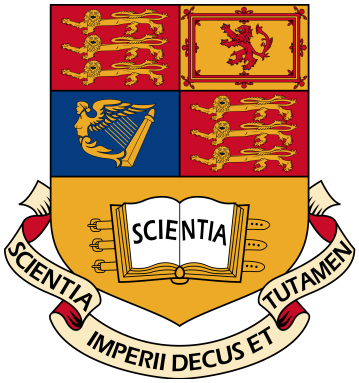
Imperial College London
Department of Computing



An Application-Aware TCP Stack

Yoann Desmouceaux

Submitted in partial fulfilment of the requirements for
the MSc Degree in Advanced Computing of Imperial College London

September 2015

Abstract

With the development of high-performance networks, using existing protocols such as TCP can become challenging. In this project, we try to tackle two issues that standard TCP implementations face in such environments: the fact that data needs to be copied several times when reading or forwarding a packet, and that applications access it as a random linear byte stream unrelated to the semantics of the contents. Coping with that can be done through the use of a stack running in userspace, which allows better performance and gives the application a direct access to packet buffers. To that aim, we extend mTCP – a userspace TCP stack – by introducing two principal features to the application layer: the ability to read packets without copying data, and the possibility to access pointers to relevant parts of the data according to a given semantic – such data being pre-parsed in the transport layer. Finally, extending the zero-copy design to packet forwarding and combining it to this application-aware scheme enables us to propose a stack that can transparently forward and inspect TCP traffic between a client and a backend server.

Acknowledgements

I would like to thank Dr. Peter Pietzuch for agreeing to supervise me during this project and giving helpful feedback, as well as Dr. Alim Abdul for proposing the subject and participating in discussions. My thanks also go to Lukas Rupprecht for kindly helping me access the LSDS testbed environment. Finally, I would like to thank Dr. Paolo Costa for giving some of his time to be the second marker of this project.

Contents

1. Introduction

1.1 Context and motivation

In use since 1974, the Transmission Control Protocol (TCP, [1]) is notoriously one of the most used transport protocols over Internet Protocol (IP, [2]) networks. Indeed, it is a connection-oriented, fault-tolerant, stream-based transport protocol, which makes it a universal candidate for reliable exchanges over IP. More precisely, it allows a connection to be established between two endpoints – each identified by a tuple $(address, port)$. Once this connection is established, hosts can start exchanging any kind of information using a stream abstraction, and an acknowledgement mechanism ensures the reliability of the data transfer.

This ability to use arbitrary data streams as payloads gives TCP a great flexibility, and it might explain why it has been chosen as the transport layer for so many protocols, from web browsing (HTTP), email handling (SMTP, POP, IMAP) and file transfer (FTP) to interactive sessions (Telnet, SSH). From the point of view of the application, it suffices to provide the TCP stack with a stream of bytes, which ensures that the other endpoint will receive the same byte stream, regardless of its size and of the network conditions.

However, this design introduces a certain amount of redundancy in the way the data is transmitted from a layer to another. Indeed, in the usual implementation of TCP, the incoming frames are held in a queue inside the kernel, and the user application also holds a buffer into which the data from these kernel buffers will be copied – this happens through the `read()` system call of the BSD socket API. Hence, the same data is stored at two different places: in the kernel buffers representing packets fetched from the Network Interface Controller (NIC), and in the application buffer.

Furthermore, the `read()` system call is unaware of the application needs and simply copies the next bytes of the TCP stream to the application. In the usual scenario, the application requests a fixed amount of data (usually 8 KB) and starts parsing this data. In the case where the relevant data cannot be entirely fetched, the application has to wait for the next `read()` call to succeed before being able to successfully deserialise the data, and might have to restart the parsing process from the beginning.

Moreover, with the development of high-performance networks, new applications arise that have expensive throughput needs, such as data processing in middleboxes or Big Data computations. Leveraging the maximum performance offered by Gigabit or 10 Gigabit Ethernet NICs is

a non-trivial challenge (since it imposes efficiently managing both CPU and memory resources [3]), but it is crucial that such hardware can run already widely-used protocols such as TCP. However, the fact that usual TCP stacks run in the kernel can be a bottleneck. Indeed, the context switch between the CPU privileged mode and the user mode that happens each time a system call is emitted is expensive [4, 5].

For this reason, it can be useful to have a TCP stack that runs directly in userspace, avoiding useless context switches between the application and the kernel. Such a design can not only reduce the overhead due to system calls, but also introduce performance gains by avoiding data copy. Indeed, with this approach there is no need to isolate packets structures (normally belonging in the kernel) from the application, since there will be no impact on the overall system stability if they become corrupted.

1.2 Contributions

In this project, we aim at addressing these issues – namely, data copy and its representation as a byte stream – by proposing a TCP stack that introduces a new paradigm for communication between the application and the transport layer. As explained above, a fruitful means towards such an aim is to use a stack running in userpace. Therefore, we base our work on mTCP [6], a userland stack based on the DPDK framework [7]. This project brings a novel approach thanks to the following features:

Zero-copy design

In the previous section, we mentioned the issue of data copy within traditional TCP implementations: the same data is present at several different places and is copied between several buffers belonging to different layers of the system. In this project, we modify the internal representation of a TCP flow in mTCP so as to adopt a zero-copy design. This way, the application is provided with a direct access to the packet buffers holding the TCP payload.

Middlebox capabilities

This zero-copy approach makes even more sense in the case of a host whose purpose is to forward data and inspect it, for example to gather statistics. We implement a mechanism to forward such streams while only needing to rewrite packet headers when moving packets, thus avoiding data copy. This way, an application can inspect the traffic flowing through two hosts without inducing a latency overhead in the end-to-end connection.

Application-aware stream representation

We introduce a new API to access the data contained in a TCP stream. Instead of accessing an array of bytes without knowing in advance if it will contain all the data required to process a semantic unit, we provide an *application-aware* API. This enables the application to be directly provided with structured data, without having to worry about the TCP byte stream underlying it. Following the aforementioned zero-copy design, these structures contain delimiters to relevant data inside the packet buffers.

Custom deserialisers

We enable the application developer to specify their own data format for the application-

aware stream introduced above. To that purpose, we implement a preprocessor-based system that generates the corresponding deserialiser and integrates it into the stack.

To summarise, we propose an instance of a TCP stack running in userspace that allows in-the-middle application-aware deserialisation using a zero-copy design.

1.3 Document outline

The rest of this document is organised as follows:

- In chapter 2, we perform a survey of related projects and notably mTCP, which has been used as a base throughout this project.
- Chapter 3 exposes the stream structure set in place to enable a zero-copy access to the TCP payload from the application.
- Chapter 4 then introduces an abstraction built on top of this stream that allows the application to access structured data. Starting from the simple example of CSV parsing, we then introduce a more complex scheme based on HTTP, and generalise by proposing a mechanism to specify arbitrary data formats.
- Finally, in chapter 5, we study the case where the stack is used in a middlebox scenario, where it transparently inspects application-specific data in a stream between a server and a client.

Each of these chapters also contain implementation details and performance evaluation of the implementation.

2. Background

In this chapter, we introduce several research projects that deal with issues related to this work. Some of them use a different approach than we have in this project, while others have been used as a starting base to build our application-aware zero-copy TCP stack.

Section 2.1 first focuses on improvements to TCP that have been introduced in the traditional kernel implementation. Then, section 2.2 describes how high-performance packets processing engines (netmap, DPDK) can leverage the capabilities of high-throughput NICs. At this point we will be able to introduce mTCP (section 2.3), a general-purpose TCP stack that uses DPDK and that we have extended in this project. We also describe FlowOS and Sandstorm, two specific-purposes TCP stacks, in section 2.4. Finally, in section 2.5 we study data serialisation protocols, especially the Google Protobuf format, which gives insight on what type of API can be used to efficiently provide an application with deserialised data.

2.1 Kernel-mode TCP improvements

Several features have been classically implemented in TCP stacks. Direct Memory Addressing (DMA) enables the hardware to copy data to (or fetch data from) the RAM without the need of the CPU, thus saving cycles and enabling asynchronous data access. Checksum offloading permits the computation of network frames checksums directly in the hardware. Alongside these techniques, two other improvements are worth noting and will be studied in this section.

2.1.1 The `sendfile()` system call

One of the main issues with conventional TCP stacks running in the kernel is that data has to be copied from kernel space to user space. While this is unavoidable in a standard scenario where the application wants to fetch data from the network, put it into a buffer and process it, this introduces a useless overhead when data is to be transferred from a socket to another socket, or between a socket and a regular file descriptor.

If we consider for instance the case of an HTTP server, where the host transfers a file to a client, the naïve implementation would proceed as such:

- perform a `read()` call on the file descriptor and store the resulting data in a buffer `buf`,

- use `buf` to perform a `write()` call on the client socket.

As a result, the data will exist in five different places:

1. in the hard disk controller memory,
2. in the buffers held by the hard disk driver in the kernel,
3. in userspace, in the `buf` buffer maintained by the application,
4. in the egress socket buffers held by the TCP stack in the kernel,
5. in the NIC egress ring buffer.

This implies that the data is copied four times – two of which are CPU copies, while the two other are DMA copies between the hardware and the kernel memory and are therefore less expensive.

In order to address this issue, the `sendfile()` system call was introduced in 1999 in Linux 2.2[1], as well as in other Unix flavours. The purpose of this system call is to move data from one file descriptor to another without copying it to and from userspace. The data now only resides in four different places:

1. in the hard disk controller memory,
2. in the buffers held by the hard disk driver in the kernel,
3. in the egress socket buffers held by the TCP stack in the kernel,
4. in the NIC egress ring buffer.

The data is now only copied three times, only one of which is a CPU copy. Furthermore, only one context switch between kernel mode and user mode is needed, instead of the two needed when using `read()` and `write()`.

In [8], the author explains how Linux 2.4 managed to eliminate this last CPU copy (thus achieving what is called *zero-copy*). The principle behind this improvement is to use the *scatter-gather* capabilities of the hardware. With such a system, the network headers and the data payload need not be contiguous in the memory before the NIC copies them using DMA. The NIC fetches the headers and the payload at *scatter* locations in memory and *gathers* them before sending them out to the wire. Therefore, the CPU copy between the hard disk buffers and the socket buffers is not necessary any more: the kernel only has to generate the network headers and append the location of the relevant payload in the hard disk buffers.

In summary, the `sendfile()` system call is a first simplistic way of performing zero-copy transfers between a disk and a socket. However, it does not permit to transfer data between two sockets. In [9], the authors introduce *TCP splicing*, which enables this feature. While their approach improves throughput, it is limited to non-intelligent proxies since the data cannot be accessed by the application. That is why a TCP stack in userspace ought to be a more flexible way of solving the issue, enabling both zero-copy transfers and data access.

2.1.2 An HTTP server inside the Linux kernel

Another idea to reduce the overhead due to context switching between user mode and kernel mode is to directly run a specific network application within the kernel [10]. This approach

[1] http://linux.die.net/man/2/sendfile

goes exactly in the opposite direction than the one we have taken in our project: while we have focussed on moving the network and transport layers (L3 and L4 of the OSI model) into the userspace, this design moves the application layer (L7 of the OSI model) inside the kernel.

An example of such an approach is TUX [11], an HTTP server running inside the kernel (another similar project is kHTTPd [12]). In TUX, a caching mechanism is implemented to hopefully reduce the overhead of I/O operations when serving content to the clients. When a file is requested by a client, it is sent to the client and corresponding socket buffers and checksums are cached. A hashtable is maintained to bind an URL to a cache entry. If the same file is requested again, only network headers and their checksums need to be recomputed, and scatter-gather IO is used to merge headers and payloads. TUX also provides an `http()` system call to communicate with userspace applications: it can for example send information about incoming HTTP requests.

Another feature introduced by TUX is the ability to run dynamic applications by providing a kernel module which will answer to HTTP requests via a callback function. This enables more diverse applications than file serving to be run, but at the cost of complexity since a kernel module must be written for each application.

While running a specialised application directly inside the kernel can have a great impact on performance by avoiding context switches and memory copy, such a design also has drawbacks.

The first one is obviously the intrinsic complexity of kernel programming: debugging a kernel module is harder than a simple application. Plus, one has to take extra precautions when dealing with kernel data structures. While current CPU designs provide a clear separation between kernel memory and application memory which prevents a userspace application from wreaking havoc when crashing, a bug in a kernel module can affect the whole kernel data structures and have disastrous effects, up to kernel panics.

The second drawback of this approach is that it reduces flexibility and reusability. Indeed, there is no BSD-like socket API in the kernel: each new application that one wants to design as a kernel module has to be rewritten from scratch and to provide its own interface to interact with the user space.

2.2 High-performance packets processing engines

Several efforts have been made to process packets at line rate on 10 Gigabit Ethernet interfaces [13, 14, 15]. In order to illustrate how such engines work, we will focus on two of them in this section, netmap and DPDK.

2.2.1 netmap

A popular high-performance network engine is netmap [5]. It consists in a framework, running in the FreeBSD and Linux kernels, aiming at providing applications with a fast access to network frames and reducing data copy overhead.

To achieve this goal, netmap allows userspace applications to directly access its kernel data structures, by mapping them into the virtual address space of each process. Those structures are of three kinds:

(i) packet buffers (pre-allocated at the initialisation of the framework), into or from which NICs copy packets using DMA;
(ii) netmap rings, which are an abstraction of NICs ring buffers and hold pointers to packet buffers;
(iii) interface descriptors, which store information about a NIC and pointers to corresponding netmap rings.

Once these structures are initialised, the user application can access them using two different APIs, a synchronous one and an asynchronous one.

The synchronous API relies on the `ioctl()` system call on a special device. The receive `ioctl()` tells the application how many packets have been received and can be read, while the transmit `ioctl()` system call tells the kernel how many packets the application has prepared for transmission.

The asynchronous API relies on the `select()` and `poll()` system calls to perform blocking I/O. Using this API can reduce the number of system calls needed – with the synchronous API, it is possible to waste a system call because no data was available.

Since the user process has access to the netmap rings of all interfaces, it can easily perform zero-copy transfer between two network interfaces, which can be useful to applications such as routing, proxying and firewalling. To do so, it simply has to pass the pointer to the packet buffer of an incoming packet to the netmap ring of the egress interface, and signal the kernel that a new output packet is ready using the aforementioned API.

Although netmap maps its structures to userspace, enabling zero-copy access from the user application, it stills runs in the kernel and therefore requires the application to communicate with the framework through the use of system calls. Hence, the overhead due to context switching between user and kernel mode is not removed.

However, this design has the advantage of introducing a robust security model. Since the only kernel memory region that the application can access is the one shared by netmap, it cannot corrupt essential kernel structures. Sanity checks are performed on netmap data structures after the application alters them, which reduces the risk for crashes. Plus, the addresses to which the DMA engine can write upon receipt of a packet are provided by netmap, not the application, and therefore the DMA engine cannot write to arbitrary memory locations that a malicious user could provide.

To conclude, netmap offers a robust framework for high-performance networking: its memory model makes it an interesting and secure compromise between classical network stacks running in the kernel and stacks running entirely in userspace. However, it does not remove the need for system calls as a communication interface between the framework and the application.

2.2.2 Intel DPDK

Another widely-used packet processing engine is the Intel® Data Plane Development Kit (DPDK) [7]. Although a commercial solution, its code source is freely available and well documented.

The main difference between netmap and DPDK is that the latter brings the NIC access down to the userspace, removing the need for system calls. Of course, the support of the kernel is still needed to enable such a functionality: DPDK relies on UIO-IXGBE [16] to this purpose. The kernel module is in charge of managing the device, allocating buffers and mapping memory between the kernel and the user space. The device PCI memory is also directly mapped to the user space through Memory-Mapped I/O, enabling the application to have a full control over the device and directly read and write frames using DMA.

Once this initialisation is done, the DPDK framework provides an interface to the application in order to efficiently deal with packet reception and transmission. DPDK uses buffers called `mbuf`s (borrowed from the FreeBSD network stack design [17]) that are pre-allocated in a memory pool at the framework initialisation. When the application needs to send a frame to the wire, it shall call `rte_pktmbuf_alloc()` to acquire a fresh `mbuf` from the memory pool, fill it with the frame content and pass it to `rte_eth_tx_one()` function, which will flush it to the wire using DMA and give it back to the memory pool. Conversely, when the application needs to read a frame from the wire, it shall call `rte_eth_rx_one()`. This will acquire a fresh `mbuf` from the memory pool, ask the NIC to fill it with the next packet content using DMA, and return it. The application can then process it and call `rte_pktmbuf_free()` when it is done, which gives it back to the memory pool.

At this point, the application could also chose to forward the frame instead of releasing it, thus performing a zero-copy transfer. To facilitate this operation, room is reserved before and after the payload so that the application can easily prepend or append data to the packet before forwarding with the help of dedicated API functions. Likewise, `mbuf`s can be chained so that several fragments can be assembled in a single packet without merging the data in memory, and cloned such that two packets with the same content can be sent to different interfaces without duplicating their content.

DPDK uses *Poll Mode Drivers* running in userspace to handle the hardware. They avoid the overhead of CPU interrupts by directly polling the NIC queues. To that purpose, two different models can be adopted. With the first one, *run-to-completion*, each time a packet is received through the driver API, it is immediately processed (and maybe forwarded) before the application requests the next packet to be fetched from the NIC. The second one is a *pipeline* model and offers greater flexibility. Under this model, the application uses one core to receive packets from the NIC. Each new packet is put in a queue, and another CPU core will process packets in this queue (potentially forwarding them).

In order to improve performance, DPDK also provides burst receive and transmit functions, allowing to process several packets in a single function call. Provided that the drivers implements these functions correctly by using cache features and reducing the number of hardware registers accesses, this can have a beneficial impact on the number of CPU cycles spent per packet.

In summary, DPDK offers a flexible and efficient way of accessing NICs directly from userspace, completely bypassing the kernel. It is worth noting that it still has two drawbacks: (i) NIC drivers need to be rewritten in userspace and (ii) once a device is attached to DPDK, it is totally disconnected from the regular kernel stack and thus cannot be used this way any more.

2.3 A general-purpose userland TCP stack: mTCP

mTCP [6] is an example of a TCP stack running in userspace and providing a general API. Previously based on the PacketShader engine [18] but now using DPDK (section 2.2.2), it aims at providing a simple way to port existing applications while improving their performance by avoiding to switch between kernel and user modes. Since we used it as a codebase in this project, technical implementation details will be included in this section.

While at first glance it could seem easier to implement such a TCP stack purely as library calls handled in the same thread as the application, the authors of mTCP chose to implement it as a two-threads model. This way, network operations are handled in a *network thread*, while the application logic is embedded in the *application thread*. This allows TCP functions that need a precise timing (such as retransmissions) to be handled properly. Since the network thread works independently from the application, it can also batch packets reads and writes, taking advantage of DPDK burst functions.

In order to take advantage of multi-cores systems, mTCP can spawn an application thread and a network thread on each CPU core. In such a case, each flow will be assigned to application and network threads belonging to the same core, so that they share the same CPU cache.

Communication between the application thread and the network thread is ensured by a BSD-like socket API, enabling to easily port existing applications to mTCP. Functions exposed by the mTCP API include `mtcp_socket()`, `mtcp_bind()`, `mtcp_connect()`, `mtcp_listen()`, `mtcp_accept()`, `mtcp_read()`, `mtcp_write()`, `mtcp_close()`. A non-blocking design is adopted: the application thread must call `mtcp_epoll_wait()` to wait for an event notification from the network thread and `mtcp_epoll_ctl()` to register events. These functions work similarly to the corresponding system calls, but are implemented with spinlocks between the network and the application thread.

Since mTCP exposes a BSD-like API, the application must provide a buffer in which data must copied when calling `mtcp_read()`, and similarly for the transmit side. Therefore, this design is inherently incompatible with the zero-copy concept. Furthermore, mTCP does not use the zero-copy paradigm in its internal implementation and maintains a linear receive buffer (`tcp_ring_buffer`) in which the TCP stream payload is merged as and when new packets are received – and a linear send buffer in which the data sent by the application is merged before being packetised and sent to the wire.

To illustrate this design, let us describe the logic used to deal with incoming data packets corresponding to established TCP connections – the logic for the transmit side works in a similar fashion.

1. In the network thread:

(a) The main loop calls the DPDK burst API to receive up to 64 `mbuf`s.

(b) Each received packet is processed: after parsing its headers and associating it to the correct TCP flow by looking in a hashtable, its payload section is copied at the right place into the `tcp_ring_buffer`, and the `rcv_nxt` variable is updated accordingly. If the buffer is full or the packet has an incorrect sequence number, an immediate ACK is enqueued on the transmit side. Otherwise, a single ACK will be transmitted for all the packets that were received at the previous step, using the `rcv_nxt` variable as the sequence number to acknowledge. It is worth noting that the thread uses a lock to protect the `tcp_ring_buffer`.

(c) A read event is raised in the application event queue, signalling that incoming data is available.

(d) Output queues are flushed (which sends the ACK packet), `mbuf`s are freed, and the loop restarts.

2. In the application thread:

(a) The application loops on `mtcp_epoll_wait()` waiting for a read event.

(b) Upon receipt of such an event, it calls `mtcp_read()` with a user-provided buffer. This will copy the required amount of data from the `tcp_ring_buffer` into this buffer, and move the tail of the ring buffer accordingly. Access to the `tcp_ring_buffer` is also protected by a lock.

This shows that the payload of the TCP stream resides in three different places: in the DPDK `mbuf`s, in the mTCP `tcp_ring_buffer`, and in the buffer provided by the user application. In chapter 3, we will explain how we enhanced this design by enabling the user to directly access the `mbuf`s, effectively providing a zero-copy design.

To sum up, mTCP provides a generic TCP stack running in userspace. It provides a BSD-like API where the network thread plays the role of the kernel, allowing applications to be ported easily. Its event-driven model allows an efficient repartition of resources between the network thread and the application thread, and tries to minimise the amount of time spent waiting in the application thread by processing packets in batches. According to the authors, it performs especially well for short transactions (saturating a 10 Gbps NIC for transactions starting at 1 Kb while the Linux kernel fails to achieve more than 1 Gbps). They also claim a good performance benefit for real applications, improving the throughput of an HTTP server (*lighttpd*[2]) by a factor of 2.2 over Linux.

2.4 Specific-purpose TCP stacks

In this section, we will present two examples of specific-purpose TCP stacks with different aims and designs, FlowOS and Sandstorm.

[2] http://www.lighttpd.net

2.4.1 FlowOS

In [19], the authors present FlowOS, a kernel mode TCP stack designed for middleboxes. FlowOS aims at providing an easy-to-use API so that services running in the middle of a TCP connection (proxies, firewalls, etc.) can easily access the payload of a TCP flow, and modify it by inserting, changing or removing some of its bytes.

To that purpose, it classifies incoming packets into different flows according to their sources and destinations. Then, a parser creates streams out of packet contents. In this abstraction, a stream is a linked list of pointers to a portion of the content of a packet. For a flow, three streams are created: (i) the IP stream, containing every byte of an IP datagram, (ii) the TCP stream, containing the IP payload of the datagram and (iii) the application stream, containing the TCP payload of the datagram.

Once these streams are created, FlowOS allows *processing modules* to handle them. A processing module is a kernel thread that works on a stream of a given flow. Such a module will only access data between the *head* and the *tail* pointer of a stream. While the FlowOS engine will move the *tail* pointer of the stream forward when it receives fresh data, a processing module will move the stream *head* pointer forward once it has finished processing data. That way, synchronisation mechanisms can be avoided while keeping a safe processing design.

Processing modules can be assembled in serial to form a processing pipeline on a flow. It is also possible to use several modules in parallel on the same flow without worrying about synchronisation issues. To achieve this, FlowOS keeps a record of all *head* pointers held by the different modules and effectively moves its pointer forward when the module owning the lowest pointer releases data.

Once the data of a flow is modified by a processing module, FlowOS needs to transfer it to its final destination. To do so, FlowOS could act as the receiver with respect to the sender, and reopen a new TCP connection between the middlebox and the final receiver. However, in order not to add a significant overhead to the process, the authors chose not to terminate the existing connection. Therefore, they implemented a mechanism to properly handle sequence number remapping when a processing module inserts or removes data from the flow.

To conclude, FlowOS provides an efficient approach for modifying TCP streams on the fly: according to the authors, it does not add any throughput overhead to the kernel routing functionality. Compared to what we wanted to achieve in this project, it has two disadvantages: (i) it runs in the kernel and (ii) it does not provide an end-to-end read or write API but only allows to handle TCP connections between two existing endpoints.

However, the stream model provided by FlowOS has been a enlightening guide for our implementation, since we aimed at providing the application with a raw payload stream as well as an application-aware stream represented the data as structured chunks.

2.4.2 Sandstorm

In this section, we will describe Sandstorm [20], an application-specific TCP stack that uses a radically different approach. Sandstorm consists in a custom TCP stack based upon netmap (section 2.2.1) and running an HTTP server serving static files.

Sandstorm aims at exploiting the zero-copy concept to the highest possible extent. Indeed, it tries to avoid packetisation on the critical path by pre-generating TCP/IP datagrams. More precisely, when the application is initialised, it browses files in the webserver directory. For each of these files, it goes through its content and linearly generates packets, including Ethernet, IP and TCP headers with correct sequence numbers and checksums. Once the application is initialised, it works as follows:

1. Upon receipt of a request from a client, Sandstorm allocates a TCP control block for this client.
2. When an ACK packet is received, its IP header is parsed to locate the TCP control block corresponding to the connection in a hashtable.
3. The ACK number of the packet is processed to locate which of the pre-generated frames should be sent next – depending on the size of the client receive window, it is likely that several frames should be sent.
4. The destination IP address and TCP port in these frames are rewritten, as well as the sequence numbers, and the corresponding checksums are gradually recomputed.
5. These packets are put in the output *netmap ring* corresponding to the outbound interface.

Recalling that netmap runs in the kernel and still needs system calls to interact with the application, Sandstorm tries to minimise the number of times it uses the `poll()` syscall so that netmap flushes its ring to the NIC. Indeed, after step 5 above, it only flushes its queue if a $\Delta t = 80\mu s$ interval elapsed since the last time it did so.

As we can see from this description, Sandstorm achieves a full zero-copy design by creating packets in advance, before they are requested. With this design, retransmissions can be handled very easily. Indeed, in case of a lost packet, it suffices to locate the data in the cache using the sequence number of the datagram and to send it again.

However, this design introduces an issue when it comes to parallel connections. If multiple clients request the same data chunk during the same Δt time interval which separates two batch transmissions, the same packet buffer cannot be used more than once. If this occurs, the destination address of the packet will be overwritten for each client in step 4, and only the last address written will be taken into account. To handle this issue, each client must be allocated its own copy of the packet, for which the authors explore two solutions. With the *pre-copy* approach, multiple copies of each packet are created in advance, while with the *memcpy* approach, a copy is created on-demand each time it is needed. According to their experimentations, the *pre-copy* technique uses less CPU and achieves higher throughputs, although obviously consuming more memory.

In summary, Sandstorm provides a novel and surprising approach to TCP specialisation in userspace. According to the authors, it can saturate 4 10 Gigabit Ethernet NICs when serving

8 KB files, superseding Linux-based HTTP server *nginx*[3] which can only do so with 128 KB files. Nevertheless, the disadvantage of this design is its excessive specialisation. Indeed, the massive pre-packetisation used by Sandstorm is only suitable for already existing static content and cannot be adapted to handle dynamic data.

2.5 Data serialisation and deserialisation

Since this project is partly about creating a mechanism to automatically deserialise data for the application layer during packet processing in the TCP stack,this section will give a brief overview of data serialisation formats. In [21], the authors compare four of them: XML, JSON, Thrift and Google Protobuf. XML and JSON are protocols that carry semantic information about their own content (though text-based, research has also been conducted to implement binary protocols compatible with XML [22]). JSON has a much more lightweight syntax than XML while not reducing its expressiveness and is, according to the authors, 6 times faster to parse. This shows that the choice of a serialisation protocol can have a huge impact on performance, especially if it runs at a line rate of 10 Gbps.

What is especially interesting in this survey is the comparison of these two text-based formats to the two others, which are binary-based. According to the authors, the Google Protobuf [23] format is up to 5 times faster to deserialise than JSON. What essentially improves performance with this format (apart from being binary-based) is that no semantic information is carried alongside the data. Since this is the kind of design that we have decided to use in our API, we will give a brief overview of how this is achieved.

With the Protobuf API, the sender and receiver applications must agree upon a data format prior to exchanging information. This is accomplished through the use of a `.proto` file, which contains a C-like description of the data format. An example of such a file would be:

```
message KeyValuePair { required string key = 1; required int32 value = 2; }
```

A specific compiler must then be called to process this file, which generates a C++ class `KeyValuePair`. This class is filled with specific getters and setters of the form `key()`, `set_key()`, `name()`, `set_name()` and are automatically implemented to fill the underlying binary representation. This class extends the `protobuf::Message` class, and can take advantage of its `SerializeToOstream (std::ostream)` and `ParseFromIstream(std::istream)` methods to automatically serialise or deserialise a C++ object to of from a stream.

The major asset of this design is that it takes advantage of knowing the data format in advance, that is why we decided to use a similar approach to build our TCP stack. Indeed, this allows an application using this framework to use a parsing code specifically crafted for the kind of data it will need to work with. Of course, this performance benefit can only happen at the cost of flexibility, since the data format must be known at compile time; nonetheless, we believe that it is not an absurd requirement given the purpose of the project.

[3] http://www.nginx.org/

3. Bringing zero-copy capabilities to the TCP stack

This work ultimately aims at creating a TCP stack that can deserialise and forward application-specific data without resorting to memory copy. To that purpose, mTCP [6] has been chosen as a codebase; however, as explained in section 2.3, mTCP does not have zero-copy abilities. Therefore, a first requirement is to provide a zero-copy mechanism for data reception in our mTCP-based stack, which is what this chapter will focus on. Section 3.1 describes the data structures and algorithms used to that purpose, while section 3.2 focuses on the implementation and section 3.3 aims at evaluating its performance.

3.1 Design considerations

3.1.1 Data structures and algorithms used

In its initial design (figure 3.1), mTCP uses a linear buffer which holds the reassembled TCP payload pending for the application, and data is copied from this buffer to a user-provided one when the application requests to read new data. In order to introduce a zero-copy paradigm, we need to modify the way mTCP represents pending TCP payload.

To that purpose, we replace the linear buffer with a linked list of nodes representing TCP packets (figure 3.2), that we will call a *zero-copy stream*. Each node contains the following information:

- the TCP sequence number associated with the packet,
- a pointer to the DPDK packet buffer corresponding to the node,
- a pointer to the beginning of the TCP payload in the DPDK packet buffer,
- the length of the TCP payload.

The use of such a design avoids copying data from DPDK buffers to an internal TCP buffer, as well as from this internal buffer to the application buffer. However, giving the user a direct access to the packet buffers means that releasing these buffers becomes the responsibility of the application.

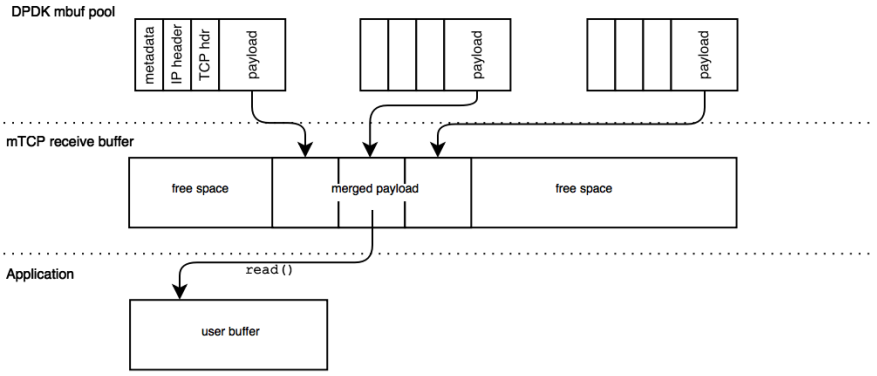


Figure 3.1: mTCP receive buffer design

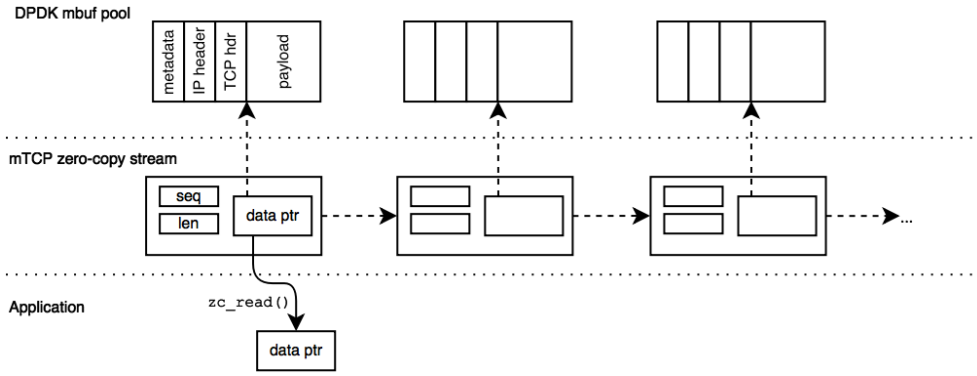


Figure 3.2: Zero-copy stream design

The algorithm used to maintain this linked list is quite simple, as described in figure 3.3. Its primary purpose is to enqueue nodes corresponding to received packets in the aforementioned linked list, and dequeue them when requested by the user. Apart from the input packets themselves, two key elements are part of the environment the algorithm: the sequence number to acknowledge (`rcv_nxt`), and the size of the receive window to advertise (`rcv_wnd`).

Sequence numbers are an intrinsic part of TCP as they guarantee transmission reliability: each packet is assigned one such number which represents the offset of the data in the whole TCP stream (up to a constant). Therefore, inspecting the sequence number of a packet provides a means of detecting out-of-order transmission. When an endpoint receives data, it must acknowledge it by sending an ACK packet with the next sequence number expected to be received. Acknowledgements can occur in a bulk to reduce network traffic.

The receive window field of a TCP packet indicates the amount of the data that a host is willing to accept without having to acknowledge it. Even if we have got rid of a fixed-size linear intermediate buffer and therefore have a virtually unbounded receive window, it is still necessary to advertise a consistent size to ensure flow control. Otherwise, it could be possible that the application does not release the packet buffers fast enough compared to the rate at which they are provided by the sender.

Figure 3.3: Algorithm used to maintain a zero-copy TCP stream

```
upon (receipt of a TCP SYN packet p)
    insert a new stream in the global stream hashtable
    stream.rcv_wnd = RECEIVE_BUFFER_SIZE //configurable by the user
    stream.rcv_nxt = p.seqnum + 1

upon (receipt of a TCP data packet p)
    stream = identify which stream p belongs to
    if (p.seqnum != stream.rcv_nxt || p.length < stream.rcv_wnd)
        drop p and send duplicate ACK
    else
        stream.list.enqueueLast(p)
        stream.rcv_nxt += p.length
        stream.rcv_wnd -= p.length

upon (request from the user to read n packets from a socket)
    stream = identify stream associated to the socket
    m = MIN(n, stream.list.size())
    return an array of pointers to the m first elements of stream.list

upon (request from the user to free n packets from a socket)
    stream = identify stream associated to the socket
    for (i = 1 to n)
        p = stream.list.dequeueFirst()
        stream.rcv_wnd += p.length
        free(DPDK packet buffer associated with p)

upon (entering packet TX loop)
    for (stream in streams)
        sendACK(ack_seq=stream.rcv_nxt, rcv_wnd=stream.rcv_wnd)
```

3.1.2 Dealing with TCP Fast Retransmission

Due to its simplicity, this algorithm quickly faces an issue when it comes to packet loss, which can lead to a great performance drop. Indeed, when a packet p is received whose sequence number is higher than expected – thus indicating packet loss –, the algorithm sends a duplicate ACK to the other endpoint with the sequence number it was expecting. This duplicate ACK will be correctly interpreted as a signal of packet loss by the sender, which will then resend packet p.

As per RFC 2581 [24], the sender will use the *TCP Fast Retransmission* mechanism and assume that only the packet referenced by the duplicate ACK was lost. Shortly after having retransmitted p, the host will resume sending the frames that it was issuing before packet loss occurred. However, the naïve version of our algorithm expects in-order frame delivery, and will be expecting to receive the packet immediately following p: those new frames will also be considered out-of-order, as shown on figure 3.4. It will therefore send a new duplicate ACK, and this vicious circle will continue. The major effect of this phenomenon is that only one packet will now be exchanged during an RTT, thus greatly reducing achievable throughput.

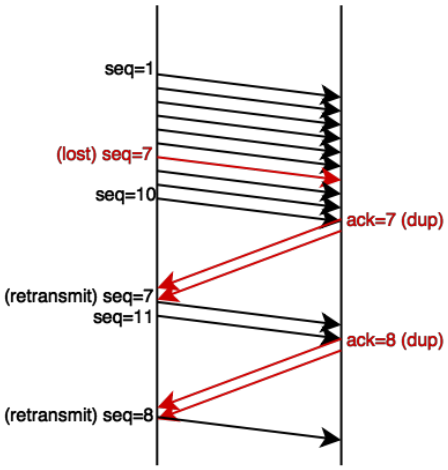


Figure 3.4: One packet per RTT phenomenon arising when ignoring TCP Fast Retransmission

In order to cope with this issue, we added an *out-of-order queue* in addition to the linked list representing the zero-copy stream. This new list will store nodes corresponding to packets that were not expected at the time of their reception, so that they can be reused later when fast retransmission occurs.

More precisely, if a packet p is lost, the stack will signal it to the sending side by issuing a duplicate acknowledgement. By the time this alert is received by the sending side, the stack will have received n successors of p, which will be put in the out-of-order queue. Once p has be retransmitted and received, the stack will enqueue p at the end of the zero-copy stream list, and dequeue its successors from the out-of-order queue to the end of the zero-copy main queue. At this time, the zero-copy list will contain p and its n successors whereas only one retransmission will have occurred (figure 3.5).

To that purpose, the algorithm presented in figure 3.3 is modified as follows:

- each time a packet is received, if it has a higher sequence number than expected, insert it in the out-of-order queue and plan to issue a duplicate ACK;
- otherwise, insert it in the zero-copy list, and try to dequeue immediately consecutive packets from the out-of-order queue and move them to the zero-copy list.

Since the out-of-order queue is probed each time a new packet is incoming, we implemented it as a doubly-linked list where sequence number ordering is maintained. That way, we can probe it and dequeue a packet from it in $O(1)$ time by looking at the first element. Insertion is done by browsing the list, which takes $O(n)$ time in the worst case (where n is the number of nodes in the list). However, it is very likely that out-of-order packets arrive in order after a loss. Hence, we chose to browse the list starting from the end when inserting new nodes, which leads to $O(1)$ insertion time in this best case.

Insertion in this queue requires extra care regarding overlapping or repeated packets. In the main zero-copy list, we only add a packet when its sequence number corresponds to the next byte to be received, thus automatically ensuring consistent ordering of the packets. In the out-of-order queue however, packets are added as they arrive. We already explained that we

keep this list ordered by the sequence numbers of its elements, nevertheless it is still possible that two identical or overlapping packets are inserted in the list. Consistency is ensured in this case by trimming incoming packets if necessary, so that the following list invariant is preserved: `node.seqnum + node.length` $\leq$ `node.next.seqnum`.

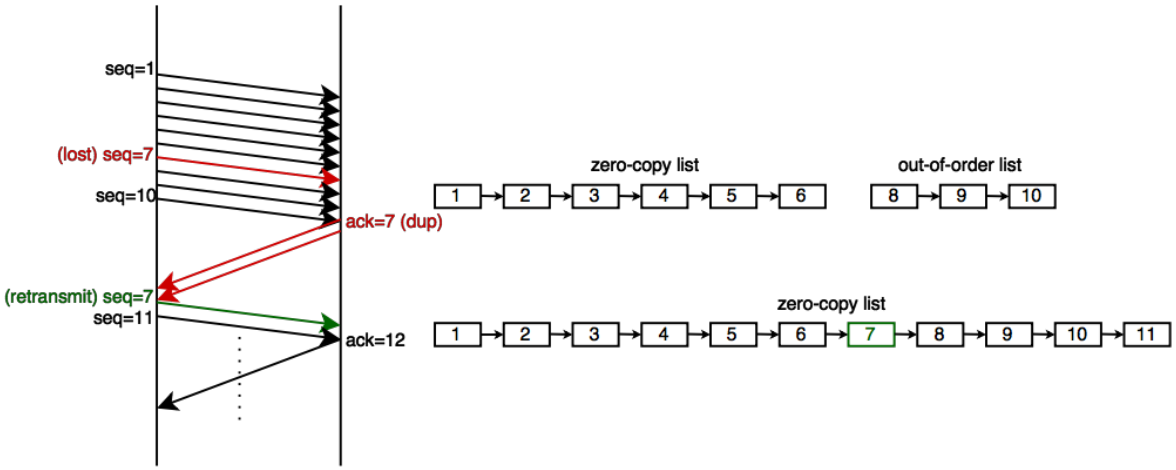


Figure 3.5: Using an out-of-order queue to cope with TCP Fast Retransmission

3.2 Implementation details

3.2.1 Integration with mTCP

In order to integrate this zero-copy mechanism into mTCP, we introduced a data structure `stream_fragment` (figure 3.6), representing a zero-copy node as explained in the previous section. These nodes are used in a structure `tcp_zc_stream`, associated with each socket and containing the zero-copy queue and the out-of-order queue mentioned in the previous description. The part of these nodes that will be transferred to the user when using the *read* API is simply a tuple $(start, end)$ of pointers delimiting the TCP payload of the node within the corresponding DPDK packet buffer.

Whenever the user chooses to use the zero-copy API for one of their sockets, the standard mTCP linear receive buffer is bypassed and our functions are used instead. The mTCP code base is still used to parse incoming packets and determine which stream they belong to. When the last function `TCPProcessPayload()` is called, if the stream is zero-copy enabled, our algorithm is then called in order to maintain the zero-copy list.

Two different threads are in charge of managing the zero-copy list: the network thread when packets are received, and the application thread when the user releases the nodes after having used them. Therefore, one could think that it would be necessary to use locks in order to protect the list from becoming corrupted. However, with a careful implementation of the different structures involved, it is possible to avoid using locking.

Indeed, the tail of the list only moves forward when new packets are received and added into the zero-copy list, and the head only moves forward when nodes are removed from it by the

Figure 3.6: Implementation of a zero-copy node within mTCP

```
struct stream_fragment_content {
        char* start; //start of the data fragment in the DPDK buffer
        char* end; //end of the data fragment in the DPDK buffer
};
struct stream_fragment {
        struct stream_fragment* prev; //NULL if beginning of the stream
        struct stream_fragment* next; //NULL if end of the stream
        uint32_t seq; //seqnum of this frame
        uint32_t rcv_nxt; //seqnum of next frame to be received
        struct rte_mbuf* mbuf; //handle to the DPDK mbuf containing this frame
        struct stream_fragment_content data; //payload contained in this frame
};
```

user. The only problem that can arise is when the queue becomes empty: with the traditional linked-list representation where `list->head == list->tail == NULL` when the queue is empty, the user thread will have to set `list->tail = NULL` when it removes the last node, which could corrupt the list if the network thread is adding a new node at this very moment. In order to solve this problem, we chose to always have a fresh node ready at `list->tail`. This way, the list is empty if and only if `list->head == list->tail` and the application thread does not have to set `list->tail` when removing the last node.

3.2.2 API exposed to the user

Since the zero-copy concept brings data from packet buffers up to the user application, the way this data is accessed by the application is a crucial part of the design. In a traditional Berkeley socket API, the application uses the `size_t read(int fd, void* buf, size_t count)` system call to fill a buffer provided by the user with up to `count` bytes of incoming data from the socket `fd`. In mTCP, the same concept is applied, except that the function is not a system call, is non-blocking by default, and moves data from two different threads in the same process rather than from kernel space to user space. However, its signature is very similar:
`int mtcp_read(mctx_t mctx, int sockid, char *buf, int len)`.

We tried to keep the same concept for our zero-copy API. We provide the following function for data reading:

```
int mtcp_zc_read_fragments(mctx_t mctx, int fd, struct stream_fragment_content* fragments, int n)
```

Similarly to the Berkeley API, the user is expected to provide an array of at least n fragments – which are actually $(start, end)$ tuples, as defined on figure 3.6. This function will fill the array with up to n such fragments, by fetching the first members of the zero-copy list. After having called this function, the user will therefore have direct pointers to the beginning and the end of the payload of the next packets of the TCP stream. We use the same conventions as the socket API: the function returns the number of nodes put in the array, 0 if the connection is closed by the remote host, and -1 with `errno = EAGAIN` if there is no data pending.

Once the application has finished handling the data received by this function call, it has to signal the stack that corresponding nodes can be removed from the zero-copy list and that

associated DPDK packet buffers can be released. This must be done before the *read* function is called again, so that consistency of the zero-copy list can be preserved. To that purpose, we provide the following function:
`int mtcp_zc_free_fragments(mctx_t mctx, int sockid, int n).`

Of course, the major disadvantage of this API is that it provides an access to the content of each packet of the stream, whereas a packet has no semantic meaning in TCP and the user might want to access data split across two consecutive packets. Nevertheless, this issue will be solved by enabling application-specific capabilities in the stack (chapter 4), and this simple API will prove helpful as a building block when implementing that solution.

3.3 Evaluation

We first ran a fine-grained experiment that aimed at evaluating the impact of putting one packet in the zero-copy stream, regardless of network conditions. To that purpose, we isolated the `TCPProcessPayload()` function and benchmarked it independently of the network stack, using fake packets held in RAM. This way, we can compare the time taken to process the payload of a TCP packet after its headers have been parsed and before the user retrieves it. Results have been averaged over one billion runs and are shown in table 3.1. Compared to the mTCP linear buffer, our zero-copy list offers better performance for incoming packet handling. This confirms that maintaining our data structure adds less overhead than what is gained by avoiding a copy of the payload.

	Average per-packet processing time (ns)	Relative performance
Zero-copy list	20.8	139%
mTCP linear buffer	29.0	100%

Table 3.1: Time spent to insert a packet in the zero-copy list versus the linear mTCP buffer

We then evaluated the soundness of the design by running a simple experiment, in which arbitrary data was transferred using both the mTCP *wget* example application and a simple zero-copy *wget* application. A standard Linux machine was used as the sending side, whereas the receiving side was a DPDK-enabled Linux machine running our modified mTCP. In order to have consistent results and to minimise the impact of TCP slow start and congestion control, experiments conducted throughout this project consisted in transferring 10 GB of data and recording the highest throughput achieved in a $1s$ time window. Data was fetched directly from RAM to avoid hard-disk I/O bottlenecks. The following hardware has been used:

- CPU: 2 Intel Xeon E5-2690 (2.90GHz, 8 physical cores, 16 logical cores)
- RAM: 32 GB
- NIC: Intel 82599EB 10 Gigabit or Broadcom NetXtreme II BCM57711 10 Gigabit

We ran the experiment with different receive buffer sizes in order to simulate different network conditions (figure 3.7). Recalling that the throughput T that can be achieved with a receive

window of w and a RTT of Δt is such that $T \leq \frac{w}{\Delta t}$, one can explain the linear behaviour of the first part of the curve. Since no data processing is performed, the main bottleneck in this experiment is w.

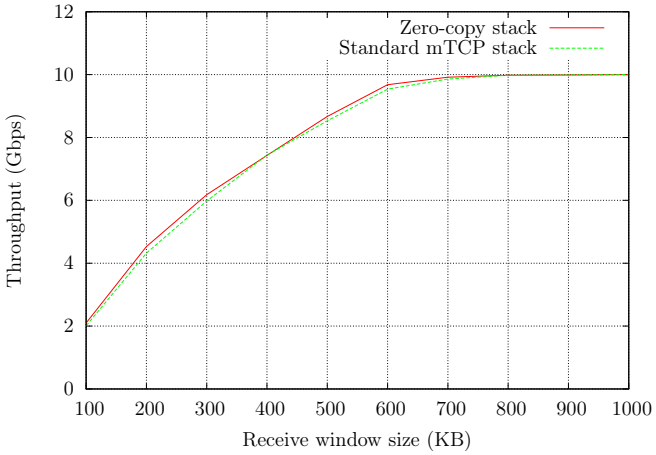


Figure 3.7: Performance of zero-copy mTCP vs standard mTCP when downloading a large file

Both versions show sensibly similar results and can process as much data as the network can provide given the w parameter. They can saturate the 10 Gbits NIC as soon as the receive window is sufficiently large. Our version has slightly better results (with a factor of up to 3%), probably because the per-packet processing overhead is lower, thus enabling the time from data reception to acknowledgement to be reduced.

To conclude, this evaluation confirms that the zero-copy design that we adopted is sound and does not induce any significant overhead. It can therefore be confidently used as a basis for an application-specific API, as we will discuss in chapter 4.

4. Application-specific data deserialisation

The essential idea underlying this project is that the standard socket API, despite its flexibility, does not necessarily fit every kind of application. When the format of the data meant to go through the wire is known in advance, using the classical stream abstraction where the application receives data byte by byte is not efficient. Section 4.1 will discuss this issue in greater details. As a solution to this problem, we introduce an approach where data deserialisation is done in the network stack and structured pointers are passed to the application as a result. In section 4.2, we first focus on a simple yet universal data format, Comma Separated Values (CSV) files, as a means to validate our design. Finally, in section 4.3 we extend this idea to richer data formats. We first focus on HTTP parsing before generalising to user-specified formats, for which we introduce a macro-based code generator.

4.1 A synchronisation issue

4.1.1 Rationale

Any flexible enough TCP stack needs two different contexts for the network and the application, so that demultiplexing can occur smoothly and that overall control can be separated from network processing (which requires timers for retransmissions). In a standard kernel mode stack, the network context will be a kernel thread and the application context will be a user process, whereas with mTCP these contexts are two different threads in a user process.

Since context switches can happen at any time, there will be a delay between the moment when a packet is processed by the network thread and the moment when it is available to the application. Parsing data in the network context could avoid wasting this time and take advantage of the fact that packets are still in the CPU cache while they are processed. Therefore, it seems likely that moving the data parsing to the network thread can improve the overall latency.

By doing so, the application will only access the data when it is ready for it. In contrast, if data such as application-layer headers is split across two different packets, with the standard approach the application could be woken up from a `read()` system call only to notice that it had not received a sufficient amount of data and that to perform another `read()` call, and upon completion possibly restart parsing from the beginning.

We ran a simple experiment to confirm the soundness of this idea. We wrote a standalone program (independent of mTCP and any real network operation) consisting in two threads affinitised on the same core, modelling a network thread and an application thread. The network thread receives a burst of packets into a buffer (if it is not full), a process simulated thanks to a dummy `for` loop, and then signals the application thread that there is data pending (using the `pthread` signalling API). On the other hand, the application thread waits for data to be available and then removes it from the buffer. This way, this simple consumer-producer model reproduces the event-based polling approach used in mTCP.

We then introduced a variable-length dummy `for` loop representing parsing data of different complexities. The experiment, whose results are shown in figure 4.1, consisted in running this data parsing loop in the network thread as well as in the application thread. Results show that it is indeed more efficient to outsource data processing to the networking thread, with a factor of 1.6×.

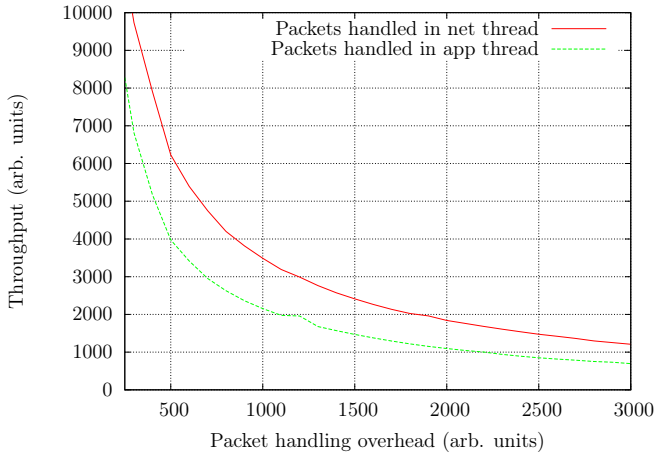


Figure 4.1: Simulation of a network stack with two threads, where data is processed either in the network thread or the application thread

4.1.2 Design validation within mTCP

The next step in verifying the validity of this idea is to check what results it gives when integrated into a real stack, where other factors can also have an important role. Using the same idea, we modified our simple zero-copy `wget` application to simulate packet processing. We inserted a dummy `for` loop representing data parsing in two different locations: (i) in the application, just after have received packets through the `mtcp_read_fragments()` call and (ii) in the network thread function `ProcessZCTCPPayload()`, just after having inserted a node in the zero-copy list.

Once again, we used different sizes for the loop to simulate parsing processes of different complexities. Of course, this approach is simplistic since it assumes that parsing a packet always takes the same amount of time, whereas in realistic scenarios this will be different according to the contents of the packet. Nevertheless, it provides a valuable first insight before actually implementing such parsing features. Results can be seen in figure 4.2 and show that parsing a packet directly after it is received rather than when it is processed by the application thread can

give a performance benefit of up to 2×. This confirms the idea that doing so avoids wasting time by efficiently managing shared resources.

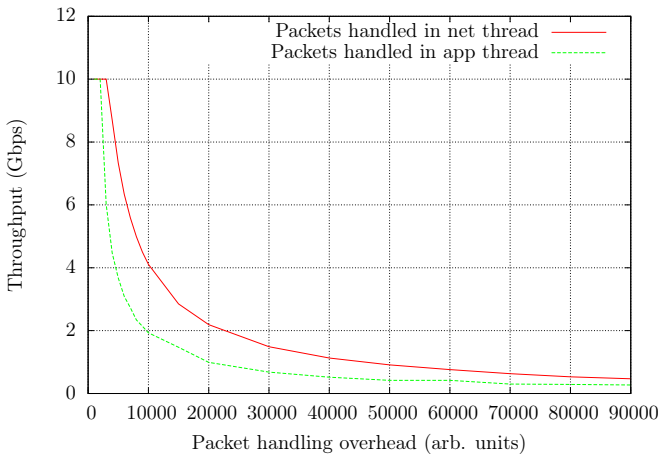


Figure 4.2: Parsing packets in the network thread versus the application thread within zero-copy mTCP

4.2 Simple application-aware processing: CSV parsing

4.2.1 Data structures used

We first focussed on a simple data format to introduce an application-aware processing mechanism, that is, comma-separated values files. This format, although simple, allows us to sketch the structure that a richer application-specific module must have. We will refer to lines in a CSV file as *records*, and entries in a line as *fields*. This point of view enables us to model various situations, for example:

- from a database point of view, an entry in a table is a record of fields,
- from a Big Data processing perspective, a $(key, value)$ tuple is a record of two fields,
- at the application-layer protocol level such as HTTP, a transaction is a record of HTTP headers.

In order to enable the application to access deserialised data using zero-copy techniques, we added a new linked list on top of the *zero-copy stream* introduced in section 3.1. We will call this new list the *application-aware stream*. Nodes in this list represent a record in the CSV file: they contain n entries, each representing a field in the current record – where n is the number of entries per line in the CSV file, fixed in advance. Fields are structures containing a pointer to the beginning and the end of a CSV column in a DPDK `mbuf`. Whereas in the simple zero-copy *read* API introduced in the previous chapter the application reads nodes representing packets, with this new application-aware stream the application will directly access nodes representing deserialised data, as shown in figure 4.3.

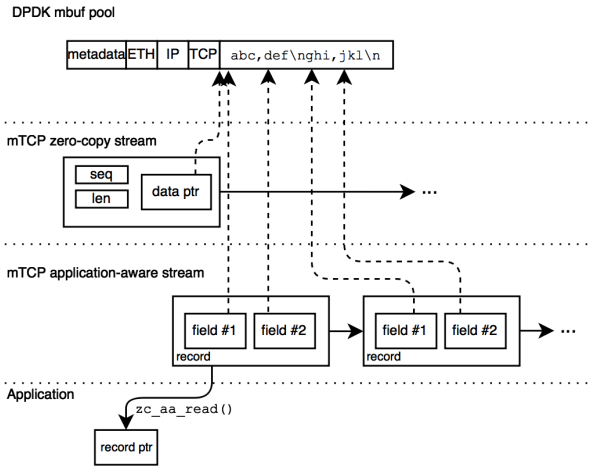


Figure 4.3: Zero-copy application-aware stream design

The algorithm introduced in figure 3.3 in charge of maintaining the zero-copy stream must now also maintain the application-aware stream. This is accomplished the following way:

1. Just after a packet is put in the *zero-copy stream* – either because it is the next packet to be received, or because fast retransmission occurred and it has been dequeued from the out-of-order queue – it is parsed. If new records are successfully parsed, they are enqueued at the end of the *application-aware stream* and a new empty node is added. If a record is not entirely parsed because it is split across two consecutive packets, it is still placed at the end of the list, and parsing will resume when the next packet is received.
2. When the user application requests to read n application-aware records, up to n pointers to such records are passed to the application, depending on how many are pending.
3. When the application has finished processing the records it has obtained through this means, it must release them. This way, they are removed from the *application-aware stream*. Then, nodes from the underlying *zero-copy stream* corresponding to packets that do not contain any more records are removed from it, and associated DPDK buffers are freed.

There are two subtleties to take into account with this algorithm. First, in step 3 above, it is necessary to track down which packets are associated to which fields, so that corresponding nodes from the *zero-copy stream* can be freed when records are released by the application. Therefore, each field structure also contains a pointer to the node owning it in the zero-copy stream. A naïve algorithm to determine whether a packet node can be deleted would be to browse the application-aware stream and check whether there are still fields owned by this packet, however such an approach would have a $O(n)$ complexity, where n is the size of the record list. Hence, we used a more efficient approach, where each packet node from the zero-copy stream has a reference counter maintained equal to the number of fields owned by the packet. Only a $O(1)$ time is thus needed to determine whether the node can be freed.

Second, it is possible that a field is separated across two consecutive packets. Therefore, our field structure not only contains a tuple indicating the beginning and the end of a CSV field in the DPDK buffer, but also a *next* pointer to another field if the field is not self-contained in a

packet. We also implemented string processing functions so that the user can manipulate a field as if it was one single entity, be it chained to another one or not.

4.2.2 Implementation details and API exposed to the user

We integrated the mechanism previously introduced into mTCP by adding a new structure representing the application-aware stream, namely `tcp_zc_aa_stream`. Similarly to the zero-copy stream introduced in chapter 3, it is a structure representing the list of records buffered to the application, with a pointer to the head and the tail. Instead of interacting with the `tcp_zc_stream`, the user interacts with this new structure and nodes from the underlying `tcp_zc_stream` will be managed automatically, as explained in the previous section.

We reused the idea of never setting the tail of the list to `NULL` but to a fresh record, so that the application thread only needs access to the head whereas the network thread only uses its tail. This enables us to use lock-free structures, improving the global latency of the system by avoiding synchronisation issues.

As explained previously, nodes of the stream represent records in the CSV file, and are implemented as a `stream_record` structure, as defined in figure 4.4. A field in a record is implemented thanks to a `field` structure and is in fact a linked list, in case it continues on another packet. In order to manage such fields, we rewrote some standard string manipulations functions to be compatible with the `field` structure, in no particular order: `get_byte()`, which gets the i-th byte of a field; `fieldstrcmp()`; which compares a field to a given string; `fieldstrstr()`, which looks for a given substring inside the field; `fieldatoi()`, which converts a field to an integer representation.

Figure 4.4: Implementation of an application-aware zero-copy node within mTCP

```
struct field {
    struct stream_fragment_content data; //delimiter of the data in the payload
    struct field* next; //if this field is not self-contained
};
struct stream_record {
    struct stream_record* next; //NULL if end of the stream
    struct field* fields[0]; //fields in this record
};
```

On the same model as the API introduced in section 3.2.2 to retrieve nodes representing packets, we expose a similar API to the application so that it can retrieve application-aware records. We introduce a function
`int mtcp_zc_read_aa_records(mctx_t mctx, int sockid, void* records, int n)` aiming at obtaining up to n records in the user-provided array `records`. When the application has finished processing them, it shall call `int mtcp_zc_free_aa_records(mctx_t mctx, int sockid, int n)` to release them before calling the *read* function again.

Even if the CSV format is quite simple, the design introduced in this section is important

because it will serve as a base for richer data formats that will be introduced in section 4.3, where the concept of records composed of fields delimiting data in the TCP stream will be kept.

4.2.3 Evaluation

In order to evaluate the performance of this implementation, we modified the mTCP *wget* application so that it uses this new API to read CSV records instead of using the standard mTCP *read* API and receive raw data from the server. To that purpose, we used a standard Linux machine as a server, and a DPDK-enabled machine as the client – the hardware features of these machines are the same as in previous experiments. The server is set to send arbitrarily long CSV files whose fields have a predetermined length. In order the server not to be a bottleneck, a chunk of CSV data is generated once and for all in RAM and the server loops over it while serving the client. Plus, a large buffer is used for the `write()` system call, so as to minimise the impact of context switching on the server.

Varying the length of the CSV fields is a means to represent different levels of complexity in the parsing process. Indeed, the longer the fields are, the less records there will be in a frame, reducing the overhead of parsing a packet. Therefore, we ran the experiment for different field sizes: for each run, a 10 GB download was completed. To compare our zero-copy implementation to a reference one, we wrote an application that uses the normal mTCP API and performed the same CSV parsing.

Results of the experiment are reported in figure 4.5. According to these, there is indeed a performance benefit to use such an approach. This can be intuitively justified by a better management of the resources, leading to more stable bufferisation. Indeed, when a standard linear byte stream is used, data risks being highly buffered before the application thread is preempted to retrieve it, and the overall system will go back and forth between buffering data and parsing it. With our approach however, data is parsed each time a packet is received, leading to a more stable distribution of the computation time.

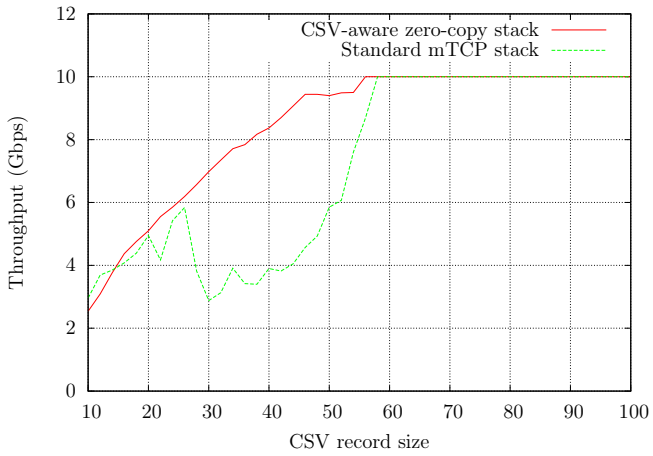


Figure 4.5: Evaluation of zero-copy application-aware CSV parsing

4.2.4 Scalability considerations

We also evaluated the scalability of the approach by running a similar experiment with multiple parallel connections. We kept our Linux-based server that generates an CSV stream. On the DPDK client side, we simultaneously connected several sockets to the server, each one being pinned to its own CPU. This is made possible through the use of the Receive Side Scaling (RSS) feature of the NIC. In a nutshell, the NIC has several receive queues, and each one is bound to a CPU at the application startup. Upon receipt of a packet, the NIC inspects its source address, source port, destination address and destination port and maps this 4-tuple onto a CPU. This way, a given incoming TCP flow is always managed by the same CPU. On the other hand, when the client initiates a connection, it has to carefully chose a source port so that further incoming packets are received on the same CPU.

As before, we launched a 10 GB download for different CSV fields size, and with different numbers of concurrent connections. Results are available in figure 4.6, and show that the in-stack parsing approach scales linearly with the number of CPU used, as one could expect. The NIC can be saturated regardless of the parsing complexity as long as there are at least 4 parallel connections, showing an efficient usage of the resources.

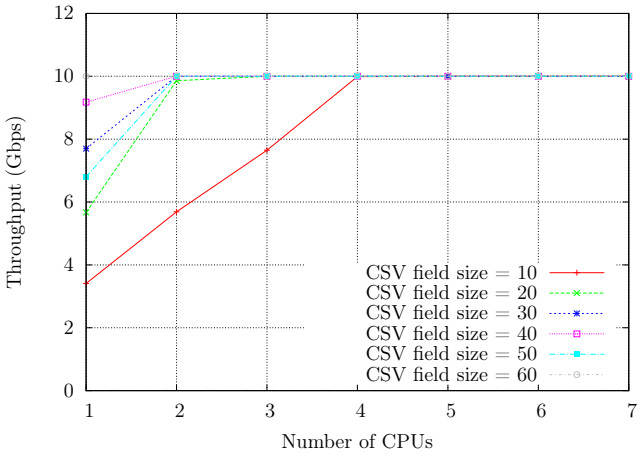


Figure 4.6: Scalability of zero-copy application-aware CSV parsing

4.3 Richer application-aware deserialisation

4.3.1 The example of HTTP parsing

Having introduced a simple application-aware stream, we can now extend this design further to handle more expressive data formats. We will first focus on an example to then generalise the approach to arbitrary formats specified by the user. To start with, we will consider a scenario where a host processes HTTP queries or answers. As with CSV parsing, we will keep records of fields as the base unit of the application-aware stream that will be passed from the stack to the application. However, they will have a richer semantic meaning than previously.

Let us first focus on the side of the connection corresponding to a server passing content to a client. As per HTTP 1.1, connections can be kept alive: a good representation for an HTTP-aware record could therefore be an entire response from the server – that is, the headers and the contents in reply to a query. However, if the file that is being transferred as a result of the query is large, the record will be spread across many packets, which will cause two problems. First, these packets will have to be kept in the zero-copy stream until completion of the transfer, and corresponding DPDK buffers will not be released to their memory pool, causing an inefficient usage of resources and possibly exhaustion of the memory pool. Second, these will cause the application to block until full retrieval of the file, while the application might be simply interested in inspecting the HTTP headers.

Therefore, we choose to introduce two different kinds of records, as shown in figure 4.7: one representing the metadata sent by the server, and the other representing a chunk of data of the actual file returned by the server. We make the assumption the the application is not interested in deserialisation of the content of the answer: answer records will just contain one field containing pointers to the raw data. If the answer is divided up on several packets, one record will be issued per packet to avoid the bufferisation phenomenon explained above. Depending on the application needs, other approaches could of course be used such as also inspecting the contents, and the mechanism that will be introduced in section 4.3.2 can cover such cases.

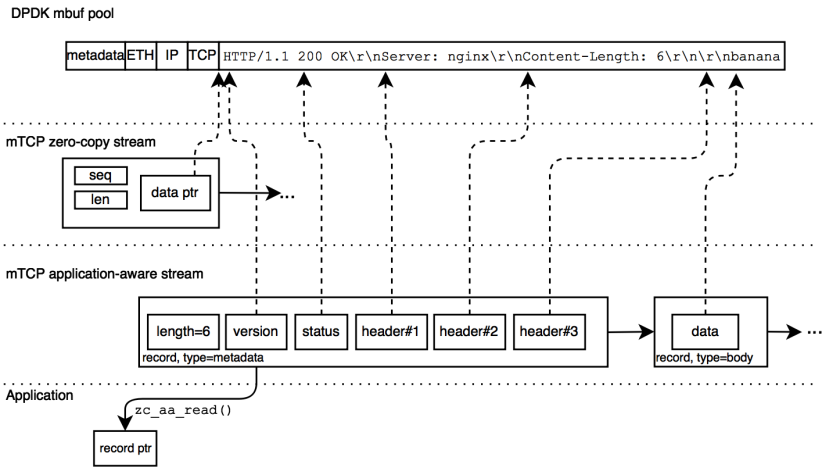


Figure 4.7: Zero-copy HTTP-aware stream design

The first type of records, representing metadata sent by a server in response to a query, contains the following fields:

- the HTTP version,
- the HTTP status code,
- an arbitrary (but fixed in advance) number of fields representing a header entry.

In addition to this, it contains an integer representing the `Content-Length` of the answer. Since there are now two different kinds of records to consider, and since fields in the metadata records are more diverse than before, the application-aware stream had to be extended so that it contains a state machine controlling the transition between the two types of records, as presented in table 4.1. Knowing the answer length enables the state machine to correctly determine the end of the

data transmission and to revert back to the header mode, waiting for the next transaction to occur.

Whereas for the CSV application-aware stream, parsing simply consisted in looking for commas or carriage returns, in this new case, each field has its own function that locates its end. Once this is done, a function is called that enables the state machine to modify some variables and choose what field or record to parse next. Even if we chose a particular implementation for our HTTP records, this approach is flexible and can be extended to a wide variety of data formats. This is made possible by the use of custom post-processing functions, which allow arbitrary operations on the state to be performed.

Record	Field	Finding end of field	Post-processing
Answer	HTTP version	`"␣"`	–
	Status code	`"\r\n"`	–
	Header i	`"\r\n"`	if header starts with `Content-Length`, parse it into `len` if empty header, start new *Body* record
Body	Data	`len`-th byte or last byte of packet	`len -= size(field)` if `len` = 0, start new *Answer* record otherwise, start new *Body* record

Table 4.1: State machine for HTTP answer parsing

If one is interested in deserialising the other side of an HTTP transfer, corresponding to the client sending queries to the server, that can be treated essentially the same way, except that *Answer* records will be *Query* records, and their format will be slightly different. This time, they would contain the following information:

- the query type (`GET`, `POST`, ...),
- the URL,
- the HTTP version,
- fields representing HTTP headers.

To summarise, extending the *record and fields* approach to data formats richer than CSV shows to be an approachable task, given that the format is well-specified. This enables us to a simple zero-copy design, where the application accesses fields representing a subset of the TCP payload. The application will be delivered a record only when it is entirely parsed, avoiding an inefficient usage of resources that can happen with the standard `read()` API since the boundaries of the data returned by the system call have no semantic meaning. On the contrary, with our method, the application is directly provided with structured data by the API. Records are indeed implemented as `structs`, whose members can be easily accessed and manipulated thanks to string processing functions previously introduced in section 4.2.2.

4.3.2 User-defined application-aware modules

Now that we have studied specific examples of data deserialisation, we can sketch out a more general method to define parsing modules in a simple way. With the previous examples in mind, such modules will share a common basis that we will describe in this section.

First, a module must define the structures that it will use as a way to store deserialised data. For this, it must define an *application-aware stream*, similar to the one introduced in section 4.2.2, whose aim will be to keep a linked list of records buffered for the application. Records will be structures containing named fields (as defined in figure 4.4) and possibly additional variables. As explained for the example of HTTP parsing, for greater flexibility it is useful to introduce several types of records. For each field, the module must first be able to locate its end; once this is done, a post-processing function can be defined to alter the state of the system and possibly switch record types. In order to put this in practise, the module must expose several functions that will be used by mTCP:

- a function to initiate the data structures, which will be called upon creation of a TCP flow, and another to destroy them upon its destruction;
- a function to pass records from the stream created by the module to the user, which will be called by the `mtcp_zc_read_aa_records()` API;
- a function to remove records from the stream, and update the reference counter of the corresponding nodes of the underlying zero-copy stream, which will be called by the `mtcp_zc_free_aa_records()` API;
- a function to parse a packet and create or update records accordingly, which will be called each time an in-order packet is received or removed from the out-of-order queue.

From this discussion it appears that application-aware modules share a similar structure. Hence, it is possible to automatically build such modules, as long as that the user provides a rather simple description of the format of the data. To that purpose, we chose to use a system based on C preprocessor macros. Another possible approaches would be to specify a small language for data format descriptions and provide a tool to translate it to a C source file. However, this would increase the complexity of the overall system. Using a macro-based system, on the contrary, enables the user of our system to write their module description inside their source file and without resorting to external tools, despite a less elegant syntax.

More precisely, the user is supposed to use certain macros to define the format they would like to use, and then include a header file `generate_aa_api.h` that will generate appropriate records and stream structures, as well as the functions mentioned above. Figure 4.8 lists the macros that must be used by the user to that purpose.

In the macro used to register fields, the `<end_of_field>` and `<post_processing>` arguments should be snippets of code aiming at finding the end of a field and to post-process it once this is done. Macros are also provided to simplify this task, as we will briefly describe. To help finding the end of a field, we provide a `STRSTR()` macro that looks for a given substring in the field, and a `OFFSET()` macro that returns a constant offset inside the field. To assist in field post-processing and therefore in the evolution of the state machine, the following macros are implemented:

- `GET_VARIABLE()` and `SET_VARIABLE()` to access or modify a global variable,
- `GET_RECORD_VARIABLE()` and `SET_RECORD_VARIABLE()` to access or modify a variable in the record, that the user will then be able to retrieve simply inside the `struct` representing the record,

Figure 4.8: Preprocessor-based definition of an application-aware module

```
//register an identifier for the format
PROTOCOL <identifier>
//register variables to control the overall state of the system
REGISTER_VARIABLES() \
    VARIABLE(<identifier>, <type>)\
    [VARIABLE()...]
//register the different records types
REGISTER_STATES()\
    STATE(<identifier>, <name>)\
    [STATE()...]
//register the fields corresponding to a given record type
REGISTER_FIELDS_<record_id>()\
    FIELD(<field_id>, <field_name>, <end_of_field>, <post_processing>)\
    [FIELD()...]
//register non-fields variables that will be included in a certain record type
REGISTER_VARIABLES_<record_id>()\
    VARIABLE(<identifier>, <type>)\
    [VARIABLE()...]
```

- `END_OF_RECORD()` and `SET_STATE()` to indicate the end of a record and set the type of the next one to come.

The data structures and functions generated by these macros will be bundled in a structure representing the application-aware module, and should be registered within mTCP after creation of the socket that will use it. The advantage of this approach is that it does not require a recompilation of the mTCP library each time the developer wants to try a new data format. Instead, the functions generated by the preprocessor-based system are exported as callbacks and will be called by mTCP each time it is needed. This way, the application can simply use the API functions that we have introduced in section 4.2.2 to read and release records.

To conclude, the generality of this macro-based approach makes it suitable to construct deserialisers for relatively different data formats. This allows the user of the system to be relieved of the burden to understanding the implementation details underlying our application-aware stream design, and to focus on describing the data rather than writing code. Furthermore, it ensures that the code generated is compatible with our implementation and will not break its functioning.

As an example, we have rewritten our HTTP answer module using this approach, which is pretty straightforward since we imagined this new approach by generalising the former. A listing of the module is given in the appendix and gives an idea of the conciseness of the approach. While about 50 lines of codes are needed to describe the HTTP module using the macro-based generator, the C code that is generated as a result is approximately 500 lines long.

4.3.3 Evaluation and scalability

Introducing data deserialisers that are more expressive than the simple CSV parser built as a starting base allows us to perform a more thorough evaluation of the stack. To that purpose, we used the HTTP parsing module on a scenario similar to previous experiments. In order to

simulate an HTTP 1.1 session with multiple downloads, we wrote a simple server that indefinitely sends HTTP answers, each consisting of seven headers and a data payload. On the other hand, the client acts as a simple sink which fetches these headers and answers and parses them. This way, the client has to perform a parsing that is less static than with CSV: the parser will alternate between times when it is very busy processing the headers and times when it has few things to do but to count bytes until the file is entirely transferred, before going back to the previous state.

The experimental testbed, as in section 4.2.3, consisted in a machine running a Linux TCP stack as the server, and a DPDK-enabled machine as the client. We ran 10 GB sessions so as to have a stable throughput and test the parser on a subsequent number of records. In order to evaluate the parser against different data complexities, we varied the size of the HTTP answer sent after each set of headers: the bigger the HTTP file size, the longer the parser is idle producing *Answers* records that require less resources to be built, hence the higher the throughput. In order to have a reference implementation, we reproduced the same parsing mechanism in a standard mTCP application, as well as in a application using the Linux TCP stack[1].

Results are shown in figure 4.9: our stack is able to handle a client continuously requesting files of size greater than 1000 bytes, and shows better results than the standard counterparts. This can be explained by the same reasons as previously: parsing packets upon their receipt instead of waiting for the application to receive them allows to avoid a bufferisation effect between the network and the application threads.

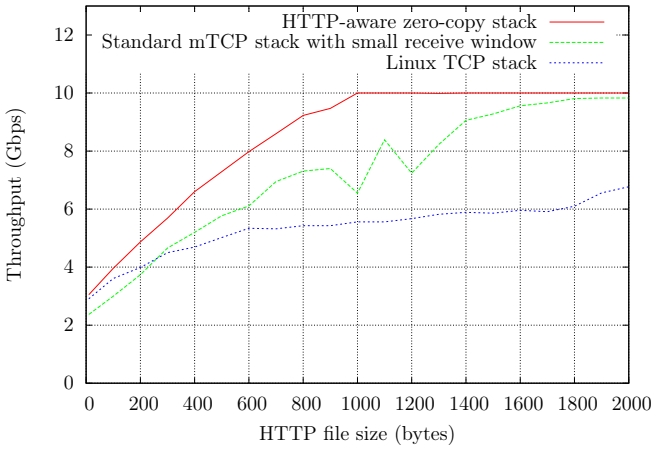


Figure 4.9: Evaluation of zero-copy HTTP parsing

Scalability of the system has been evaluated by running the same experiment on our zero-copy stack, this time with multiple concurrent connections. As with the experiment conducted with CSV parsing, each socket was affinitised with its own CPU. This allows a better exploitation of the capabilities of the NIC by giving more processing power to the overall system. In cases where the NIC cannot be saturated due to a too expansive parsing, the remaining capacity of the NIC can now be exploited by another CPU to handle another connection. Figure 4.10 shows that, with 5 parallel connections, the link is saturated regardless of the complexity of the parsing.

[1]Since DPDK does not have such a feature, we had to disable the Generic Receive Offload on the Linux client, which aims at reassembling many packets into one before giving them to the stack so as to diminish the per-packet overhead

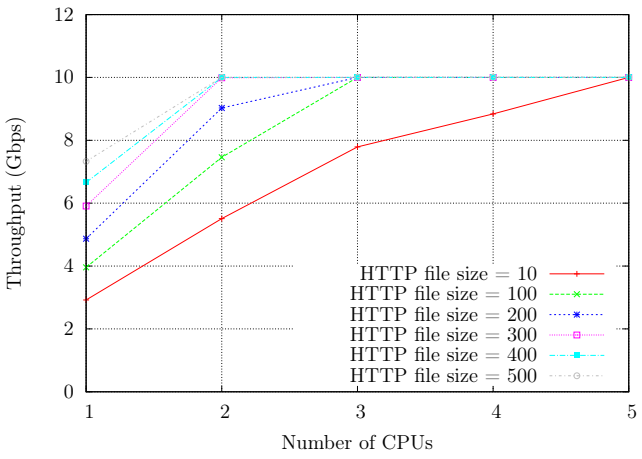


Figure 4.10: Scalability of zero-copy HTTP parsing

To summarise, we extended the zero-copy stream introduced in chapter 3 by creating another stream abstraction on top of it. These streams represent structured data that the user application can directly retrieve instead of having to go through a byte array with no semantic meaning. Each record of data consists of a certain number of fields, each of these effectively made of pointers to the relevant parts of the TCP payload in DPDK buffers. This way, no data is ever copied from the packet buffers, and the user application is not interrupted until a complete record is available: it interacts with TCP as an object provider, not as a blackbox that delivers a stream of bytes. We then introduced a macro-based interface so that the user can declare which data format they want to interact with. Evaluation of the resulting system reveals that the underlying design is robust and allows a performance benefit over a standard implementation. In chapter 5, we will study how this mechanism can be integrated in a middlebox environment, where the stack inspects traffic on the fly.

5. Application-specific processing in middleboxes

The final step of this work consists in using the zero-copy design introduced in chapter 3 and the application-aware processing mechanism of chapter 4 in a middlebox scenario, which will be discussed in this chapter. More precisely, our stack will run on a host that will act as a relay between a client and a server. While forwarding the data between the server and the client, the stack will be able to deserialise it in order to gather information or to modify it on the fly.

To that purpose, we implement a forwarding mechanism on top of our previous API for data reception, keeping on using zero-copy ideas. In fact, this is the very scenario where the zero-copy paradigm can be exploited to its fullest benefit: the same packet buffer can be used for inbound and outbound purposes with only the headers needing be modified. This avoids four data copies between intermediate buffers (figure 3.1) and allows an efficient usage of CPU caches.

In the rest of this chapter, we will first discuss the structures needed to provide a simple TCP proxy (section 5.1). Section 5.2 will then focus on adding application-specific processing abilities to the system and evaluate the impact it has on performance.

5.1 Simple zero-copy TCP proxying

5.1.1 An outbound zero-copy packet list

The first step towards building a zero-copy forwarding mechanism is to introduce relevant data structures for outbound stream representation. As for input streams, mTCP uses a linear buffer to store data waiting to be sent or acknowledged. In the case of a standard proxy that consecutively uses `mtcp_read()` and `mtcp_write()` on two different sockets, it means that data will be copied four times:

1. from inbound DPDK buffers to the linear mTCP buffer associated with the server socket,
2. from this mTCP buffer to the user-owned buffer,
3. from the user-owned buffer to the linear mTCP buffer associated with the client socket,
4. from that mTCP buffer to outbound DPDK buffers.

On the other hand, our design aims at avoiding any data copy by modifying packet headers directly in DPDK `mbufs`. To that purpose, we use the same idea as for the zero-copy input stream

(section 3.1) and maintain a linked list of nodes representing pending egress packets, as shown in figure 5.1. As for the input zero-copy stream, these nodes contain a pointer to the corresponding DPDK packet buffer, a pointer to the beginning of the TCP payload and its length, as well as the sequence number of the packet. Upon insertion of a node in the list, the DPDK `mbuf` should come from an inbound TCP stream and the packet headers will thus be obsolete. Only when the list is flushed will the headers be overwritten with appropriate ones by mTCP functions.

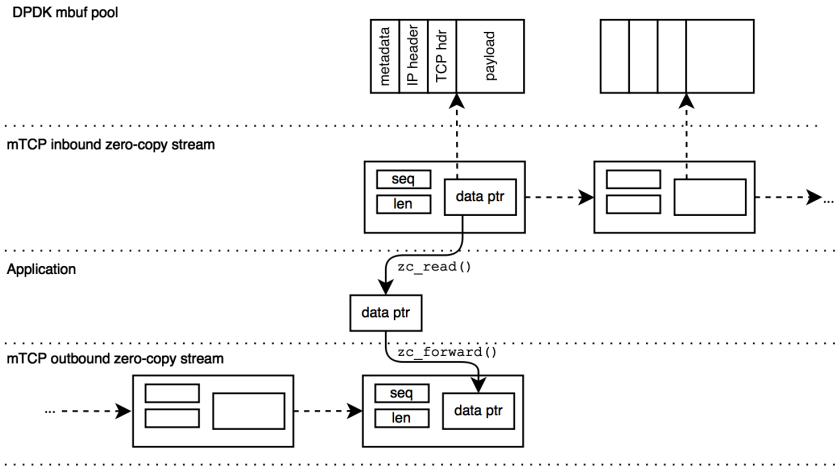


Figure 5.1: Zero-copy forwarding design

The list actually comprises two parts: the first one consists in sent but unacknowledged data, whereas the second one represents buffered data yet to be sent. This is implemented thanks to the use of three pointers: `list->head`, `list->snd_nxt` and `list->tail`. The algorithm used to maintain the list is the following:

1. When the application requests to forward a packet p on a socket s, a node representing p is enqueued at the end of the list corresponding to s. The sequence number of the node is set to `list->tail->seq + list->tail->len`, *i.e.* the first sequence number available at the end of the stream.
2. When mTCP enters its transmit loop, the list is flushed.
   (a) This is done by first looking for the `list->snd_nxt` node (if existing) and checking whether its sequence number is equal to the one of the next packet expected by the other host.
   (b) If not, it means that a duplicate ACK has been received and that retransmission must occur. In such a case, `list->snd_nxt` is reset to `list->head` and the list is browsed until the node whose sequence number matches the one expected is found.
   (c) Once `list->snd_nxt` is set appropriately, the list is browsed node by node until its tail is reached or the client send window is full – or the maximum of packets to send in one burst has been reached. For each node browsed, TCP headers are rewritten accordingly and the packet is sent to the wire.
   (d) At the end of the process, `list->snd_nxt` is set to the next packet to send (or `NULL` if not applicable).
3. Upon receipt of an ACK packet, nodes are removed from the list whose sequence number

is lower than the sequence number acknowledged. For each node removed, the associated DPDK buffer is released.

A subtlety arises when it comes to managing the memory occupied by the `mbufs`. Indeed, when a packet buffer is given to DPDK for transmission, it is the responsibility of the driver transmit function to release it; however, we need to keep the buffer in the unacknowledged queue in case retransmission should occur. Fortunately, DPDK uses a reference counting mechanism for its buffers, and we can therefore increase the reference count of a buffer before flushing it to the wire in order to avoid a premature release by the framework.

5.1.2 Merging two connections

From the point of view of the application, an API is needed to utilise these new data structures and actually achieving data forwarding between two sockets. We explored two different approaches with different properties, which will be described in this section.

The standard API

The natural idea for writing data to a socket is to imitate the Berkeley socket API and provide resembling functions – we will call this the *standard API*. With this approach, presented in figure 5.2, it is the responsibility of the application to initiate the data forwarding. After having received fragments from the server socket through the `mtcp_zc_read_fragments()` function (introduced in section 3.2.2), the user is free to inspect them and possibly modify their content. Once this is done, instead of calling `mtcp_zc_free_fragments()`, we provide a function `int mtcp_zc_forward(int sockid_from, int sockid_to, int n)`.

This function will act similarly to `mtcp_zc_free_fragments()` with respect to the ingress socket, except that it will not release the DPDK buffers corresponding to the packets. Instead, it will place nodes representing them in the output zero-copy stream associated to the egress socket.

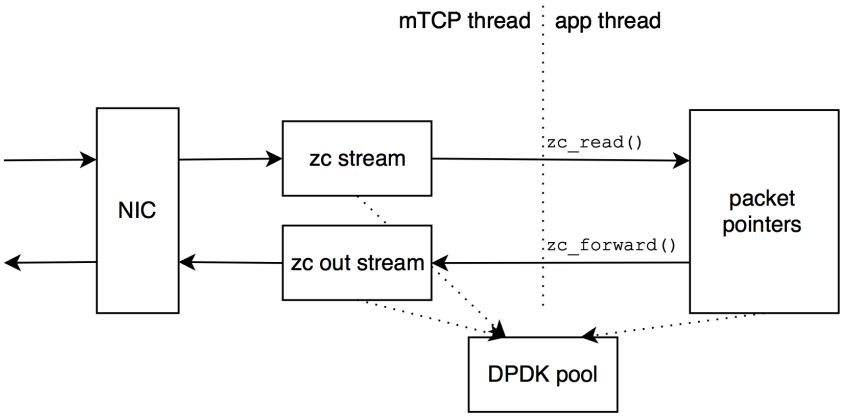


Figure 5.2: Overview of the *standard* forward API

As with a regular TCP stack, flow control is provided thanks to the use of a send window w, which in this case is computed as $w = w_{max} - s$, where s is the size of the output zero-copy stream

and w_{max} a configurable constant. If the client cannot process the data fast enough, eventually we will have $w = 0$. When this occurs, our function `mtcp_zc_forward()` blocks until new room is available. Until this, the application will not be able to call `mtcp_zc_read_fragments()` again, which will likely fill the receive window, causing the server to stop sending data and regulating the flow.

The advantages of this approach is its resemblance to a standard TCP API and its flexibility: for each packet, the application can choose whether and where to forward it, and to modify it before doing so. Nevertheless, it does not offer satisfying performance, as shown in table 5.1. This can be explained by two main factors. First, the fact that the packets are not sent until the application thread reads them and decides to forward them induces latency and, more critically, at least two thread switches between data receipt and forwarding. This can also have an impact on caching: packet buffer might be invalidated from the CPU cache before they are forwarded, removing some of the benefits of a zero-copy approach.

Second, a bufferisation phenomenon occurs that leads to an inefficient use of resources. Indeed, in on of our test setups, the Linux server stack could send data faster than the other Linux client stack could receive it. Therefore, an oscillating process takes place: in a first step, the server sends as much data as possible to the middlebox, which transfers it to the client at the maximum rate possible. When the middlebox sending buffer is full, it cannot empty its receive buffer, which quickly becomes full. At this point, a second phase occurs where a zero-window is advertised to the server, which stops sending data, while the client can process the remaining of the data. This process goes back and forth, alternating between burst transfers and idle periods, leading to poor performance.

The splice API

In order to overcome this issue, we introduced a second API with which the user states once and for all that two different sockets should be merged, following the ideas of [9] – we will refer to this second approach as the *splice API*. In this case, a function `int mtcp_zc_splice(int sockid_from, int sockid_to)` shall be called before connecting the socket from which to transfer the data. Then, each time the stack receives it an incoming data packet from the server socket and adds it to the inbound zero-copy stream, it immediately adds it to the outbound zero-copy stream associated with the client socket, as shown in figure 5.3. Provided that the client receive window or congestion window is not full, the packet will be forwarded on-the-fly. The packet will also remain in the inbound zero-copy stream until it is read and freed by the application thanks to the normal `mtcp_zc_read_fragments()` and `mtcp_zc_free_fragments()` functions.

The same technique as before cannot be used for flow control. Whereas with the *standard API*, the `mtcp_zc_forward()` call could be made blocking since it originated from the application thread, here it is not possible to block before putting a packet in the outbound zero-copy stream, since this occurs in the network thread just after one packet is received. Not putting the packet in the outbound list if a threshold is reached is also not an option since it would mean that the client would never receive it.

The simplest method to ensure flow control is therefore to outsource it to the two endpoints

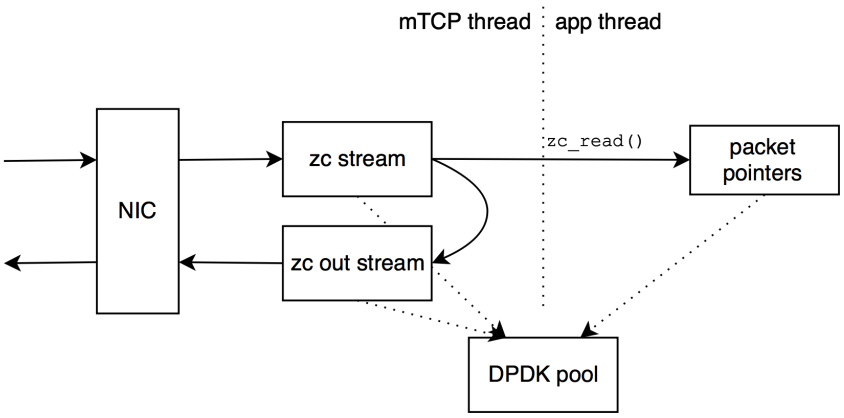


Figure 5.3: Overview of the *splice* forward API

of the connection. To that purpose, the middlebox can advertise to the server the receive window that the client has advertised to the middlebox. For the rest of the discussion, let w_s be the receive window advertised to the server and w_c the received window advertised by the client. This method consists in simply setting $w_s = w_c$.

However, this might not always achieve proper flow control, as we could observe within one of our testbeds. This is because of the delay introduced by the middlebox: when the server receives the w_s advertisement, it will be further in the stream than the client when it sent w_c, because it will have continued to send data to the middlebox in the meantime. The equation that we have is more precisely $w_s(t) = w_c(t - \Delta t)$, where Δt is the round-trip time between the client and the server. In the end, data could still be sent slightly faster by the server than received by the client. Since every packet is put in the outbound zero-copy stream by the middlebox, its size would diverge to $+\infty$ thus causing memory exhaustion, which is not acceptable.

A simple solution to account for the delay introduced by the middlebox is to introduce a coefficient $\alpha \in (0, 1)$ and to advertise $w_s = \alpha w_c$. Experiments in our setup show that the flow can be properly regulated given that α is below a certain threshold. However, choosing a too small α has a negative impact on throughput, whereas choosing α too high can still lead to memory exhaustion.

The problem with this approach is that it shifts the difficulty towards setting the right value of α. Using a predefined constant value does not seem to be an acceptable solution, since changes in the network conditions can invalidate this choice and lead to memory exhaustion. For example, it suffices that a second flow is processed in parallel for α to become non-optimal. For these reasons we chose not to opt for this method.

Therefore, the natural solution that arises is to dynamically adjust α, so that it automatically adapts itself to network conditions. This way, throughput can hopefully be optimal, while avoiding bufferisation due to the server sending data too fast. To do so, we use a rather simple feedback method. Let b be the size of the outbound zero-copy stream (*i.e.* the size of buffered unacknowledged and unsent data), and b_0 a configurable threshold (in the order of 1 megabyte).

Let also $f(x) = \dfrac{1}{1+9x}, \forall x > 0$. We then set $\alpha = f\left(\dfrac{b}{b_0}\right)$, *i.e.* we advertise:

$$w_s = \frac{1}{1+9\frac{b}{b_0}} w_c$$

This way, $\alpha = 1$ when $b = 0$, $\alpha = 0.1$ when $b = b_0$, and $\lim\limits_{b \to +\infty} \alpha(b) = 0$. This allows to greatly reduce the inbound throughput when the buffer exceeds the threshold b_0, hopefully reducing its size and converging to a steady state where the sending rate of the server is optimal.

A very rough mathematical model can justify this approach. Let $b(t)$ be the size of the middlebox send buffer. We have seen that the client receives data at a rate δ while the server sends at a rate $\beta > \delta$. Introducing the α factor, the sending rate becomes $\alpha\beta$, and we ideally want $\alpha\beta = \delta$. The equation governing $b(t)$ is:

$$b(t + \Delta t) = b(t) + \alpha(t)\beta - \delta = b(t) + \frac{\beta}{1+9\frac{b(t)}{b_0}} - \delta \tag{5.1}$$

where Δt is the round-trip time between the client and the server. Since we can consider that $\Delta t \to 0$, we can linearise the equation, which yields[1]:

$$\frac{\mathrm{d}}{\mathrm{d}t} b(t) = \alpha(t)\beta - \delta = \frac{\beta}{1+9\frac{b(t)}{b_0}} - \delta \tag{5.2}$$

This is a first-order autonomous differential equation $b'(t) = F(b(t))$, where the map $F : [0, +\infty) \to \mathbb{R}$ cancels out exactly one time and is such that $F'(b) < 0, \forall b \geq 0$. Therefore, the system will converge to a stable equilibrium, which justifies the intuition that this method allows to find the optimal α factor.

In a real environment, the value of α computed thanks to this method oscillates around an equilibrium, and can adapt itself in response to an exogenous change of network conditions (*i.e.* a change in β or δ). Figure 5.4 shows an excerpt of the evolution of α in a sample run of the middlebox scenario in our test setup. In this plot one can see that α is indeed adjusted when the environment changes, as illustrated by the two down peaks.

[1] up to a proportional change in β and δ that has no effect on their semantic

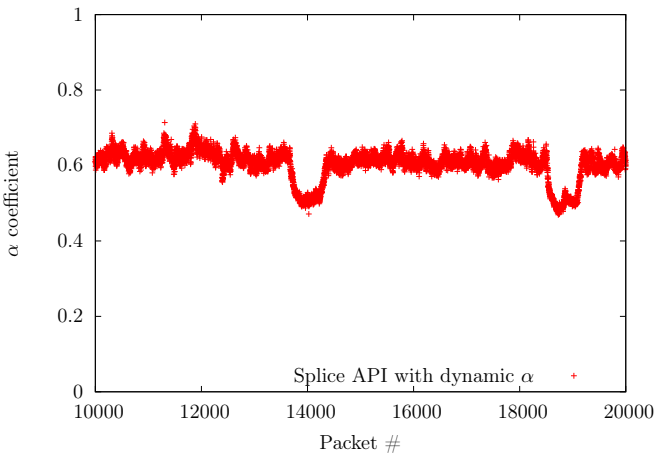


Figure 5.4: Sample evolution of α using the adaptive algorithm in the *splice API*

5.1.3 Evaluation

In this section, we evaluate the performance of the previously introduced forwarding API. We used the following testbed (the machines have the same hardware as in the previous tests):

1. a PC running a Linux TCP stack as the server,
2. a DPDK-enabled PC running our modified mTCP stack as the proxy
3. a PC running a Linux TCP stack as the client.

Comparison: To start with, we compared the performance of the *standard API* versus the *splice API* on a 10 GB data transfer: results are shown in table 5.1.

	Throughput (Gbps)	Relative performance
Direct connection	10.0	100%
Middlebox scenario (*standard API*)	7.1	71%
Middlebox scenario (*splice API*)	10.0	100%

Table 5.1: Comparison of the *standard* and the *splice* forward APIs

Results show that a direct connection between the client and the server running Linux TCP stacks can saturate the link. When adding our middlebox with the *standard API*, we observe a performance drop of 29%. As explained in previous section, despite its flexibility, this API does not offer optimal performance. However, the *splice API* that we introduced to overcome this issue shows satisfying properties. Indeed, when using it in the middlebox, the NIC remains saturated: our TCP proxy does not add any significant overhead to the end-to-end connection. For these reasons, we will thereafter focus on the *splice API*.

The splice API: As shown in table 5.1, this API can saturate the NIC while forwarding data for a single TCP connection between two endpoints, showing that its design is sound. In order to verify that this approach scales well to multiple connections, we ran the same experiment with 10 and 100 concurrent connections. In this experiment, the link was still saturated (table

5.2), showing that the stack can indeed adapt itself to frequent network environment changes that can occur with multiple parallel streams.

	Throughput (Gbps)
Single connection	10.0
10 concurrent connections	10.0
100 concurrent connections	10.0

Table 5.2: Scalability evaluation of the *splice API*

In order to validate the flow regulation algorithm with slow clients and to check whether it achieves its promises (*i.e.* providing optimal throughput without introducing bufferisation), we ran another set of experiments. In these, we used another machine as a client, whose NIC had faulty drivers and could only process inbound data at around 6.6 Gbps. This allowed us to validate that our approach was functional in non-ideal network conditions. Results are shown in figure 5.5, for different sizes of the `read()` buffer of the `netcat` client. In the best case, the data transfer runs at 97% of the throughput of the direct connection: the overhead introduced by the middlebox is negligible.

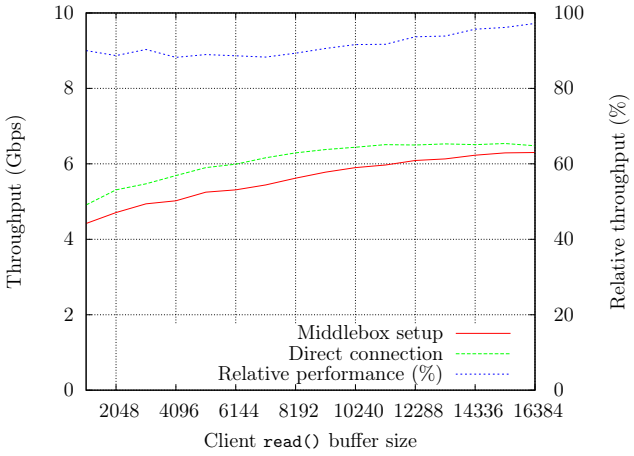


Figure 5.5: Evaluation of the *splice API* with a slow client

To conclude, the *splice API* overcame the challenges that it was supposed to solve: it proved to be efficient and scalable in multiple scenarios. When the end-to-end connection runs at line-rate, adding our TCP proxy still saturates the link, and with slow clients it only adds an overhead of a few percent. Therefore, we can use it as a base for application-aware traffic inspection, as will be discussed in the next section.

5.2 In-the-middle application-aware deserialisation

5.2.1 Design

In this section, we now consider the scenario of a TCP proxy that needs not only to forward data between a server and a client, but also to inspect the traffic flowing through both ends in a more structured way than simply examining packets one by one. For instance, such a middlebox could be interested in inspecting HTTP headers to gather statistics on user-agents, or SMTP headers to collect data on most frequently used destination domains in an email infrastructure. To do so, it suffices to use the *splice API* introduced above, but to register an application-aware module and use appropriate functions, as explained in chapter 4. Combining the two methods enables the application to perform in-the-middle traffic inspection.

More precisely, the user of the application should proceed the following way:

- define a data format for deserialisation using the preprocessor-based generator,
- allocate a socket and listen to new connections,
- upon receipt of a new connection, create a socket and connect it to the background server, splice it with the client socket and register the application-aware module on it,
- wait for read events, and use the zero-copy API to read records from the socket.

Therefore, the following will occur:

1. Upon receipt of an in-order packet or dequeuing of an previously out-of-order packet:
   (a) the packet is put in the zero-copy stream,
   (b) it is then parsed, generating records that are put in the application-aware stream,
   (c) finally, it is put in the outbound stream, after incrementing its reference counter, and a read event is generated.
2. When mTCP enters its forwarding loop and actually sends the packet to the wire, the corresponding node of the outbound stream is marked as sent. When it is acknowledged by the client, it is removed from the outbound stream. However, this might not release the DPDK `mbuf` associated with it if the application has not yet read corresponding records.
3. When the application requests application-aware records after receiving a read event, such records are passed form the application-aware stream to the application. When it requests to free them, they are removed from the stream, and nodes representing packets owning these records are removed from the inbound zero-copy stream. This might or might not release corresponding `mbuf`s, depending on whether packets have been acknowledged by the client and removed from the outbound stream.

The main advantage of this approach is its simplicity. From the point of view of the user of the stack, it suffices to write an application-aware parser as introduced in chapter 4, and integrate it in an application that listens for new connections and contacts a background server when a new one is incoming. Using the *splice API* relieves the user of the responsibility of having to manage flow forwarding: they only have to call the `mtcp_zc_read_aa_records()` API to inspect the traffic going between the client and the server, and gather structured data.

5.2.2 Evaluation

In order to evaluate this approach, we ran an experiment in a fashion similar to section 4.3.3, where we evaluated the HTTP deserialiser in a unidirectional client-server scenario. In this new experiment, we reused our custom server which continuously sends HTTP responses – seven headers followed by an arbitrary answer – to simulate an arbitrary long HTTP 1.1 session. On the client side, we used a variant of `netcat` as a simple sink. Finally, we added our DPDK-enabled proxy between the server and the client, configured to perform in-the-middle deserialisation of the HTTP session, so that the traffic can be transparently inspected.

We tested different parsing complexities by varying the size of the HTTP answer sent after the headers, so that the parser spends more or less time idle. Using a direct connection, the link is saturated regardless of the size of the answer, since the client does nothing and the data stream is pre-generated on the server. When plugging the middlebox in, the throughput varies as a function of the size of the answer, as reported in figure 5.6. It can reach 9.3 Gbps (representing an overhead of 7%) provided that the answer size is large enough so that the parser is not too busy.

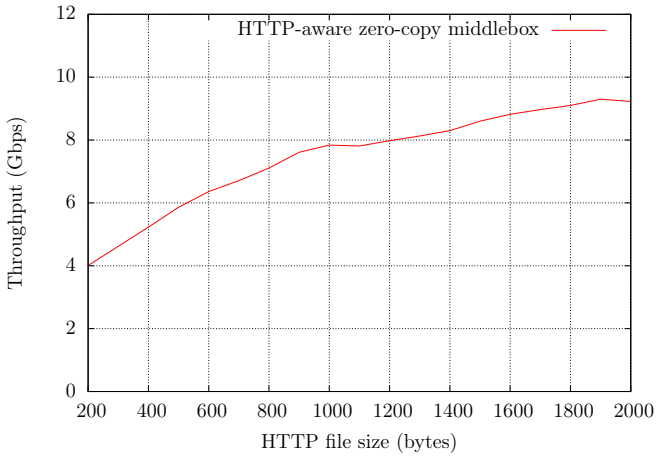


Figure 5.6: Evaluation of zero-copy HTTP parsing in a middlebox scenario

To summarise, this approach allows a transparent inspection of traffic going between a server and a client. The limit under which the parser starts inducing an overhead is approximately when 1 query is performed every 2000 bytes. At a line rate of 10 Gbps, this represents approximately half a million queries per second.

In a real environment, it seems doubtful that an HTTP server can sustain such a load. Therefore, the HTTP parsing performed inside our middlebox is unlikely to add any significant overhead compared to simple proxying. In order to verify this, we conducted another experiment involving a more realistic environment. This time, we used a real HTTP server (*nginx*, known to be lightweight and to scale well) on the server machine and `wget` on the client machine instead of a simple sink. This way, we can evaluate the performance of the deserialiser in real transactions, where the flow alternates between the client and the server, and where they perform real processing while issuing a query or an answer.

We configured `wget` to keep the same HTTP connection alive and download a file ten thou-

sands times in a row, so as to avoid the overhead of opening new connections and spawning new processes. We chose a file size of 2 KB since we have seen that it is the saturation limit of our parser. We varied the number of parallel connections to compare different loads of the server. Results are reported in figure 5.7, both for the HTTP-aware proxy and the simple proxy as a reference.

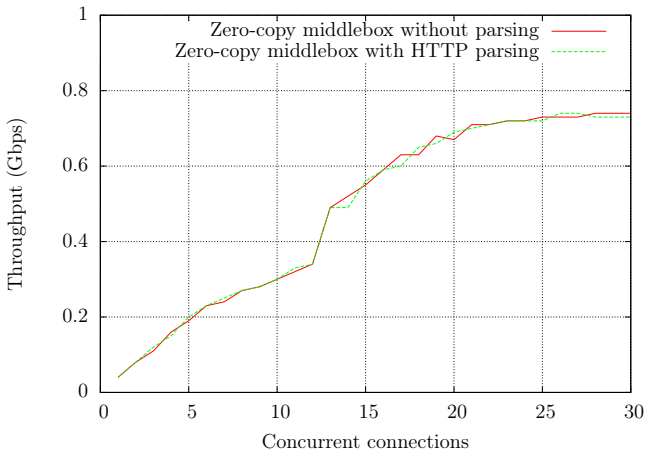


Figure 5.7: Impact of HTTP-aware proxying vs simple proxying in a real environment

From this plot, we can infer that *nginx* cannot process a stream of 2 KB requests at line rate, thus our parser does not incur any overhead. Should we have chosen a smaller file size, *nginx* would process it at an even lower rate, and higher file sizes correspond to the case where our parser can process headers at line rate. This experiment shows that, in real environments such as web browsing, adding the HTTP parser module to our zero-copy proxy has no noticeable consequences on performance.

6. Conclusion

6.1 Summary

In this document, we have presented a new approach to cope with the issue of handling TCP in high-performance environments. We aimed at dealing with two different issues inherent to the standard way TCP is implemented. First, a standard TCP stack running in the kernel cannot offer optimal performance, because of the overhead induced by context switching between kernel- and usermode, as well as the need to copy data between different abstraction layers. Second, the usual Berkeley socket API is presenting data between the transport layer and the application layer as a byte stream, whereas the application layer might be interested in a more structured manner to represent data. This has been addressed by using two main techniques: (i) we based our code on mTCP, a userspace TCP stack, to which we added zero-copy abilities and (ii) we implemented in-stack parsers that enable the application to access structured records instead of a byte stream. We will now briefly summarise how this has been achieved.

In chapter 3, we introduced a *zero-copy stream* as a way to internally represent a TCP stream. This data structure comprises two parts, an in-order list, and an out-of-order list that handles TCP Fast Retransmission. Those lists are maintained ordered with respect to the underlying TCP sequence numbers, and their nodes contain pointers to the data in packet buffers. We replaced the standard `read()` function with a one that provides the application with such pointers, effectively allowing it to access data without moving it across linear buffers.

Such an approach is not very useful in itself, as the application receives data in MTU-sized chunks, but can serve as a base to cope with the problem of structured data identified above. Thus, in chapter 4 we added an *application-aware stream* on top of the *zero-copy stream*. The base unit of this stream is not a packet any more, but data records containing a certain number of fields, each of them finally composed of delimiters pointing in packet buffers even if the data is split across several of them. Such records are parsed upon receipt of each packet, and a *read* API allows the application to directly fetch them. This design prevents the application from not knowing in advance if a *read* call will return a sufficient amount of data to be processed in a single pass, minimising the switches between the network and the user contexts. Lastly, we created a macro-based system for the user to specify the data format that they want to use.

Finally, we added zero-copy forwarding capabilities to the stack, so that it can transfer packets between two TCP flows by only modifying the headers, as described in chapter 5. We

experimented two different methods, the *standard API*, in which forwarding is to be initiated by the user application, and the *splice API*, in which it is automatically done upon receipt of a packet. The former gives more control to the application, while the latter showed to be more efficient. Combining this with our deserialisation mechanism allows the application to transparently inspect the traffic going between two hosts, and access such information in a structured fashion. Use cases for such a scenario are numerous, and evaluation shows that in a real environment, no noticeable overhead is added.

6.2 Future work

Although this work gives a good insight on what an application-aware zero-copy TCP stack can be, some features could be improved.

The socket model

In this project, we kept the socket model introduced by mTCP, thanks to which the application thread communicates with the network thread. The ambition of the authors was mainly to provide an API that is the closest possible to the BSD one. However, for the kind of use cases that we investigated such as traffic inspection, it would be useful to get rid of this model and make the application a supervisor rather than a consumer. A possibility would be to pre-configure mTCP so that the it knows in advance what to do with each flow and what actions to perform when data is available. This would however require to rethink the whole mTCP design.

Modifying the content

A challenge that has been not addressed in this project is to enable modification of the payload of a stream. With the approach that we proposed, the only way to perform data modification is to replace the contents of a field by something of equal length. Of course, inserting data at an arbitrary point in the packet is still possible but requires moving the rest of the payload.

The issue with this is that it breaks the zero-copy design used across the project: if the packets are to be modified, it becomes useless to move them untouched from the ingress to the egress stream. There are some cases, nonetheless, where this could make sense: for instance, when data is to be prepended or appended to a packet. If one actually wants to perform rich content modification while keeping the zero-copy design, it would potentially be possible to split the packet into several pieces, modify the required one, and use scatter-gather I/O to reassemble the packet. It is however likely that the overhead of doing so would overcome the benefits of avoiding data copy.

Introducing a simpler format description

Although the system introduced to specify data formats in chapter 4 thanks to C macros is idiomatic and well-suited for the usages that we have explored, it can become cumbersome to use for long grammars. This approach could scale better by introducing a custom language to specify grammars and providing a program generating the C code of the parser out of such a description.

Appendix

The following is a usage example of the macro-based parser generator defined in section 4.3.2. This snippet defines a module that deserialises the server-to-client side of simple HTTP sessions. The other side of the session can be deserialised in a similar fashion.

```
#define POST_PROCESS_ANSWER() \
    if (GET_RECORD_VARIABLE(answer.content_length) == 0 &&
        STRCMP("Content-Length:␣") == 0) { \
        SET_RECORD_VARIABLE(answer.content_length,
            ATOI_WITH_OFFSET(strlen("Content-Length:␣"))); \
    } \
    if (STRCMP("\r\n") == 0) { \
        SET_VARIABLE(remaining_content_length,
            GET_RECORD_VARIABLE(answer.content_length)); \
        SET_STATE(HTTP_BODY); \
        END_OF_RECORD(); \
    }

#define POST_PROCESS_BODY() \
    GET_VARIABLE(remaining_content_length) -= FIELD_LEN(); \
    if (GET_VARIABLE(remaining_content_length) <= 0) { \
        SET_VARIABLE(remaining_content_length, 0); \
        SET_STATE(HTTP_ANSWER); \
    } \
    END_OF_RECORD();

/* BEGIN PROTOCOL DECLARATION */
#define PROTOCOL http
#define NUM_FIELDS 13

#define REGISTER_STATES() \
        STATE(HTTP_ANSWER, answer) \
        STATE(HTTP_BODY, body)

#define REGISTER_VARIABLES() \
        VARIABLE(remaining_content_length, int)

#define REGISTER_FIELDS_HTTP_ANSWER \
        FIELD(HTTP_ANSWER_VERSION, version, STRSTR("␣"), VOID())\
        FIELD(HTTP_ANSWER_STATUS_CODE, status_code, STRSTR("\r\n"), VOID())\
        FIELD(HTTP_ANSWER_HEADER_0, header0, STRSTR("\r\n"),
```

```
            POST_PROCESS_ANSWER())\
        //...
        FIELD(HTTP_ANSWER_HEADER_9, header9, STRSTR("\r\n"),
            POST_PROCESS_ANSWER())

#define REGISTER_VARIABLES_HTTP_ANSWER \
        VARIABLE(content_length, int)

#define REGISTER_FIELDS_HTTP_BODY \
        FIELD(HTTP_BODY_DATA, data,
            OFFSET(GET_VARIABLE(remaining_content_length)), POST_PROCESS_BODY())

#define REGISTER_VARIABLES_HTTP_BODY

#include "generate_aa_api.h"
```

Writing the code above will generate the record structure listed below, which will be retrieved by the application upon request. It will also generate the corresponding *application-aware stream* as well as callbacks to integrate it into mTCP.

```
typedef struct http_record {
    struct http_record* next;
    uint8_t type;
    union {
        struct {
            int content_length;
        } answer;
        struct {} body;
    } variables;

    union {
        struct {
            struct stream_field* version;
            struct stream_field* status_code;
            struct stream_field* header0;
            struct stream_field* header1;
            //...
            struct stream_field* header9;
        } answer;
        struct {
            struct stream_field* data;
        } body;
        struct stream_field* raw_fields[13];
    } fields;

} * http_record_t;
```

List of Figures

Bibliography

[1] Jon Postel. Transmission Control Protocol. In *Requests For Comments*, number 793. IETF, 1981.

[2] Jon Postel. Internet Protocol. In *Requests For Comments*, number 791. IETF, 1981.

[3] Pavan Balaji, Hemal V Shah, and Dhabaleswar K Panda. Sockets vs RDMA interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck. In *RAIT workshop*, volume 4, page 2004, 2004.

[4] David D Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *Communications Magazine, IEEE*, 27(6):23–29, 1989.

[5] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[6] E Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and K Park. mTCP: a highly scalable user-level TCP stack for multicore systems. *Proc. 11th USENIX NSDI*, 2014.

[7] Intel. Intel Data Plane Development Kit Overview. http://www.intel.co.uk/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf, 2012.

[8] Dragan Stancevic. Zero copy I: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

[9] David A Maltz and Pravin Bhagwat. TCP Splice for application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, 1999.

[10] Philippe Joubert, Robert B King, Richard Neves, Mark Russinovich, John M Tracey, et al. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *USENIX Annual Technical Conference, General Track*, pages 175–187, 2001.

[11] Chuck Lever, Marius Aamodt Eriksen, and Stephen P Molloy. An analysis of the TUX web server. Technical report, Center for Information Technology Integration, 2000.

[12] Moshe Bar. Kernel Korner: kHTTPd, a kernel-based web server. *Linux Journal*, 2000(76es):21, 2000.

[13] Luigi Rizzo, Luca Deri, and Alfredo Cardigliano. 10 Gbit/s Line Rate Packet Processing Using Commodity Hardware: Survey and new Proposals. 10.

[14] Ian Pratt and Keir Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 67–76. IEEE, 2001.

[15] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *OSDI*, pages 135–148, 2012.

[16] Max Krasnyansky. UIO-IXGBE. https://opensource.qualcomm.com/wiki/UIO-IXGBE, 2013.

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

[18] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[19] Mehdi Bezahaf, Abdul Alim, and Laurent Mathy. FlowOS: a flow-based platform for middleboxes. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, pages 19–24. ACM, 2013.

[20] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 175–186. ACM, 2014.

[21] Audie Sumaray and S Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, page 48. ACM, 2012.

[22] Jaakko Kangasharju, Sasu Tarkoma, and Tancred Lindholm. Xebu: A binary format with schema-based optimizations for XML data. In *Web Information Systems Engineering–WISE 2005*, pages 528–535. Springer, 2005.

[23] Kenton Varda. Protocol Buffers: Google's data interchange format. http://google-opensource.blogspot.com/2008/07/protocol-huffers-googles-data.html, 2008.

[24] Mark Allman, Vern Paxson, and William Stevens. TCP Congestion Control. In *Requests For Comments*, number 2581. IETF, 1999.