Tracing the Way of Data in a TCP Connection through the Linux Kernel

Seminar Organic Computing

Richard Sailer
Matrikelnummer: 1192352

Universität Augsburg
Lehrstuhl für Organic Computing
richard.willi.sailer@student.uni-augsburg.de

Abstract

TODO

1 Motivation and Introduction

Linux kernel programming is complex and difficult for most students or people with experience in application programming. The C programming language lacks many of the high level features these people are used to and even common C libraries are not accessible in kernel code, actually no C libraries at all is available in kernel space[3].

Aditionally the kernel is a complex piece of software consisting of many modules working together in non-trivial ways. To change or extend a part of the kernel semantic knowledge on how and why this parts work together is necesarry.

While the first two obstacles, the often unfamiliar C programming language and the lack of libraries, can be overcome with some experience and get smaller and smaller after some time, the possibility of understanding the in-kernel mechanics is an question of good documentation.

While there exist several good books on this topic, all of them have one of the following three problems:

1. They grow old very fast

   Or put another way: The linux kernel evolves too fast. For example UNDERSTANDING LINUX NETWORK INTERNALS by Christian Benvenuti[1], a really comprehensive books, was written in 2006 covering linux kernel 2.6. Since it took some time writing some examples and parts cover even older parts, like the bottom halve interupt handling. Many details in the tcp stack or the interupt handling have changed since. At the time of writing the current version of the linux kernel is 4.3

2. They have a certain focus and do preselection of content

   Since the linux kernel is a very large project consisting of over 19 Million lines of code[1] obviously a preselection is necesarry even if the author is focussing on a subsystem. For example in "Linux Kernel Networking" by Rami Rosen[5] TCP is covered quite brief since the lower networking layers get a much deeper coverage.

3. They are not freely available

   Most of these books have to be purchased and ordered. Some of them are quite expensive and ordering them can take several days. These two facts make it more difficult for interested programmers to start linux kernel programming.

This seminar paper tries to solve two of these three issues. In this paper we will try to provide an free (licensed under the GNU Free Documentation License) and up to date overview of one very specific topic: The Way of Data in a TCP Connection through the Linux kernel. Aditionally we will introduce and explain the method used to examine and understand these kernel mechanics in order to give every reader a tool helpful in understanding other parts of the kernel his or herself.

1. Version 4.1, see http://www.phoronix.com/scan.php?page=news_item&px=Linux-19.5M-Stats for details.

2 State of Research and Related Work

There are several papers and linux kernel documentation documents covering related or similiar topics. This section tries to list the most important ones and will explain the similiarities and differences in scope, focus, and other aspects compared to this seminar paper.

2.1 The performance analysis of Linux networking--packet receiving[7]

This work done in 2006 by Wenji Wu and Matt Crawford from the fermilab in Illinois, USA focuses entirely on the packet recieve process and performance issues. While providing many valuable insights for performance engineering and answering some on the "tracing the way of data" questions on this work it barely covers the "which function are involved and what are they doing"-question, which is part of the focus in this work, since their primary target was performance optimization. Also it contains the following diagram of the buffer structure, which is quite helpful for getting an overview and context for the results of the next chapters:

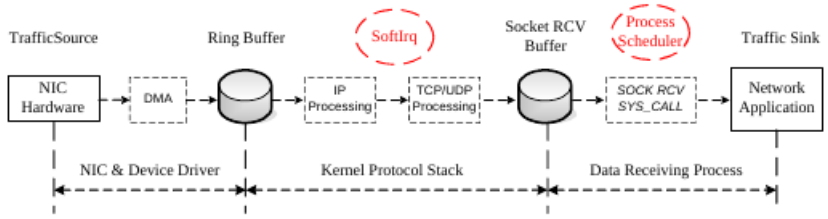

Figure 1. Buffers and Copying in the Linux Kernel

The 2 Buffers shown in Figure 1 will appear again in the results section.

2.2 The "kernel_flow" article in the official Linux Fundation Documentation[4]

In 2009 the linux foundation realeased this documentation for the linux kernel networking stack together with a quite tall and comprehensive diagram (which for layouting and scope reasons is not included in this paper). With a size of 3489x1952 pixels even on a full HD monitor it's not possible to look at the full diagram. Besides it suffers from "putting absolutely everything in one picture" which makes it difficult to get an overview. While beeing an primary and valuable source for this work it was a goal of this paper to produce an up-to-date and simplified version of this diagram as poster. Simplified in this case means, split into two distinct diagrams one for the receive path and one for the send path.

4. **See: http://www.linuxfoundation.org/collaborate/workgroups/networking/kernel_flow or use pdf-href**

3 About the Measuring Method: ftrace

3.1 About ftrace

3.1.1 What is tracing and ftrace

Ftrace is a kernel-builtin tracer for function calls and events inside the linux kernel.[6]

Definition. *tracer (software engineering) [2]*
A tracer is a tool for analyzing the behaviour of a given software at runtime. It provides an output similiar to an log of what happens inside the program. In most cases this output is an sequential structured list of all function calls which happened during execution. But there also exist other kinds of tracing like events tracing or I/O tracing. For VM-languages like Java or C# tracing is an feature of the runtime environment which can be turned on or off at runtime. For compiled languages like C tracing support has to be added via compiler switches or additional modules.

In the case of ftrace the tracing support is part of the software, the linux kernel. On most architectures ftrace uses hardware support for better performance.[4]

3.1.2 A Short Overview of ftrace Capabilities and Usage

ftrace is used and configured via the *debugfs* virtual filesystem. With the following shell command *debugfs* is made available:

Ftrace can be used via the *trace-cmd* programm. *trace-cmd* is packad and available in all big linux distributions.[5] Since tracing in kernel operation is still a quite major intervention into a running system only the root user is allowed to use trace-cmd, so all the following examples have to be executed as root.

To simply start recording all the function calls happening in the linux without any filtering use:

```
trace-cmd record -p function_graph
```

Figure 2. Example: Start tracing of all function calls in linux kernel

This writes all the results into a trace.dat in your working directory. `record` is one of the several subcommands of trace-cmd, in our examples and later measurments only the `record` and the `report` subcommands are needed. You shouldn't run this command (in the unfiltered version) too long, since it produces quite big files, about 900 MB after 30 seconds of tracing appeared in all tests using unfiltered tracing.

5. At the time of this writing (03.01.2016) *trace-cmd* is available in Ubuntu (since 12.04), Debian testing and stable and Fedora.

To view the content of the `trace.dat` file in a human readable format use:

```
trace-cmd report > results
```

Figure 3. Converting the Results into an humen readable format.

The report subcommands automatically uses the trace.dat file in the working directory and writes it's content to STDOUT which the redirection operator redirects to the `results` file.

The human readable output contains several columns about: the name of the process on behalf of the in kernel function call happened, the id of the CPU, an absolute timestamp, info if it's a function exit or entry event, the time the function needed (most below one micro-second) and the function name. The function names are graphically indentend to display the call hierarchy, so if B is called by A, B is indented relative to A by 2 spaces.

Usually these are much more columns than needed so in the results you will see in these paper, some of these columns were removed for layouting reasons.

3.1.3 Filtering For analyzing the way of data of a TCP connection we do not need information of all functions called in the overall kernel. We're only interested in traceing of the function calls happening on behalf of one single application. Kernel side filtering after a specific pid is possible using:

```
trace-cmd -p function_graph -P <pid>
```

Figure 4. Tracing all in-kernel function callls happening on behalf of <pid>

This way, the log file sizes are much smaller and more focused than previously. Tracing netcat for some time, while sending and recieving 3 small text messages produced a trace log file of 2,3 MB.

3.2 Why ftrace? Comparison to other Measurment Methods

Why did we choose ftrace? There are several tools for analysing and understing what happens inside the linux kernel.

4 Test Setup and Results

4.1 Test Setup

To produce measurable network traffic the *BSD netcat* programm has been used. Netcat is a small unix command line programm which opens a TCP (or

UDP or Unix Domain) Socket, either as listening socket, or as "client" to connect to another socket. The IP and Port to connect to (or the port to listen on) are supplied as command line parameters. For example: `nc 17.17.17.17 1055` connects to the IP 17.17.17.17 on port 1055 via an IPv4 TCP connection. Complementary with `nc -l 1055` the process opens an listening socket on the local machine on port 1055, waiting for a IPv4 TCP connection. After establishing a connection netcat sends all data it gets from STDIN through the socket and prints all data it recieves through the socket to STDOUT.

For this experiment 2 netcat instances were used, one on the measurment computer another on an remote linux server. The command used on the measurment computer was `nc -l 1337` and the remote server connected via `nc <ip> 1337`. After the connection was established in another terminal tracing was started using:

```
trace-cmd record -p function_graph -P <nc-pid>
```

Then two short messages (strings of 9 Byte and 167 Byte) were sent from the measurement computer. Following two messages of equal size were sent from the remote server and recieved by the measurement computer. Finalizing the tracing was stopped and the results translated into a human readable file using `trace-cmd report`.

4.2 Test results

Since the full trace of all function calls happening on behalf of netcat contained about 3000 lines (which subtracting all the empty lines drawn for the ascii art graph and closing brackets are about 2100 function calls), post editing got necesarry. Most of the function calls involved scheduling, terminal I/O or kernel internal locking of resources, so the sequences belonging to sending or recieving one packet were located and extracted. This happened by following the `SyS_write()` and `sys_read()` calls, which are the syscalls netcat uses to send and receive packets. This was gathered through traceing all the syscalls netcat does using strace.

The receive sequence consisted of 37 function calls and 56 lines which is small enough to print the complete trace in this document. Contrastingly the send sequence comprised 510 lines, so shortening got necesarry. The shortening included removing most of the locking and mutex function calls. Also in many cases where `function()` , did some locking and then called `__function()` for doing the internal work were simplified by only keeping the `function()` call. As a last step, the indentation and superflous columns of both results were removed, so both traces fit into this document side by side.

The final simplified traces are visible in Figure 5 and Figure 6. The full and unedited trace results are available via

```
SyS_write() {
__fdget_pos() [...]
vfs_write() {
rw_verify_area() [...]
__vfs_write() {
sock_write_iter() {
sock_sendmsg() {
inet_sendmsg() {
tcp_sendmsg() {
lock_sock_nested() [...]
tcp_send_mss() [...]
sk_stream_alloc_skb() [...]
skb_entail() [...]
skb_put();
tcp_push() {
__tcp_push_pending_frames() {
tcp_write_xmit() {
tcp_init_tso_segs();
tcp_transmit_skb() {
skb_clone() [...]
skb_push();
tcp_v4_send_check() [...]
bictcp_cwnd_event();
ip_queue_xmit() {
skb_push();
ip_local_out_sk() {
__ip_local_out_sk() {
ip_send_check();
nf_hook_slow() [...]
ip_output() {
nf_hook_slow() [...]
ip_finish_output() {
ip_finish_output2() {
skb_push();
dev_queue_xmit_sk() {
__dev_queue_xmit() {
skb_clone() {
kmem_cache_alloc();
__skb_clone() {
__copy_skb_header();
skb_release_all() {
skb_release_head_state();
skb_release_data();
kfree_skbmem()
kmem_cache_free();
e1000_xmit_frame() [...]
} } } } } } }
tcp_event_new_data_sent() {
tcp_rearm_rto() {
tcp_rearm_rto.part.59() {
sk_reset_timer() {
mod_timer() [...]
} } } } } }
release_sock() {
} } } }
fsnotify();
} }
```

Figure 5. Sending a TCP packet, simplified kernel trace result

```
sys_read() {
__fdget_pos() {
__fget_light();
}
vfs_read() {
rw_verify_area() {
security_file_permission() {
__fsnotify_parent();
fsnotify();
}
}
__vfs_read() {
sock_read_iter() {
sock_recvmsg() {
security_socket_recvmsg();
inet_recvmsg() {
tcp_recvmsg() {
lock_sock_nested() {
_cond_resched();
_raw_spin_lock_bh();
__local_bh_enable_ip();
}
skb_copy_datagram_iter();
tcp_rcv_space_adjust();
__kfree_skb() {
skb_release_all() {
skb_release_head_state() {
sock_rfree();
}
skb_release_data() {
kfree();
}
}
kfree_skbmem() {
kmem_cache_free();
}
}
tcp_cleanup_rbuf() {
__tcp_select_window();
}
release_sock() {
_raw_spin_lock_bh();
tcp_release_cb();
_raw_spin_unlock_bh() {
__local_bh_enable_ip();
}
}
}
}
}
}
}
__fsnotify_parent();
fsnotify();
}
}
```

Figure 6. Receiving a TCP packet complete kernel trace results

5 Evaluation and Discussion of the Results

5.1 Send Flow

Asuming a TCP connection is already established and we have a socket for sending data in our programm. Before the TCP implementation of the kernel can process and send the data, it has to get the data from userspace, the next two subsections will cover how this handing over happens and how the kernel will process the data.

5.1.1 Syscalls and Kernel Entry

There are 4[6] syscalls available for sending data through a TCP socket, namely:

- write()
- send()
- sendto()
- sendmsg()

They all need a file handle to the socket and a pointer to the send-data-buffer as arguments. They differ in the number of additional parameters and the fact that sendmsg() needs a more complex `msghdr` data structure for the input data instead of a simple buffer.

5.1.2 In Kernel Flow

5.2 Recieve Flow

5.2.1 Syscalls and Kernel Entry

From userspace there are several entry points to the kernel for recieving data:

- read()
- recv()
- recvfrom()
- recvmsg()

5.2.2 In Kernel Flow

6 Conclusion

The two goals of this work were reached. This text contains an freely available and up to date overview over the linux TCP networking internals and the ftrace kernel event tracing toolkit. Regarding the third issue, the scope, we tried give a quite general scope, covering no part of the TCP stack explicitely (or solely sending/recieving), so every ascending kernel developer gets some general context of the part functioniality he or she wants to change or improve. Nevertheless TCP Networking is already a quite focused topic, so the third issue was not completely solved.

6. Actually there are 6, but `writev()` and `fwrite()` are not mentioned seperately because from a kernel developer point of view they are similiar to the discussed ones. (See main text, below enumeration).

Concerning the decision for using ftrace it must be said that using ftrace solely was no very good idea, since ftrace (or trace-cmd) can not report any information about the function parameters used.[7][6] But knowing what data a function is using (or what pointers to data) is quite important for tracing the way of data. So a combined method of using ftrace and looking up function prototypes in the linux source code was employed and worked fairly well.

So for generally understanding how these functions work together ftrace and prototype lookup is a reasonable method. But if the goal is fixing errors and knowledge about the concrete values passed to a function get necessary other tools like *kgdb* and *systemtap* are more well-suited.

7. There exists no option or capability in the ftrace documentation regarding function parameters.

Bibliography

[1] Christian Benvenuti. *Understanding Linux network internals*. O'Reilly, Sebastapol, Calif, 2006.

[2] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: lessons learned from five industrial projects. In *Proceedings of the First International Conference on Runtime Verification (RV 2010)*. Springer-Verlag (Lecture Notes in Computer Science), November 2010. Original publication is available at www.springerlink.com.

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

[4] Mike Frysinger . Function tracer guts. Doc-file in Linux source tree: linux/Documentation/trace/ftrace-design.txt.

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

[6] Steven Rostedt . Ftrace - function tracer. Doc-file in Linux source tree: linux/Documentation/trace/ftrace.txt.

[7] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking–packet receiving. *Computer Communications*, 30(5):1044–1057, 2007.