A Comparison of TCP Implementations

Linux TCP vs. lwIP

Richard Sailer

Universität Augsburg
Lehrstuhl für Organic Computing
Forschungsmodul im Studiengang Informatik

Table of contents

Chapter 1
Introduction

TCP implementations share their fate with system administrators: they only get attention when something doesn't work. While shiny enterprise cloud web applications lie in the focus of most software manager and perhaps even developers TCP is the silent workhorse under every web application and every browser. Over 30 years of academic research and engineering effort are the reason for this network connection reliability people take for granted. Reliable and correct file downloads and web surfing have become naturally as well as electric power, water or german cars and trains[1.1].

This "it works without having to spend one thought on it" property is in general a good thing, but nevertheless there will be cases when people are needed that understand TCP implementations. This can be when another extension to TCP is wanted or, well... something doesn't work. For these people this document is made. We give an overview over two TCP implementations and compare them. This does not include an function for function explanation, more a general picture and some code quality study.

Depending on what you want to do perhaps not every part of the work is interesting for you, this why we took good care of putting everything in separate chapters and sub chapters. We have one sub chapter on TCP features comparing the two implementations feature wise. The next is on user bases telling how much and which people use these implementations.

In our fourth chapter we start looking at the TCP implementations and their files, we describe and isolate the files from their surrounding network stack code to have and examine them separately in the next chapters.

Subsequent we give an overview over different static code analysis and code quality metric measurement tools, since this is the second focus of this work. Besides getting an overview of TCP implementations and conceptually understanding them, we also want to try to quantitative measure and classify their implementation quality. During this we not only want to present the user our details but also explain the tools and how to "do this at home" with other software or code. This includes a little overview over available tools and a short tutorial how we used them.

In the next sub chapter we show the comparative results regarding project modularization, function length and complexity and density of code documentation.

Then in chapter six we take a closer look at the data structures these two implementations use for packets and compare them, quantitatively and from an software architecture point of view.

1.1 Target Audience

In general this text can be helpful for the following categories of people:

1. People interested in TCP implementations. What features are there? How are they structured? What is the code quality of existing implementations.
2. People interested in code quality metrics. How to create them? How to interpret and visualise them?
3. People who often have to dive into and understand a large existing code base. Since the methodology presented here for isolating parts and getting an overview is usable for every software project, not only TCP implementations.

1.1. Well the last two are perhaps no so good examples anymore.

1.2 Required Knowledge

This work is in general thought for graduate student's having a bachelor degree in computer science or related fields. But it should also be easy to understand for everyone having a few years of practical experience with computer networks and C. To be more precise the most relevant concrete necessary skills are:

- Knowing the C programming language
- Understanding TCP[1.2]
- Some familiarity with the Linux command line

1.2. A very good and comprehensive description of TCP can be found in Tanenbaum's Computer Networks book[Tan03]. For an short overview the wikipedia article on TCP (https://en.wikipedia.org/wiki/Transmission_Control_Protocol) is also a good reference.

Chapter 2
Related Work

2.1 A Tale of Four Kernels

In 2008 Diomidis Spinellis released the paper "A Tale of Four Kernels"[Spi08]. There he extensively compared the four operating systems:

1. FreeBSD
2. GNU/Linux
3. OpenSolaris[2.1]
4. The Windows Research Kernel[2.2]

His comparison regards code quality metrics. His final result is that there are no big quality differences between the 4 systems, but Linux scores a bit bad in documentation, while windows scores bad in code structure and function length. Spinellis work has been a been a major inspiration for our research regarding concepts and methodology. We will even use one of his tools for collecting metrics cqmetrics.

2.2 TCP/IP Illustrated Volume 2

In TCP/IP Illustrated Volume 2[WS95] Gary Wright and Richard Stevens carefully describe and explain the TCP/IP implementation of 4.4BSDLite. The usual common TCP/IP reference implementation of their time[WS95][Ste93]. This BSD TCP/IP implementation was the first complete and freely available TCP/IP implementation and was for long time referenced as "the standard". Even the Linux TCP source code references it's design and behaviour in many comments several times.[2.3]

Unfortunately this work and the TCP/IP implementation is from 1995 and there exist no newer release. Since TCP/IP implementations have evolved this book is not so recent anymore. But it's still a good starting point, since the fundamental algorithms and concepts have rarely been changed. In most cases they were rather augmented.[2.4]

The authors explain the data structures and logic in a very clear, understandable and visual way using many illustrations and careful source code discussion.

Since describing and explaining TCP implementations is also a part of our work, in all the relevant places we tried to use a comparable presentation approach.

2.1. Oracle has bought Sun Microsystems meanwhile and discontinued the open development of OpenSolaris. Nevertheless there exists a free fork of OpenSolaris continued by some of its developers called OpenIndiana.

2.2. The Windows Research Kernel is a cleaned up and reduced release of the Windows Server 2003 kernel. This means it's a 64 bit NT kernel. Microsoft released this as a step towards academica, to make it possible to use this NT code in Operating system lectures and curses. Nevertheless it's still not open source and obtaining the source code is a bit laborious, since it is only given to academic staff under certain conditions.

2.3. See `tcp.c` and `tcp_input.c` for examples.

2.4. A look at the history of RFC standards and also the SVN revision history of the BSD releases is good conceptual proof for this.

Chapter 3
Feature Comparison and User Base

3.1 Feature Comparison

Since TCP is over 30 years old[Ins81] and widely used there exist lots of extensions. While the original TCP is already quite complex and powerful, all these extensions add even more functionality but also complexity.[3.1] To understand and compare TCP implementations not only code discussion, but also a semantic look at the functionality is useful. Understanding the range of functionality also means understanding size and several design decisions.

Table 3.1 list the most common TCP features and extensions and if they're implemented by one of our TCP implementations. A x means "yes", a - means "no". Our primary sources were the Linux TCP man page[Pro15] and the official LWIP wiki[lwia].

Feature	In LWIP?	In Linux TCP?
Congestion Control	x	x
Fast Retransmit	x	x
Fast Recovery	x	x
Fast Open	-	x
Delayed ACK	-	x
Additional Pluggable Congestion Control Algorithms (Westwood, Vegas, NewReno, Cubic, etc.)	-	x
Multithreading Support	-	x
Portable	x	-
Firewall Support	-	x (netfilter, conntrack)
Instrumentation	-	x (ftrace, perf, etc.)
Dynamic User Space Configuration	-	x
ECN	-	x
TCP Sack	-	x
TCP Fack	-	x
TCP RACK	-	x
TCP Keepalive	x	x
PAWS (Protection Against Wrapping Sequence numbers)	x	x
Window scaling (RFC 1323)	x	x
Timestamps	x	x
Nagle Algorithms	x	x

Table 3.1. Feature Comparison of the Two TCP Implementations

What do these differences mean? Well several conclusions:

1. TCP Sack/RACK/Fack

3.1. For brevity and conciseness we chose to omit a full description of every TCP extension. You can find a short and informal description of most of them on https://en.wikipedia.org/wiki/Transmission_Control_Protocol and a detailed full description in their respective RFCs.

TCP Sack is "Selective Acknowledgements" a very powerful but also complex extension. Omitting it means considerable less code, but also less functionality. Omitting Sack means often re-sending more packets than necessary, because acknowledgements are less accurate. You can find a further discussion of the Linux TCP Sack functions in Appendix A and in the File Organisation Section of the Code Quality Results.

The same effect on code length and functionality as with SACK also holds for RACK and Fack but in a less grave way.

2. Firewall Support
   Omitting firewall support also saves a lot of functionality and code complexity.
3. Multithreading Support
   The LWIP TCP implementation works entirely single threaded and uses a lot of global variables.[3.2] While using global variables in general is bad practise in a embedded, memory constrained, per design single threaded environment it's a understandable design decision.
   Working single threaded has the following advantages
   - No locking overhead (less code complexity)
   - No possible race condition, or starvation, or deadlock problems

   And the following disadvantages:
   - Less throughput on multi core systems

   Since the embedded systems LWIP targets are mostly single core system this makes sense.

So summarised, LWIP implements considerable less functionality and so we expect the code to be smaller and less complex. The next chapters will show if this comes true.

3.2 User Base

How relevant is all this? Are we evaluating a academic curiosity or widespread productivity software? The next sections will answer these questions.

3.2.1 Linux TCP

The Linux TCP implementation is not very portable, it has only one target software system: the Linux kernel. Nevertheless the Linux kernel is really portable and widespread so this compensates. According to a hitlinks study the Linux kernel has one of the largest installation base of all general purpose systems[hit]. The detailed reason for this are:

- Android phones (65 % of all smartphones[hit])
- Also most server run Linux
- And a considerable share of network home routers, video game consoles and smartwatches

All these devices also run the Linux TCP implementations, which make it quite widespread and relevant.

3.2.2 LWIP TCP

LWIP is a lot more specialised (and portable), it's main use case are embedded devices and embedded operating systems. It's used by many embedded device manufacturers like:

- Altera
- Analog Devices[San08]
- Xilinx[Vel08]

3.2. Which you can see, for example in `tcp_in.c` lines `67-84`

- Honeywell
- and some more

Among other things in avionic and automotive systems.

LwIP may also become even more widespread with the progress of the internet of things[3.3], since most of the involved devices are small and have limited resources. For them a lightweight TCP/IP stack is a seasonable infrastructure to connect to the internet.

3.3. The "internet of things" is the idea of connecting physical devices to the internet. Like cars, fridges, coffee maker, lights, rooms, doors etc. What this means for it users and if it makes sense is a separate question, not discussed in this document.

Chapter 4
TCP Code Overview and Isolation

Since both

- the TCP implementation in the Linux kernel
- and the TCP implementation inside the LWIP TCP/IP stack

are part of a larger code base, at first we had to locate and isolate the source files which are the TCP implementation. The following subsections describe how we identified them and which particular files they are.

We describe our course of action quite detailed. This may be of use to you when you will have to identify and isolate some functionality in a bigger project some day, for reuse or some other purpose. On the other hand, if you're just interested in the results, you can find the isolated file lists at the end of each sub chapter.

During the selection and isolation we also had to look at the files to isolate. So, as a side effect, this chapter also gives a good overview of the isolated files and their content/functionality.

The same goes for directory structure and file organisation. We looked through the whole project while searching and also explain what we found and how we used this to "navigate and find". So you can (to some degree) get an idea how these project files are organised and how to find stuff.

Speaking of effort, the isolation was relatively straightforward for LWIP and a bit more complicated for Linux.

4.1 Which Versions

We decided to use recent stable versions[4.1] [4.2] of both TCP implementations which are at the time of this writing[4.3]:

TCP Stack	Version
Linux Kernel TCP	4.8.14
LWIP TCP	Release 2.0.0

Table 4.1. Software Versions Used for Analysis

4.2 LWIP

Our first approach was to use the directory structure and filenames. The LWIP repository contains a directory called `src/` containing all source code and header files. In `src/` there resides a text file `FILES` containing the following information:

4.1. Proof for correct Linux Version Number: [lin] This is the web interface to the version control system of the Linux source code repository. This special sites lists all refs, which are tags and branch pointers, Linux uses tags for version numbering. Version numbers of the form (for example) v3.x.y (three numbers) are stable releases. Stable release numbers are sorted lexicographically (Source for all of these statements about version numbers: [Lov10]). So to find the most recent stable release, one has to pick the lexicographically highest stable version number in this list of tags. Which at the moment is 4.8.14. A dump of this repository's web interface refs site of the current day is also attached to this document in the "web-sources" folder.

4.2. Proof for correct LWIP Version Number: [lwib] This again, like in the proof for the Linux Version Number, is the web interface to the version control system of a source code repository's. But this time of course the LWIP source code repository's. Here also tags are used for version numbers. The highest version number you can find on this page is Release 2.0.0. A dump of this page is also included in the "web-sources" folder.

4.3. 14.12.2016

```
api/      - The code for the high-level wrapper API. Not needed if
            you use the lowel-level call-back/raw API.

apps/     - Higher layer applications that are specifically programmed
            with the lwIP low-level raw API.

core/     - The core of the TPC/IP stack; protocol implementations,
            memory and buffer management, and the low-level raw API.

include/  - lwIP include files.

netif/    - Generic network interface device drivers are kept here.

For more information on the various subdirectories, check the FILES
file in each directory.
```

4.2.1 C Source Files

Obviously `api/, apps/ and netif/` and `netif/` aren't relevant if we are looking for the TCP implementation. We also won't consider `include/` here, because header files will be treated extra later. So the directory relevant for us is `core/`. `core/` contains the following files:

```
 > tree -L 1
.
├── def.c
├── dns.c
├── inet_chksum.c
├── init.c
├── ip.c
├── ipv4
├── ipv6
├── mem.c
├── memp.c
├── netif.c
├── pbuf.c
├── raw.c
├── stats.c
├── sys.c
├── tcp.c
├── tcp_in.c
├── tcp_out.c
├── timeouts.c
└── udp.c
```

(with the **strong** printed ones being directories). Here there lie three files, `tcp.c, tcp_in.c, tcp_out.c` their naming strongly suggests they contain a tcp implementation. And the naming of all the other files suggest they contain something else than tcp. (The subfolders **ipv4** and **ipv6** contain nothing by filename tcp-related either. Their full content was not displayed for brevity).

But what if the modularization is not that clean? What if `pbuf.c` or `ip.c` also contains some TCP logic? Just assuming the separation is clean would mean a lot of good faith and not very much critical scientific spirit. This is why we looked for a second method to get or verify the the selection of these three files. And we found one!

Fortunately the build process of LWIP is highly customisable. It's even possible not to include TCP in the final build at all[lwia] (like, for embedded devices which only need UDP). So we could look at how this in/exclusion works and which files where affected.

The build process configuration works via a optionally user-provided file called `lwipopts.h`. Here it is possible to disable TCP via the following line:[lwic]

```
#define LWIP_TCP 0
```

This all uses the C preprocessor. It's a technique called *conditional compilation* [RKL88], which means, several other files contain the structure surrounding code:

```
#if LWIP_TCP /* don't build if not configured for use in lwipopts.h */
...
#endif /* LWIP_TCP */
```

Figure 4.1. Conditional Compilation Example

This means only if the preprocessor Symbol `LWIP_TCP` is true (defined and not null) the code between #if and #endif will be included in the final build.

So we can look which and how much code in the LWIP repository is surrounded by this structure and therefore deduce which files belong to the TCP implementation. This is great because that's not that difficult using some regex magic. To search all files in `src/core/` we used the tool `ag` which is like grep, but faster and with nicer formatted output.[4.4]:

```
ag -i  '#if.*LWIP_TCP[\s\S]*#endif.*LWIP_TCP.*'
 > /tmp/LWIP_TCP_macro_scanning
```

Figure 4.2. Ag Command Used to Search for Uses of This Special Conditional Compilation Structure

This searches for all text being enclosed by the previously described structure and saves the results in the file `/tmp/LWIP_TCP_macro_scanning`[4.5]. Some explanations about `ag` and the regexp:

- ag searches multi-line per default so it is possible to match bigger sections enclosed by the structure
- `.*` means 0 or more arbitrary symbols but no newline
- `[\s\S]` is Perl regex specific and means:

  `\s`. all whitespaces

  `\S`. everything except whitespaces

  So together `[\s\S]` means "absolutely every symbol"
- why `-i` (case insensitive)? Well, we were not sure if the preprocessor keywords are case sensitive. However, the results showed that we didn't match "too much" because of lower/higher-case variations.
- The result file contains **all** matches, including the lines between the structure, which is why it's quite long and not directly included in this work. But it's easy too navigate and orientate in it since ag draws the corresponding filename and line number in front of every line. This makes it also easy to tell apart the different matches.

The result of this pattern matching journey was very pleasant. The files in `src/core/` containing the `LWIP_TCP` structure, can be divided into two categories:

1. Files containing one or more matching blocks of only a few lines length.

   Like `timeouts.c` and `init.c`. These are files fulfilling other tasks than TCP, but having little TCP related parts.

   Like the general timer in timeouts.c which can also be asked if "is the tcp timer currently scheduled" an needs some code to be in sync with the TCP timer.

4.4. See https://github.com/ggreer/the_silver_searcher if interested. It's packaged and available in at least: Ubuntu, Arch Linux and Debian.

4.5. Also included in the attached dir/archive under the same name

Or the code in `init.c` initialising the TCP stack at start time.

2. Files being almost the entire matching block (minus some copyright header).
   These are our candidates. Fortunately these match with the 3 files discovered by name and directory structure earlier which means: LWIP has a clear and clean module distinction. This can also be seen as a first result metric!

So now we know the .c files containing the TCP implementation for sure and have learned some facts about the LWIP directory structure, the other files in `src/core/` and how they relate to the TCP implementation. To complete this little section here is a synopsis of these 3 files with a short description taken from their comment header:

tcp.c.
Common functions for the TCP implementation, such as functions for manipulating the data structures and the TCP timer functions.

tcp_in.c.
The input processing functions of the TCP layer.

tcp_out.c.
The output functions of TCP.

4.2.2 Header Files (in `src/include/lwip/`)

Unfortunately the pattern matching approach didn't work that well for the header files. Well, it worked for 2 of 3 files. We could find `tcp.h` and `priv/tcp_priv.h` with this method. But further code reading in a later stage of this work revealed, that one of this file includes a further header file: `prot/tcp.h`.

Of course this somehow makes sense, if you want to ensure that a unneeded header files is not included you do not need to "block" it directly, it suffices if all files including it are blocked. Fortunately .c files do not get included this way so this does not apply to .c files, therefore we don't have to go back and correct this for the .c files.

At last again, a listing and short description of all the files we took into account:

tcp.h.
TCP API. The function calls with which the TCP implementation can be accessed from the "outside". Included by the tcp*.c files because they also have to know the API they have to fulfil.

priv/tcp_priv.h.
TCP internal implementations (do not use in application code). Private as in "not for the LWIP applications like the little web server or the TFTP server.

prot/tcp.h.
TCP protocol definitions, like TCP header and flags.

4.3 Linux TCP

For the Linux TCP implementation this was more difficult.

4.3.1 C Source Files (Using Directory Structure and File Names)

Again at First we decided to use the directory structure and filenames. In the Linux root folder there lies a folder called `net/`. In this folder we first looked for all files containing "`tcp`" in their filename and ending with `.c`. We did this via:

```
find . -iname '*tcp*.c' > /tmp/linux-filenames-containing-tcp
```

With the following result:

```
./rds/tcp.c                    ./rds/tcp_listen.c
./rds/tcp_recv.c               ./rds/tcp_send.c
```

```
./rds/tcp_connect.c
./rds/tcp_stats.c
./ipv4/tcp_htcp.c
./ipv4/tcp.c
./ipv4/tcp_pr.c
./ipv4/tcp_highspeed.c
./ipv4/tcp_nv.c
./ipv4/tcp_metrics.c
./ipv4/tcp_vegas.c
./ipv4/tcp_probe.c
./ipv4/tcp_cong.c
./ipv4/tcp_westwood.c
./ipv4/tcp_cubic.c
./ipv4/tcp_output.c
./ipv4/tcp_recovery.c
./ipv4/tcp_yeah.c
./ipv4/tcp_dctcp.c
./ipv4/tcp_offload.c
./ipv4/tcp_cdg.c
./ipv4/tcp_lp.c
./ipv4/tcp_bic.c
./ipv4/tcp_timer.c
./ipv4/tcp_fastopen.c
./ipv4/tcp_diag.c
./ipv4/tcp_veno.c
./ipv4/tcp_hybla.c
./ipv4/tcp_input.c
./ipv4/tcp_scalable.c
./ipv4/tcp_ipv4.c
./ipv4/tcp_illinois.c
./ipv4/tcp_minisocks.c
./ipv6/tcpv6_offload.c
./ipv6/tcp_ipv6.c
./netfilter/xt_tcpmss.c
./netfilter/nf_nat_proto_tcp.c
./netfilter/ipvs/ip_vs_proto_tcp.c
./netfilter/xt_tcpudp.c
./netfilter/xt_TCPOPTSTRIP.c
./netfilter/xt_TCPMSS.c
./netfilter/nf_conntrack_[..]_tcp.c
```

Looking at this list we can eliminate several entries via a process of elimination:

1. In `netfilter/` lies the packet filter implementation. (Iptables and firewalling functionality)
   Therefore the listed files in the netfilter folder should be related to "matching tcp packets" but not be the TCP implementation, therefore we can omit them.
2. The files in `ipv6/`:
   `./ipv6/tcp_ipv6.c` contains the glue code to make tcp work in IPv6 packets. Therefore it's not relevant for us. (Source: The Header in the file)
   `./ipv6/tcpv6_offload.c` contains TCP offloading code. Offloading means "letting specialised hardware do stuff, that usually the CPU does". Offloading is a good and very performance-conducive thing, but also something not directly related to a TCP implementation. Therefore we can ignore this file too.
3. The files in `rds/`:
   Those were the biggest surprise. A short look at the file `rds/Kconfig` (see Note 4.1, for details about Kconfig files) revealed what `rds/` is at all:

   ```
   config RDS
     tristate "The RDS Protocol"
     depends on INET
     ---help---
      The RDS (Reliable Datagram Sockets) protocol provides
      reliable, sequenced delivery of datagrams over Infiniband or
      TCP.
   ```

   So RDS is a own network protocol somehow using TCP (or Infiniband) for its functionality. We can therefore safely assume that all the files like `./rds/tcp_connect.c` contain the glue code to make RDS work *using TCP* but no TCP implementation critical files. Therefore we can omit them too.

Note 4.1. `Kconfig` files and `make menuconfig`
Kconfig files are the info files used by the kernel configuration tools (like make menuconfig or make nconfig). These tools allow you to select which functionality you want to include in your kernel and which to omit. Somehow a more sophisticate way of doing conditional compilation, like with LWIP in the last sub chapter. Those tools use the Kconfig files in the subdirectories to know and display all the available functionality modules, with a short description, this description helped us here.

This elimination now leaves us with the files in `ipv4/`, which are still quite a lot. Some filenames look suspicious like `tcp_yeah.c` or `tcp_vegas.c`. Both sound suspiciously related to congestion control algorithms (like YEt Another Highspeed TCP[BCV] and Vegas[BOP94]). A short look at their content and the headers confirms this suspicion. Actually a lot of these files sounded like additional congestion control algorithms. We decided not to include them in our analysis for several reasons:

1. They're often contributed from external research projects, so their quality may vary a lot from the rest of the core-TCP implementation (in a good way as well as in a bad way). So they should be analysed separately.
2. The Linux default is TCP Cubic. Also most widespread distributions use Cubic for default too[4.6](Although at least Ubuntu and Debian include them as modules in `/lib/modules/$kernel-version/`). Therefore we currently assume they are not that widely used. This is an assumption and may be not true, so neither less it was just a focus decision to only include Core-TCP.

So how do we isolate them from the Core-TCP implementation? We decided to use conditional compilation and the kernel config feature again. Fortunately in the semi-graphical kernel configuration utility `make nconfig` you can select or deselect every TCP congestion control algorithm (or chose compilation as a module). Therefore we created a new kernel config, being the default Debian one but having the option `"TCP: advanced congestion control"` (`CONFIG_TCP_CONG_ADVANCED`) disabled, to get the absolute basic and no extra congestion control algorithms[4.7]. At next we compiled the kernel (or at least all configured files in `net/ipv4`) using

```
make clean
make net/ipv4
```

A whole terminal log of the commands and the compile output is attached in the `linux-compilation-no-advanced-tcp-cc` folder. The relevant findings can be obtained when we grep the compile log for "tcp" which results in:

```
> grep "tcp" compile-log
  CC      net/ipv4/tcp.o
  CC      net/ipv4/tcp_input.o
  CC      net/ipv4/tcp_output.o
  CC      net/ipv4/tcp_timer.o
  CC      net/ipv4/tcp_ipv4.o
  CC      net/ipv4/tcp_minisocks.o
  CC      net/ipv4/tcp_cong.o
  CC      net/ipv4/tcp_metrics.o
  CC      net/ipv4/tcp_fastopen.o
  CC      net/ipv4/tcp_recovery.o
  CC      net/ipv4/tcp_offload.o
  CC      net/ipv4/tcp_cubic.o
  CC [M]  net/ipv4/tcp_diag.o
```

This list is way more handy and clear than the last one. But this could still contain "too much" i.e. files containing something else than the core TCP implementation. Too be more sure these are the correct files (and to get a first semantic overview over the Linux TCP implementation) lets start with a short description of each file, together with **[yes]** if we decided to include it in our analysis and **[no]** if we decided against it:

tcp.c.
TCP implementation all common "base" functions
[yes]

tcp_input.c.
All functions for processing incoming TCP segments.

4.6. See the `/boot/config-$kernel_version` file on a arbitrary Ubuntu, Debian or Fedora system.

4.7. The full config lies in the attached archive in the folder `linux-compilation-no-advanced-tcp-cc`.

[yes]

tcp_output.c.
All functions for processing outgoing TCP packets.

[yes]

tcp_timer.c.
All TCP related timers which include:

tcp_write_timer. The default TCP retransmission timeout. Re-transmits a packet if no ACK has arrived in time

tcp_delack_timer. The delayed ACK timer. This makes it possible to avoid sending an ACK for every packet (which is expensive). Using this timer it's possible to "wait a little time" before sending an ACK. Maybe an additional data packet arrives and we can send an ACK for 2 packets, which is way more efficient. See RFC 2581[APS99] for more details on delayed ACK.

tcp_keepalive_timer. Used for the TCP keepalive feature, see TCP/IP Illustrated Vol 1[Ste93] for details.

[yes]

tcp_ipv4.c.
Functions necessary to make TCP work with IPv4. IPv4 specific functions. Since binding to IPv4 is not really part of a core-TCP functionality we decided not to include it.

[no]

tcp_minisocks.c.
We are not entirely sure what this file does. Currently we assume it does something related to glueing TCP and the Socket layer together, also it's not very long, therefore we decided to keep it.

[yes]

tcp_cong.c.
TCP congestion control. This contains the main interface for concrete congestion control algorithm implementations, as well as a fallback implementation (New Reno)

[yes]

tcp_metrics.c.
Seems to generate and maintain connection and data flow metrics (like these read out by the `netstat` tool). Since we could not decide whether these metrics are needed by the TCP Stack for normal operation or not we decided to keep them.

[yes]

tcp_fastopen.c.
The Fast Open algorithm as specified in [CCRJ14]. Since Fast Open is turned on by default since Kernel 3.13 we decided to include it too

[yes]

tcp_recovery.c.
Implementation of another TCP extension: "RACK: a time-based fast loss detection algorithm for TCP". We decided to include it in our analysis. It's also very short (2 functions).

[yes]

tcp_offload.c.
Offload functionality for Linux TCP. Since offloading is no part of core-TCP functionality, we decided to exclude this file.

[no]

tcp_cubic.c.
The cubic TCP congestion control algorithm. Since it's the Linux default we decided to include it.

[yes]

tcp_diag.c.
Module for monitoring TCP transport protocols sockets. Since diagnostics are not needed for TCP operation we decided to exclude this file too.

[no]

So our final analysis set of .c files is:

```
tcp.c
tcp_cong.c
tcp_fastopen.c
tcp_input.c
tcp_metrics.c
tcp_minisocks.c
tcp_output.c
tcp_recovery.c
tcp_timer.c
```

Figure 4.3. Final Analysis set of Linux TCP Implementation

4.3.2 Header files

The relevant TCP header files are: (relatively to the repository root directory)

```
include/net/tcp_states.h
include/net/tcp.h
include/linux/tcp.h
include/uapi/linux/tcp.h
```

Figure 4.4. Linux TCP Header Files

These were obtained using the following methods:

1. Using find, filenames, directory structure and file contents/comments like in the previous section
2. Looking at the header files included in all the .c files of the previous section.

4.3.3 Evaluation of Results and Methodology

Have we reached our goal? Have we isolated all relevant files. Could we have missed some? Could we have included too much? It's necessary and sensible too ask this because the method used here (for the Linux kernel) relies on several conditions and we have to be aware of that:

1. The Linux kernel uses proper modularization and functionality is cleanly separated.
2. The Linux kernel source files are named sensible

We want to give these points a short discussion:

About modularization. Two recent books about Linux kernel programming contain the statement "the Linux kernel is properly organised and modularised and you should look for your code too be be as good modularised to get in merged" [Lov10][QK12]. We have chosen to believe this.

About naming. Especially this point is worry-some, since you may already have noticed from the logs and search results that kernel developers love abbreviations and short-naming things. Fortunately TCP is already quite short (3 characters), and for all function names and variables we have observed so far it was "non-abbreviated" i.c. `tcp_`. So we can assume this, at least for TCP with some certainty.

Both of these thoughts are quite but not finally convincing. Luckily there are other good reasons for our selection to be correct:

1. The two books "Linux Kernel Networking: Implementation and Theory"[Ros13] and "TCP/IP Architecture, design and implementation in Linux"[SV09] both contain discussions of the Linux TCP source code. All the files mentioned there are part of our analysis set. So this is a strong indication we have not forgotten anything.

2. The contents of the files (and their header comment) also seem to include a TCP implementation according to [Ins81]. It didn't really look like we included anything to much. This discussion was the reason why we included a short description of every file.

To summarise we have no formal proof our analysis set is correct, but all empirically measured results and literature comparison strongly suggests it. We also did not find any indication contradicting the selection. So we decided that the given proof provided enough certainty to move and use this analysis set.

Chapter 5
Code Quality Metrics

5.1 Why Code Quality?

Why do we look at code quality metrics? Why do we care for code quality? You could object that this is all wasted time and purely academic. Well it isn't there exist several arguments and evidence for paying attention to code quality.

1. Teamwork
   Well structured and understandable code makes it much easier for a new colleague or open source project to join and be productive.
2. Security
   While there is no hard evidence for it, there is a lot of evidence suggesting that many critical security vulnerabilities would never have grown in clean code. One of the best and most recent examples for this is the heart bleed bug of OpenSSL which grew in a very unclean code.[Kam14]
3. Software Maintenance Cost (Bug Fixing and Adding of New Features)
   Sun once did an internal study of their software project costs and coding conventions. They found the following results[Mic]:
   - 40%–80% of the lifetime cost of a piece of software goes to maintenance
   - Hardly any software is maintained for its whole life by the original author.
   - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

   Robert Glass notes similar results[Gla02].

Code quality is a complex issue with lots of facets. In this work we primarily look at:

- Code organisation (How long are the files, how good is functionality modularised)
- Function length and Complexity (LOC and cyclomatic complexity)
- Comment density (Function descriptions as well as comments in functions)

5.2 Static Code Analysis Tools Evaluation

5.2.1 Static Code Analysis Tools in General

When we looked for static code analysis tools wikipedia occurred as a good starting point to get a overview[5.1]. The wikipedia list of tools for static code analysis contained over 24 different tools. After looking closer or trying out many of them we would suggest to roughly group them into the 3 groups displayed in figure 5.1.

5.1. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

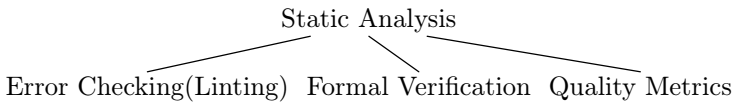

Figure 5.1. The three Categories of Static Analysis Tools

Which are:

Error Checking(Linting).
These tools check the code for potential errors and potential causes for trouble, way stricter than a compiler. These are infos like:

- Value stored to 'x' is never read
- The right operand of '!=' is a garbage value
- Potential leak of memory pointed to by 'x'
- Function call argument is an uninitialised value
- Use of memory after it is freed

Examples for tools like this are:

- clang analyser
- splint
- sparse
- cppdepend

Formal Verification.
These tools make it possible to prove the correctness of a program regarding a specification. Which means, they prove that the program really does x under given well defined input and circumstances. The problem with these tools is that you have to know and specify exactly what the program should do, and what these preconditions are. They also require additional work and code annotations in many cases. Example of such tools are:

- Frama-C
- PolySpace
- Eclair
- related: Isabelle

Code Quality Metrics.
These tools generated quantitative data regarding the code quality. These include metrics like:

- Comment density
- Cyclomatic Complexity
- Lines of Code per Function
- Halstead Complexity

Examples are:

- cccc[5.2]
- The method used by in "A Tale of Four Kernels"[Spi08]
- cmetrics[5.3]

5.2. http://cccc.sourceforge.net/

5.3. https://github.com/MetricsGrimoire/CMetrics

- cqmetrics[5.4]

Please note that this grouping is rough and incomplete.We only noted a few examples for each category and some of the tools noted here fit into several categories. There also existed more more fine grained categories depending on how close you want to look these tools.

Since we had decided to compare code quality of implementations only tools of the last category were relevant for us.

5.2.2 Code Quality Metrics Tools

Our next step was to test and compare the different available tools. Our experiences were:

cccc.
CCCC - a code counter for C and C++ was easily to obtain since it's packaged in most major GNU/Linux distributions. It also generates nice HTML pages. But it failed to recognise many function in the Linux kernels and did not display them in the result HTML pages. It also reported a lot of parse errors. Additionally the data was difficult to extract to generate further statistics therefore we decided not to use cccc.

"A Tale of Four Kernels" Method.
The method used by Diomidis Spinellis in his paper heavily relies on his tool cscout. While it produced good result from its description, we got the impression it is also quite laborious and needs background knowledge on the cscout tool. Additionally it seemed to need several additional bash and Perl scripts as glue and a SQL database to setup. All this may pay for examination for a very large code base but for our small scope we decided against it.

cmetrics.
Cmetrics is a small tool set consisting of C and shell scripts. While the result output looked promising, it could not parse many functions of Linux tcp therefore we decided not to use it.

cqmetrics.
cqmetrics is a C++ tool written by the author of the "A Tale of Four Kernels" paper. After self compilation it quickly generated statistics in the way we wanted them. Unfortunately cqmetrics only generates full per-file metrics but not per function. This makes it difficult to get a differentiated picture of the deviation or to reason about single functions.

In the end we chose cqmetrics and also constructed a way to get per function metrics. Which is described in the next section.

5.3 How we Obtained The Results

5.3.1 Command Usage

After building cqmetrics its usage is quite simple:

```
cat tcp.c | qmcalc > qmcalc_results.tsv
```

The result is a file of tab separated values, which by itself are quite difficult to interpret. Fortunately the repository contains a header line which can be prepended using:

```
cat header.tab qmcalc_results.tsv > qmcalc_results_+_header.tsv
```

Now we see the semantics of the results including:

- cyclomatic complexity
- Function Length

5.4. https://github.com/dspinellis/cqmetrics

- Usage of preprocessor directives
- and many more

For a detailed listing of the output metrics see the `README.md` file in the cqmetrics repository.

5.3.2 About the Output Format

.tsv is quite a low level format, the results looked like the following (abbreviated)

```
 nchar   nline   line_length_min line_length_mean
188562   5526    0      33.1227 29
```

The lines are displaced because one tab character not always has the same length as the description word in the header.

Fortunately we found a good tool to comfortably work with this data. gnumeric can directly open these files via `gnumeric qmcalc_results+header.tsv` and comfortably allows working with them or generating more sophisticated statistics.

5.3.3 Getting per Function Metrics

With the file format handled there remains only one last problem, how do we get per function metrics to get a differentiated picture of the deviation or to reason about single functions. To accomplish this we wrote a little Perl script which breaks a .c file apart into all single functions using a regular. Although in theory C grammar is not parse-able with regular expression in our case this worked trouble-free. Perhaps because both project adhere to a clear code style regarding opening and closing braces in the first column.

This functions are then fed separately into qmcalc. The qmcalc output is then (with the function an the file name) appended to our result file. So qmcalc is invoked for every function, but this was not really a performance problem, even for Linux TCP with its 409 functions this ran in under 2 seconds. This Perl script also adds the header to the final result.

The command to ran qmcalc with this Perl wrapper was:

```
qcmetrics_wrapper.pl tcp.c > results_+_header.tsv
```

The full Perl script can be found in the accompanying data in the folder `cqmetrics_wrapper/`

Our final result tsv file contained a slightly modified header and much more lines, one line for every function. The result files lie in `tcp_implementations/$implementation/results/` this folder also contains the later unused results generated with `cccc` and `cmetrics`. These result files were the starting point for our further analysis and interpretation, covered in the next section.

5.4 Results

5.4.1 Directory Structure

The point of a good directory structure is is to semantically group things together, to make finding and orientation in a big project easier. This is helpful for new colleagues or programmers, but even for experienced programmers if they move to another sub domain in a big project. Directories are one kind of modularity in C projects, besides compilation units.[Spi08]

As you may remember from from searching the LWIP TCP Code in section 4.2.1 the LWIP directory structure was clear and well arranged. The functionalities where well grouped and code could be found fast and straightforward. Many Folders contained a text file FILES giving more info about the files in this directory. This all may be a side effect of LWIP being quite small in general but nevertheless LWIP made a good impression here.

We can't say the same for Linux[5.5]. The `ipv4/` subfolder contains 94 files, with 30 of them TCP related. Additionally the TCP Code belongs to two different groups: a) tcp-core and b) congestion-control algorithms. This all make the `ipv4/` directory a bit unclear. Putting the 30 TCP related files into a own `tcp/` sub directory with a own `cc-algos/` sub directory would be possible way of improvement here.

5.4.2 File Lengths

C Files along with directories and functions are one of the few ways to establish modularity and separation in C projects[TM10]. Therefore looking at them can tell you a lot about a c-projects modularity. For good Modularization it's important to choose the modules not to be too big, but also not as too small. But what are appropriate sizes for C files? It's difficult to find a definitive recommendation in the literature. D. Spinellis says

> Most files are less than 2000 lines long. Overly long files (such as the C files in OpenSolaris and the WRK) are often problematic because they can be difficult to manage, they may create many dependencies and they may violate modularity.[Spi08]

And then goes on by just using "lesser is better" in the following analysis. We would like to establish a limit of 2500 lines[5.6] here. And define files over 2500 lines as "perhaps problematic" besides they have a good reason for.

5.4.2.1 LWIP

Let's start with LWIP and count the lines using the UNIX command `wc -l`

```
> wc -l *.c | sort -n

 1616 tcp_out.c
 1813 tcp_in.c
 2097 tcp.c

 5526 total
```

Figure 5.2. File Lengths in the LWIP TCP Implementation

So again, LWIP scores quite good here.

5.4.2.2 Linux TCP

The same procedure for Linux TCP in figure 5.3:

```
> wc -l *.c | sort -n

  109 tcp_recovery.c
  327 tcp_fastopen.c
  450 tcp_cong.c
  707 tcp_timer.c
  836 tcp_minisocks.c
 1186 tcp_metrics.c
 3334 tcp.c
 3578 tcp_output.c
 6366 tcp_input.c

16893 total
```

Figure 5.3. File Lengths in the Linux TCP Implementation

5.5. See section 4.3.1 for reference

5.6. This number was established by a small survey of our colleagues and fellow hackers. Nevertheless it's just a proposal, no established standard.

Of the 9 analysed files 6 are quite small and only implement specialised features, which is good and nicely organised. Nevertheless there are 2 somehow problematic files (`tcp.c` and `tcp_output.c`) and problematic file `tcp_input.c`.

It's interesting that processing incoming tcp segments requires about double the code of processing outgoing tcp segments. This can perhaps be, because processing incoming segments also includes a lot of:

- Parsing
- Decision making on what to do with the segment
- Verifying sequence numbers
- and perhaps some extended TCP features like TCP Sack or Fack

Nevertheless we decided to give the file `tcp_input.c` a look using the statistics we obtained for the closer function analysis. We filtered these to only contain functions in `tcp_input.c` and discovered something: 22 functions[5.7] there contain the string `sack` in them, as in selective Acknowledgements one of the TCP Extensions implemented in Linux TCP [MMFR96][Pro15]. They comprise a semantic group good to be put into a own file like `tcp_sack.c`. The same goes for parsing, or ECN.

5.4.3 Function Lengths

Overly long function are bad for readability, understanding and maintainability.

When talking about function lengths the same thoughts about modularization and scope as for file lengths apply. Additionally now we have a standard! At least for Linux, because the Coding Style file in the Linux tree[cod] gives us something to work with:

> `Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.`

So now we have a concrete number: 48 lines. He then goes into one exception:

> `The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's OK to have a longer function.`

Which makes sense. We will consider this when we discover a longer function. What Linus describes here as "complexity and indentation level" fits well with what we note as "cyclomatic complexity in the next section.

5.4.3.1 LWIP

First some general stats about the functions in the LWIP TCP code:

Number of Functions	79
Longest Function (Lines)	692
Shortest Function (Lines)	5

Table 5.1. General Statistics About The Functions in LWIP TCP Code

The longest function looks quite worrisome, but let's look at the distribution of function lengths first:

5.7. See Appendix A

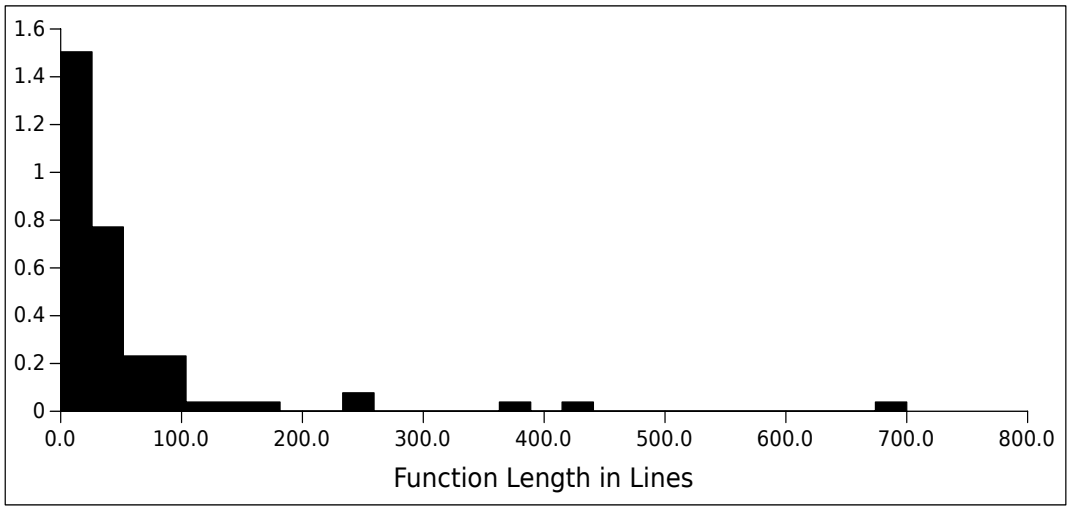

Figure 5.4. LWIP: Histogram of Function Lengths

In figure 5.4 the histogram bins have a width of 25 , meaning, the first bar contains all functions with a length in $[0, 25]$ the second all functions with a length in $]25, 50]$, etc. The number on the y-Axis is the relative density, i.e. the number of functions in this bin, divided by the width of this bin (i.e. 25).

In this figure we recognise several facts:

1. Most of the functions are shorter than 50 lines (59 functions of 79)
2. There are some black sheep with lengths between 200 and 500 lines
3. There is one black sheep with a length of almost 700 lines (697 to be precise)

The high number of functions with length below 50 lines is pleasant. Nevertheless 697 is quite long, lets take a short look at this function it's `tcp_receive()` from `tcp_in.c`.

To better understand the meaning of `tcp_receive()` a comment from the header of `tcp_in.c` may be helpful:

```
/**
 * @file
 * Transmission Control Protocol, incoming traffic
 *
 * The input processing functions of the TCP layer.
 *
 * These functions are generally called in the order (ip_input() ->)
 * tcp_input() -> * tcp_process() -> tcp_receive() (-> application).
 *
 */
```

So `tcp_receive()` is called quite late in the TCP processing path. And from the Doxygen comment describing `tcp_receive()`

```
/**
 * Called by tcp_process. Checks if the given segment is an ACK for outstanding
 * data, and if so frees the memory of the buffered data. Next, it places the
 * segment on any of the receive queues (pcb->recved or pcb->ooseq). If the
 * segment
 * is buffered, the pbuf is referenced by pbuf_ref so that it will not be freed
 * until
 * it has been removed from the buffer.
 *
 * If the incoming segment constitutes an ACK for a segment that was used for RTT
```

```
 * estimation, the RTT is estimated here as well.
 *
 * Called from tcp_process().
 */
```

So `tcp_receive()` does **a lot**. Completely different from Torvalds advice `"be short and sweet, and do just one thing"`. But why? Well, one reason could be, that the LWIP authors completely forgot all good design rules and this function just got out of hand. We don't believe this, simply because all the other code is structured and documented so well. So why then? This function lies on a hot code path and is called for almost[5.8] every incoming TCP packet, so for performance? Perhaps, but then, usually functions can be split up without any performance penalty when every split up function uses the `inline` keyword. But why is this not done here? A short search over the whole LWIP code showed that LWIP never uses the inline keyword, on no single function, so perhaps this is some internal standard. And because of this standard they had to keep `tcp_receive()` that long. But don't forget that's just our hypothesis, we don't know the real cause.

5.4.3.2 Linux

Let's start with the same general statistics as stated for LWIP:

Number of Functions	408
Longest Function (Lines)	336
Shortest Function (Lines)	4

Table 5.2. General Statistics About The Functions in LWIP TCP Code

So here we have a black sheep too. Well maybe not as black as the previous one rather a dark-gray sheep. But let's look at the distribution:

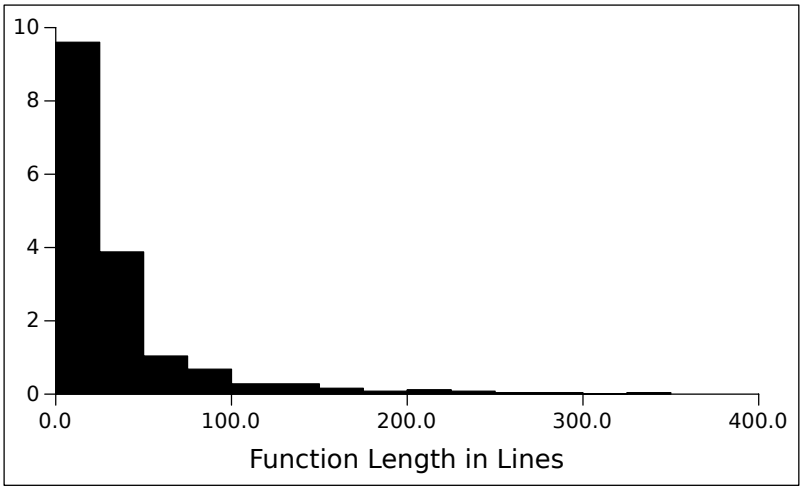

Figure 5.5. Linux TCP: Histogram of Function Lengths

In figure 5.5 the histogram bins have a width of 25 , meaning, the first bar contains all functions with a length in $[0, 25]$ the second all functions with a length in $]25, 50]$, etc. The number on the y-Axis is the relative density, i.e. the number of functions in this bin, divided by the width of this bin (i.e. 25).

From this graph we see several things:

1. Most of the functions are shorter than 50 lines, which is good. (337 from 408, so over $3/4$)
   Probably a lot of them are small accessor and converter expressions, which would otherwise be directly used. Putting such expressions in a own function is good because it adds a lot semantic about what this code does, which is not contained in the pure expression.

5.8. it's not called for incoming packets that are empty ACKs

2. There are some gray sheep around 200 lines
3. And one (barely visible) gray sheep at 325-350 (336 lines to be precise)

So in general the distribution is quite similar, with the difference that the black sheep of Linux is less black.

But lets take a look at this black sheep it's `tcp_recvmsg()` in `tcp.c`. It has a short header comment saying:

```
/*
 *      This routine copies from a sock struct into the user buffer.
 *      [...]
 */
```

Copying data from the socket to the user buffer happens at the very end of TCP processing. Technically it happens even after TCP processing. Since TCP processing ends and succeeds when the unboxed data has been placed in the socket's buffer. `tcp_recvmsg()` is called independently, from the `read()` syscall in the user space.[Ros13]

So `tcp_recvmsg()` does something different, from the black sheep in LWIP. This is a bit disappointing, because if two functions doing roughly the same would be very long that would indicate there is a reason for their size.

But let's discuss a bit about `tcp_recvmsg()`, looking at it's full head:

```
int tcp_recvmsg(struct sock *sk, struct msghdr *msg, size_t len, int nonblock,
                int flags, int *addr_len)
```

we see that it takes 6 arguments, which is also no good sign [Spi08]. Here we think this function is historical inherited bad code, because in Linux `static inline` functions[5.9] are available and quite common, They are also used in this very file. Since this function uses 7 goto labels, it would be possible to put at least some of this label specific code and put it into a own inline function. It would be possible that we're overlooking something here (cache bloat of inline functions? [cod]) but currently that would be our measurement.

5.4.4 Cyclomatic Complexity (CC)

Another more sophisticated tool for understanding the complexity of a function is the "cyclomatic complexity" or "McCabe Complexity". It is the number of independent control flow paths in a function.[McC76] So for example the following function from the Linux kernel has a CC of 1

```
static inline bool forced_push(const struct tcp_sock *tp)
{
        return after(tp->write_seq, tp->pushed_seq + (tp->max_window >> 1));
}
```

and this one has a CC of 4:

```
void tcp_shutdown(struct sock *sk, int how)
{
        /*      We need to grab some memory, and put together a FIN,
         *      and then put it into the queue to be sent.
         *              Tim MacKenzie(tym@dibbler.cs.monash.edu.au) 4 Dec '92.
         */
        if (!(how & SEND_SHUTDOWN))
                return;

        /* If we've already sent a FIN, or it's a closed state, skip this. */
```

5.9. static before a function means "private to this compilation unit". Actually the static keyword in C is a bit confusing since it's meaning depends on whether it is placed in global or function local scope.

```
        if ((1 << sk->sk_state) &
            (TCPF_ESTABLISHED | TCPF_SYN_SENT |
             TCPF_SYN_RECV | TCPF_CLOSE_WAIT)) {
                /* Clear out any half completed packets.  FIN if needed. */
                if (tcp_close_state(sk))
                        tcp_send_fin(sk);
        }
}
```

because there are four possible ways to go through this function.

From this little example it may also be obvious that functions with a lower CC are easier to understand. And since difficult to understand functions mean havoc for maintainability a lower CC is better for maintainability[Spi08]. But is there a sensible limit to not overstep for a function to be good. Perhaps. In a second paper the inventor of this metric McCabe recommends to use a limit of 10 during development. Every time a function oversteps this it should be split up, or if there is a good reason for overstepping, at least add a comment why.[MMAS83]

So with this background established lets look at the two TCP implementations:

5.4.4.1 LWIP

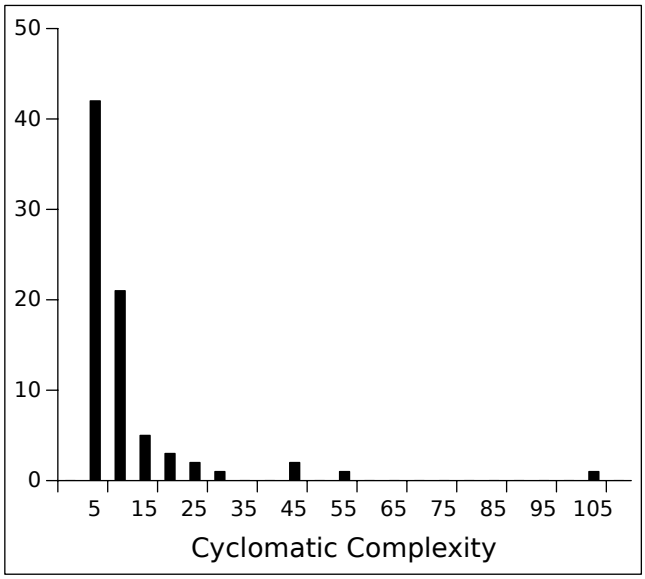

Figure 5.6. Histogram of the Cyclomatic Complexity distribution of the LWIP TCP Functions

In figure 5.6 we chose a width of 5 for each bin, so the first bar contains all functions with a CC in $[0, 5[$, the second all functions in $[5, 10[$, etc.

Please note the following:

- For one function `tcp_input()` our tool could not calculate the cyclomatic complexity. So we are missing an value here. `tcp_input()` is the second longest of all functions in LWIP TCP with a length of 428 Lines. So we estimate it probably has a high CC too perhaps around 80. But since that is a pure guess we decided not to take it into our results

These results tell us:

1. The majority (63 of 79) of the functions lies in the sweet spot of CC $\in[0, 10]$, which is great
2. There are a few (3) black sheep with CC values between 40 and 55
3. There is one very black sheep with a cyclomatic Complexity of 104

Interestingly this is the same black sheep as before when we looked at function lengths namely: `tcp_receive()`. Which makes this function look problematic once more.

5.4.4.2 Linux TCP

Now let's look at the distribution of cyclomatic complexity in the Linux kernel:

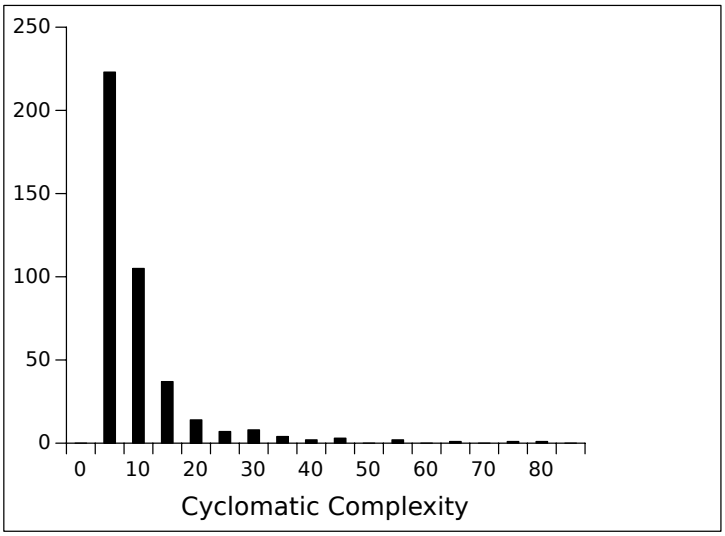

Figure 5.7. Histogram of the Cyclomatic Complexity Distribution of the Linux TCP Functions

In figure 5.7 the width of each histogram bin (and starting point, etc.) is exactly the same as in the LWIP cc histogram (figure 5.6)

These results tell us:

1. The CC distribution is very similar to the LWIP CC distribution
2. Most functions (328 of 408) lie in the sweet spot of CC $\in [0, 10]$
3. There are a few black sheep
4. And 3 very black sheep (the very small bars between 60 and 80, barely visible). But
    a. These 3 are less black than the very black sheep of LWIP (63,71,78 instead of 104)
    b. Linux TCP is comprised of 408 functions, LWIP TCP of 79. Which is roughly 5 to 1. So while Linux TCP contains 5 times the functions LWIP contains, it only contains 3 times the "very black sheep" LWIP contains. So Linux TCP has "percentile less black sheep".

To summarise regarding cyclomatic complexity Linux scores comparable to LWIP but has its worst functions scoring better than the worst functions of LWIP.

5.4.5 Comment Density

Since LWIP TCP contains Doxygen Comments for every function and Linux does this for a few we decided that only comparing overall comment lines would be unfair. Therefore we added a second sub-study comparing the amounts of function internal comments.

5.4.5.1 In Function Comments

Another common measurement for Code Quality is comment density, the quotient of comment chars vs. statements [Spi08]. While it's quite difficult to quantify and measure the quality of comments, it's quite simple to measure their existence and frequency.

While we studied the comment density we realised that it's value vary significantly. These variations depend on function length in LOC. A lot of short functions contain no comments at all, which is not really bad from a code quality point of view. The inner workings of short functions like `tcp_setprio()` (figure 5.8) are understandable without a comment.

```
void
tcp_setprio(struct tcp_pcb *pcb, u8_t prio)
{
  pcb->prio = prio;
}
```

Figure 5.8. tcp_setprio() in LWIP tcp.c

Some further semantics of the internal comment density:

1. In general LWIP scores better here containing more inline comment characters than Linux
2. The plottings indicate a loose constant comment density with lots of outliers. Since they did not contain any more interesting information we chose to omit them.
3. Linux includes some bad examples of over 100 lines with no or almost no comments.
4. In Linux TCP the comment density is varying more significantly.

5.4.5.2 In Function Comments and Function Descriptions (All Comments)

A summarising measurement over all functions in all files gives the following results:

	Linux TCP	LWIP TCP
Comments per Function	2.52	8.70
Doxygen Comments per Function	0.026	1.12
Average Comment Length (Chars/Comment)	149.8	99.9

Table 5.3. Overall Comment Density Statistics

This indicates several results:

1. LWIP has a very good Doxygen coverage covering every single function[5.10], some have even two Doxygen comments. Linux has its own Doxygen like system[5.11] but scores quite bad here.
2. In general LWIP contains more but shorter comments per function. While this may be subjective we assume that many short comments are better for comprehension than a few long once.

In summary the LWIP TCP implementation contains better documentation than Linux TCP. This result hardens if you compare the comprehensive Doxygen documentation and the wiki (containing additional conceptual documents) to the few conceptual files in the Linux `Documentation` folder, or the Linux auto generated restructured Text Documentation.[5.12]

5.5 A Short Qualitative Evaluation of the Header Files

A short manual evaluation of the header files revealed that they are of good quality in both projects. Function prototypes were noted consistently in both projects. Regarding data structure definitions LWIP and Linux score comparably good. Every strict member has a short comment explaining its meaning in both projects.

5.10. We verified this manually by looking through the LWIP code and could not find a single function without a Doxygen comment.

5.11. It's called kernel-doc. It consists of Doxygen like comments above functions which are processed by a Perl script for further usage in man pages or other documentation.

5.12. The Linux auto-generated documentation (via `make htmldocs` for example), like the LWIP documentation contains some additional info on TCP, but way less.

5.6 Lines of Code and Cyclomatic Complexity - A Correlation?

You may have realised too, if you look at the previous histograms for function length and cyclomatic complexity of one implementation, they look pretty similar. Almost like this two parameters were somehow related or correlated.

This suspicion hardened when we where looking at the raw data set in our spreadsheet software, suggesting many examples of correlation. Especially when we sorted the functions, first by Lines of Code, then by cyclomatic complexity, the order was almost the same only, a few entries changed places. Also functions with low CC also were always very short, as well as all long function had a CC.

Finally, the data hardening this suspicion was so much, so we decided to look if someone else has already looked into this correlation. And we were successful:

In 2009 Jay, Hale, and Smith measured about 1,2 Million source code files and created comprehensive evidence and proved it: There is a stable linear relationship between cyclomatic complexity and lines of code. They also go one step further by concluding:[GHS+09]

> `We conclude that CC has no explanatory power of its own and that LOC and CC measure the same property.`

Which is a very practical and way to little known fact worth spreading. Knowing this simplifies code quality analysis, since obtaining the length of functions is way easier than obtaining the cyclomatic complexity. For a small code base this way a estimation is possible even without using any specialised tools.

Chapter 6
A short Discussion of Data Structures

In our last chapter let's take a look at one of the central data structures of every tcp implementation (and every network stack): the data structure for a packet.

6.1 The LWIP Packet Data Structure (PBUF)

The LWIP packet data structure is the `pbuf`, packet buffer. It's surprisingly simple and contains only 7 members. It's also well documented as visible in figure 6.1.

```
/** Main packet buffer struct */
struct pbuf {
  /** next pbuf in singly linked pbuf chain */
  struct pbuf *next;

  /** pointer to the actual data in the buffer */
  void *payload;

  /**
   * total length of this buffer and all next buffers in chain
   * belonging to the same packet.
   *
   * For non-queue packet chains this is the invariant:
   * p->tot_len == p->len + (p->next? p->next->tot_len: 0)
   */
  u16_t tot_len;

  /** length of this buffer */
  u16_t len;

  /** pbuf_type as u8_t instead of enum to save space */
  u8_t /*pbuf_type*/ type;

  /** misc flags */
  u8_t flags;

  /**
   * the reference count always equals the number of pointers
   * that refer to this pbuf. This can be pointers from an application,
   * the stack itself, or pbuf->next pointers from a chain.
   */
  u16_t ref;
};
```

Figure 6.1. The Definition of the LWIP PBUF structure in `include/lwip/pbuf.h`

Some explanations:

1. One network packet can be stored in several `pbuf`s, which then are stored together in a `pbuf chain`. A `pbuf chain` is a singly linked list of `pbuf`s. This is what the first member the `next` pointer is intended for.[lwia]
2. `payload` points to the actual packet data. It's type is a void pointer so it can easily be casted to every other type, which becomes necessary when parsing the data.
3. About data types: `u8_t` is a 8 bit unsigned integer, `u16_t` a 16 byte unsigned integer etc.
4. For further details of the `pbuf` structure see the `infrastructure > pbuf` section in the lwIP Doxygen documentation[lwia]

Besides the struct definition there are also the prototypes of several `pbuf` accessor function listed in `pbuf.h`, giving this data structure an nice encapsulated API. Since they have descriptive signatures like:

- `struct pbuf *pbuf_alloc(pbuf_layer l, u16_t length, pbuf_type type);`
- `err_t pbuf_copy(struct pbuf *p_to, const struct pbuf *p_from);`

we decided to not describe them any further. In addition all `pbuf` API functions are well documented in the LWIP Doxygen documentation[lwia].

6.2 The Linux Packet Data Structure (SKB)

The naming for the Linux packet data structure is a bit confusing:

1. In prose texts it's called `Socket Buffer`
2. The C datatype is `struct sk_buff`
3. The corresponding header file is `skbuff.h`
4. All the variables referencing it are called `skb`

Which means we have almost every permutation of the name covered. Great! In this text we will stick to the name `skb`.

Compared to the lwIP `pbuf` the `skb` is overwhelmingly complex, containing over 60 members. Since it's definition in `include/linux/skbuff.h` is over 150 lines long we decided to not print it here, but in a special Appendix, Appendix B.

Some explanations:

1. In every place where `skb`s get stored and handled (like the input/output queues of sockets or NIC drivers), they are handled as double linked list.[Ros13]

   This is why there are the `*next` and `*previous` pointers it the beginning of the data structure.

   It is interesting to note that the kernel has a own double linked list data structure, which every "things-which-wants-to-be-in-a-list" should use according to convention[Lov10], but `skb`s don't. This is perhaps because `skb`s are way older than this new official data type.
2. `skb` also contains function pointers like `void (*destructor)(struct sk_buff *skb)` which lift this data structure in a entire new complexity range, not only containing data definitions but also pointers to variable logic.
3. `skb` also contains a lot more information about internal packet structure. This includes information about protocol headers (transport and network protocol headers) or VLAN headers. Regarding this `skb` uses a completely different approach than LWIP, which only contains an opaque `payload` void pointer.

Regarding accessor functions `skb` is comparable to `pbuf`.

6.3 Comparative Thoughts

Our comparison starts with some quantitative data, shown in table 6.1.[6.1]

	LWIP TCP	Linux TCP
Members	7	>60
Approximate Memory Size (Assuming a 64 bit CPU for Linux)	64 bit + 2 pointers	1000 bit + 7 pointers
Contains packet payload	no	no
Contains pointers to net device and socket	no	yes
Contains list handling pointers	yes	yes
Contains packet structure info	no	yes
Contained function pointers	0	3

Table 6.1. Quantitative Data on the Packet Data Structures

As we see in table 6.1 the `skb` data structure is way larger than the `pbuf`, in terms of member count as well in term of memory size. While the difference of about 1000 bit to 64 bit seems extreme you should take a closer look at the members. Many of the `skb` members are necessary like the "timestamp", queuing infos, VLAN tags, or the checksum. So while some of this members are only a result of Linux TCP having more features than lwIP TCP, a lot of them is data lwIP has to store somewhere too. Having them elsewhere does not really save memory at run-time at all, it only means storing them elsewhere, like in the payload or in some separate (maybe global) variables. So this memory difference isn't as extreme as it seems.

Having all this related infos as separate members of the struct is also a good thing in terms of software architecture. It means having a clear and comfortable abstraction and data type. While having these infos in external variables is a leaky abstraction. Since this can totally make sense when working single threaded and being lightweight, this may be was a deliberate performance vs. architecture trade-off.

Regarding Code Quality both data structures share a high standard. Both are well documented and formatted. The extreme length of `skb` (over 150 lines) could be considered a problem, but since this is a central highly performance relevant structure it's quite likely this is a well considered design decision.

Regarding performance we should note that most Linux accessor functions are inline functions (and usually only 2-3 lines long), which saves some small function call and return overhead. The lwIP functions aren't since they don't use inline in their project, so this may make a small performance difference every time a packet is accessed, which happens often, because everything a network stack does is accessing and modifying packets.

To summarise we can say that the lwIP `pbuf` is much smaller than the Linux `skb` but also less sophisticated and more of a low level data structure. The memory size difference should not matter that much, because all the data contained in the additional `skb` fields has to be contained somewhere anyway. Like in external variables or the payload member.

6.1. The Linux memory size is approximated by counting the members sizes and assuming a Intel 64 bit CPU and the gcc type sizes. Where conditional compilation was relevant we decided to include 50 % of the ifdef guarded members. For LWIP counting was way simpler since most of the 7 members had hard-coded size.

Chapter 7
Conclusion

We comprehensively covered several aspects of the two TCP implementations. From features and user base, directory structure and code quality to the data structures of the packets.

Features.
In general, Linux TCP supports way more features, regarding several TCP extensions of the last 10 years, lwIP doesn't. But the most important additional features are multi-threading and firewall support. Since multi-threading changes several fundamental design concepts this difference was visible in several other points of our study. The only feature of lwIP Linux TCP hasn't is portability, while the lwIP stack plays with several operating systems or minimal embedded system configurations, the Linux TCP stack only plays with the Linux kernel.

User Base.
Both systems have huge real wold user bases. While the home of lwIP is the embedded world it is used in many small devices, as well as cars and air planes. The linux kernel is present in over 60% of all smart phones, most servers and also many embedded device. So almost omni present. Because of the high distribution of smartphones in all western and many non-western countries we currently assume the Linux TCP stack has a larger user base, but it's very difficult to reason about this.

Files and Directory Structure.
Recognising and isolating the TCP implementation was easy for lwIP, since it consists of only three files, and difficult for Linux. Since the Linux TCP implementation is more interwoven with the rest of the network stack or contain TCP extensions or additional pluggable congestion control algorithms. Our final set contained 9 C files and 4 header files for Linux and 3 C files and 3 header files for lwIP. We also gave a short description of the functionality and meaning of every file for the implementation.

Code Quality.
First we evaluated code quality metric measurement tools. The tooling situation here is surprisingly bad, finally we chose `cqmetrics` and a own wrapper script.

Our subsequent study revealed several results grouped by category:

Code Organisation.
In lwIP the code is well organised into subdirectories where everything can easily be found. This isn't the case for Linux, the `ipv4/` folder contains 94 files.

In lwIP all files are below 2100 lines which is good. In Linux 2 files are over 3000 Lines long which is not so good, speaking of modularization.

Function Length and Complexity.
In general both implementations score very good here with most functions being und 40 lines long. However both implementations contained black sheep, both responsible for processing incoming TCP packets. But while the Linux black sheep had somehow tolerable 336 lines, the lwIP black sheep had way worse 692 Lines.

The cyclomatic complexity (cc) graphs and results were very similar to the function length ones.

So we did some research and found out that in a huge recent study it was discovered that there is a direct linear correlation between lines and cc. So in most cases it is not necessary to even measure cc.

Comment Density.

lwIP contains Doxygen comments for every single function, Linux does for some (2.6%). Counting this and also the in-function comments lwIP TCP is well documented, while Linux TCP is under documented.

Data Structures.
The lwIP data structure for packets (`pbuf`) is very small and low level style (7 members) while the Linux data structure (`skb`) is quite big and more high level like (>60 members). The size difference in bytes should not matter though, because all the data stored in the additional members of `skb` has also to be stored somewhere in lwIP like in additional variables or the payload member.

7.1 Ending Thoughts

After ending this work there remain several conclusions:

1. About Learning TCP
   If you want to learn how to implement TCP read the lwIP source code it's small, neat and well documented
2. What did the Code Quality study unveil?
   For lwIP there is this one 692 lines function which perhaps should be refactored. For Linux there are several small things which could be done in small commits, like taking all the tcp files and putting them into one sub directory, or adding short describing headers to every tcp related file.
3. How helpful was the quality metrics methodology for understanding a new project?
   While the cyclomatic complexity was not helpful at all knowing the LOC of every function was. It gave a good overview over the project somehow telling which function is how difficult or relevant. So whereas the tooling for cyclomatic complexity measurement was horrible this was not really a problem. Since measuring the LOC of a function is not really difficult this could be a great addition to an IDE, including a small view which shows all the functions with and sorted by their lengths, instantly providing a better "feeling" for a project.

Appendix A
TCP SACK Functions in tcp_input.c

```
tcp_dsack_seen
tcp_sack_cache_ok
tcp_reset_reno_sack
tcp_dsack_extend
tcp_add_reno_sack
tcp_try_undo_dsack
tcp_sack_extend
tcp_limit_reno_sacked
tcp_check_sack_reneging
tcp_sacktag_skip
tcp_remove_reno_sacks
tcp_maybe_skipping_dsack
tcp_dsack_set
tcp_sack_maybe_coalesce
tcp_sack_remove
tcp_check_dsack
tcp_is_sackblock_valid
tcp_sack_new_ofo_skb
tcp_match_skb_to_sack
tcp_sacktag_walk
tcp_sacktag_one
tcp_sacktag_write_queue
```

Appendix B
The Linux Socket Buffer Data Structure

B.1 C Definition

From `include/linux/skbuff.h`:

```
struct sk_buff {
        union {
                struct {
                        /* These two members must be first. */
                        struct sk_buff          *next;
                        struct sk_buff          *prev;

                        union {
                                ktime_t         tstamp;
                                struct skb_mstamp skb_mstamp;
                        };
                };
                struct rb_node  rbnode; /* used in netem & tcp stack */
        };
        struct sock             *sk;
        struct net_device       *dev;

        /*
         * This is the control buffer. It is free to use for every
         * layer. Please put your private variables there. If you
         * want to keep them across layers you have to do a skb_clone()
         * first. This is owned by whoever has the skb queued ATM.
         */
        char                    cb[48] __aligned(8);

        unsigned long           _skb_refdst;
        void                    (*destructor)(struct sk_buff *skb);
#ifdef CONFIG_XFRM
        struct  sec_path        *sp;
#endif
#if defined(CONFIG_NF_CONNTRACK) || defined(CONFIG_NF_CONNTRACK_MODULE)
        struct nf_conntrack     *nfct;
#endif
#if IS_ENABLED(CONFIG_BRIDGE_NETFILTER)
        struct nf_bridge_info   *nf_bridge;
#endif
        unsigned int            len,
                                data_len;
        __u16                   mac_len,
                                hdr_len;
```

```
        /* Following fields are _not_ copied in __copy_skb_header()
         * Note that queue_mapping is here mostly to fill a hole.
         */
        kmemcheck_bitfield_begin(flags1);
        __u16                   queue_mapping;
        __u8                    cloned:1,
                                nohdr:1,
                                fclone:2,
                                peeked:1,
                                head_frag:1,
                                xmit_more:1;
        /* one bit hole */
        kmemcheck_bitfield_end(flags1);

        /* fields enclosed in headers_start/headers_end are copied
         * using a single memcpy() in __copy_skb_header()
         */
        /* private: */
        __u32                   headers_start[0];
        /* public: */

/* if you move pkt_type around you also must adapt those constants */
#ifdef __BIG_ENDIAN_BITFIELD
#define PKT_TYPE_MAX    (7 << 5)
#else
#define PKT_TYPE_MAX    7
#endif
#define PKT_TYPE_OFFSET()       offsetof(struct sk_buff, __pkt_type_offset)

        __u8                    __pkt_type_offset[0];
        __u8                    pkt_type:3;
        __u8                    pfmemalloc:1;
        __u8                    ignore_df:1;
        __u8                    nfctinfo:3;

        __u8                    nf_trace:1;
        __u8                    ip_summed:2;
        __u8                    ooo_okay:1;
        __u8                    l4_hash:1;
        __u8                    sw_hash:1;
        __u8                    wifi_acked_valid:1;
        __u8                    wifi_acked:1;

        __u8                    no_fcs:1;
        /* Indicates the inner headers are valid in the skbuff. */
        __u8                    encapsulation:1;
        __u8                    encap_hdr_csum:1;
        __u8                    csum_valid:1;
        __u8                    csum_complete_sw:1;
        __u8                    csum_level:2;
        __u8                    csum_bad:1;

#ifdef CONFIG_IPV6_NDISC_NODETYPE
        __u8                    ndisc_nodetype:2;
#endif
        __u8                    ipvs_property:1;
```

```
        __u8                    inner_protocol_type:1;
        __u8                    remcsum_offload:1;
        /* 3 or 5 bit hole */

#ifdef CONFIG_NET_SCHED
        __u16                   tc_index;       /* traffic control index */
#ifdef CONFIG_NET_CLS_ACT
        __u16                   tc_verd;        /* traffic control verdict */
#endif
#endif

        union {
                __wsum          csum;
                struct {
                        __u16   csum_start;
                        __u16   csum_offset;
                };
        };
        __u32                   priority;
        int                     skb_iif;
        __u32                   hash;
        __be16                  vlan_proto;
        __u16                   vlan_tci;
#if defined(CONFIG_NET_RX_BUSY_POLL) || defined(CONFIG_XPS)
        union {
                unsigned int    napi_id;
                unsigned int    sender_cpu;
        };
#endif
        union {
#ifdef CONFIG_NETWORK_SECMARK
                __u32           secmark;
#endif
#ifdef CONFIG_NET_SWITCHDEV
                __u32           offload_fwd_mark;
#endif
        };

        union {
                __u32           mark;
                __u32           reserved_tailroom;
        };

        union {
                __be16          inner_protocol;
                __u8            inner_ipproto;
        };

        __u16                   inner_transport_header;
        __u16                   inner_network_header;
        __u16                   inner_mac_header;

        __be16                  protocol;
        __u16                   transport_header;
        __u16                   network_header;
        __u16                   mac_header;
```

```
        /* private: */
        __u32                   headers_end[0];
        /* public: */

        /* These elements must be at the end, see alloc_skb() for details.  */
        sk_buff_data_t          tail;
        sk_buff_data_t          end;
        unsigned char           *head,
                                *data;
        unsigned int            truesize;
        atomic_t                users;
};
```

B.2 Struct Member Documentation

```
/**
 *      struct sk_buff - socket buffer
 *      @next: Next buffer in list
 *      @prev: Previous buffer in list
 *      @tstamp: Time we arrived/left
 *      @rbnode: RB tree node, alternative to next/prev for netem/tcp
 *      @sk: Socket we are owned by
 *      @dev: Device we arrived on/are leaving by
 *      @cb: Control buffer. Free for use by every layer. Put private vars
here
 *      @_skb_refdst: destination entry (with norefcount bit)
 *      @sp: the security path, used for xfrm
 *      @len: Length of actual data
 *      @data_len: Data length
 *      @mac_len: Length of link layer header
 *      @hdr_len: writable header length of cloned skb
 *      @csum: Checksum (must include start/offset pair)
 *      @csum_start: Offset from skb->head where checksumming should start
 *      @csum_offset: Offset from csum_start where checksum should be stored
 *      @priority: Packet queueing priority
 *      @ignore_df: allow local fragmentation
 *      @cloned: Head may be cloned (check refcnt to be sure)
 *      @ip_summed: Driver fed us an IP checksum
 *      @nohdr: Payload reference only, must not modify header
 *      @nfctinfo: Relationship of this skb to the connection
 *      @pkt_type: Packet class
 *      @fclone: skbuff clone status
 *      @ipvs_property: skbuff is owned by ipvs
 *      @peeked: this packet has been seen already, so stats have been
 *              done for it, don't do them again
 *      @nf_trace: netfilter packet trace flag
 *      @protocol: Packet protocol from driver
 *      @destructor: Destruct function
 *      @nfct: Associated connection, if any
 *      @nf_bridge: Saved data about a bridged frame - see br_netfilter.c
 *      @skb_iif: ifindex of device we arrived on
 *      @tc_index: Traffic control index
 *      @tc_verd: traffic control verdict
 *      @hash: the packet hash
```

```
*      @queue_mapping: Queue mapping for multiqueue devices
*      @xmit_more: More SKBs are pending for this queue
*      @ndisc_nodetype: router type (from link layer)
*      @ooo_okay: allow the mapping of a socket to a queue to be changed
*      @l4_hash: indicate hash is a canonical 4-tuple hash over transport
*              ports.
*      @sw_hash: indicates hash was computed in software stack
*      @wifi_acked_valid: wifi_acked was set
*      @wifi_acked: whether frame was acked on wifi or not
*      @no_fcs:  Request NIC to treat last 4 bytes as Ethernet FCS
 *     @napi_id: id of the NAPI struct this skb came from
*      @secmark: security marking
*      @offload_fwd_mark: fwding offload mark
*      @mark: Generic packet mark
*      @vlan_proto: vlan encapsulation protocol
*      @vlan_tci: vlan tag control information
*      @inner_protocol: Protocol (encapsulation)
*      @inner_transport_header: Inner transport layer header (encapsulation)
*      @inner_network_header: Network layer header (encapsulation)
*      @inner_mac_header: Link layer header (encapsulation)
*      @transport_header: Transport layer header
*      @network_header: Network layer header
*      @mac_header: Link layer header
*      @tail: Tail pointer
*      @end: End pointer
*      @head: Head of buffer
*      @data: Data head pointer
*      @truesize: Buffer size
*      @users: User count - see {datagram,tcp}.c
*/
```

Bibliography

[APS99] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control. 1999.

[BCV] Andrea Baiocchi, Angelo P Castellani, and Francesco Vacirca. Yeah-tcp: yet another highspeed tcp.

[BOP94] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[CCRJ14] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. Tcp fast open. RFC 7413, RFC Editor, December 2014. http://www.rfc-editor.org/rfc/rfc7413.txt.

[cod] Linux kernel documentation on coding style. Documentation/CodeStyle file in Linux source tree.

[GHS+09] JAY Graylin, Joanne E Hale, Randy K Smith, HALE David, Nicholas A Kraft, WARD Charles et al. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications*, 2(03):137, 2009.

[Gla02] Robert L Glass. *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2002.

[hit] Hitslink.com. "operating system market share". Http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8.

[Ins81] Information Sciences Institute. RFC 793. 1981. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html.

[Kam14] Poul-Henning Kamp. Please put openssl out of its misery. *Queue*, 12(3):20–20, apr 2014.

[lin] Linux-repo-cgit. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/refs/.

[Lov10] Robert Love. *Linux kernel development*. Addison-Wesley, Upper Saddle River, NJ, 2010.

[lwia] *LWIP Doxygen Documentation*.

[lwib] Lwip-repo-cgit. http://git.savannah.gnu.org/cgit/lwip.git/refs/.

[lwic] Lwip-wiki-lwipopts.h. http://lwip.wikia.com/wiki/Lwipopts.h.

[McC76] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[Mic] Sun Microsystems. "code conventions for the java programming language". *http://www.oracle.com/technetwork/java/index-135089.html*.

[MMAS83] T.J. MacCabe, McCabe, Associates, and IEEE Computer Society. *Structured testing*. Tutorial Texts Series. IEEE Computer Society Press, 1983.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. RFC 2018, RFC Editor, October 1996.

[Pro15] Linux Man Pages Project. *Linux Programmers Manuel - TCP (man 7 tcp)*. 12 2015.

[QK12] Jürgen Quade and Eva-Katharina Kunst. *Linux-Treiber entwickeln*. Dpunkt Verlag, 2012.

[RKL88] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[Ros13] R. Rosen. *Linux Kernel Networking: Implementation and Theory*. Books for professionals by professionals. Apress, 2013.

[San08] Kaushal Sanghai. Building complex vdk/lwip applications using blackfin processors. *Analog Devices Inc*, 2008.

[Spi08] Diomidis Spinellis. A tale of four kernels. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 381–390. New York, may 2008. Association for Computing Machinery.

[Ste93] W. Richard Stevens. *TCP/IP Illustrated (Vol. 1): The Protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[SV09] Sameer Seth and M Ajaykumar Venkatesulu. *TCP/IP architecture, design and implementation in Linux*, volume 68. John Wiley & Sons, 2009.

[Tan03] Andrew S Tanenbaum. Computer networks, 4-th edition. *Ed: Prentice Hall*, 2003.

[TM10] Neil Thomas and Gail Murphy. How effective is modularization. *Making Software: What Really Works, and Why We Believe It*, page 373, 2010.

[Vel08] Siva Velusamy. Lightweight ip (lwip) application examples. 2008.

[WS95] Gary R Wright and W Richard Stevens. *TcP/IP Illustrated*, volume 2. Addison-Wesley Professional, 1995.

About the Sources Used in this Work

This work uses, additionally to conventional sources like books, articles and web sources, two rather unconventional sources which are:

1. The auto-generated doxygen documentation in the LWIP repository

2. Source Code files

Both additional sources deserve a little annotation and comment:

Auto-generated doxygen documentation of LWIP 2.0.0 [lwia] .
This source is neither a classical written source, nor a web source.

It is generated from the coments in the LWIP source code files and some extra text files in the LWIP directory. Since it contains a lot of helpful and comprehensive information, we chose to use it.

In the attachment to this work, you can find a folder called `lwip-doc`. In this folder there is the sub-subfolder `doxygen/output/` in which you can find the file `index.html`

If you open this file with your favorite web-browser you can browse through the whole lwip doxygen documentation, which is quite comprehensive. It contains information of alle imporntant data structures, almost every function and several important concepts.

We refer to this documentation using the bibtex sigil [lwia]

Source Code files.
In several places we argue and reason using source code files.

Since using the bibliography to reference these source would be quite impractical we decided to indicate the concrete filename directly in this places using `verbatim font`.

All the source files used (and probably some more) are also part of the attachment of this work, in exactly the version we used.