP options reside between the TCP header and the data of a segment. Since the original TCP specification [Pos81c] a number of additions to TCP has been defined as TCP options. This includes the TCP selective acknowledgment SACK [MMFR96] and the TCP extensions for high speed networks [JBB92] which define TCP time-stamps and window scaling options.

The only TCP option defined in the original TCP specification was the Maximum Segment Size (MSS) option which specifies how large the largest TCP segment may be in a connection. The MSS option is sent by both parties during the opening of a connection.

Reliable stream transfer

Each byte in the byte stream is assigned a sequence number starting at some arbitrary value. The stream is partitioned into arbitrary sized segments. The TCP sender will try however, to fill each segment with enough data so that the segment is as large as the maximum segment size of the connection. This is shown in Figure 2.4 (refer to the paragraphs on opening and closing a connection later in this section for a description of the SYN and FIN segments). Each segment is prepended with a TCP header and transmitted in separate IP packets. In theory, for each received segment the receiver produces an ACK. In practice however, most TCP implementations send an ACK only on every other incoming segment in order to reduce ACK traffic. ACKs are also piggybacked on outgoing TCP segment. The ACK contains the next sequence number expected in the continuous stream of bytes. Thus, the ACKs do not acknowledge the reception of any individual segment, but rather acknowledges the transmission of a continuous range of bytes.

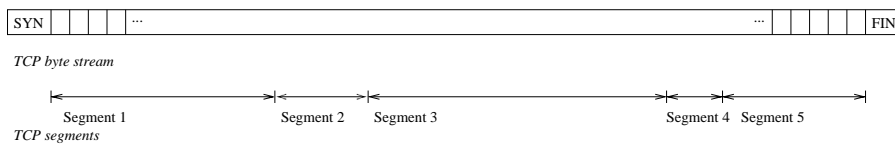


Figure 2.4. A segmented TCP byte stream

Consider a TCP receiver that has received all bytes up to and including sequence number x , as well as the bytes $x + 20$ to $x + 40$, with a gap between $x + 1$ and $x + 19$, as in the top figure of Figure 2.5. The ACK will contain the sequence number $x + 1$, which is the next sequence number expected in the continuous stream. When the segment containing bytes $x + 1$ to $x + 19$ arrives, the next ACK will contain the sequence number $x + 41$. This is shown in the bottom figure of Figure 2.5.

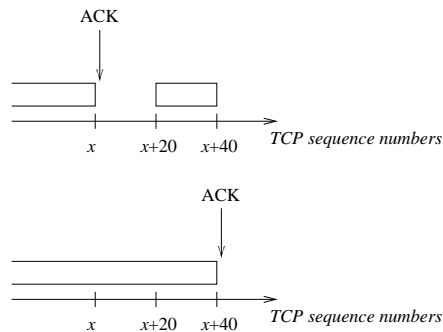


Figure 2.5. TCP byte stream with a gap and corresponding ACKs

The sending side of a TCP connection keeps track of all segments sent that have not yet been ACKed by the receiver. If an ACK is not received within a certain time, the segment is retransmitted. This process is referred to as a time-out and is depicted in Figure 2.6. Here we see a TCP sender sending segments to a TCP receiver. Segment 3 is lost in the network and the receiver will continue to reply with ACKs for the highest sequence number of the continuous stream of bytes that ended with segment 2. Eventually, the sender will conclude that segment 3 was lost since no ACK has been received for this segment, and will retransmit segment 3. The receiver has now received all bytes up to and including segment 5, and will thus reply with an ACK for segment 5. (Even though TCP ACKs are not for individual segments it is sometimes convenient to discuss ACKs as belonging to specific segments.)

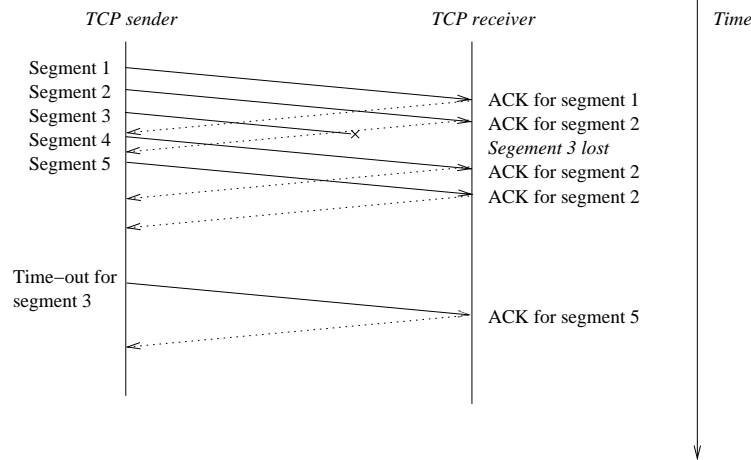


Figure 2.6. Loss of a TCP segment and the corresponding time-out

Round-trip time estimation

A critical factor of any reliable protocol is the round-trip time estimation, since the round-trip time is used as a rule of thumb when determining a suitable time to wait for an ACK before retransmitting a segment. If the round-trip time estimate is much lower than the actual round-trip time of the connection, segments will be retransmitted before the original segment or its corresponding ACK has propagated through the network. If the round-trip time estimation is too high, time-outs will be longer than necessary thus degrading performance.

TCP uses the feedback provided by its acknowledgment mechanism to measure round-trip times and calculates a running average of the samples. Round-trip time measurements are taken once per window, since it is assumed that all segments in one window's flight should have approximately the same round-trip time. Also, taking round-trip samples for every segment does not yield better measurements [AP99]. If a segment for which a round-trip time was measured is a retransmission, that round-trip time measurement is discarded [KP87]. This is because the ACK for the retransmitted segment may have been sent either in response to the original segment or to the retransmitted segment. This makes the round-trip time estimation ambiguous.

Flow control

The flow control mechanism in TCP assures that the sender will not overwhelm the receiver with data that the receiver is not ready to accept. Each outgoing TCP segment includes an indication of the size of the available buffer space and the sender must not send more data than the receiver can accommodate. The available buffer space for a connection is referred to as the window of the connection. The window principle ensures proper operation even between two hosts with drastically different memory resources.

The TCP sender tries to have one receiver window's worth of data in the network at any given time provided that the application wishes to send data at the appropriate rate (this is not entirely true; see the next section on congestion control). It does this by keeping track of the highest sequence number s ACKed by the receiver, and makes sure not to send data with sequence number larger than $s + r$, where r is the size of the receiver's window.

Returning to Figure 2.6, we see that the TCP sender stopped sending segments after segment 5 had been sent. If we assume that the receiver's window was 1000 bytes in this case and that the individual sizes of segments 3, 4 and 5 was exactly 1000 bytes, we can see that since the sender had not received any ACK for segments 3, 4 and 5, the sender refrained from sending any more segments. This is because the sequence number of segment 6 would in this case be equal to the

sum of the highest ACKed sequence number and the receiver's window.

Congestion control

While flow control tries to avoid that buffer space will be overrun at the end points, the congestion control mechanisms [Jac88, APS99] tries to prevent the overrun of router buffer space. In order to achieve this TCP uses two separate methods:

- slow start, which probes the available bandwidth when starting to send over a connection, and
- congestion avoidance, which constantly adapts the sending rate to the perceived bandwidth of the path between the sender and the receiver.

The congestion control mechanism adds another constraint on the maximum number of outstanding (unacknowledged) bytes in the network. It does this by adding another state variable called the *congestion window* to the per-connection state. The *minimum* of the congestion window and the receiver's window is used when determining the maximum number of unacknowledged bytes in the network.

TCP uses packet drops as a sign of congestion. This is because TCP was designed for wired networks where the main source of packet drops (> 99%) are due to buffer overruns in routers. There are two ways for TCP to conclude that a packet was dropped, either by waiting for a time-out, or to count the number of duplicate ACKs that are received. If two ACKs for the same sequence number is received, this could mean that the packet was duplicated within the network (not an unlikely event under certain conditions [Pax97]). It could also mean that segments were reordered on their way to the receiver. However, if three duplicate ACKs are received for the same sequence number, there is a good chance that this indicates a lost segment. Three duplicate ACKs trigger a mechanism known as fast retransmit and the lost segment is retransmitted without waiting for its time-out.

During slow start, the congestion window is increased with one maximum segment size per received ACK, which leads to an exponential increase of the size of the congestion window¹. When the congestion window reaches a threshold, known as the slow start threshold, the congestion avoidance phase is entered.

When in the congestion avoidance phase, the congestion window is increased linearly until a packet is dropped. The drop will cause the congestion window to be reset to one segment, the slow start threshold is set to half of the current window, and slow start is initiated. If the drop was indicated by three duplicate ACKs the fast recovery mechanism is triggered. The fast recovery mechanism will halve the congestion window and keep TCP in the congestion avoidance phase, instead of falling back to slow start.

Increasing the congestion window linearly is in fact harder than increasing the window exponentially, since a linear increase requires an increase of one segment per round-trip time, rather than one segment per received ACK. Instead of using the round-trip time estimate and using a timer to increase the congestion window, many TCP implementations, including the BSD implementations, increase the congestion window by a fraction of a segment per received ACK.

The TCP state diagram

TCP not only provides a reliable stream transfer, but also a reliable way to set up and take down connections. This process is most commonly captured as a state diagram and the TCP state diagram is shown in Figure 2.7 on page 10, where the boxes represent the TCP states and the arcs represent the state transitions with the actions taken as a result of the transitions. The bold face text shows the actions taken by the application program.

¹Despite its name, slow start opens the congestion window quite rapidly; the name was coined at a time when TCP senders started with sending a whole receiver's window worth of data.

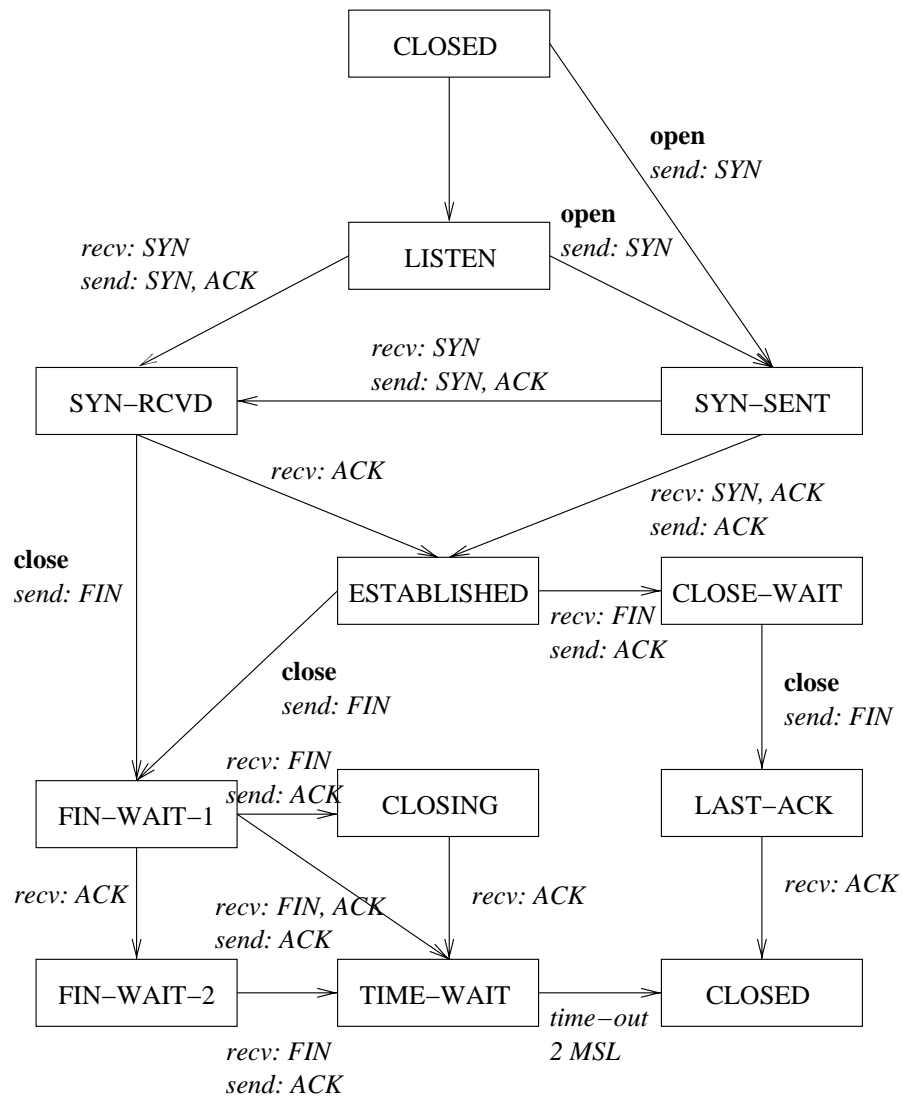


Figure 2.7. The TCP state diagram

Opening a connection

In order for a connection to be established, one of the participating sides must act as a server and the other as a client. The server enters the LISTEN state and waits for an incoming connection request from a client. The client, being in the CLOSED state, issues an **open**, which results in a TCP segment with the SYN flag set to be sent to the server and the client enters the SYN-SENT state. The server will enter the SYN-RCVD state and responds to the client with a TCP segment with both the SYN and ACK flags set. As the client responds with an ACK both sides will be in the ESTABLISHED state and can begin sending data.

This process is known as the three way handshake (Figure 2.8), and will not only have the effect of setting both sides of the connection in the ESTABLISHED state, but also synchronizes the sequence numbers for the connection.

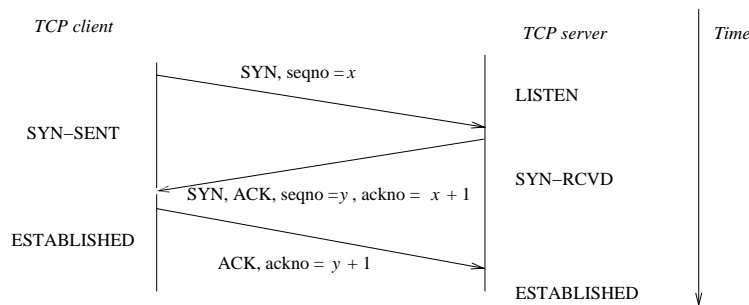


Figure 2.8. The TCP three way handshake with sequence numbers and state transitions

Both the SYN and FIN segments occupy one byte position in the byte stream (refer back to Figure 2.4) and will therefore be reliably delivered to the other end point of the connection through the use of the retransmission mechanism.

Closing a connection

The process of closing a connection is rather more complicated than the opening process since all segments must be reliably delivered before the connection can be fully closed. Also, the TCP close function will only close one end of the connection, meaning that both ends of the connection will have to close before the connection is completely terminated.

When a connection end point issues a close on the connection, the connection state on the closing side of the connection will traverse the FIN-WAIT-1 and FIN-WAIT-2 states, and optionally passing the CLOSING state, after which it will end up in the TIME-WAIT state. The connection is required to stay in the TIME-WAIT state for twice the maximum segment lifetime (MSL) in order to account for duplicate copies of segments that might still be in the network (see the discussion in Section 3.3.3 on page 20). The remote end goes from the ESTABLISHED state to the CLOSE-WAIT state in which it stays until the connection is closed by both sides. When the remote end issues a **close**, the connection passes the LAST-ACK state and the connection will be removed at the remote end.

2.2 The BSD implementations

In the early eighties the TCP/IP protocol suite was implemented at BBN technologies for the University of Berkeley, California as a part of their BSD operating system. The source code for their implementation was later published freely and the code could be included free of charge in any commercial products. This led to the code being used in many operating systems from large vendors. Also, due to the availability of the code, the BSD implementation of the TCP/IP protocol

suite is the de facto reference implementation, and is the most well documented implementation (see for example [WS95]).

Since the first release in 1984, the BSD TCP/IP implementation has evolved and many different versions have been released. The first release to incorporate the TCP congestion control mechanisms described above was called TCP Tahoe. The Tahoe release still forms the basis of many TCP/IP implementations found in modern operating systems. The Tahoe release did not implement the fast retransmit and fast recovery algorithms, which were developed after the release. The BSD TCP/IP release which incorporated those algorithms, as well as many other performance related optimizations, was called TCP Reno. TCP Reno has been improved with better retransmission behavior and those TCP modifications are known as NewReno [FH99].

2.3 Buffer and memory management

At the heart of every network stack implementation lies the memory buffer management subsystem. Memory buffers are used to hold every packet in the system, and are therefore allocated and deallocated very frequently. Every time a packet arrives a memory buffer must be allocated, and every time a packet leaves the host the memory buffer associated with it must be freed.

The BSD TCP/IP implementations use a buffer scheme where the buffers are known as mbufs [MBKQ96]. Mbufs were designed as a buffer system for use in any interprocess communication including network communication. In the BSD implementation, mbufs are small buffers of 128 bytes each which includes both user data and management information. 108 bytes can be used for user data in each mbuf. For large messages, a larger memory chunk of fixed size (1 or 2 kilobytes) known as an mbuf cluster can be referenced by the mbuf. The buffers are of fixed size to make allocation faster and simpler and to reduce external fragmentation.

Mbufs can be linked to form an mbuf chain. This is useful for appending or prepending headers or trailers to a message. Headers can be appended by allocating an mbuf, filling the mbuf with the header, and chaining the mbufs containing the data to the header mbuf.

2.4 Application Program Interface

The Application Program Interface, API, is a fundamental part of any implementation of a particular service. The API is the entry points used by an application program to utilize the services provided by a module or library. Since the API is used in every communication between the application and the module, tailoring the API to suite the implementation of the module reduces the communication overhead.

The de-facto standard TCP/IP API is the BSD socket API [MBKQ96], which abstracts the network communication such that sending or receiving data from the network is not different from writing or reading from an ordinary file. From the application's point of view, TCP connections are just a continuous stream of bytes, rather than segmented.

Even though the BSD socket API is not formally defined as a standard API, the success of BSD has resulted in a large number of applications written for the BSD socket API.

2.5 Performance bottlenecks

Early research on efficiency of the implementation of communication protocols [Cla82a] found numerous key factors that degrade the performance of a protocol implementation. One of the main points is that in order to achieve reasonable throughput the protocol implementation will have to be put in the operating system kernel. There are numerous reasons for doing this. First, kernel code is in general not swapped out to secondary storage, nor paged out through virtual memory mechanisms. If a communication protocol will have to be fetched from disk, this will cause a serious delay when servicing either an incoming packet or a request from an application program. Also, since kernel code is often cannot be preempted, once a protocol has begun processing it will

complete in the shortest possible amount of time. Moreover, when having communication protocols reside in a processes, the protocol might have to compete with other processes for CPU resources and the protocol might have to wait for a scheduling quantum before servicing a request.

Another key point in protocol processing is that the processing time of a packet depends on factors such as CPU scheduling and interrupt handling. The conclusions drawn from this observation is that the protocol should send large packets and that unneeded packets should be avoided. Unneeded packets will in general require almost the same amount of processing as a useful packet but does not do anything useful.

The design of a protocol stack implementation, where protocols are layered on top of each other can be done in different ways. Depending on the way the implementation of the layering is designed, the efficiency of the implementation varies. The key issue is the communication overhead between the protocol layers.

2.5.1 Data touching

One of the largest bottlenecks in any protocol processing, however, is the data touching, in particular for large packets [KP96]. This pertains to the common operations of checksumming and copying. Checksumming and copying needs to pick up every byte in a packet and process it. Since end to end checksumming is essential [SRC84] it cannot be optimized away. Data copying is also needed in some cases, in particular when moving data from a network interface into main memory. Also, when data passes protection domains, such as when incoming data is passed from the operating system kernel into an application process, the data is usually copied. In [PP93] it is shown that combining necessary data copying with checksumming can increase performance.

2.6 Small TCP/IP stacks

There exists numerous very small TCP/IP implementations. Many of them have been made by individuals as hobby projects, whereas others have been developed by software companies for use in commercial embedded systems. For many of these small TCP/IP stacks the source code is not available, and it has therefore not been able to study them.

One of the most notable implementations is the iPic web server [Shr], which implements a web server on a PIC 12C509A which is a chip of the size of a match-head. For this, they claim to have implemented a standards compliant (as defined by [Bra89]) TCP/IP stack in 256 bytes of code. The files for the web server are stored on an EEPROM chip. The source code for the TCP/IP stack is not available.

In order to make such a small implementation one will have to make certain shortcuts. It could for example be possible to store precomputed TCP headers in ROM and only make small modifications of the acknowledgment numbers and the TCP checksum when transmitting them. Also, reducing the number of multiple connections to one would greatly simplify the code.

Chapter 3

The proxy based architecture

In a small client system that is to operate in a wireless network, there are essentially four quantities worth optimizing,

- power consumption,
- code efficiency in terms of execution time,
- code size, and
- memory utilization.

Power consumption can be reduced by, e.g., tailoring the network protocols or engineering of the physical network device and is not covered in this work. The efficiency of the code will, however, effect the power consumption in that more efficient code will require less electrical CPU power than less efficient code. Code efficiency requires careful engineering, especially in order to reduce the amount of data copying. The size of the code can be reduced by careful design and implementation of the TCP/IP stack in terms of both the protocol processing and the API. Since a typical embedded system has more ROM than RAM available the most profitable optimization is to reduce the RAM utilization in the client system. This can be done by letting the proxy do a lot of the buffering that otherwise would have to be done by the client.

Most of the basic protocols in the TCP/IP suite, such as IP, ICMP, and UDP are fairly simple by design and it is easy to make a small implementation of these protocols. Additionally, since they are not designed to be reliable they do not require that end-hosts buffer data. TCP, on the other hand, is more expensive both in terms of code size and memory consumption mostly due to the reliability of TCP, which requires it to buffer and retransmit data that is lost in the network.

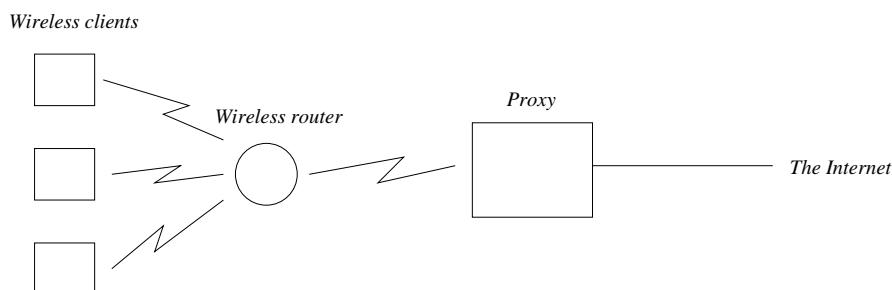


Figure 3.1. The proxy environment

The proxy is designed to operate in an environment as shown in Figure 3.1, where one side of the proxy is connected to the Internet through a wired link, and the other side to a wireless

network with zero or more routers and possibly different wireless link technologies. The fact that there may be routers in the wireless network means that all packet losses behind the proxy cannot be assumed to stem from bit errors on the wireless links, since packets also can be dropped if the routers are congested. Although routers may appear in the wireless network, the design of the proxy does not depend on their existence, and the proxy may be used in an environment with directly connected clients as well.

In an environment as in Figure 3.1 the wireless clients and the router, which are situated quite near each other, can communicate using a short range wireless technology such as Bluetooth. The router and the proxy communication can use a longer range and more power consuming technology, such as IEEE 802.11b.

An example of this infrastructure is the Arena project [ARN] conducted at Luleå University of Technology. In this project, ice hockey players of the local hockey team will be equipped with sensors for measuring pulse rate, blood pressure, and breathing rate as well as a camera for capturing full motion video. Both the sensors and the camera will carry an implementation of the TCP/IP protocol suite, and information from the sensors and the camera will be transmitted to receivers on the Internet. The sensors, which corresponds to the wireless clients in Figure 3.1, communicates using Bluetooth technology with the camera, which is the wireless router. The camera is connected with a gateway, which runs the proxy software, using wireless LAN IEEE 802.11b technology.

Apart from this very concrete example, other examples of this environment are easily imagined. In an office environment, people at the office has equipment such as hand held computers, and at each desk a wireless router enables them to use the hand held devices on the corporate network. In an industrial environment, the machines might be equipped with sensors for measurement and control. Each machine also has one sensor through which the others communicate. The sensors might run some distributed control algorithm for controlling the machine, and the process can be monitored from a remote location via a local IP network or over the Internet.

The proxy does not require any modifications to TCP in either the wireless clients or the fixed hosts in the Internet. This is advantageous since any TCP/IP implementation may be used in the wireless clients, and also simplifies communication between clients in the wireless network behind the proxy.

3.1 Architecture

The proxy operates as an IP router in the sense that it forwards IP packets between the two networks to which it is connected, but also captures TCP segments coming from and going to the wireless clients. Those TCP segments are not necessarily directly forwarded to the wireless hosts, but may be queued for later delivery if necessary. IP packets carrying protocols other than TCP are forwarded immediately. The path of the packets can be seen in Figure 3.2. The proxy does both per-packet processing and per-connection processing in order to offload the client system. Per-connection processing pertains only to TCP connections.

3.2 Per-packet processing

Per-packet processing is done, as the name implies, on every packet forwarded by the proxy. When an IP packet is received by the proxy, it checks whether any per-packet processing should be done for the packet before it is passed up to the per-connection processing module or forwarded to the wireless host.

3.2.1 IP fragment reassembly

When an end host receives an IP fragment, it has to buffer the fragment and wait until all fragments have arrived before the packet can be reassembled and passed to the upper layer protocols. Since it is possible that some fragments have been lost in the network, the end host does not wait infinitely

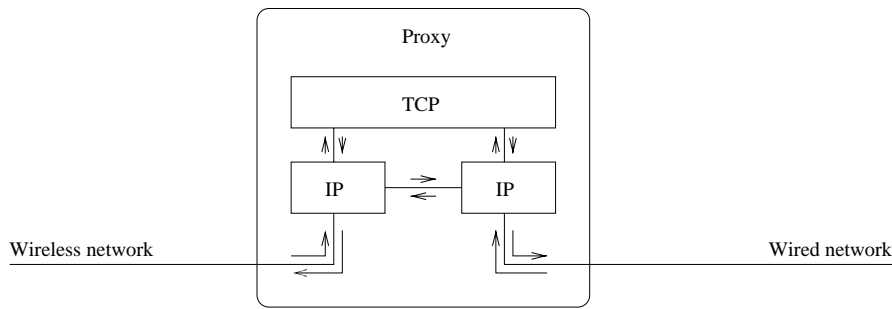


Figure 3.2. The proxy architecture

long for all fragments. Rather, each IP packet which lacks one or more fragments is associated with a lifetime, and if the missing fragments have not been received within the lifetime, the packet is discarded. This means that if one or more fragments were lost on its way to the receiver, the other fragments are kept in memory for their full lifetime in vain.

Since reassembly of IP fragments might use useful memory for doing useless work in the case of lost fragments, this process can be moved to the proxy. Also, since the loss of a fragment of an IP packet implies the loss of all fragments of the packet, IP fragmentation does not work well with lossy links, such as wireless links. Therefore, by making the reassembly of fragmented IP packets at the proxy the wireless links are better utilized.

The problem with reassembling, potentially large, IP packets at the proxy is that the reassembled packet might be too large for the wireless links behind the proxy. No suitable solution to this problem has been found, and finding a better solution has been postponed to future work (Section 5.4).

3.2.2 Removing IP options

For IP packets that do not carry IP options the IP header has a fixed size. This can be exploited by the TCP/IP implementation in the final recipient of the packet since for those packets, the offset to the transport layer header is fixed. Having a fixed offset simplifies the code in the end hosts in many aspects, in particular if the implementation integrates the layers in the protocol processing. Since IP options are not common and many hosts do not implement them, they can safely be removed from the packets going to the wireless hosts.

3.3 Per-connection processing

The per-connection processing function uses three different mechanisms to reduce the load on the client system. These are

- acknowledging data sent by the client so that it will not need to wait for an entire round-trip time (or more) for outstanding data to be acknowledged,
- reordering TCP segments so that the client need not buffer out-of-sequence data, and
- distributed state, which relieves some of the burden of closing connections.

Of these, the first is most useful in connections where the wireless client acts mostly as a TCP sender, e.g., when the wireless client hosts an HTTP server. The second is most useful when the client is the primary receiver of data, e.g., when downloading email to the client, and the third when the client is the first end-point to close down connections, such as when doing HTTP/1.0 [BLFF96] transfers.

For every active TCP connection being forwarded, the proxy has a Protocol Control Block (PCB) entry which contains state of the connection. This includes variables such as the IP addresses and port numbers of the endpoints, the TCP sequence numbers, etc. The PCB entries themselves are soft-state entities in that each PCB entry has an associated lifetime, which is updated every time a TCP segment belonging to the connection arrives. If no segments arrive within the lifetime, the PCB will be completely removed. This ensures that PCBs for inactive connections and connections that have terminated because of end host reboots will not linger in the proxy indefinitely. The lifetime depends on the state of the connection; if the proxy holds cached data for the connection, the lifetime is prolonged.

When a TCP segment arrives, the proxy tries to find a PCB with the exact same IP addresses and port numbers as the TCP segment. This is similar to the process of finding a PCB match in an end host, but differs in the way that both connection endpoints are completely specified, i.e., there are no wild-card entries in the list of PCBs. A new PCB is created if no match is found. If a PCB match is found, the proxy will process the TCP segment as described in the following sections.

3.3.1 Caching unacknowledged data

For a connection between the wireless client and a remote host in the Internet, the round-trip time between the wireless client and the proxy always is shorter than that between the wireless client and the remote host. This fact is used to reduce the buffering needs in the wireless client by letting the proxy acknowledge data from the client. When the client receives an acknowledgment for sent, and therefore buffered data, the buffer space associated with the data can be deallocated in the client, since the data is known to be successfully received¹. The proxy will, upon reception of a data segment from the wireless client, forge an ACK so that the client believes that the ACK came from the remote host. When the proxy has acknowledged a segment, the proxy has assumed responsibility for retransmitting the segment should it be lost on its way to the remote host. The general idea is that data as fast as possible should be moved from the client to the proxy.

By allowing the proxy to prematurely acknowledge data from the client the end to end semantics of TCP are broken in that the acknowledgments are no longer end to end. In other words, the wireless client sending data believes that the data has reached its destination, when in reality the data only has reached the proxy. In order to keep some of the end to end semantics, the proxy does not acknowledge the SYN and FIN segments, as seen in Figure 3.3. This means that the SYN and FIN segments will have a longer round-trip time than normal TCP segments. For the SYN segment this is not a problem since the following data segments will have a shorter round-trip time, and the sender will adjust to this. The longer round-trip time for the FIN will make the sender retransmit the FIN a few times, but since the FIN segment is small this will not be a large problem. Moreover, since the wireless client may be aware of the proxy, the retransmission time-out can be increased for the FIN segment.

If the proxy finds that the remote host is not responding, i.e., the proxy does not receive ACKs for the segments it has buffered after the proper number of retransmissions, it sends a TCP RST to the client. This is the equivalent of letting the connection time out in the client, but in this case the connection times out in the proxy. The proxy will wait for a fairly long time before timing-out a connection, preferably a few times longer than end host would wait.

The SYN segment cannot be acknowledged by the proxy since the proxy does not know anything about the remote host to which the SYN segment was intended. The proxy does not know in what way the remote host will respond (i.e., if it responds with a SYN-ACK or a RST) or whether it will respond at all. Also, if the SYN would be acknowledged by the proxy it would have to do sequence number translation on all further segments in the connection, and the connection state would therefore have to be hard rather than soft.

As a side effect of the proxy acknowledgments, the client will perceive a shorter round-trip time than the actual round-trip time of the connection and will therefore have a lower retransmission

¹Since the data is acknowledged by the proxy, the client cannot know that the data has been received by the remote host. It does know, however, that the data has been successfully received by someone.

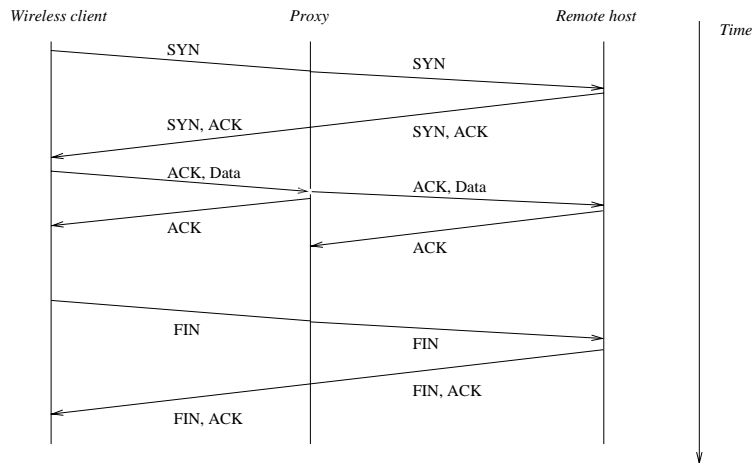


Figure 3.3. The proxy acknowledging data from the client

time-out. This will lead to faster retransmission of segments that are lost due to bit errors over the wireless links behind the proxy and higher overall throughput.

Congestion control

Since the proxy is responsible for the retransmission of prematurely acknowledged segments, the wireless client is unaware of any congestion in the wired internet and is therefore unable to respond to it. One approach to solve this problem would be to let the proxy sense the congestion, and use Explicit Congestion Notification [RF99] (ECN) to inform the client of the congestion. The client would then reduce its sending rate appropriately. The disadvantage of this approach is that the client is forced to buffer data that the proxy could have assumed responsibility for. Also, it contradicts the idea of having the data moved to the proxy as fast as possible.

Instead, the proxy assumes responsibility of the congestion control over the wired Internet. Since the proxy has the responsibility for retransmitting segments that it has acknowledged, the same congestion control mechanisms that are used in ordinary TCP can be used by the proxy. When the congestion window at the proxy does not allow the proxy to send more segments any segments coming from the client are acknowledged, and the advertised receiver's window is artificially closed. To the wireless client this seems as if the application at the remote host does not read data at the same rate that the wireless client is sending. When doing this, the congestion control problem of the wired links is mirrored as a flow control problem in the wireless network behind the proxy.

3.3.2 Ordering of data

TCP segments may arrive out of order for two reasons, either because the Internet has reordered the segments such that a segment with a higher sequence number arrives prior to a segment with a lower sequence number, or because a segment is lost thus causing a gap in the sequence numbers. Of those, the latter is the more likely and happens regularly even in short-lived TCP connections.

If the TCP receiver does not buffer the out-of-order segments the sender is forced to retransmit all of those, even those received, thus wasting bandwidth as well as causing unnecessary delays. The Internet host requirements [Bra89] states that a TCP receiver may refrain from buffering out-of-order segments, but strongly encourages hosts to do so.

In the wireless clients, buffering out-of-order segments may use a large part of the available memory. The proxy will therefore intercept out-of-order segments and instead of forwarding them to the wireless client queue them for later delivery. When an in-order segment arrives the proxy

forwards all previously queued out-of-order segments to the client, while trying not to congest any wireless routers. If the proxy is installed to operate in an environment without wireless routers, the congestion control features can be switched off.

Using this mechanism, the clients are likely to receive all TCP segments in order. This will not only relieve burden of the memory, but also work well with Van Jacobson's header prediction optimization [Jac90], which makes processing of in-order segments more efficient than processing of out-of-order segments.

Since the client is receive most of its segments in order, it can refrain from buffering out-of-order segments. If an out-of-order segment do arrive at the client, it will produce an immediate ACK. This duplicate ACK will be able to trigger a fast retransmit from the proxy.

Buffering out-of-order segments

When a TCP segment destined for the wireless client arrives at the proxy, the corresponding PCB is found and the sequence number of the segment is checked to see if it is the next sequence number expected, based on the information in the PCB. If the sequence number is higher than expected, the segment is queued and is not forwarded to the client.

Since the client does not receive the out-of-order segments, it cannot produce any duplicate ACKs that would trigger a fast retransmit from the remote host. Instead, the proxy will forge ACKs for every incoming out-of-order segment and send them to the remote host. The forged ACKs will acknowledge the last segment received and acknowledged by the wireless client. The proxy will not acknowledge segments that is buffered in the proxy, thus maintaining the end to end semantics.

Transmitting in-order segments

When an in-order segment arrives, the in-order segment and any contiguous earlier queued out-of-order segments are transmitted to the client. From the time when the proxy starts to send the previously buffered in-sequence segments until the client has acknowledged them, the fast retransmit threshold in the proxy is lowered from three duplicate ACKs to one duplicate ACK. If one of the segments is lost on its way to the client, two duplicate ACKs will be received from the client. This will trigger a retransmission of all segments that have higher sequence numbers that the sequence number acknowledged by the duplicate ACK. Since we assume that the client does not buffer out-of-order segments, this will not mean that we retransmit segments that has been received by the client. Also, if the client does buffer out-of-order segments, the retransmissions will waste bandwidth but will not be harmful to the operation of the TCP connection.

If any of the segments buffered in the proxy are retransmitted by the original sender of the data, those retransmissions will not be forwarded by the proxy.

Sending an uncontrolled burst of buffered segments might cause congestion if there are routers in the wireless network behind the proxy. Therefore, if the proxy is configured to operate in an environment with routers in the wireless network, the proxy uses the same congestion control mechanisms as for an ordinary TCP connection when transmitting the in-order segments. Since the in-order segment in a row of out-of-order segments most probably is the result of a time-out and retransmission, the proxy has not probed the wireless links for some time, and has no idea of the current congestion status in the wireless network. Therefore, the proxy always does a slow-start when transmitting the in-order segments.

Figure 3.4 shows how the reordering process works. In the top figure, the proxy has received three out-of-order segments (shown as shaded boxes) from the remote host which has just retransmitted the first in-order segment. The wireless client has buffered two previously received segments which have not yet been consumed by the application. The remote host has buffered the four segments which has not yet been acknowledged. The bottom figure shows the situation some short time later. Here, the proxy has already sent one of the buffered segments to the wireless client, and two more segments are in flight. The client has acknowledged the first segment and this segment is therefore no longer buffered in either the proxy or the sender. Due to slow start

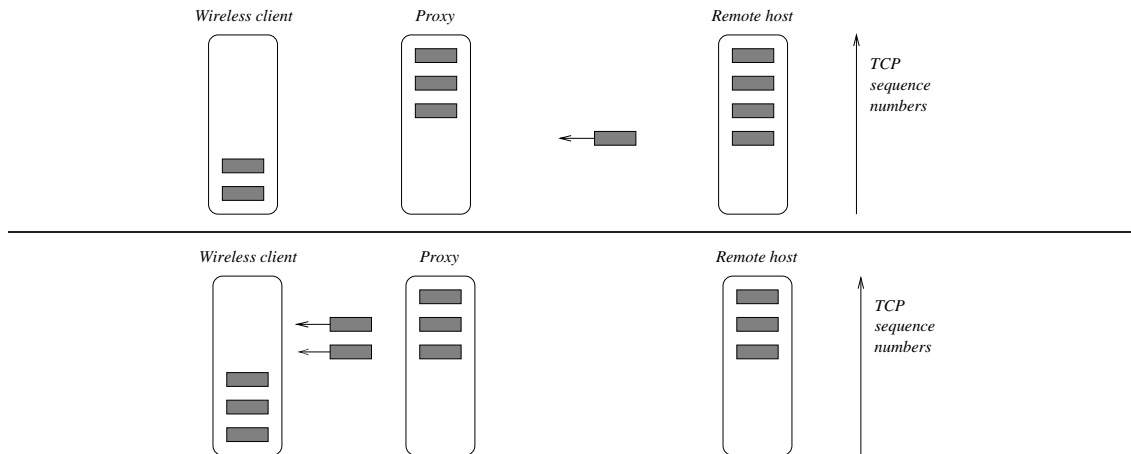


Figure 3.4. The proxy ordering segments

the proxy started with sending one segment and has now doubled its congestion window, therefore sending twice as many segments. Notice that even if the proxy has buffered the out-of-order segments, they have not yet been acknowledged to the sender, and therefore still are buffered in the sender.

3.3.3 Distributed state

Since the proxy captures all TCP segments to and from the wireless network, it is possible for the proxy to follow the state transitions made by the TCP in the wireless hosts. By allowing the proxy to handle TCP connections in certain states, the wireless hosts can be relieved of some burden. This pertains only to TCP states in which the wireless host does not send or receive any user data, i.e., states during the closing of a connection. In particular this pertains to the TIME-WAIT state in which the connection must linger for 2 times the maximum segment lifetime. This is typically configured to be between 30 and 120 seconds. This means that the total time in TIME-WAIT is between 1 and 4 minutes.

The TIME-WAIT state

The TIME-WAIT state is entered when an application issues an active close on a connection before the connection is closed by the peer. During the TIME-WAIT state any incoming segment is ACKed and dropped. The purpose of the TIME-WAIT state is to protect from delayed duplicate segments from the connection to interfere with new connections. As described in [Bra92], such interference can lead to de-synchronization of the new connection, failure of the new connection, or acceptance of delayed data from the old connection. During the lingering period in the TIME-WAIT state all old segments will die in the network.

To see how such a problem might occur, consider a connection c opened between hosts A and B (Figure 3.5). Some time after c has been closed, a new connection c' is opened. If a delayed segment from c with sequence and acknowledgment numbers fitting within the window of c' arrives this segment will be accepted by c' and there will be no way of knowing that this segment is erroneous.

The problem with TCP connections in TIME-WAIT is that they occupy an amount of memory and given the scarcity of memory in the wireless hosts, this can lead to new connections being rejected due to lack of resources for as long as four minutes. This is particularly severe for wireless hosts running an HTTP server, which does an active close on the connection when all HTTP data has been sent thus making the connection go into TIME-WAIT at the wireless host. Since HTTP

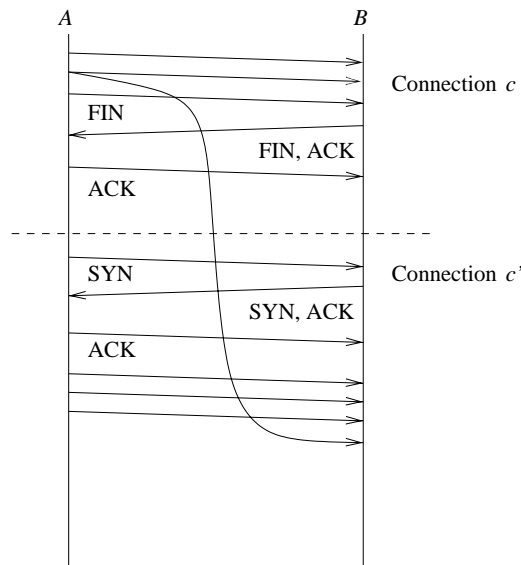


Figure 3.5. A delayed segment arriving in the wrong connection.

clients often open many simultaneous connections to the server, the memory consumed by the TIME-WAIT connections can be a significant amount. Also, since every TIME-WAIT connection occupy a PCB, the time for finding a PCB match when demultiplexing incoming packets will increase with the number of TIME-WAIT connections.

The naive approach to solving the TIME-WAIT problem is to shorten the time a connection is in TIME-WAIT. While this reduces memory costs, it can be dangerous due to reasons described above. Other approaches include keeping TIME-WAIT connections in a smaller data structure than other connections, to modify TCP so that the client keeps the connection in TIME-WAIT instead of the server [FTY99], or to modify HTTP so that the client does an active close before the server [FTY99].

While the above approaches are promising in a quite specialized case, none of them are directly applicable here. Keeping TIME-WAIT connections in a smaller data structure will still involve using valuable memory. Modifying TCP contradicts with the purpose of this work in that it produces a solution that do not match the standards, and more importantly requires changing TCP in every Internet host. Since a general solution is sought, modifying HTTP is not a plausible solution either.

The approach taken in this work is to let the proxy handle connections in TIME-WAIT on behalf of the wireless hosts. Here, the wireless hosts can remove the PCB and reclaim all memory associated with the connection when entering TIME-WAIT. The relative cost of keeping a TIME-WAIT connection in the proxy is very small compared to the cost of keeping it in the wireless host.

When the proxy sees that the wireless client has entered the TIME-WAIT state, it sends an RST to the client, which kills the connection in the client². The proxy then refrains from forwarding any TCP segments in that particular connection to the client.

Following state transitions

The proxy follows the state transitions made by the wireless host. This is done in a manner similar to how it is done in an end-host, but with a few modifications. The TCP state machine (Figure 2.7) cannot be used directly, since we are capturing TCP segments from both ends of the

²For this to work, the TCP implementation in the client must not have implemented the TIME-WAIT assassination work arounds suggested in [Bra92].

connection. Also, since packets may be lost on their way from the proxy to the wireless host there are some uncertainties with what state transitions that are actually made in the wireless host. For example, consider a connection running over the proxy in which the wireless host has closed the connection and is in FIN-WAIT-1, and the other host is in CLOSE-WAIT. When the wireless client receives a FINACK segment acknowledging the FIN it sent, it should enter the TIME-WAIT state (see Figure 2.7). Even if the proxy has seen the FIN segment, we cannot be sure that the wireless host has entered TIME-WAIT until we know that the FIN has been successfully received. Thus we cannot conclude that the wireless host is in TIME-WAIT until an acknowledgment for the FIN has arrived at the proxy.

The state diagram describing the state transitions in the proxy is seen in Figure 3.6. The abbreviation *c* stands for “the wireless client” and the abbreviation *h* stands for “the remote host”. The remote host is a host on the Internet. The notation $SYN + 1$ means “the next data byte in the sequence after the SYN segment”.

This state diagram is similar to the TCP state diagram in Figure 2.7, but with more states. Notice that there is no LISTEN state in Figure 3.6. This is because there is no way for the proxy to know that a connection has gone into LISTEN at the wireless host since no segments are sent when doing the transition from CLOSED to LISTEN.

Explanations for the states are as follows.

CLOSED No connection exists.

SYN-RCVD-1 The remote host has sent a SYN, but the wireless client has not responded.

SYN-RCVD-2 The wireless client has responded with a SYNACK to a SYN from the remote host.

SYN-RCVD-3 An ACK has been sent by the remote host for the SYNACK sent by the wireless client. It is uncertain whether the wireless client has entered ESTABLISHED or not.

SYN-SENT-1 The wireless client has sent a SYN.

SYN-SENT-2 The remote host has sent a SYNACK in response to the SYN, but it is uncertain whether the wireless client has entered ESTABLISHED or not.

ESTABLISHED The wireless client is known to have entered the ESTABLISHED state.

CLOSE-WAIT The remote host has sent a FIN.

FIN-WAIT-1 The wireless client has sent a FIN and is thus in FIN-WAIT-1.

FIN-WAIT-2 The remote host has acknowledged the FIN, but we do not know if the wireless client is in FIN-WAIT-1 or FIN-WAIT-2.

FIN-WAIT-3 The remote host has sent a FIN, but we do not know if the wireless client is in FIN-WAIT-2 or TIME-WAIT.

CLOSING-1 The remote host has sent a FIN but it is uncertain whether the wireless client is in FIN-WAIT-1 or in CLOSING.

CLOSING-2 The wireless client has acknowledged the FIN, and is in CLOSING.

TIME-WAIT The wireless client is known to be in TIME-WAIT.

Since the proxy does not prematurely acknowledge the SYN or FIN segments, the proxy will acknowledge segments from the wireless client only in the states SYN-RCVD-3, ESTABLISHED and CLOSE-WAIT. Segments from the remote host will be acknowledged in the states ESTABLISHED, FIN-WAIT-1 and FIN-WAIT-2.

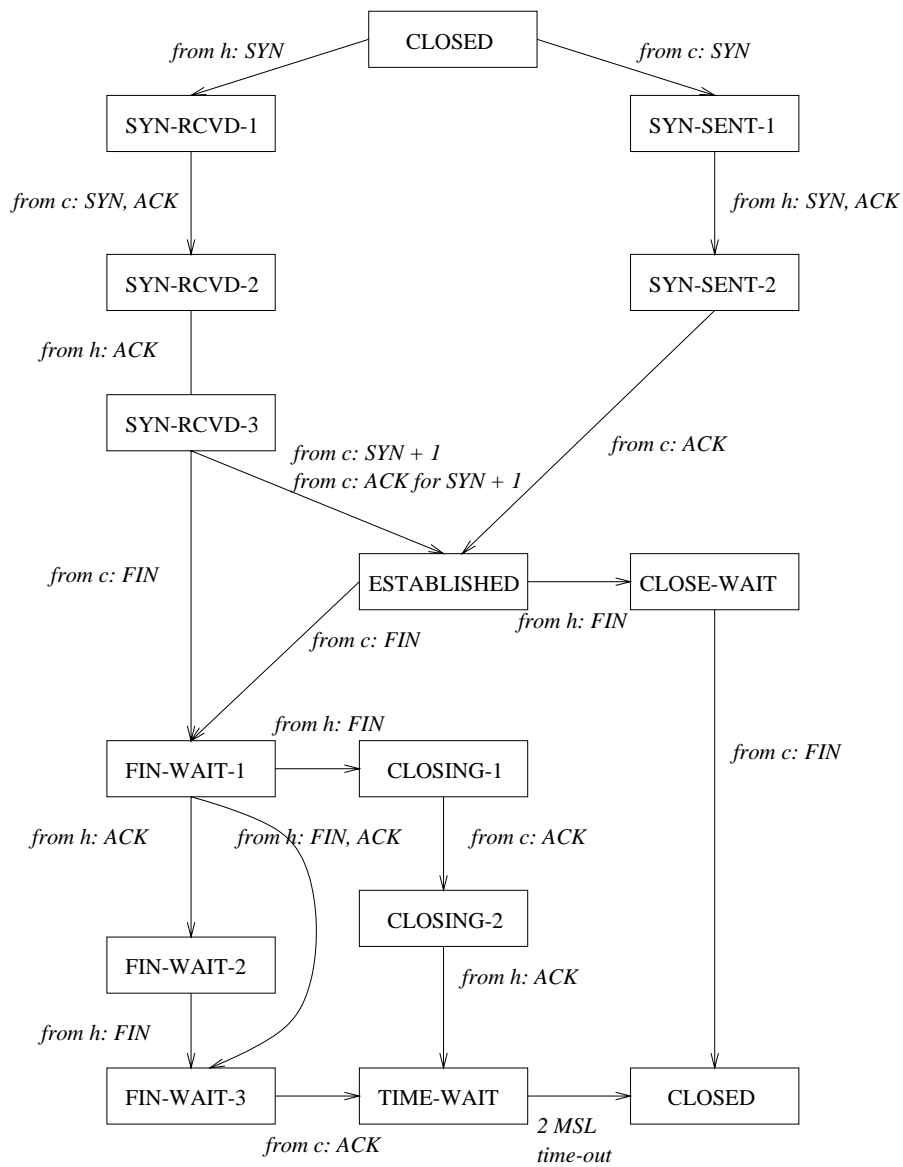


Figure 3.6. The TCP state machine in the proxy.

3.4 Alternative approaches

TCP has shown very poor performance over error prone wireless links and research has been done on how to improve wireless TCP performance. The indirect TCP, I-TCP [BB95], uses a split connection approach with a proxy scheme to improve TCP performance over error prone wireless links and also copes with long durations of disconnectivity. In the split connection approach the proxy terminates the connection and opens a separate connection with the other end host. M-TCP [BS97] is another example of the split connection approach.

Indeed, the mechanisms described in Sections 3.3.1 and 3.3.2 have many similarities with a split connection approach in that the proxy buffers and acknowledges TCP segments from the client. There are however a few significant differences that are worth pointing out. With the split connection approach:

- The end to end semantics are totally gone. In a split connection approach the proxy terminates the connection and thus acknowledges both the SYN and the FIN segments as well as all data segment.
- The asymmetry of the approach described here could not be exploited as easily. With the proxy scheme described here, out of sequence segments going from the wireless client to the remote Internet host are not queued but rather forwarded immediately. A split connection scheme would queue out of sequence segments from the wireless client until an in-sequence segment arrived.
- Soft state cannot be used for the connections in the proxy. A split connection approach needs to have state even for inactive connections.

3.5 Reliability

By buffering and acknowledging segments at the proxy, the clients are led to believe that the data has been successfully delivered, even though that may not be the case. If a PCB with cached TCP data times out, the data will be discarded. Since this data is not kept in any of the end hosts, discarding it would kill the connection. With a sufficiently long life time, however, a time out of a PCB with cached data means that either one of or both the end hosts has powered off, or a permanent network failure has occurred. In either case the connection has timed out at the end hosts, and cannot be used any longer.

Due to the fact that data is acknowledged at the proxy, a crash of the proxy would have a severe effect on all active TCP connections over the proxy, since any buffered data would be lost. It would not be a viable solution to save the cached data to stable storage, such as a hard disk, due to the enormous overhead involved. Instead, the proxy should be configured to refrain from caching data for those connections that require higher reliability. Such connections could be identified by the port numbers or IP addresses of the end-points. The filter could be implemented by adding a filter PCB in the PCB list, which would carry a flag indicating that the connection should not be processed by the proxy. Since the proxy already searches the PCB list for each incoming TCP segment, this solution would not add any complexity to the proxy. This solution is not general enough to be universally applicable however, since it can in some cases be hard to know in advance what port numbers that will be used for a connection (FTP data being an example of this). Finding a better solution for the reliability problem has been postponed as future work.

3.6 Proxy implementation

The proxy is in many cases similar to a fully functional TCP implementation. It has to be able to receive, transmit and retransmit segments as well as estimating round-trip times and doing congestion control. Since a fully functional TCP implementation has been developed within the

scope of this thesis (Section 4.8), much of the same code has been used in the implementation of the proxy.

The proxy has been implemented as a user processes running under FreeBSD 4.1. Implementing the proxy in user space rather than in the operating system kernel has numerous advantages:

- Development and debugging is easier.
- Deployment of the proxy is much easier since it does not involve rebuilding the kernel of the machine on which the proxy will run.
- Configuration and management of the proxy at runtime is easier.
- Failure of the proxy due to bugs in the code will not compromise the entire system.
- Porting the proxy to other systems than FreeBSD is much easier.

One disadvantage is that packets have to be copied multiple times; from kernel space to user space, and back again to kernel space, after having been processed by the proxy. Also, context has to be switched between the kernel and the proxy twice per packet. This substantially increases the delay of packets going through the proxy.

3.6.1 Interaction with the FreeBSD kernel

Incoming packets are handled by the FreeBSD kernel. They are routed from the network interfaces through the IP layer of the kernel to one of two tunnel interfaces, `tun0` and `tun1`, as can be seen in Figure 3.7. The tunnel interfaces act as any normal kernel network interface and are configured using the standard tool `ifconfig`. Packets forwarded over a tunnel interface can however, be read from a special device file `/dev/tunX`, where X is the number of the tunnel interface. Data written to the file is handled as a packet by the kernel, and can be forwarded to any other interface. Incoming packets are read from the device file.

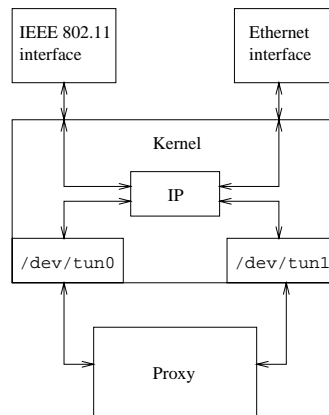


Figure 3.7. The proxy interacting with the kernel

Packets destined to the wireless network are forwarded to the tunnel interface `tun1`, and packets to the Internet, from the wireless network, are forwarded to `tun0`. When using tunnel interfaces to capture and send packets the network sniffer program `tcpdump` can easily be used to inspect traffic through the proxy.

Chapter 4

Design and implementation of the TCP/IP stack

The protocols in the TCP/IP suite are designed in a layered fashion, where each protocol layer solves a separate part of the communication problem. This layering can serve as a guide for designing the implementation of the protocols, in that each protocol can be implemented separately from the other. Implementing the protocols in a strictly layered way can however, lead to a situation where the communication overhead between the protocol layers degrades the overall performance [Cla82a]. To overcome these problems, certain internal aspects of a protocol can be made known to other protocols. Care must be taken so that only the important information is shared among the layers.

Most TCP/IP implementations keep a strict division between the application layer and the lower protocol layers, whereas the lower layers can be more or less interleaved. In most operating systems, the lower layer protocols are implemented as a part of the operating system kernel with entry points for communication with the application layer process. The application program is presented with an abstract view of the TCP/IP implementation, where network communication differs only very little from inter-process communication or file I/O. The implications of this is that since the application program is unaware of the buffer mechanisms used by the lower layers, it cannot utilize this information to, e.g., reuse buffers with frequently used data. Also, when the application sends data, this data has to be copied from the application process' memory space into internal buffers before being processed by the network code.

The operating systems used in minimal systems such as the target system of LWIP most often do not maintain a strict protection barrier between the kernel and the application processes. This allows using a more relaxed scheme for communication between the application and the lower layer protocols by the means of shared memory. In particular, the application layer can be made aware of the buffer handling mechanisms used by the lower layers. Therefore, the application can more efficiently reuse buffers. Also, since the application process can use the same memory as the networking code the application can read and write directly to the internal buffers, thus saving the expense of performing a copy.

4.1 Overview

As in many other TCP/IP implementations, the layered protocol design has served as a guide for the design of the implementation of LWIP. Each protocol is implemented as its own module, with a few functions acting as entry points into each protocol. Even though the protocols are implemented separately, some layer violations are made, as discussed above, in order to improve performance both in terms of processing speed and memory usage. For example, when verifying the checksum of an incoming TCP segment and when demultiplexing a segment, the source and destination IP addresses of the segment has to be known by the TCP module. Instead of passing

these addresses to TCP by the means of a function call, the TCP module is aware of the structure of the IP header, and can therefore extract this information by itself.

LWIP consists of several modules. Apart from the modules implementing the TCP/IP protocols (IP, ICMP, UDP, and TCP) a number of support modules are implemented. The support modules consists of the operating system emulation layer (described in Section 4.3), the buffer and memory management subsystems (described in Section 4.4), network interface functions (described in Section 4.5), and functions for computing the Internet checksum. LWIP also includes an abstract API, which is described in Section 4.10.

4.2 Process model

The process model of a protocol implementation describes in which way the system has been divided into different processes. One process model that has been used to implement communication protocols is to let each protocol run as a stand alone process. With this model, a strict protocol layering is enforced, and the communication points between the protocols must be strictly defined. While this approach has its advantages such as protocols can be added at runtime, understanding the code and debugging is generally easier, there are also disadvantages. The strict layering is not, as described earlier, always the best way to implement protocols. Also, and more important, for each layer crossed, a context switch must be made. For an incoming TCP segment this would mean three context switches, from the device driver for the network interface, to the IP process, to the TCP process and finally to the application process. In most operating systems a context switch is fairly expensive.

Another common approach is to let the communication protocols reside in the kernel of the operating system. In the case of a kernel implementation of the communication protocols, the application processes communicate with the protocols through system calls. The communication protocols are not strictly divided from each other but may use the techniques of crossing the protocol layering.

LWIP uses a process model in which all protocols reside in a single process and are thus separated from the operating system kernel. Application programs may either reside in the LWIP process, or be in separate processes. Communication between the TCP/IP stack and the application programs are done either by function calls for the case where the application program shares a process with LWIP, or by the means of a more abstract API.

Having LWIP implemented as a user space process rather than in the operating system kernel has both its advantages and disadvantages. The main advantage of having LWIP as a process is that is portable across different operating systems. Since LWIP is designed to run in small operating systems that generally do not support neither swapping out processes nor virtual memory, the delay caused by having to wait for disk activity if part of the LWIP process is swapped or paged out to disk will not be a problem. The problem of having to wait for a scheduling quantum before getting a chance to service requests still is a problem however, but there is nothing in the design of LWIP that precludes it from later being implemented in an operating system kernel.

4.3 The operating system emulation layer

In order to make LWIP portable, operating system specific function calls and data structures are not used directly in the code. Instead, when such functions are needed the operating system emulation layer is used. The operating system emulation layer provides a uniform interface to operating system services such as timers, process synchronization, and message passing mechanisms. In principle, when porting LWIP to other operating systems only an implementation of the operating system emulation layer for that particular operating system is needed.

The operating system emulation layer provides a timer functionality that is used by TCP. The timers provided by the operating system emulation layer are one-shot timers with a granularity of at least 200 ms that calls a registered function when the time-out occurs.

The only process synchronization mechanism provided is semaphores. Even if semaphores are not available in the underlying operating system they can be emulated by other synchronization primitives such as conditional variables or locks.

The message passing is done through a simple mechanism which uses an abstraction called *mailboxes*. A mailbox has two operations: post and fetch. The post operation will not block the process; rather, messages posted to a mailbox are queued by the operating system emulation layer until another process fetches them. Even if the underlying operating system does not have native support for the mailbox mechanism, they are easily implemented using semaphores.

4.4 Buffer and memory management

The memory and buffer management system in a communication system must be prepared to accommodate buffers of very varying sizes, ranging from buffers containing full-sized TCP segments with several hundred bytes worth of data to short ICMP echo replies consisting of only a few bytes. Also, in order to avoid copying it should be possible to let the data content of the buffers reside in memory that is not managed by the networking subsystem, such as application memory or ROM.

4.4.1 Packet buffers — pbufs

A pbuf is LWIP's internal representation of a packet, and is designed for the special needs of the minimal stack. Pbufs are similar to the mbufs used in the BSD implementations. The pbuf structure has support both for allocating dynamic memory to hold packet contents, and for letting packet data reside in static memory. Pbufs can be linked together in a list, called a pbuf chain so that a packet may span over several pbufs.

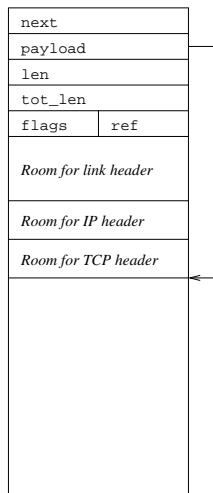


Figure 4.1. A PBUF_RAM pbuf with data in memory managed by the pbuf subsystem.

Pbufs are of three types, PBUF_RAM, PBUF_ROM, and PBUF_POOL. The pbuf shown in Figure 4.1 represents the PBUF_RAM type, and has the packet data stored in memory managed by the pbuf subsystem. The pbuf in Figure 4.2 is an example of a chained pbuf, where the first pbuf in the chain is of the PBUF_RAM type, and the second is of the PBUF_ROM type, which means that it has the data located in memory not managed by the pbuf system. The third type of pbuf, PBUF_POOL, is shown in Figure 4.3 and consists of fixed size pbufs allocated from a pool of fixed size pbufs. A pbuf chain may consist of multiple types of pbufs.

The three types have different uses. Pbufs of type PBUF_POOL are mainly used by network device drivers since the operation of allocating a single pbuf is fast and is therefore suitable for

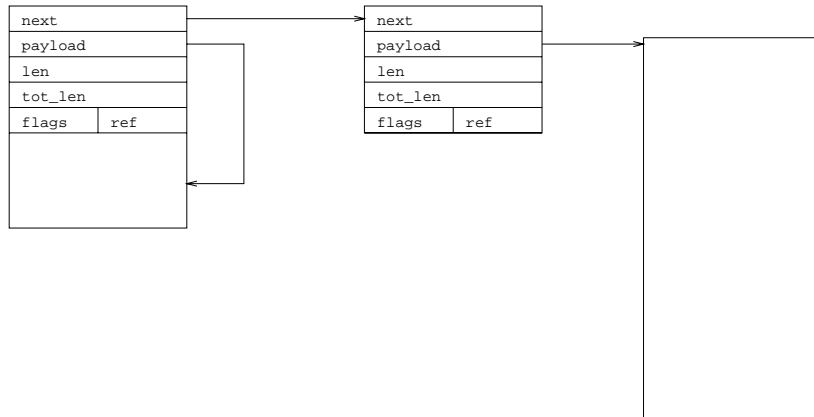


Figure 4.2. A PBUF_RAM pbuf chained with a PBUF_ROM pbuf that has data in external memory.

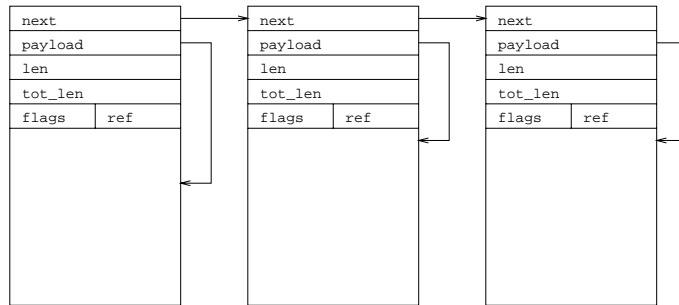


Figure 4.3. Chained PBUF_POOL pbufs from the pbuf pool.

use in an interrupt handler. PBUF_ROM pbufs are used when an application sends data that is located in memory managed by the application. This data may not be modified after the pbuf has been handed over to the TCP/IP stack and therefore this pbuf type main use is when the data is located in ROM (hence the name PBUF_ROM). Headers that are prepended to the data in a PBUF_ROM pbuf are stored in a PBUF_RAM pbuf that is chained to the front of the PBUF_ROM pbuf, as in Figure 4.2.

Pbufs of the PBUF_RAM type are also used when an application sends data that is dynamically generated. In this case, the pbuf system allocates memory not only for the application data, but also for the headers that will be prepended to the data. This is seen in Figure 4.1. The pbuf system cannot know in advance what headers will be prepended to the data and assumes the worst case. The size of the headers is configurable at compile time.

In essence, incoming pbufs are of type PBUF_POOL and outgoing pbufs are of the PBUF_ROM or PBUF_RAM types.

The internal structure of a pbuf can be seen in the Figures 4.1 through 4.3. The pbuf structure consists of two pointers, two length fields, a flags field, and a reference count. The `next` field is a pointer to the next pbuf in case of a pbuf chain. The `payload` pointer points to the start of the data in the pbuf. The `len` field contains the length of the data contents of the pbuf. The `tot_len` field contains the sum of the length of the current pbuf and all `len` fields of following pbufs in the pbuf chain. In other words, the `tot_len` field is the sum of the `len` field and the value of the `tot_len` field in the following pbuf in the pbuf chain. The `flags` field indicates the type of the pbuf and the `ref` field contains a reference count. The `next` and `payload` fields are native pointers and the size of those varies depending on the processor architecture used. The two length fields

are 16 bit unsigned integers and the `flags` and `ref` fields are 4 bit wide. The total size of the `pbuf` structure depends on the size of a pointer in the processor architecture being used and on the smallest alignment possible for the processor architecture. On an architecture with 32 bit pointers and 4 byte alignment, the total size is 16 bytes and on an architecture with 16 bit pointers and 1 byte alignment, the size is 9 bytes.

The `pbuf` module provides functions for manipulation of `pbufs`. Allocation of a `pbuf` is done by the function `pbuf_alloc()` which can allocate `pbufs` of any of the three types described above. The function `pbuf_ref()` increases the reference count. Deallocation is made by the function `pbuf_free()`, which first decreases the reference count of the `pbuf`. If the reference count reaches zero the `pbuf` is deallocated. The function `pbuf_realloc()` shrinks the `pbuf` so that it occupies just enough memory to cover the size of the data. The function `pbuf_header()` adjusts the `payload` pointer and the `length` fields so that a header can be prepended to the data in the `pbuf`. The functions `pbuf_chain()` and `pbuf_dechain()` are used for chaining `pbufs`.

4.4.2 Memory management

The memory manager supporting the `pbuf` scheme is very simple. It handles allocations and deallocations of contiguous regions of memory and can shrink the size of a previously allocated memory block. The memory manager uses a dedicated portion of the total memory in the system. This ensures that the networking system does not use all of the available memory, and that the operation of other programs is not disturbed if the networking system has used all of it's memory.

Internally, the memory manager keeps track of the allocated memory by placing a small structure on top of each allocated memory block. This structure (Figure 4.4) holds two pointers to the next and previous allocation block in memory. It also has a `used` flag which indicates whether the allocation block is allocated or not.

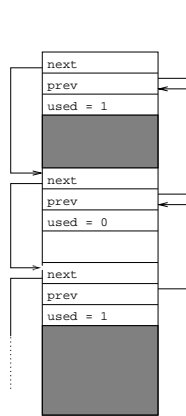


Figure 4.4. The memory allocation structure.

Memory is allocated by searching the memory for an unused allocation block that is large enough for the requested allocation. The first-fit principle is used so that the first block that is large enough is used. When an allocation block is deallocated, the `used` flag is set to zero. In order to prevent fragmentation, the `used` flag of the next and previous allocation blocks are checked. If any of them are unused, the blocks are combined into one larger unused block.

4.5 Network interfaces

In LWIP device drivers for physical network hardware are represented by a network interface structure similar to that in BSD. The network interface structure is shown in Figure 4.5. The

network interfaces are kept on a global linked list, which is linked by the `next` pointer in the structure.

```
struct netif {
    struct netif *next;
    char name[2];
    int num;
    struct ip_addr ip_addr;
    struct ip_addr netmask;
    struct ip_addr gw;
    void (*input)(struct pbuf *p, struct netif *inp);
    int (*output)(struct netif *netif, struct pbuf *p,
                 struct ip_addr *ipaddr);
    void *state;
};
```

Figure 4.5. The `netif` structure.

Each network interface has a name, stored in the `name` field in Figure 4.5. This two letter name identifies the kind of device driver used for the network interface and is only used when the interface is configured by a human operator at runtime. The name is set by the device driver and should reflect the kind of hardware that is represented by the network interface. For example, a network interface for a Bluetooth driver might have the name `bt` and a network interface for IEEE 802.11b WLAN hardware could have the name `wl`. Since the names not necessarily are unique, the `num` field is used to distinguish different network interfaces of the same kind.

The three IP addresses `ip_addr`, `netmask` and `gw` are used by the IP layer when sending and receiving packets, and their use is described in the next section. It is not possible to configure a network interface with more than one IP address. Rather, one network interface would have to be created for each IP address.

The `input` pointer points to the function the device driver should call when a packet has been received.

A network interface is connected to a device driver through the `output` pointer. This pointer points to a function in the device driver that transmits a packet on the physical network and it is called by the IP layer when a packet is to be sent. This field is filled by the initialization function of the device driver. The third argument to the `output` function, `ipaddr`, is the IP address of the host that should receive the actual link layer frame. It does not have to be the same as the destination address of the IP packet. In particular, when sending an IP packet to a host that is not on the local network, the link level frame will be sent to a router on the network. In this case, the IP address given to the `output` function will be the IP address of the router.

Finally, the `state` pointer points to device driver specific state for the network interface and is set by the device driver.

4.6 IP processing

LWIP implements only the most basic functionality of IP. It can send, receive and forward packets, but cannot send or receive fragmented IP packets nor handle packets with IP options. For most applications this does not pose any problems.

4.6.1 Receiving packets

For incoming IP packets, processing begins when the `ip_input()` function is called by a network device driver. Here, the initial sanity checking of the IP version field and the header length is

done, as well as computing and checking the header checksum. It is expected that the stack will not receive any IP fragments since the proxy described in Chapter 3 is assumed to reassemble any fragmented packets, thus any packet that is an IP fragment is silently discarded. Packets carrying IP options are also assumed to be handled by the proxy, and are dropped.

Next, the function checks the destination address with the IP addresses of the network interfaces to determine if the packet was destined for the host. The network interfaces are ordered in a linked list, and it is searched linearly. The number of network interfaces is expected to be small so a more sophisticated search strategy than a linear search has not been implemented.

If the incoming packet is found to be destined for this host, the protocol field is used to decide to which higher level protocol the packet should be passed to.

4.6.2 Sending packets

An outgoing packet is handled by the function `ip_output()`, which uses the function `ip_route()` to find the appropriate network interface to transmit the packet on. When the outgoing network interface is determined, the packet is passed to `ip_output_if()` which takes the outgoing network interface as an argument. Here, all IP header fields are filled in and the IP header checksum is computed. The source and destination addresses of the IP packet is passed as an argument to `ip_output_if()`. The source address may be left out, however, and in this case the IP address of the outgoing network interface is used as the source IP address of the packet.

The `ip_route()` function finds the appropriate network interface by linearly searching the list of network interfaces. During the search the destination IP address of the IP packet is masked with the netmask of the network interface. If the masked destination address is equal to the masked IP address of the interface, the interface is chosen. If no match is found, a default network interface is used. The default network interface is either pre-configured or configured at runtime by a human operator¹. If the network address of the default interface does not match the destination IP address, the `gw` field in the network interface structure (Figure 4.5) is chosen as the destination IP address of the link level frame. (Notice that the destination address of the IP packet and the destination address of the link level frame will be different in this case.) This primitive form of routing glosses over the fact that a network might have many routers attached to it. For the most basic case, where a local network only has one router, this works however.

Since the transport layer protocols UDP and TCP need to have the destination IP address when computing the transport layer checksum, the outgoing network interface must in some cases be determined before the packet is passed to the IP layer. This is done by letting the transport layer functions call the `ip_route()` function directly, and since the outgoing network interface is known already when the packet reaches the IP layer, there is no need to search the network interface list again. Instead, those protocols call the `ip_output_if()` function directly. Since this function takes a network interface as an argument, the search for an outgoing interface is avoided.

4.6.3 Forwarding packets

If none of the network interfaces has the same IP address as an incoming packet's destination address, the packet should be forwarded. This is done by the function `ip_forward()`. Here, the TTL field is decreased and if it reaches zero, an ICMP error message is sent to the original sender of the IP packet and the packet is discarded. Since the IP header is changed, the IP header checksum needs to be adjusted. There is no need to recompute the entire checksum, however, since simple arithmetic can be used to adjust the original IP checksum [MK90, Rij94]. Finally, the packet is forwarded to the appropriate network interface. The algorithm used to find the appropriate network interface is the same that is used when sending IP packets.

¹Human configuration of LWIP during runtime requires an application program that is able to configure the stack. Such a program is not included in LWIP.

4.6.4 ICMP processing

ICMP processing is fairly simple. ICMP packets received by `ip_input()` are handed over to `icmp_input()`, which decodes the ICMP header and takes the appropriate action. Some ICMP messages are passed to upper layer protocols and those are taken care of by special functions in the transport layer. ICMP destination unreachable messages can be sent by transport layer protocols, in particular by UDP, and the function `icmp_dest_unreach()` is used for this.

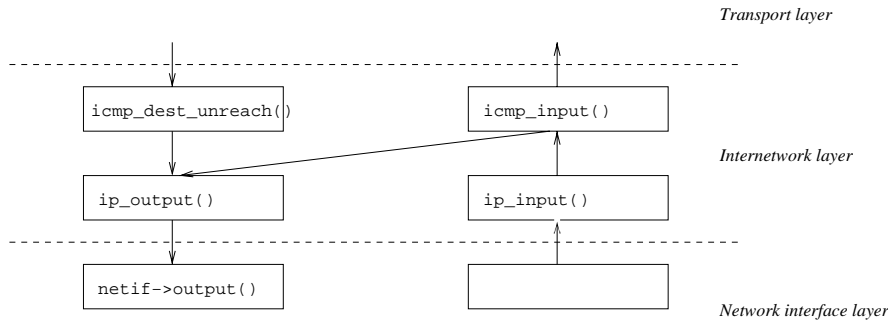


Figure 4.6. ICMP processing

Using ICMP ECHO messages to probe a network is widely used, and therefore ICMP echo processing is optimized for performance. The actual processing takes place in `icmp_input()`, and consists of swapping the IP destination and source addresses of the incoming packet, change the ICMP type to echo reply and adjust the ICMP checksum. The packet is then passed back to the IP layer for transmission.

4.7 UDP processing

UDP is a simple protocol used for demultiplexing packets between different processes. The state for each UDP session is kept in a Protocol Control Block, PCB, structure as shown in Figure 4.7. The UDP PCBs are kept on a linked list which is searched for a match when a UDP datagram arrives.

```

struct udp_pcb {
    struct udp_pcb *next;
    struct ip_addr local_ip, dest_ip;
    u16_t local_port, dest_port;
    u8_t flags;
    u16_t checksum_len;
    void (*recv)(void *arg, struct udp_pcb *pcb, struct pbuf *p);
    void *recv_arg;
};

```

Figure 4.7. The `udp_pcb` structure

The UDP PCB structure contains a pointer to the next PCB in the global linked list of UDP PCBs. 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` and `dest_port` fields. The `flags` field indicates what UDP checksum policy that should be used for this session. This can be either to switch UDP checksumming off completely, or to use UDP Lite [LDP99] in which the checksum

covers only parts of the datagram. If UDP Lite is used, the `chksum_len` field specifies how much of the datagram that should be checksummed.

The last two arguments, `recv` and `recv_arg`, are used when a datagram is received in the session specified by the PCB. The function pointed to by `recv` is called when a datagram is received.

Due to the simplicity of UDP, the input and output processing is equally simple and follows a fairly straight line (Figure 4.8). To send data, the application program calls `udp_send()` which calls upon `udp_output()`. Here the necessary checksumming is done and UDP header fields are filled. Since the checksum includes the IP source address of the IP packet, the function `ip_route()` is in some cases called to find the network interface to which the packet is to be transmitted. The outgoing network interface could be cached in the PCB, but this is currently not done. The IP address of this network interface is used as the source IP address of the packet. Finally, the packet is turned over to `ip_output_if()` for transmission.

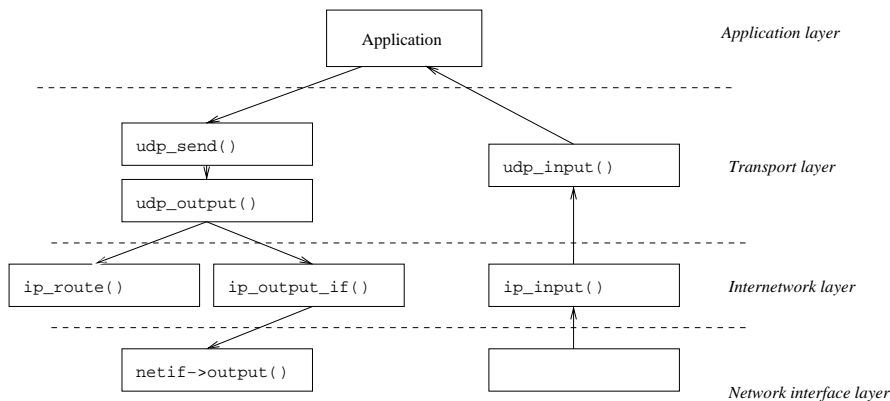


Figure 4.8. UDP processing

When a UDP datagram arrives, the IP layer calls the `udp_input()` function. Here, if checksumming should be used in the session, the UDP checksum is checked and the datagram is demultiplexed. When the corresponding UDP PCB is found, the `recv` function is called.

4.8 TCP processing

TCP is a transport layer protocol that provides a reliable byte stream service to the application layer. TCP is more complex than the other protocols described here, and the TCP code constitutes 50% of the total code size of LWIP.

4.8.1 Overview

The basic TCP processing (Figure 4.9) is divided into six functions; the functions `tcp_input()`, `tcp_process()`, and `tcp_receive()` which are related to TCP input processing, and `tcp_write()`, `tcp_enqueue()`, and `tcp_output()` which deals with output processing.

When an application wants to send TCP data, `tcp_write()` is called. The function `tcp_write()` passes control to `tcp_enqueue()` which will break the data into appropriate sized TCP segments if necessary and put the segments on the transmission queue for the connection. The function `tcp_output()` will then check if it is possible to send the data, i.e., if there is enough space in the receiver's window and if the congestion window is large enough and if so, sends the data using `ip_route()` and `ip_output_if()`.

Input processing begins when `ip_input()` after verifying the IP header hands over a TCP segment to `tcp_input()`. In this function the initial sanity checks (i.e., checksumming and TCP

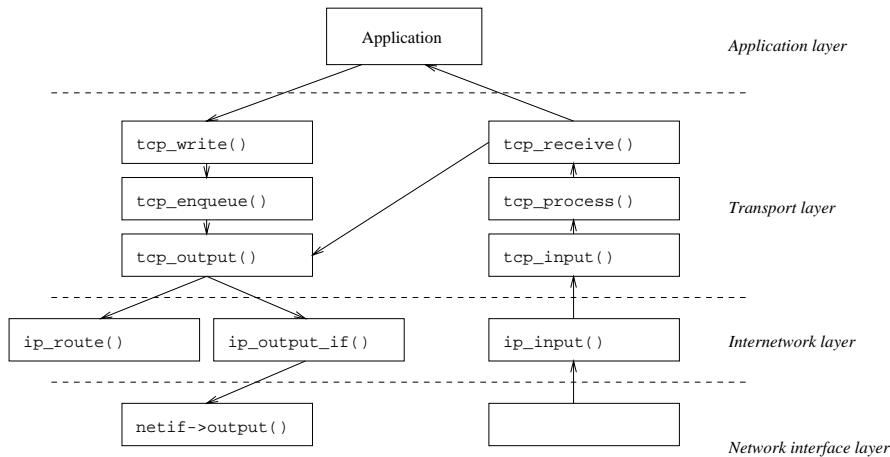


Figure 4.9. TCP processing

options parsing) are done as well as deciding to which TCP connection the segment belongs. The segment is then processed by `tcp_process()`, which implements the TCP state machine, and any necessary state transitions are made. The function `tcp_receive()` will be called if the connection is in a state to accept data from the network. If so, `tcp_receive()` will pass the segment up to an application program. If the segment constitutes an ACK for unacknowledged (thus previously buffered) data, the data is removed from the buffers and its memory is reclaimed. Also, if an ACK for data was received the receiver might be willing to accept more data and therefore `tcp_output()` is called.

4.8.2 Data structures

The data structures used in the implementation of TCP are kept small due to the memory constraints in the minimal system for which LWIP is intended. There is a tradeoff between the complexity of the data structures and the complexity of the code that uses the data structures, and in this case the code complexity has been sacrificed in order to keep the size of the data structures small.

The TCP PCB is fairly large and is shown in Figure 4.10. Since TCP connections in the LISTEN and TIME-WAIT states need to keep less state information than connections in other states, a smaller PCB data structure is used for those connections. This data structure is overlaid with the full PCB structure, and the ordering of the items in the PCB structure in Figure 4.10 is therefore somewhat awkward.

The TCP PCBs are kept on a linked list, and the `next` pointer links the PCB list together. The `state` variable contains the current TCP state (Figure 2.7) of the connection. Next, the IP addresses and port numbers which identify the connection are stored. The `mss` variable contains the maximum segment size allowed for the connection.

The `rcv_nxt` and `rcv_wnd` fields are used when receiving data. The `rcv_nxt` field contains the next sequence number expected from the remote end and is thus used when sending ACKs to the remote host. The receiver's window is kept in `rcv_wnd` and this is advertised in outgoing TCP segments. The field `tmr` is used as a timer for connections that should be removed after a certain amount of time, such as connections in the TIME-WAIT state. The maximum segment size allowed on the connection is stored in the `mss` field. The `flags` field contains additional state information of the connection, such as whether the connection is in fast recovery or if a delayed ACK should be sent.

```

struct tcp_pcb {
    struct tcp_pcb *next;
    enum tcp_state state;    /* TCP state */
    void (* accept)(void *arg, struct tcp_pcb *newpcb);
    void *accept_arg;
    struct ip_addr local_ip;
    u16_t local_port;
    struct ip_addr dest_ip;
    u16_t dest_port;
    u32_t rcv_nxt, rcv_wnd; /* receiver variables */
    u16_t tmr;
    u32_t mss;              /* maximum segment size */
    u8_t flags;
    u16_t rttest;          /* rtt estimation */
    u32_t rtseq;           /* sequence no for rtt estimation */
    s32_t sa, sv;          /* rtt average and variance */
    u32_t rto;             /* retransmission time-out */
    u32_t lastack;         /* last ACK received */
    u8_t dupacks;          /* number of duplicate ACKs */
    u32_t cwnd, u32_t ssthresh; /* congestion control variables */
    u32_t snd_ack, snd_nxt, /* sender variables */
        snd_wnd, snd_wl1, snd_wl2, snd_lbb;
    void (* recv)(void *arg, struct tcp_pcb *pcb, struct pbuf *p);
    void *recv_arg;
    struct tcp_seg *unsent, *unacked, /* queues */
        *ooseq;
};

```

Figure 4.10. The `tcp_pcb` structure

The fields `rttest`, `rtseq`, `sa`, and `sv` are used for the round-trip time estimation. The sequence number of the segment that is used for estimating the round-trip time is stored in `rtseq` and the time this segment was sent is stored in `rttest`. The average round-trip time and the round-trip time variance is stored in `sa` and `sv`. These variables are used when calculating the retransmission time-out which is stored in the `rto` field.

The two fields `lastack` and `dupacks` are used in the implementation of fast retransmit and fast recovery. The `lastack` field contains the sequence number acknowledged by the last ACK received and `dupacks` contains a count of how many ACKs that has been received for the sequence number in `lastack`. The current congestion window for the connection is stored in the `cwnd` field and the slow start threshold is kept in `ssthresh`.

The six fields `snd_ack`, `snd_nxt`, `snd_wnd`, `snd_wl1`, `snd_wl2` and `snd_lbb` are used when sending data. The highest sequence number acknowledged by the receiver is stored in `snd_ack` and the next sequence number to send is kept in `snd_nxt`. The receiver's advertised window is held in `snd_wnd` and the two fields `snd_wl1` and `snd_wl2` are used when updating `snd_wnd`. The `snd_lbb` field contains the sequence number of the last byte queued for transmission.

The function pointer `recv` and `recv_arg` are used when passing received data to the application layer. The three queues `unsent`, `unacked` and `ooseq` are used when sending and receiving data. Data that has been received from the application but has not been sent is queued in `unsent` and data that has been sent but not yet acknowledged by the remote host is held in `unacked`. Received data that is out of sequence is buffered in `ooseq`.

The `tcp_seg` structure in Figure 4.11 is the internal representation of a TCP segment. This structure starts with a `next` pointer which is used for linking when queuing segments. The `len`

```

struct tcp_seg {
    struct tcp_seg *next;
    u16_t len;
    struct pbuf *p;
    struct tcp_hdr *tcphdr;
    void *data;
    u16_t rtime;
};

```

Figure 4.11. The `tcp_seg` structure

field contains the length of the segment in TCP terms. This means that the `len` field for a data segment will contain the length of the data in the segment, and the `len` field for an empty segment with the SYN or FIN flags set will be 1. The pbuf `p` is the buffer containing the actual segment and the `tcphdr` and `data` pointers points to the TCP header and the data in the segment, respectively. For outgoing segments, the `rtime` field is used for the retransmission time-out of this segment. Since incoming segments will not need to be retransmitted, this field is not needed and memory for this field is not allocated for incoming segments.

4.8.3 Sequence number calculations

The TCP sequence numbers that are used to enumerate the bytes in the TCP byte stream are unsigned 32 bit quantities, hence in the range $[0, 2^{32} - 1]$. Since the number of bytes sent in a TCP connection might be more than the number of 32-bit combinations, the sequence numbers are calculated modulo 2^{32} . This means that ordinary comparison operators cannot be used with TCP sequence numbers. The modified comparison operators, called $<_{seq}$ and $>_{seq}$, are defined by the relations

$$s <_{seq} t \Leftrightarrow s - t < 0$$

and

$$s >_{seq} t \Leftrightarrow s - t > 0,$$

where s and t are TCP sequence numbers. The comparison operators for \leq and \geq are defined equivalently. The comparison operators are defined as C macros in the header file.

4.8.4 Queuing and transmitting data

Data that is to be sent is divided into appropriate sized chunks and given sequence numbers by the `tcp_enqueue()` function. Here, the data is packeted into pbufs and enclosed in a `tcp_seg` structure. The TCP header is built in the pbuf, and filled in with all fields except the acknowledgment number, `ackno`, and the advertised window, `wnd`. These fields can change during the queuing time of the segment and are therefore set by `tcp_output()` which does the actual transmission of the segment. After the segments are built, they are queued on the `unsent` list in the PCB. The `tcp_enqueue()` function tries to fill each segment with a maximum segment size worth of data and when an under-full segment is found at the end of the `unsent` queue, this segment is appended with the new data using the pbuf chaining functionality.

After `tcp_enqueue()` has formatted and queued the segments, `tcp_output()` is called. It checks if there is any room in the current window for any more data. The current window is computed by taking the maximum of the congestion window and the advertised receiver's window. Next, it fills in the fields of the TCP header that was not filled in by `tcp_enqueue()` and transmits the segment using `ip_route()` and `ip_output_if()`. After transmission the segment is put on the `unacked` list, on which it stays until an ACK for the segment has been received.

When a segment is on the `unacked` list, it is also timed for retransmission as described in Section 4.8.8. When a segment is retransmitted the TCP and IP headers of the original segment is kept and only very little changes has to be made to the TCP header. The `ackno` and `wnd` fields of the TCP header are set to the current values since we could have received data during the time between the original transmission of the segment and the retransmission. This changes only two 16-bit words in the header and the whole TCP checksum does not have to be recomputed since simple arithmetic [Rij94] can be used to update the checksum. The IP layer has already added the IP header when the segment was originally transmitted and there is no reason to change it. Thus a retransmission does not require any recomputation of the IP header checksum.

Silly window avoidance

The Silly Window Syndrome [Cla82b] (SWS) is a TCP phenomena that can lead to very bad performance. SWS occurs when a TCP receiver advertises a small window and the TCP sender immediately sends data to fill the window. When this small segment is acknowledged the window is opened again by a small amount and sender will again send a small segment to fill the window. This leads to a situation where the TCP stream consists of very small segments. In order to avoid SWS both the sender and the receiver must try to avoid this situation. The receiver must not advertise small window updates and the sender must not send small segments when only a small window is offered.

In LWIP SWS is naturally avoided at the sender since TCP segments are constructed and queued without knowledge of the advertised receiver's window. In a large transfer the output queue will consist of maximum sized segments. This means that if a TCP receiver advertises a small window, the sender will not send the first segment on the queue since it is larger than the advertised window. Instead, it will wait until the window is large enough for a maximum sized segment.

When acting as a TCP receiver, LWIP will not advertise a receiver's window that is smaller than the maximum segment size of the connection.

4.8.5 Receiving segments

Demultiplexing

When TCP segments arrive at the `tcp_input()` function, they are demultiplexed between the TCP PCBs. The demultiplexing key is the source and destination IP addresses and the TCP port numbers. There are two types of PCBs that must be distinguished when demultiplexing a segment; those that correspond to open connections and those that correspond to connections that are half open. Half open connections are those that are in the LISTEN state and only have the local TCP port number specified and optionally the local IP address, whereas open connections have the both IP addresses and both port numbers specified.

Many TCP implementations, such as the early BSD implementations, use a technique where a linked list of PCBs with a single entry cache is used. The rationale behind this is that most TCP connections constitute bulk transfers which typically show a large amount of locality [Mog92], resulting in a high cache hit ratio. Other caching schemes include keeping two one entry caches, one for the PCB corresponding to the last packet that was sent and one for the PCB of the last packet received [PP93]. An alternative scheme to exploit locality can be done by moving the most recently used PCB to the front of the list. Both methods have been shown [MD92] to outperform the one entry cache scheme.

In LWIP, whenever a PCB match is found when demultiplexing a segment, the PCB is moved to the front of the list of PCBs. PCBs for connections in the LISTEN state are not moved to the front however, since such connections are not expected to receive segments as often as connections that are in a state in which they receive data.

Receiving data

The actual processing of incoming segments is made in the function `tcp_receive()`. The acknowledgment number of the segment is compared with the segments on the `unacked` queue of the connection. If the acknowledgment number is higher than the sequence number of a segment on the `unacked` queue, that segment is removed from the queue and the allocated memory for the segment is deallocated.

An incoming segment is out of sequence if the sequence number of the segment is higher than the `rcv_nxt` variable in the PCB. Out of sequence segments are queued on the `ooseq` queue in the PCB. If the sequence number of the incoming segment is equal to `rcv_nxt`, the segment is delivered to the upper layer by calling the `recv` function in the PCB and `rcv_nxt` is increased by the length of the incoming segment. Since the reception of an in-sequence segment might mean that a previously received out of sequence segment now is the next segment expected, the `ooseq` queue is checked. If it contains a segment with sequence number equal to `rcv_nxt`, this segment is delivered to the application by a call to `recv` function and `rcv_nxt` is updated. This process continues until either the `ooseq` queue is empty or the next segment on `ooseq` is out of sequence.

If a supporting proxy is used, the proxy mechanism for ordering TCP segments described in Section 3.3.2 will lessen the need for the client to buffer out of sequence segments. Therefore, LWIP may be configured to refrain from buffering such segments.

4.8.6 Accepting new connections

TCP connections that are in the LISTEN state, i.e., that are passively open, are ready to accept new connections from a remote host. For those connections a new TCP PCB is created and must be passed to the application program that opened the initial listening TCP connection. In LWIP this is done by letting the application register a callback function that is to be called when a new connection has been established.

When a connection in the LISTEN state receives a TCP segment with the SYN flag set, a new connection is created and a segment with the SYN and ACK flags are sent in response to the SYN segment. The connection then enters the SYN-RCVD state and waits for an acknowledgment for the sent SYN segment. When the acknowledgment arrives, the connection enters the ESTABLISHED state, and the `accept` function (the `accept` field in the PCB structure in Figure 4.10) is called.

4.8.7 Fast retransmit

Fast retransmit and fast recovery is implemented in LWIP by keeping track of the last sequence number acknowledged. If another acknowledgment for the same sequence number is received, the `dupacks` counter in the TCP PCB is increased. When `dupacks` reaches three, the first segment on the `unacked` queue is retransmitted and fast recovery is initialized. The implementation of fast recovery follows the steps laid out in [APS99]. Whenever an ACK for new data is received, the `dupacks` counter is reset to zero.

4.8.8 Timers

As in the the BSD TCP implementation, LWIP uses two periodical timers that goes off every 200 ms and 500 ms. Those two timers are then used to implement more complex logical timers such as the retransmission timers, the TIME-WAIT timer and the delayed ACK timer.

The fine grained timer, `tcp_timer_fine()` goes through every TCP PCB checking if there are any delayed ACKs that should be sent, as indicated by the `flag` field in the `tcp_pcb` structure (Figure 4.10). If the delayed ACK flag is set, an empty TCP acknowledgment segment is sent and the flag is cleared.

The coarse grained timer, implemented in `tcp_timer_coarse()`, also scans the PCB list. For every PCB, the list of unacknowledged segments (the `unacked` pointer in the `tcp_seg` structure

in Figure 4.11), is traversed, and the `rttime` variable is increased. If `rttime` becomes larger than the current retransmission time-out as given by the `rto` variable in the PCB, the segment is retransmitted and the retransmission time-out is doubled. A segment is retransmitted only if allowed by the values of the congestion window and the advertised receiver's window. After retransmission, the congestion window is set to one maximum segment size, the slow start threshold is set to half of the effective window size, and slow start is initiated on the connection.

For connections that are in TIME-WAIT, the coarse grained timer also increases the `tmr` field in the PCB structure. When this timer reaches the $2 \times MSL$ threshold, the connection is removed.

The coarse grained timer also increases a global TCP clock, `tcp_ticks`. This clock is used for round-trip time estimation and retransmission time-outs.

4.8.9 Round-trip time estimation

The round-trip time estimation is a critical part of TCP since the estimated round-trip time is used when determining a suitable retransmission time-out. In LWIP round-trip times measurements are taken in a fashion similar to the BSD implementations. Round-trip times are measured once per round-trip and the smoothing function described in [Jac88] is used for the calculation of a suitable retransmission time-out.

The TCP PCB variable `rtseq` hold the sequence number of the segment for which the round-trip time is measured. The `rttest` variable in the PCB holds the value of `tcp_ticks` when the segment was first transmitted. When an ACK for a sequence number equal to or larger than `rtseq` is received, the round-trip time is measured by subtracting `rttest` from `tcp_ticks`. If a retransmission occurred during the round-trip time measurement, no measurement is taken.

4.8.10 Congestion control

The implementation of congestion control is surprisingly simple and consists of a few lines of code in the output and input code. When an ACK for new data is received the congestion window, `cwnd`, is increased either by one maximum segment size or by $mss^2/cwnd$, depending on whether the connection is in slow start or congestion avoidance. When sending data the minimum value of the receiver's advertised window and the congestion window is used to determine how much data that can be sent in each window.

4.9 Interfacing the stack

There are two ways for using the services provided by the TCP/IP stack; either by calling the functions in the TCP and UDP modules directly, or to use the LWIP API presented in the next section.

The TCP and UDP modules provide a rudimentary interface to the networking services. The interface is based on callbacks and an application program that uses the interface can therefore not operate in a sequential manner. This makes the application program harder to program and the application code is harder to understand. In order to receive data the application program registers a callback function with the stack. The callback function is associated with a particular connection and when a packet arrives on the connection, the callback function is called by the stack.

Furthermore, an application program that interfaces the TCP and UDP modules directly has to (at least partially) reside in the same process as the TCP/IP stack. This is due to the fact that a callback function cannot be called across a process boundary. This has both advantages and disadvantages. One advantage is that since the application program and the TCP/IP stack are in the same process, no context switching will be made when sending or receiving packets. The main disadvantage is that the application program cannot involve itself in any long running computations since TCP/IP processing cannot occur in parallel with the computation, thus degrading communication performance. This can be overcome by splitting the application into two

parts, one part dealing with the communication and one part dealing with the computation. The part doing the communication would then reside in the TCP/IP process and the computationally heavy part would be a separate process. The LWIP API presented in the next section provides a structured way to divide the application in such a way.

4.10 Application Program Interface

Due to the high level of abstraction provided by the BSD socket API, it is unsuitable for use in a minimal TCP/IP implementation. In particular, BSD sockets require data that is to be sent to be copied from the application program to internal buffers in the TCP/IP stack. The reason for copying the data is that the application and the TCP/IP stack usually reside in different protection domains. In most cases the application program is a user process and the TCP/IP stack resides in the operating system kernel. By avoiding the extra copy, the performance of the API can be greatly improved [ABM95]. Also, in order to make a copy, extra memory needs to be allocated for the copy, effectively doubling the amount of memory used per packet.

The LWIP API was designed for LWIP and utilizes knowledge of the internal structure of LWIP to achieve effectiveness. The LWIP API is very similar to the BSD API, but operates at a slightly lower level. The API does not require that data is copied between the application program and the TCP/IP stack, since the application program can manipulate the internal buffers directly.

Since the BSD socket API is well understood and many application programs have been written for it, it is advantageous to have a BSD socket compatibility layer. Appendix B presents the BSD socket functions rewritten using the LWIP API. A reference manual of the LWIP API is found in Appendix A.

4.10.1 Basic concepts

From the application's point of view, data handling in the BSD socket API is done in continuous memory regions. This is convenient for the application programmer since manipulation of data in application programs is usually done in such continuous memory chunks. Using this type of mechanism with LWIP would not be advantageous, since LWIP usually handles data in buffers where the data is partitioned into smaller chunks of memory. Thus the data would have to be copied into a continuous memory area before being passed to the application. This would waste both processing time and memory. Therefore, the LWIP API allows the application program to manipulate data directly in the partitioned buffers in order to avoid the extra copy.

The LWIP API uses a connection abstraction similar to that of the BSD socket API. There are very noticeable differences however; where an application program using the BSD socket API need not be aware of the distinction between an ordinary file and a network connection, an application program using the LWIP API has to be aware of the fact that it is using a network connection.

Network data is received in the form of buffers where the data is partitioned into smaller chunks of memory. Since many applications want to manipulate data in a continuous memory region, a convenience function for copying the data from a fragmented buffer to continuous memory exists.

Sending data is done differently depending on whether the data should 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. The TCP/IP stack will partition the data into appropriately sized packets and queue them for transmission. When sending UDP datagrams, the application program will have to explicitly allocate a buffer and fill it with data. The TCP/IP stack will send the datagram immediately when the output function is called.

4.10.2 Implementation of the API

The implementation of the API is divided into two parts, due to the process model of the TCP/IP stack. As shown in Figure 4.12, parts of the API is implemented as a library linked to the application program, and parts are implemented in the TCP/IP process. The two parts communicate

using the interprocess communication (IPC) mechanisms provided by the operating system emulation layer. The current implementation uses the following three IPC mechanisms:

- shared memory,
- message passing, and
- semaphores.

While these IPC types are supported by the operating system layer, they need not be directly supported by the underlying operating system. For operating systems that do not natively support them, the operating system emulation layer emulates them.

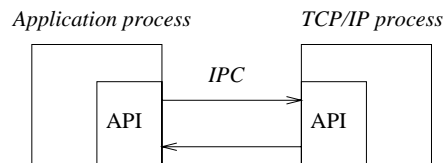


Figure 4.12. Division of the API implementation

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.

The buffer management is located in the library part of the API implementation. Buffers are created, copied and deallocated in the application process. Shared memory is used to pass the buffers between the application process and the TCP/IP process. The buffer data type used in communication with the application program is an abstraction of the `pbuf` data type.

Buffers carrying referenced memory, as opposed to allocated memory, is also passed using shared memory. For this to work, it has to be possible to share the referenced memory between the processes. The operating systems used in embedded systems for which LWIP is intended usually do not implement any form of memory protection, so this will not be a problem.

The functions that handle network connections are implemented in the part of the API implementation that resides in the TCP/IP process. The API functions in the part of the API that runs in the application process will pass a message using a simple communication protocol to the API implementation in the TCP/IP process. The message includes the type of operation that should be carried out and any arguments for the operation. The operation is carried out by the API implementation in the TCP/IP process and the return value is sent to the application process by message passing.

4.11 Statistical code analysis

This section analyzes the code of LWIP with respect to compiled object code size and number of lines in the source code. The code has been compiled for two processor architectures:

- The Intel Pentium III processor, henceforth referred to as the Intel x86 processor. The code was compiled with gcc 2.95.2 under FreeBSD 4.1 with compiler optimizations turned on.
- The 6502 processor [Nab, Zak83]. The code was compiled with cc65 2.5.5 [vB] with compiler optimizations turned on.

The Intel x86 has seven 32-bit registers and uses 32-bit pointers. The 6502, which main use today is in embedded systems, has one 8-bit accumulator as well as two 8-bit index registers and uses 16-bit pointers.

4.11.1 Lines of code

Table 4.1. Lines of code.

Module	Lines of code	Relative size
TCP	1076	42%
Support functions	554	21%
API	523	20%
IP	189	7%
UDP	149	6%
ICMP	87	3%
Total	2578	100%

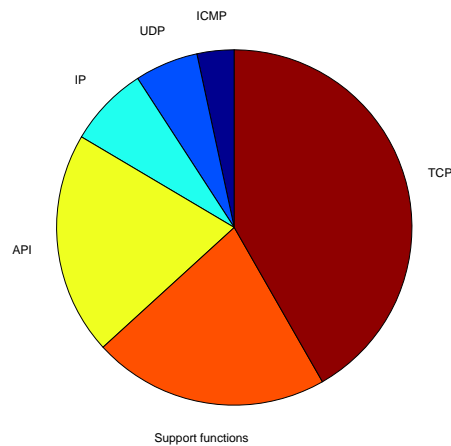


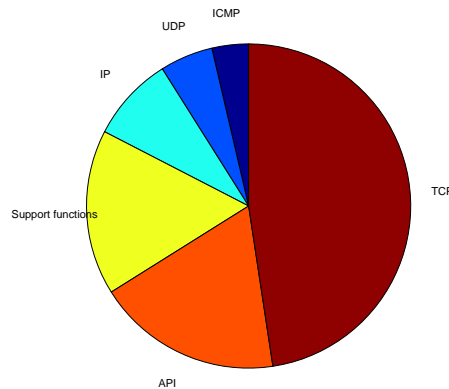
Figure 4.13. Lines of code.

Table 4.1 summarizes the number of lines of source code in LWIP and Figure 4.13 shows the relative number of lines of code. The category “Support functions” include buffer and memory management functions as well as the functions for computing the Internet checksum. The checksumming functions are generic C implementations of the algorithm that should be replaced with processor specific implementations when actually deployed. The category “API” includes both the part of the API that is linked with the applications and the part that is linked with the TCP/IP stack. The operating system emulation layer is not included in this analysis since its size varies heavily with the underlying operating system and is therefore not interesting to compare.

For the purpose of this comparison all comments and blank lines have been removed from the source files. Also, no header files were included in the comparison since those files mostly contain declarations that are repeated in the source code. We see that TCP is vastly larger than the other protocol implementations and that the API and the support functions taken together are as large as TCP.

Table 4.2. LWIP object code size when compiled for the Intel x86.

Module	Size (bytes)	Relative size
TCP	6584	48%
API	2556	18%
Support functions	2281	16%
IP	1173	8%
UDP	731	5%
ICMP	505	4%
Total	13830	100%

**Figure 4.14.** LWIP object code size when compiled for the x86.

4.11.2 Object code size

Table 4.2 summarizes the sizes of the compiled object code when compiled for the Intel x86 and Figure 4.14 shows the relative sizes. We see that the order of the items are somewhat different from Table 4.1. Here, the API is larger than the support functions category, even though the support functions has more lines of code. We also see that TCP constitutes 48% of the compiled code but only 42% of the total lines of code. Inspection of the assembler output from the TCP module shows a probable cause for this. The TCP module involve large amounts of pointer dereferencing, which expands into many lines of assembler code, thus increasing the object code size. Since many pointers are dereferenced two or three times in each function, this could be optimized by modifying the source code so that pointers are dereferenced only once and placed in a local variable. While this would reduce the size of the compiled code, it would also use more RAM as space for the local variables is allocated on the stack.

Table 4.3. LWIP object code size when compiled for the 6502.

Module	Size (bytes)	Relative size
TCP	11461	51%
Support functions	4149	18%
API	3847	17%
IP	1264	6%
UDP	1211	5%
ICMP	714	3%
Total	22646	100%

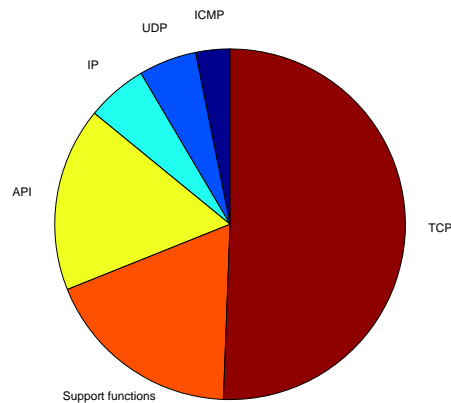


Figure 4.15. LWIP object code size when compiled for the 6502.

Table 4.3 shows the sizes of the object code when compiled for the 6502 and in Figure 4.14 the relative sizes are shown. We see that the TCP, the API, and the support functions are nearly twice as large as when compiled for the Intel x86, whereas IP, UDP and ICMP are approximately the same size. We also see that the support functions category is larger than the API, contrary to Table 4.2. The difference in size between the API and the support functions category is small though.

The reason for the increase in size of the TCP module is that the 6502 does not natively support 32-bit integers. Therefore, each 32-bit operation is expanded by the compiler into many lines of assembler code. TCP sequence numbers are 32-bit integers and the TCP module performs numerous sequence number computations.

The size of the TCP code can be compared to the size of TCP in other TCP/IP stacks, such as the popular BSD TCP/IP stack for FreeBSD 4.1 and the independently derived TCP/IP stack for Linux 2.2.10. Both are compiled for the Intel x86 with gcc and compiler optimizations turned on. The size of the TCP implementation in LWIP is almost 6600 bytes. The object code size of the TCP implementation in FreeBSD 4.1 is roughly 27000 bytes, which is four times as large as in LWIP. In Linux 2.2.10, the object code size of the TCP implementation is even larger and consists of 39000 bytes, roughly six times as much as in LWIP. The large difference in code size between LWIP and the two other implementations arise from the fact that both the FreeBSD and the Linux implementations contain more TCP features such as SACK [MMFR96] as well as parts of the implementation of the BSD socket API.

The reason for not comparing the sizes of the implementation of IP is that there is vastly more features in the IP implementations of FreeBSD and Linux. For instance, both FreeBSD and Linux includes support for firewalling and tunneling in their IP implementations. Also, those implementations support dynamic routing tables, which is not implemented in LWIP.

The LWIP API constitutes roughly one sixth of the size of LWIP. Since LWIP can be used without inclusion of the API, this part can be left out when deploying LWIP in a system with very little code memory.

4.12 Performance analysis

The performance of LWIP in terms of RAM usage or code efficiency have not been formally tested in this thesis, and this has been noted in future work. Simple tests have been conducted, however, and these have shown that LWIP running a simple HTTP/1.0 server is able to serve at least 10 simultaneous requests for web pages while consuming less than 4 kilobytes of RAM. In those tests, only the memory used by the protocols, buffering system, and the application program has been taken into account. Thus memory used by a device driver would add to the above figure.

Chapter 5

Summary

With this thesis, the following work has been done:

- The design and implementation of a small TCP/IP stack, LWIP, that uses very little RAM and that has a very small code footprint. The TCP/IP stack is written from scratch and has been designed with the restrictions of a minimal client system in mind.
- The design and implementation of an API for the LWIP stack that utilizes knowledge of the internal structure of the TCP/IP stack to reduce data copying.
- The design and implementation of a proxy based scheme for offloading a TCP implementation in a small client system. The scheme does not require any modifications of either the TCP sender or the TCP receiver thus making it possible to use the proxy with any TCP implementation.

5.1 The small TCP/IP stack

LWIP, the small TCP/IP stack, has been shown to have a small code size. It is designed for systems with little RAM and simple tests have shown that it can operate in an environment with very little RAM available. Operating system dependencies are moved into a separate module to simplify porting of LWIP to other operating systems.

The TCP/IP stack has support for TCP, UDP, ICMP, and IP. The TCP layer is fully functional and implements congestion control, round-trip time estimation, fast retransmit, and fast recovery. The IP layer has support for IP forwarding and can be used as a simple router.

5.2 The API

The LWIP API is very similar to the BSD socket API, but since the internal LWIP buffers are handled by the application copying can be reduced to a minimum. LWIP has been designed so that it is possible to run LWIP without the API presented here thus saving code memory.

5.3 The proxy scheme

The proxy scheme offloads the memory of the client system. The proxy is designed to work with TCP connections which require the client to buffer data, thus using memory in the client. The proxy scheme provides a way for the proxy to do some of the buffering on behalf of the client. The proxy uses the following three mechanisms to offload the client.

- Premature acknowledgment of TCP segments from the client.

- Reordering of TCP segments destined to the client.
- Distributed TCP state.

Care has been taken so that some of the end to end semantics of TCP are kept.

5.4 Future work

This thesis does not include any performance analysis of the proxy based scheme. Executing a performance analysis would be the natural next step of this work. For this, usage scenarios would have to be set up

It would also be interesting to integrate the proxy scheme designed in this thesis with a scheme for improving TCP throughput over wireless links such as I-TCP [BB95], M-TCP [BS97], or the Berkeley Snoop protocol [BSAK95].

The handling of fragmented IP packets that are larger than the link MTU discussed in Section 3.2.1 is not good since large IP packets always are dropped. A better solution might be to transmit large IP packets from the proxy to the client over a dedicated TCP connection. With this, the client does not have to be prepared to accept the full IP packet at once, and may use TCP flow control to limit the amount of large IP packets that is received.

More work needs to be done on the proxy reliability issue discussed in Section 3.5. As is stands, the suggested work-around in Section 3.5 can be very awkward in some situations.

The TCP implementation in LWIP needs to be tested in a more formal manner. Tools for conformance testing of TCP implementations exist [PS98] and those could be used to test the implementation. Furthermore, performance testing of the LWIP with respect to memory consumption and execution time should be conducted.

Appendix A

API reference

A.1 Data types

There are two data types that are used for the LWIP API. These are

- `netbuf`, the network buffer abstraction, and
- `netconn`, the abstraction of a network connection.

Each data type is represented as a pointer to a C `struct`. Knowledge of the internal structure of the `struct` should not be used in application programs. Instead, the API provides functions for modifying and extracting necessary fields.

A.1.1 Netbufs

Netbufs are buffers that are used for sending and receiving data. Internally, a netbuf is associated with a pbuf as presented in Section 4.4.1. Netbufs can, just as pbufs, accomodate both allocated memory and referenced memory. Allocated memory is RAM that is explicitly allocated for holding network data, whereas referenced memory might be either application managed RAM or external ROM. Referenced memory is useful for sending data that is not modified, such as static web pages or images.

The data in a netbuf can be fragmented into differently sized blocks. This means that an application must be prepared to accept fragmented data. Internally, a netbuf has a pointer to one of the fragments in the netbuf. Two functions, `netbuf_next()` and `netbuf_first()` are used to manipulate this pointer.

Netbufs that have been received from the network also contain the IP address and port number of the originator of the packet. Functions for extracting those values exist.

A.2 Buffer functions

`netbuf_new()`

Synopsis

```
struct netbuf * netbuf_new(void)
```

Description

Allocates a netbuf structure. No buffer space is allocated when doing this, only the top level structure. After use, the netbuf must be deallocated with `netbuf_delete()`.

netbuf_delete()

Synopsis

```
void netbuf_delete(struct netbuf *)
```

Description

Deallocates a netbuf structure previously allocated by a call to the `netbuf_new()` function. Any buffer memory allocated to the netbuf by calls to `netbuf_alloc()` is also deallocated.

Example

This example shows the basic mechanisms for using netbufs.

```
int
main()
{
    struct netbuf *buf;

    buf = netbuf_new();    /* create a new netbuf */
    netbuf_alloc(buf, 100); /* allocate 100 bytes of buffer */

    /* do something with the netbuf */
    /* [...] */

    netbuf_delete(buf);   /* deallocate netbuf */
}
```

netbuf_alloc()

Synopsis

```
void *netbuf_alloc(struct netbuf *buf, int size)
```

Description

Allocates buffer memory with `size` number of bytes for the netbuf `buf`. The function returns a pointer to the allocated memory. Any memory previously allocated to the netbuf `buf` is deallocated. The allocated memory can later be deallocated with the `netbuf_free()` function. Since protocol headers are expected to precede the data when it should be sent, the function allocates memory for protocol headers as well as for the actual data.

netbuf_free()

Synopsis

```
int netbuf_free(struct netbuf *buf)
```

Description

Deallocates the buffer memory associated with the netbuf `buf`. If no buffer memory has been allocated for the netbuf, this function does nothing.

netbuf_ref()

Synopsis

```
int netbuf_ref(struct netbuf *buf, void *data, int size)
```

Description

Associates the external memory pointer to by the `data` pointer with the netbuf `buf`. The size of the external memory is given by `size`. Any memory previously allocated to the netbuf is deallocated. The difference between allocating memory for the netbuf with `netbuf_alloc()` and allocating memory using, e.g., `malloc()` and referencing it with `netbuf_ref()` is that in the former case, space for protocol headers is allocated as well which makes processing and sending the buffer faster.

Example

This example shows a simple use of the `netbuf_ref()` function.

```
int
main()
{
    struct netbuf *buf;
    char string[] = "A string";

    /* create a new netbuf */
    buf = netbuf_new();

    /* refernce the string */
    netbuf_ref(buf, string, sizeof(string));

    /* do something with the netbuf */
    /* [...] */

    /* deallocate netbuf */
    netbuf_delete(buf);
}
```

netbuf_len()

Synopsis

```
int netbuf_len(struct netbuf *buf)
```

Description

Returns the total length of the data in the netbuf `buf`, even if the netbuf is fragmented. For a fragmented netbuf, the value obtained by calling this function is not the same as the size of the first fragment in the netbuf.

netbuf_data()

Synopsis

```
int netbuf_data(struct netbuf *buf, void **data, int *len)
```

Description

This function is used to obtain a pointer to and the length of a block of data in the netbuf `buf`. The arguments `data` and `len` are result parameters that will be filled with a pointer to the data and the length of the data pointed to. If the netbuf is fragmented, this function gives a pointer to one of the fragments in the netbuf. The application program must use the fragment handling functions `netbuf_first()` and `netbuf_next()` in order to reach all data in the netbuf.

See the example under `netbuf_next()` for an example of how use `netbuf_data()`.

`netbuf_next()`

Synopsis

```
int netbuf_next(struct netbuf *buf)
```

Description

This function updates the internal fragment pointer in the netbuf `buf` so that it points to the next fragment in the netbuf. The return value is zero if there are more fragments in the netbuf, > 0 if the fragment pointer now points to the last fragment in the netbuf, and < 0 if the fragment pointer already pointed to the last fragment.

Example

This example shows how to use the `netbuf_next()` function. We assume that this is in the middle of a function and that the variable `buf` is a netbuf.

```
/* [...] */
do {
    char *data;
    int len;

    /* obtain a pointer to the data in the fragment */
    netbuf_data(buf, &data, &len);

    /* do something with the data */
    do_something(data, len);
} while(netbuf_next(buf) >= 0);
/* [...] */
```

`netbuf_first()`

Synopsis

```
void netbuf_first(struct netbuf *buf)
```

Description

Resets the fragment pointer in the netbuf `buf` so that it points to the first fragment.

`netbuf_copy()`

Synopsis

```
void netbuf_copy(struct netbuf *buf, void *data, int len)
```

Description

Copies all of the data from the netbuf `buf` into the memory pointed to by `data` even if the netbuf `buf` is fragmented. The `len` parameter is an upper bound of how much data that will be copied into the memory pointed to by `data`.

Example

This example shows a simple use of `netbuf_copy()`. Here, 200 bytes of memory is allocated on the stack to hold data. Even if the netbuf `buf` has more data than 200 bytes, only 200 bytes are copied into `data`.

```
void
example_function(struct netbuf *buf)
{
    char data[200];
    netbuf_copy(buf, data, 200);

    /* do something with the data */
}
```

netbuf_chain()**Synopsis**

```
void netbuf_chain(struct netbuf *head, struct netbuf *tail)
```

Description

Chains the two netbufs `head` and `tail` together so that the data in `tail` will become the last fragment(s) in `head`. The netbuf `tail` is deallocated and should not be used after the call to this function.

netbuf_fromaddr()**Synopsis**

```
struct ip_addr *netbuf_fromaddr(struct netbuf *buf)
```

Description

Returns the IP address of the host the netbuf `buf` was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromport()` can be used to obtain the port number of the remote host.

netbuf_fromport()**Synopsis**

```
unsigned short netbuf_fromport(struct netbuf *buf)
```

Description

Returns the port number of the host the netbuf `buf` was received from. If the netbuf has not been received from the network, the return value of this function is undefined. The function `netbuf_fromaddr()` can be used to obtain the IP address of the remote host.

A.3 Network connection functions

netconn_new()

Synopsis

```
struct netconn * netconn_new(enum netconn_type type)
```

Description

Creates a new connection abstraction structure. The argument can be one of `NETCONN_TCP` or `NETCONN_UDP`, yielding either a TCP or a UDP connection. No connection is established by the call to this function and no data is sent over the network.

netconn_delete()

Synopsis

```
void netconn_delete(struct netconn *conn)
```

Description

Deallocates the netconn `conn`. If the connection is open, it is closed as a result of this call.

netconn_type()

Synopsis

```
enum netconn_type netconn_type(struct netconn *conn)
```

Description

Returns the type of the connection `conn`. This is the same type that is given as an argument to `netconn_new()` and can be either `NETCONN_TCP` or `NETCONN_UDP`.

netconn_peer()

Synopsis

```
int netconn_peer(struct netconn *conn,  
struct ip_addr **addr, unsigned short *port)
```

Description

The function `netconn_peer()` is used to obtain the IP address and port of the remote end of a connection. The parameters `addr` and `port` are result parameters that are set by the function. If the connection `conn` is not connected to any remote host, the results are undefined.

netconn_addr()

Synopsis

```
int netconn_addr(struct netconn *conn,  
struct ip_addr **addr, unsigned short *port)
```

Description

This function is used to obtain the local IP address and port number of the connection `conn`.

netconn_bind()

Synopsis

```
int netconn_bind(struct netconn *conn,
                struct ip_addr *addr, unsigned short port)
```

Description

Binds the connection `conn` to the local IP address `addr` and TCP or UDP port `port`. If `addr` is `NULL` the local IP address is determined by the networking system.

netconn_connect()

Synopsis

```
int netconn_connect(struct netconn *conn,
                   struct ip_addr *remote_addr, unsigned short remote_port)
```

Description

In case of UDP, sets the remote receiver as given by `remote_addr` and `remote_port` of UDP messages sent over the connection. For TCP, `netconn_connect()` opens a connection with the remote host.

netconn_listen()

Synopsis

```
int netconn_listen(struct netconn *conn)
```

Description

Puts the TCP connection `conn` into the TCP LISTEN state.

netconn_accept()

Synopsis

```
struct netconn * netconn_accept(struct netconn *conn)
```

Description

Blocks the process until a connection request from a remote host arrives on the TCP connection `conn`. The connection must be in the LISTEN state so `netconn_listen()` must be called prior to `netconn_accept()`. When a connection is established with the remote host, a new connection structure is returned.

Example

This example shows how to open a TCP server on port 2000.

```
int
main()
{
    struct netconn *conn, *newconn;
```

```

/* create a connection structure */
conn = netconn_new(NETCONN_TCP);

/* bind the connection to port 2000 on any local
   IP address */
netconn_bind(conn, NULL, 2000);

/* tell the connection to listen for incoming
   connection requests */
netconn_listen(conn);

/* block until we get an incoming connection */
newconn = netconn_accept(conn);

/* do something with the connection */
process_connection(newconn);

/* deallocate both connections */
netconn_delete(newconn);
netconn_delete(conn);
}

```

netconn_recv()

Synopsis

```
struct netbuf * netconn_recv(struct netconn *conn)
```

Description

Blocks the process while waiting for data to arrive on the connection `conn`. If the connection has been closed by the remote host, `NULL` is returned, otherwise a `netbuf` containing the received data is returned.

Example

This is a small example that shows a suggested use of the `netconn_recv()` function. We assume that a connection has been established before the call to `example_function()`.

```

void
example_function(struct netconn *conn)
{
    struct netbuf *buf;

    /* receive data until the other host closes
       the connection */
    while((buf = netconn_recv(conn)) != NULL) {
        do_something(buf);
    }

    /* the connection has now been closed by the
       other end, so we close our end */
    netconn_close(conn);
}

```

netconn_write()

Synopsis

```
int netconn_write(struct netconn *conn, void *data,
int len, unsigned int flags)
```

Description

This function is only used for TCP connections. It puts the data pointed to by `data` on the output queue for the TCP connection `conn`. The length of the data is given by `len`. There is no restriction on the length of the data. This function does not require the application to explicitly allocate buffers, as this is taken care of by the stack. The `flags` parameter has two possible states, as shown below.

```
#define NETCONN_NOCOPY 0x00
#define NETCONN_COPY 0x01
```

When passed the flag `NETCONN_COPY` the data is copied into internal buffers which is allocated for the data. This allows the data to be modified directly after the call, but is inefficient both in terms of execution time and memory usage. If the flag `NETCONN_NOCOPY` is used, the data is not copied but rather referenced. The data must not be modified after the call, since the data can be put on the retransmission queue for the connection, and stay there for an indeterminate amount of time. This is useful when sending data that is located in ROM and therefore is immutable.

If greater control over the modifiability of the data is needed, a combination of copied and non-copied data can be used, as seen in the example below.

Example

This example shows the basic usage of `netconn_write()`. Here, the variable `data` is assumed to be modified later in the program, and is therefore copied into the internal buffers by passing the flag `NETCONN_COPY` to `netconn_write()`. The `text` variable contains a string that will not be modified and can therefore be sent using references instead of copying.

```
int
main()
{
    struct netconn *conn;
    char data[10];
    char text[] = "Static text";
    int i;

    /* set up the connection conn */
    /* [...] */

    /* create some arbitrary data */
    for(i = 0; i < 10; i++)
        data[i] = i;

    netconn_write(conn, data, 10, NETCONN_COPY);
    netconn_write(conn, text, sizeof(text), NETCONN_NOCOPY);

    /* the data can be modified */
    for(i = 0; i < 10; i++)
        data[i] = 10 - i;
```

```
/* take down the connection conn */
netconn_close(conn);
}
```

netconn_send()

Synopsis

```
int netconn_send(struct netconn *conn, struct netbuf *buf)
```

Description

Send the data in the netbuf `buf` on the UDP connection `conn`. The data in the netbuf should not be too large since IP fragmentation is not used. The data should not be larger than the maximum transmission unit (MTU) of the outgoing network interface. Since there currently is no way of obtaining this value a careful approach should be taken, and the netbuf should not contain data that is larger than some 1000 bytes.

No checking is made whether the data is sufficiently small and sending very large netbufs might give undefined results.

Example

This example shows how to send some UDP data to UDP port 7000 on a remote host with IP address 10.0.0.1.

```
int
main()
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr addr;
    char *data;
    char text[] = "A static text";
    int i;

    /* create a new connection */
    conn = netconn_new(NETCONN_UDP);

    /* set up the IP address of the remote host */
    addr.addr = htonl(0x0a000001);

    /* connect the connection to the remote host */
    netconn_connect(conn, &addr, 7000);

    /* create a new netbuf */
    buf = netbuf_new();
    data = netbuf_alloc(buf, 10);

    /* create some arbitrary data */
    for(i = 0; i < 10; i++)
        data[i] = i;

    /* send the arbitrary data */
    netconn_send(conn, buf);
}
```

```
/* reference the text into the netbuf */
netbuf_ref(buf, text, sizeof(text));

/* send the text */
netconn_send(conn, buf);

/* deallocate connection and netbuf */
netconn_delete(conn);
netconn_delete(buf);
}
```

netconn_close()

Synopsis

```
int netconn_close(struct netconn *conn)
```

Description

Closes the connection `conn`.

Appendix B

BSD socket library

This appendix provides a simple implementation of the BSD socket API using the LWIP API. The implementation is provided as a reference only, and is not intended for use in actual programs. There is for example no error handling.

Also, this implementation does not support the `select()` and `poll()` functions of the BSD socket API since the LWIP does not have any functions that can be used to implement those. In order to implement those functions, the BSD socket implementation would have to communicate directly with the LWIP stack and not use the API.

B.1 The representation of a socket

In the BSD socket API sockets are represented as ordinary file descriptors. File descriptors are integers that uniquely identifies the file or network connection. In this implementation of the BSD socket API, sockets are internally represented by a `netconn` structure. Since BSD sockets are identified by an integer, the `netconn` variables are kept in an array, `sockets[]`, where the BSD socket identifier is the index into the array.

B.2 Allocating a socket

B.2.1 The `socket()` call

The `socket()` call allocates a BSD socket. The parameters to `socket()` are used to specify what type of socket that is requested. Since this socket API implementation is concerned only with network sockets, these are the only socket type that is supported. Also, only UDP (`SOCK_DGRAM`) or TCP (`SOCK_STREAM`) sockets can be used.

```
int
socket(int domain, int type, int protocol)
{
    struct netconn *conn;
    int i;

    /* create a netconn */
    switch(type) {
    case SOCK_DGRAM:
        conn = netconn_new(NETCONN_UDP);
        break;
    case SOCK_STREAM:
        conn = netconn_new(NETCONN_TCP);
```

```
    break;
}

/* find an empty place in the sockets[] list */
for(i = 0; i < sizeof(sockets); i++) {
    if(sockets[i] == NULL) {
        sockets[i] = conn;
        return i;
    }
}
return -1;
}
```

B.3 Connection setup

The BSD socket API calls for setting up a connection are very similar to the connection setup functions of the minimal API. The implementation of these calls mainly include translation from the integer representation of a socket to the connection abstraction used in the minimal API.

B.3.1 The `bind()` call

The `bind()` call binds the BSD socket to a local address. In the call to `bind()` the local IP address and port number are specified. The `bind()` function is very similar to the `netconn_bind()` function in the LWIP API.

```
int
bind(int s, struct sockaddr *name, int namelen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr;
    unsigned short remote_port;

    remote_addr = (struct ip_addr *)name->sin_addr;
    remote_port = name->sin_port;

    conn = sockets[s];
    netconn_bind(conn, remote_addr, remote_port);

    return 0;
}
```

B.3.2 The `connect()` call

The implementation of `connect()` is as straightforward as that of `bind()`.

```
int
connect(int s, struct sockaddr *name, int namelen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr;
    unsigned short remote_port;
```

```

remote_addr = (struct ip_addr *)name->sin_addr;
remote_port = name->sin_port;

conn = sockets[s];
netconn_connect(conn, remote_addr, remote_port);

return 0;
}

```

B.3.3 The listen() call

The `listen()` call is the equivalent of the LWIP API function `netconn_listen()` and can only be used for TCP connections. The only difference is that the BSD socket API allows the application to specify the size of the queue of pending connections (the backlog). This is not possible with LWIP and the `backlog` parameter is ignored.

```

int
listen(int s, int backlog)
{
    netconn_listen(sockets[s]);
    return 0;
}

```

B.3.4 The accept() call

The `accept()` call is used to wait for incoming connections on a TCP socket that previously has been set into LISTEN state by a call to `listen()`. The call to `accept()` blocks until a connection has been established with a remote host. The arguments to `listen` are result parameters that are set by the call to `accept()`. These are filled with the address of the remote host.

When the new connection has been established, the LWIP function `netconn_accept()` will return the connection handle for the new connection. After the IP address and port number of the remote host has been filled in, a new socket identifier is allocated and returned.

```

int
accept(int s, struct sockaddr *addr, int *addrlen)
{
    struct netconn *conn, *newconn;
    struct ip_addr *addr;
    unsigned short port;
    int i;

    conn = sockets[s];

    newconn = netconn_accept(conn);

    /* get the IP address and port of the remote host */
    netconn_peer(conn, &addr, &port);

    addr->sin_addr = *addr;
    addr->sin_port = port;

    /* allocate a new socket identifier */
}

```

```

for(i = 0; i < sizeof(sockets); i++) {
    if(sockets[i] == NULL) {
        sockets[i] = newconn;
        return i;
    }
}

return -1;
}

```

B.4 Sending and receiving data

B.4.1 The `send()` call

In the BSD socket API, the `send()` call is used in both UDP and TCP connection for sending data. Before a call to `send()` the receiver of the data must have been set up using `connect()`. For UDP sessions, the `send()` call resembles the `netconn_send()` function from the LWIP API, but since the LWIP API require the application to explicitly allocate buffers, a buffer must be allocated and deallocated within the `send()` call. Therefore, a buffer is allocated and the data is copied into the allocated buffer.

The `netconn_send()` function of the LWIP API cannot be used with TCP connections, so this implementation of the `send()` uses `netconn_write()` for TCP connections. In the BSD socket API, the application is allowed to modify the sent data directly after the call to `send()` and therefore the `NETCONN_COPY` flag is passed to `netconn_write()` so that the data is copied into internal buffers in the stack.

```

int
send(int s, void *data, int size, unsigned int flags)
{
    struct netconn *conn;
    struct netbuf *buf;

    conn = sockets[s];

    switch(netconn_type(conn)) {
    case NETCONN_UDP:
        /* create a buffer */
        buf = netbuf_new();

        /* make the buffer point to the data that should
           be sent */
        netbuf_ref(buf, data, size);

        /* send the data */
        netconn_send(sock->conn.udp, buf);

        /* deallocated the buffer */
        netbuf_delete(buf);
        break;
    case NETCONN_TCP:
        netconn_write(conn, data, size, NETCONN_COPY);
        break;
    }
}

```

```

    return size;
}

```

B.4.2 The `sendto()` and `sendmsg()` calls

The `sendto()` and `sendmsg()` calls are similar to the `send()` call, but they allow the application program to specify the receiver of the data in the parameters to the call. Also, `sendto()` and `sendmsg()` only can be used for UDP connections. The implementation uses `netconn_connect()` to set the receiver of the datagram and must therefore reset the remote IP address and port number if the socket was previously connected. An implementation of `sendmsg()` is not included.

```

int
sendto(int s, void *data, int size, unsigned int flags,
       struct sockaddr *to, int tolen)
{
    struct netconn *conn;
    struct ip_addr *remote_addr, *addr;
    unsigned short remote_port, port;
    int ret;

    conn = sockets[s];

    /* get the peer if currently connected */
    netconn_peer(conn, &addr, &port);

    remote_addr = (struct ip_addr *)to->sin_addr;
    remote_port = to->sin_port;
    netconn_connect(conn, remote_addr, remote_port);

    ret = send(s, data, size, flags);

    /* reset the remote address and port number
       of the connection */
    netconn_connect(conn, addr, port);
}

```

B.4.3 The `write()` call

In the BSD socket API, the `write()` call sends data on a connection and can be used for both UDP and TCP connections. For TCP connections, this maps directly to the LWIP API function `netconn_write()`. For UDP, the BSD socket function `write()` function is equivalent to the `send()` function.

```

int
write(int s, void *data, int size)
{
    struct netconn *conn;

    conn = sockets[s];

    switch(netconn_type(conn)) {
    case NETCONN_UDP:

```

```

    send(s, data, size, 0);
    break;
case NETCONN_TCP:
    netconn_write(conn, data, size, NETCONN_COPY);
    break;
}
return size;
}

```

B.4.4 The `recv()` and `read()` calls

In the BSD socket API, the `recv()` and `read()` calls are used on a connected socket to receive data. They can be used for both TCP and UDP connections. A number of flags can be passed by the call to `recv()`. None of these are implemented here, and the `flags` parameter is ignored.

If the received message is larger than the supplied memory area, the excess data is silently discarded.

```

int
recv(int s, void *mem, int len, unsigned int flags)
{
    struct netconn *conn;
    struct netbuf *buf;
    int buflen;

    conn = sockets[s];
    buf = netconn_recv(conn);
    buflen = netbuf_len(buf);

    /* copy the contents of the received buffer into
       the supplied memory pointer mem */
    netbuf_copy(buf, mem, len);
    netbuf_delete(buf);

    /* if the length of the received data is larger than
       len, this data is discarded and we return len.
       otherwise we return the actual length of the received
       data */
    if(len > buflen) {
        return buflen;
    } else {
        return len;
    }
}

int
read(int s, void *mem, int len)
{
    return recv(s, mem, len, 0);
}

```

B.4.5 The `recvfrom()` and `recvmsg()` calls

The `recvfrom()` and `recvmsg()` calls are similar to the `recv()` call but differ in that the IP address and port number of the sender of the data can be obtained through the call.

An implementation of `recvmsg()` is not included.

```
int
recvfrom(int s, void *mem, int len, unsigned int flags,
         struct sockaddr *from, int *fromlen)
{
    struct netconn *conn;
    struct netbuf *buf;
    struct ip_addr *addr;
    unsigned short port;
    int buflen;

    conn = sockets[s];
    buf = netconn_recv(conn);
    buflen = netbuf_len(conn);

    /* copy the contents of the received buffer into
       the supplied memory pointer */
    netbuf_copy(buf, mem, len);

    addr = netbuf_fromaddr(buf);
    port = netbuf_fromport(buf);
    from->sin_addr = *addr;
    from->sin_port = port;
    *fromlen = sizeof(struct sockaddr);
    netbuf_delete(buf);

    /* if the length of the received data is larger than
       len, this data is discarded and we return len.
       otherwise we return the actual length of the received
       data */
    if(len > buflen) {
        return buflen;
    } else {
        return len;
    }
}
```

Appendix C

Code examples

C.1 Using the API

This section presents a simple web server written using the LWIP API. The application code is given below. The application implements only the bare bones of an HTTP/1.0 [BLFF96] server and is included only to show the principles in using the LWIP API for an actual application.

The application consists of a single process that accepts connections from the network, responds to HTTP requests, and closes the connection. There are two functions in the application; `main()` which does the necessary initialization and connection setup, and `process_connection()` that implements the small subset of HTTP/1.0. The connection setup procedure is a fairly straightforward example of how connections are initialized using the minimal API. After the connection has been created using `netconn_new()` the connection is bound to TCP port 80 and put into the LISTEN state, in which it waits for connections. The call to `netconn_accept()` will return a `netconn` connection once a remote host has connected. After the connection has been processed by `process_connection()` the `netconn` must be deallocated using `netconn_delete()`.

In `process_connection()` a netbuf is received through a call to `netconn_recv()` and a pointer to the actual request data is obtained via `netbuf_data()`. This will return the pointer to the data in the first fragment of the netbuf, and we hope that it will contain the request. Since we only read the first seven bytes of the request, this is not an unreasonable assumption. If we would have wanted to read more data, the simplest way would have been to use `netbuf_copy()` and copy the request into a continuous memory and process it from there.

This simple web server only responds to HTTP GET requests for the file “/”, and when the request has been checked the response it sent. We send the HTTP header for HTML data as well as the HTML data with two calls to the functions `netconn_write()`. Since we do not modify either the HTTP header or the HTML data, we can use the `NETCONN_NOCOPY` flag with `netconn_write()` thus avoiding any copying.

Finally, the connection is closed and the function `process_connection()` returns. The connection structure is deallocated after the call.

The C code for the application follows.

```
/* A simple HTTP/1.0 server using the minimal API. */

#include "api.h"

/* This is the data for the actual web page.
   Most compilers would place this in ROM. */
const static char indexdata[] =
"<html> \
<head><title>A test page</title></head> \
```

```
<body> \  
This is a small test page. \  
</body> \  
</html>";  
  
const static char http_html_hdr[] =  
"Content-type: text/html\r\n\r\n";  
  
/* This function processes an incoming connection. */  
static void  
process_connection(struct netconn *conn)  
{  
    struct netbuf *inbuf;  
    char *rq;  
    int len;  
  
    /* Read data from the connection into the netbuf inbuf.  
     * We assume that the full request is in the netbuf. */  
    inbuf = netconn_recv(conn);  
  
    /* Get the pointer to the data in the first netbuf  
     * fragment which we hope contains the request. */  
    netbuf_data(inbuf, &rq, &len);  
  
    /* Check if the request was an HTTP "GET /\r\n". */  
    if(rq[0] == 'G' && rq[1] == 'E' &&  
        rq[2] == 'T' && rq[3] == ' ' &&  
        rq[4] == '/' && rq[5] == '\r' &&  
        rq[6] == '\n') {  
  
        /* Send the header. */  
        netconn_write(conn, http_html_hdr, sizeof(http_html_hdr),  
                      NETCONN_NOCOPY);  
  
        /* Send the actual web page. */  
        netconn_write(conn, indexdata, sizeof(indexdata),  
                      NETCONN_NOCOPY);  
  
        /* Close the connection. */  
        netconn_close(conn);  
    }  
}  
  
/* The main() function. */  
int  
main()  
{  
    struct netconn *conn, *newconn;  
  
    /* Create a new TCP connection handle. */  
    conn = netconn_new(NETCONN_TCP);
```

```

/* Bind the connection to port 80 on any
   local IP address. */
netconn_bind(conn, NULL, 80);

/* Put the connection into LISTEN state. */
netconn_listen(conn);

/* Loop forever. */
while(1) {
    /* Accept a new connection. */
    newconn = netconn_accept(conn);

    /* Process the incoming connection. */
    process_connection(newconn);

    /* Deallocate connection handle. */
    netconn_delete(newconn);
}
return 0;
}

```

C.2 Directly interfacing the stack

Since the basic web server mechanism is very simple in that it only receives one request and services it by sending a file to the remote host, it is well suited to be implemented using the internal event based interface of the stack. Also, since there is no heavy computation involved, the TCP/IP processing is not delayed. The following example shows how to implement such an application. The implementation of the application is very similar to the above example.

```

/* A simple HTTP/1.0 server directly interfacing the stack. */

#include "tcp.h"

/* This is the data for the actual web page. */
static char indexdata[] =
"HTTP/1.0 200 OK\r\n\
Content-type: text/html\r\n\
\r\n\
<html> \
<head><title>A test page</title></head> \
<body> \
This is a small test page. \
</body> \
</html>";

/* This is the callback function that is called
   when a TCP segment has arrived in the connection. */
static void
http_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p)
{
    char *rq;

    /* If we got a NULL pbuf in p, the remote end has closed

```

```
    the connection. */
if(p != NULL) {

    /* The payload pointer in the pbuf contains the data
       in the TCP segment. */
    rq = p->payload;

    /* Check if the request was an HTTP "GET /\r\n". */
    if(rq[0] == 'G' && rq[1] == 'E' &&
        rq[2] == 'T' && rq[3] == ' ' &&
        rq[4] == '/' && rq[5] == '\r' &&
        rq[6] == '\n') {

        /* Send the web page to the remote host. A zero
           in the last argument means that the data should
           not be copied into internal buffers. */
        tcp_write(pcb, indexdata, sizeof(indexdata), 0);
    }

    /* Free the pbuf. */
    pbuf_free(p);
}

/* Close the connection. */
tcp_close(pcb);
}

/* This is the callback function that is called when
   a connection has been accepted. */
static void
http_accept(void *arg, struct tcp_pcb *pcb)
{
    /* Set up the function http_recv() to be called when data
       arrives. */
    tcp_recv(pcb, http_recv, NULL);
}

/* The initialization function. */
void
http_init(void)
{
    struct tcp_pcb *pcb;

    /* Create a new TCP PCB. */
    pcb = tcp_pcb_new();

    /* Bind the PCB to TCP port 80. */
    tcp_bind(pcb, NULL, 80);

    /* Change TCP state to LISTEN. */
    tcp_listen(pcb);

    /* Set up http_acct() function to be called
       when a new connection arrives. */
}
```

```
    tcp_accept(pcb, http_accept, NULL);  
}
```

Appendix D

Glossary

ACK

The acknowledgment signal used by TCP.

API

Application Program Interface. A set of functions that specifies the communication between an application program and a system service.

checksum

A checksum is a function that computes a specific number by summing all bytes in a packet. Used for detection of data corruption.

congestion

Congestion occurs when a router drops packets due to full buffers, i.e., when the network is overloaded.

datagram

A chunk of information. Analogous to a packet.

demultiplexing

The opposite of multiplexing. Extracting one of the information streams from a combined stream of information streams.

header

Control information for a packet located at the beginning of the packet.

ICMP

Internet Control Message Protocol. An unreliable signaling protocol used together with IP.

internet

An interconnected set of networks using IP for addressing.

Internet

The global Internet.

IP

Internet Protocol. The protocol used for addressing packets in an internet.

IPv4

Internet Protocol version 4. The version of IP mainly used in the global Internet.

IPv6

Internet Protocol version 6. The next generation IP. Expands the address space from 2^{32} combinations to 2^{128} and also supports auto-configuration.

multiplexing

A technique that enables two or more information streams to use the same link. In the TCP/IP case this refers to the process of, e.g., using the IP layer for many different protocols such as UDP or TCP.

packet

A chunk of information. Analogous to a datagram.

PCB

Protocol Control Block. The data structure holding state related information of a (possibly half-open) UDP or TCP connection.

proxy

An intermediate agent that utilizes knowledge of the transportation mechanism to enhance performance.

RFC

Request For Comments. A paper specifying a standard or discussing various mechanisms in the Internet.

round-trip time

The total time it takes for a packet to travel from the sender to the receiver, and for the reply to travel back to the sender.

router

A node in an internet. Connects two or more networks and forwards IP packets across the connection point.

UDP

User Datagram Protocol. An unreliable datagram protocol on top of IP. Mainly used for delay sensitive applications such as real time audio and video.

TCP

Transmission Control Protocol. Provides a reliable byte stream on top of IP. The most commonly used transportation protocol in today's Internet. Used for email as well as file and web services.

TCP/IP

The internet protocol suite which includes the basic delivery protocols such as IP, UDP and TCP as well as some application level protocols such as the email transfer protocol SMTP and the file transfer protocol FTP.

Bibliography

- [ABM95] B. Ahlgren, M. Björkman, 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.
- [AP99] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proceedings of the SIGCOMM '99 Conference*, Cambridge, MA, September 1999.
- [APS99] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. RFC 2581, Internet Engineering Task Force, April 1999.
- [ARN] Summary of the Arena project. Web page. 2000-11-21.
URL: <http://www.cdt.luth.se/projects/summary/arena.html>
- [BB95] A. Bakre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.
- [BIG⁺97] 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.
- [BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC 1945, Internet Engineering Task Force, May 1996.
- [Bra89] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, Internet Engineering Task Force, October 1989.
- [Bra92] R. Braden. TIME-WAIT assassination hazards in TCP. RFC 1337, Internet Engineering Task Force, May 1992.
- [BS97] K. Brown and S. Singh. M-TCP: TCP for mobile cellular networks. *ACM Computer Communications Review*, 27(5):19–43, October 1997.
- [BSAK95] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the first ACM Conference on Mobile Communications and Networking*, Berkeley, California, November 1995.
- [Car96] B. Carpenter. Architectural principles of the Internet. RFC 1958, Internet Engineering Task Force, June 1996.
- [Cla82a] D. D. Clark. Modularity and efficiency in protocol implementation. RFC 817, Internet Engineering Task Force, July 1982.
- [Cla82b] D. D. Clark. Window and acknowledgement strategy in TCP. RFC 813, Internet Engineering Task Force, July 1982.
- [FH99] S. Floyd and T. Henderson. The NewReno modifications to TCP’s fast recovery algorithm. RFC 2582, Internet Engineering Task Force, April 1999.

- [FTY99] T. Faber, J. Touch, and W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *Proceedings of IEEE INFOCOM '99*, New York, March 1999.
- [HNI⁺98] J. Haartsen, M. Naghshineh, J. Inouye, O. Joeressen, and W. Allen. Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, 2(4):38–45, October 1998.
- [Jac88] V. Jacobson. Congestion avoidance and control. In *Proceedings of the SIGCOMM '88 Conference*, Stanford, California, August 1988.
- [Jac90] V. Jacobson. 4.3BSD TCP header prediction. *ACM Computer Communications Review*, 20(2):13–15, April 1990.
- [JBB92] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. RFC 1323, Internet Engineering Task Force, May 1992.
- [KP87] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of the SIGCOMM '87 Conference*, Stowe, Vermont, August 1987.
- [KP96] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions of Networking*, 4(6):817–828, December 1996.
- [LDP99] L. Larzon, M. Degermark, and S. Pink. UDP Lite for real-time multimedia applications. In *Proceedings of the IEEE International Conference of Communications*, Vancouver, British Columbia, Canada, June 1999.
- [MBKQ96] 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.
- [MD92] Paul E. McKenney and Ken F. Dove. Efficient demultiplexing of incoming TCP packets. In *Proceedings of the SIGCOMM '92 Conference*, pages 269–279, Baltimore, Maryland, August 1992.
- [MK90] T. Mallory and A. Kullberg. Incremental updating of the internet checksum. RFC 1141, Internet Engineering Task Force, January 1990.
- [MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. RFC 2018, Internet Engineering Task Force, October 1996.
- [Mog92] J. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.
- [Nab] M. Naberezny. The 6502 microprocessor resource. Web page. 2000-11-30.
URL: <http://www.6502.org/>
- [Pax97] Vern Paxson. End-to-end internet packet dynamics. In *Proceedings of the SIGCOMM '97 Conference*, Cannes, France, September 1997.
- [Pos80] J. Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.
- [Pos81a] J. Postel. Internet control message protocol. RFC 792, Internet Engineering Task Force, September 1981.
- [Pos81b] J. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.
- [Pos81c] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

- [PP93] C. Partridge and S. Pink. A faster UDP. *IEEE/ACM Transactions in Networking*, 1(4):429–439, August 1993.
- [PS98] S. Parker and C. Schmechel. Some testing tools for TCP implementors. RFC 2398, Internet Engineering Task Force, August 1998.
- [RF99] K. Ramakrishnan and S. Floyd. A proposal to add Explicit Congestion Notification (ECN) to IP. RFC 2481, Internet Engineering Task Force, January 1999.
- [Rij94] A. Rijsinghani. Computation of the internet checksum via incremental update. RFC 1624, Internet Engineering Task Force, May 1994.
- [Shr] H. Shrikumar. IPic - a match head sized web-server. Web page. 2000-11-24.
URL: <http://www-ccs.cs.umass.edu/~shri/iPic.html>
- [SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [vB] U. von Bassewitz. cc65 - a freeware c compiler for 6502 based systems. Web page. 2000-11-30.
URL: <http://www.cc65.org/>
- [WS95] G. Wright and W. Stevens. *TCP/IP Illustrated Volume 2*. Addison-Wesley, Reading, MA, 1995.
- [Zak83] R. Zaks. *Programming the 6502*. Sybex, Berkeley, California, 1983.