Université de Liège

Faculty of Engineering

Very fast web traffic generator on a Tilera device

Raphael Javaux

Master thesis submitted for the degree of MSc in Computer Science

Academic year 2014 - 2015

Very fast web traffic generator on a Tilera device

Abstract

This document summarizes the design and development of a lightweight, user-space, event-driven and highly-scalable TCP/IP network stack.

A simple web-server using this new framework got a 2.6× performance improvement, when compared to the same application layer running on the new reusable *TCP* sockets introduced in *Linux 3.9*. The web-server was able to deliver static HTML pages at a rate of 12 Gbps on a Tilera TILE-Gx36 36-core processor.

This number could be significantly improved by carrying out some optimizations that have not been implemented yet (because of a limitation of time resources) or by porting the network stack to a faster *CPU* architecture.

The network stack was developed with the ultimate goal of creating a very fast HTTP traffic simulator to certify layer-4 middleboxes for very high network load.

Raphael Javaux

Master thesis submitted for the degree of MSc in Computer Science

Academic year 2014 - 2015

Content

Introduction

This document summarizes the software I developed as my final year project.

I was asked to design a very fast *HTTP* traffic generator. The purpose of this traffic generator was to simulate a very large amount of *HTTP* sessions to qualify layer-4 middleboxes for deployment on production networks. The goal was not to detect any error caused by the middleboxes, but to generate *HTTP* traffic at a sufficient rate to validate the ability of the middleboxes to endure a such throughput.

Desire was expressed that *HTTP* traffic should be generated at a rate of up to 40 Gbps, using four 10 Gbps *Ethernet* links.

It was required that the software should run on a *Tilera TILE-Gx36* device, an extension card orchestrated by a proprietary 36-core *TILE-Gx* microprocessor. The device is designed to run high performance network applications and has four 10 Gbps network interfaces

The project resulted in the design and development of *Rusty*, an user-space, light-weight, event-driven and highly-scalable *TCP/IP* network stack.

Although network applications powered by *Rusty* are not able reach the 40 Gbps goal, they benefit from an huge increase in performances when compared to the *Linux* network stack running on the same hardware.

The high-performance web-server developed to generate *HTTP* traffic gets a 2.6× improvement in throughput when running on Rusty instead of the new high-performance reusable *TCP* sockets introduced in *Linux 3.9*.

The web-server written for the traffic generator was able to deliver static *HTML* pages at a rate of 12 Gbps on the *Tilera TILE-Gx36* device. This number could be significantly improved by carrying out some additional optimizations that have not been implemented yet, because of a shortage of time.

The network stack directly calls the proprietary network driver of the *Tilera* device, by-passing the operating system. It has been designed to be easily adapted to other user-space network drivers, available with some professional NICs. As performances are currently bounded by the relatively low processing power of the *TILE-Gx* microprocessor, an huge increase in performances could also be expected by porting *Rusty* to a faster *CPU* architecture.

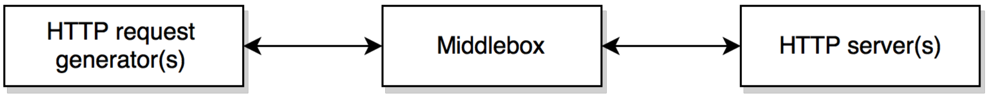
Source code of the new *TCP/IP* stack and of the web-server is available at `github.com/RaphaelJ/rusty/`.

A very fast HTTP traffic generator

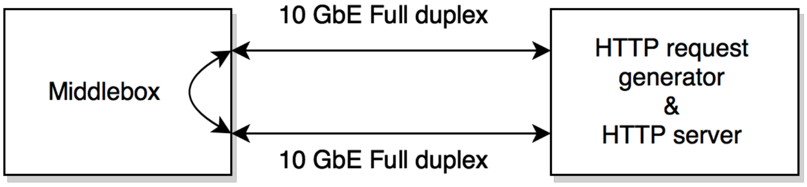
The goal of this final year project was to develop a test-bed to evaluate the capacity of network middleboxes currently developed at the *University of Liège* to sustain high loads of *HTTP* traffic.

The middleboxes are to be located between one (or several) *HTTP* request generator(s) and one (or several) *HTTP* server(s). *HTTP* request gerator(s) and *HTTP* server(s) should generate as much *HTTP* sessions as possible, and should be able to tell at which rate traffic was delivered.

The ultimate goal of the project was to make up to 40 Gbps of *HTTP* traffic going through the middlebox.



Middleboxes are expected to have four 10 Gbps *Ethernet* links. To benefit from the full-duplex capacity of the links, it was strongly suggested that *HTTP* requests and responses should transit in both directions of a single link. Suppose that you have two full-duplex links between the middlebox and a single testing device (such as in the following diagram), the request generator running on the first link could send requests to the web-server running on the second link, and vice-versa. This doubles the traffic passing through the middlebox per link.

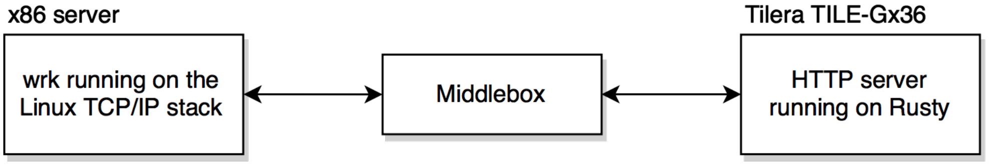


The university network laboratory has a *Tilera TILE-Gx36* device with four 10 Gbps *Ethernet* links. It was requested that the entire test-bed should run on this device. The device uses a proprietary 36-core *CPU*, runs the *Linux* operating system, and has network co-processors for packet classification.

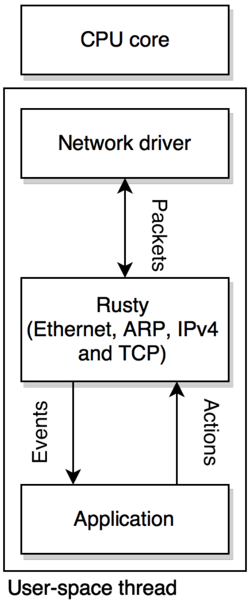
It appeared at a very early stage in the project that the *Linux* network stack would never be able to sustain the requested amount of traffic on the requested device. To address this issue, a new high-performance *TCP/IP* stack was developed from scratch. This new stack, named *Rusty*, significantly improved the performances, but I was not able to reach the targeted 40 Gbps throughput.

Rusty is not yet able to initiate connections, because of an unresolved issue with the *Tilera*'s driver when using multiple cores. The issue is described in the *Implementation details* chapter. Because of that, the *HTTP* request generator runs on another device, an *x86* server with several 10 Gbps *Ethernet* NICs. This server is fast enough to sufficiently load the web-server running on the *TILE-Gx36*.

The web-server running on the *TILE-Gx36* has been written especially for this project. The request generator running on the *x86* server is *wrk*, a scriptable *HTTP* benchmarking tool [Gloz15].


x86 server
wrk running on the Linux TCP/IP stack
Middlebox
Tilera TILE-Gx36
HTTP server running on Rusty

Rusty



Rusty is the name of the *TCP/IP* stack that was developed to increase the throughput of the traffic simulator. The software is written using the *C++* programming language.

It follows the original *TCP* specification [RFC793] with the congestion control algorithms described in [RFC5681] (slow start, fast retransmit and and fast recovery).

It has an architecture designed for efficiency and high-scalability:

It is an user-space network stack. Apart from its initialisation and some memory allocations, it does not rely on the operating system. It uses the network driver while in user-space and does not produce any system-call (except for some rare cases such as page fault handling). Once initialized, it binds itself to some *CPU* cores and disables preemptive multi-tasking for these cores (so the operating system does not issue context-switches). This is detailed in the *Implementation details* chapter.

- It is an highly-parallelizable network stack. It spawns as much instances of itself as the number of *CPU* cores it can use. Each instance handles only a subset of the *TCP* connections — a given connection is always handled by the same core —, and does not share anything with other stacks. There is no mutual exclusion mechanism and the software scales almost linearly with respect to the number of processing cores (*Rusty* is 31 times faster on 35 cores than on a single core !). The architecture is presented in the *Implementation details* chapter while the observed scalability is discussed in the *Performance analysis* chapter.

Rusty also distances itself from the original *TCP* specification by ignoring *Urgent* and *Push* flags. They do not make much sense in an event-driven *TCP/IP* stack, as segment payloads are always delivered to the application layer at the instant they are received.

- It is an event-driven network stack. The application layer reacts to events (such as a new connection, a new arrival of data ...) by providing function pointers to event handlers. As the *Rusty* instance that manages the connection and the application event handlers run on the same core and thread, in user-space, the overhead of passing data from the network stack to the application layer is only that of a function pointer call. It is radically different from traditional stacks where blocking calls such as `recv()` or `accept()` produce system calls, and often context-switches.
- It has a zero-copy interface. The application layer directly accesses the memory buffers used by the network interface. `send()` and `recv()` calls, as implemented in most network stacks, require the data to be copied from and to application buffers. The machinery behind this zero-copy interface is detailed in the *Implementation details* chapter.

The next chapter gives an example of an event-driven network application written with *Rusty*.

There are other user-space and highly-parallelizable network stacks, of which two are briefly introduced in the *Similar network stacks* chapter.

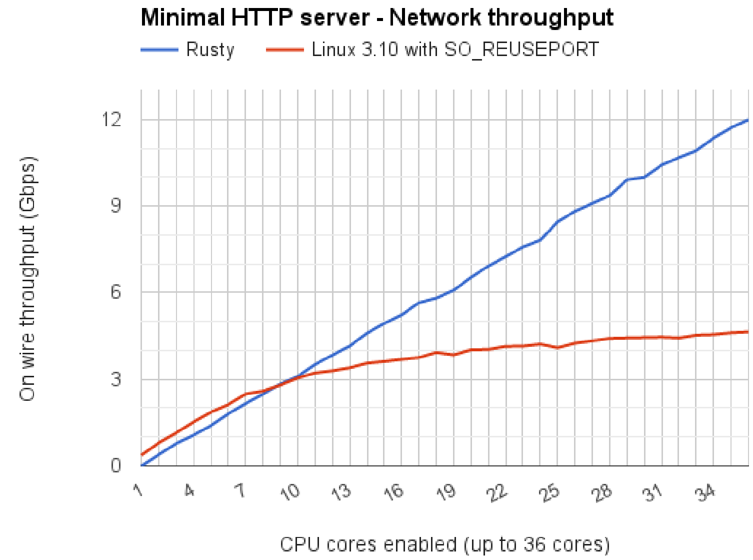
To improve performances, *Rusty* also provides a way to pre-compute *Internet* checksums of transmitted data. This gives a noticeable speedup for applications, such as the web-server used to generate *HTTP* traffic, for which the pages that can be transmitted are determined in advance. Pre-computing checksums is not as easy as it may look, because you do not know in advance how the data will be chunked in *TCP* segments. The technique used is described in the *Implementation details* chapter.

Performance overview

This section only summarizes the performances and the scalability of *Rusty*. A detailed analysis and the methodology applied are available in the *Performance analysis* chapter.

Rusty shows an exceptional ability to scale up with additional processing cores. This exhibits the efficiency of the multi-threading model that has been put in place.

The following graph compares the maximum throughput of the same *HTTP* server, on the *Tilera TILE-Gx36*, for a given number of *CPU* cores, when using *Rusty* and when using the *Linux* network stack. The *Linux* version uses *epoll* with *reusable sockets*, which is the advised method to write efficient network applications by the *Linux* kernel developers [Kerr13].



With all the 36 cores enabled, the *Rusty* implementation puts 12 Gbps on the wire (about 11,000 *HTTP* requests second). This is a 2.6× improvement against the *Linux* version which is only able to deliver 4.6 Gbps (about 4,000 HTTP requests second).

While these performances are engaging, the *TCP/IP* stack has not been seriously optimised yet (because of a shortage of time). Dynamic memory allocations (currently required to handle timers, transmission queues and functionnal closures) consume a large fraction of the *CPU* cycles required to handle a segment, and could be widely optimized.

Performances are also relatively low because of the relative low performances of the *TILE-Gx36* processor. This chip is several times slower than a modern high-end *x86* microprocessor (some benchmarks comparing three *x86* computers and the *TILE-Gx36* are also given in the

Performance analysis chapter).

Chapter 1

Event-driven network applications

Rusty interacts with the application layer through an event-driven interface. Basically, the application layer provides functions to handle events such as a new incoming connection (on a listening *TCP* port), a new arrival of data on an established connection, or the closing of a connection. It then reacts to these events, by doing actions such as requesting to send data or to close a connection. This is not a brand new way of developing network applications: the network API of node.js has a similar event-driven interface [Node15] (but is not an user-space network stack, as it relies on the sockets provided by the operating system).

In this chapter, you will be presented how to write a simple multi-threaded echo server. The application accepts new connections on a listening socket and echoes back the data received on these connections.

Closures

In this chapter, we will use a new *C++* feature introduced in the 2011 revision of the language: *closures*. Closures are functions that can be defined inside another function (as an expression) and that can capture the environment of the parent function (i.e. the values of the variables). Closures are also known as lambda expressions, and exist in almost any modern language (well, actually, they somehow existed *before* programming languages).

As an example, suppose that you have a function that numericaly computes the integral of another function `f`, between `a` and `b`:

$$\int_a^b f(x)\ dx$$

This integration function has been written so it can be used with any `f` function. In other words, the function accepts a function as argument, and its signature is:

```
double integral(double a, double b, function<double(double)> f);
```

The `function<double(double)>` type of the third argument tells us that `f` is a function that accepts a `double` and returns a `double`. If we wish to compute the integral of `sqrt(x)` between `0` and `10`, all we would have to do is doing this call:

```
integral(0, 10, sqrt);
```

But now suppose that we want to write a function that compute this integral for any value of `a`, `b` and `n`:

$$\int_a^b x^n\ dx$$

We would not be able to do that with a simple function pointer as we did with `sqrt`. We need to find a way to transmit the value of the `n` parameter to `integral()`.

Closures provides a way to "create" a function that will capture this `n` parameter:

```
double integral_pow(double a, double b, double n)
{
    function<double(double)> f =
        [n](double x)
        {
            return pow(x, n);
        };

    return integral(0, 10, f);
}
```

We created a single argument function called `f` that captured the n parameter (`[n]` tells the compiler that `n` has been captured), then we used it in the call `integral()`.

Closures can be used to write event handlers for *Rusty*.

Initializing the network stack

Now that you know about closures, we can come back to our echo server.

The first step to implement our echo server is to initialize the network stack. This is done by initializing the lower layer of the network stack, namely the abstraction layer over the network driver.

Three parameters must be supplied to the constructor of the network stack data-type:

- The network interface and its associated *IPv4* address on which the stack will run. In this example, we will use the `xgbe1` 10 Gbps *Ethernet* interface. The *ARP* layer will announce the address, and outgoing *IPv4* datagrams will be delivered from this address.
- The number of cores that Rusty will use to run this network stack. Here we chose to use 35 cores of the *TILE-Gx36* device. We can not use more than 35 cores on this 36-core device, as *Rusty* takes total execution control of these cores, and as at least one core must be left free for the operating system to run.

This following call constructs a stack object using the `mpipe_t` network driver (*mPIPE* is the name of the driver provided by *Tilera* for its NICs). The call only allocates resources and initializes the network driver. It does not actually start the network stack.

```
rusty::driver::mpipe_t stack("xgbe1", "10.0.2.1", 35);
```

If we wished to use several network interfaces, we would have created several stack objects, one per interface.

Adding a listening port

We would like to echo data coming on incoming connections on port

23.

First, we need to open the port for new connections. To do that, we call the `tcp_listen()` method on the previously initialized stack object. In addition to the port on which the stack must listen, the method accepts a function that will be called each time an new connection is established. Let's call `tcp_listen()` to listen on port 23 with a function that we will define later:

```
stack.tcp_listen(23, new_connection_handler);
```

Handling connection events

It is time to define the `new_connection_handler` function we used in the `tcp_listen()` call. The function must accept a connection as argument (as a `cont_t` value) and must return a new `conn_handlers_t` value containing a set of handlers for this connection. Its type signature must be:

```
conn_handlers_t new_connection_handler(conn_t conn);
```

The `conn_handlers_t` data-type contains four functions, each one handling one type of event that can happen on an established connection:

```
struct conn_handlers_t {
    // Called when the connection receives new data.
    function<void(cursor_t)>                new_data;

    // Called when the remote asked to close the connection.
    function<void()>                        remote_close;

    // Called when both ends closed the connection. Resources
    // allocated for the connection should be released.
    function<void()>                        close;

    // The connection has been unexpectedly closed (connection
    // reset). Resources allocated for the connection should be
    // released.
    function<void()>                        reset;
};
```

The `new_data` handler is given a `cursor_t` value. Cursors are an abstraction over network buffers that enable zero-copy reading and writing. As of now, just assume they are similar to *Streams* in *Java* or *C++*. Cursors are detailed in the *Implementation details* chapter to follow.

So, now, we can write the `new_connection_handler` function. The function constructs a new set of handlers for the connection that has been received in argument.

```
conn_handlers_t new_connection_handler(conn_t conn)
{
    conn_handlers_t handlers;

    handlers.new_data =
        // ‘mutable’ tells the compiler that the closure
        // changes the state of the ‘conn’ variable.
        [conn](cursor_t in) mutable
        {
            // This closure sends the received data back to the
            // remote device.

            size_t size = in.size();

            // The 'send' method receives three values:
            // * The size of the data to send.
            // * A function to execute to write the data in the
            //   network buffer.
            //   ‘Writers’ are detailed in the
            //   'Implementation details' chapter.
            // * A function to execute when the data has been
            //   acknowledged.
            conn.send(
                size,
                [in](size_t offset, cursor_t out)
                {
                    out.write(in.drop(offset).take(out.size()));
                },
                do_nothing // Does nothing when data is
                           // acknowledged.
            );
        };

    handlers.remote_close =
        [conn]() mutable
        {
            // Closes the connection when the remote half-closed
            // the connection.
            conn.close();
        };

    handlers.close = do_nothing;
    handlers.reset = do_nothing;

    return handlers;
}
```

do_nothing is a function that, well, does nothing. It is used to ignore an event.

Handlers should not execute for too long, as they take the execution control from the network stack. If an handler takes too much time to execute, this could delay the processing of incoming packets, and causes a retransmission of data from the remote TCP. A a similar problem arises in event-driven GUIs when a time consuming event handler can freeze the whole graphical interface.

Rusty carries out an optimization permitted by the *TCP* protocol when one sends data while in the new_data handler, like we did. It will combine our response with the acknowledgement segment of the received data, saving one segment. That is because the network stack has not acknowledged the received data yet. Similarly, the *FIN* segment the stack sends when we call conn.close() will be combined with the acknowledgement of the received *FIN* segment that triggered the remote_close event (if the transmission queue is empty).

Starting the network stack

Now that we have initatialized *Rusty* with our application layer, we can start the stack on the worker threads by calling the run() method. As the run() method returns immediately after the threads have been launched, we also need to call the join() method that wait for the termination of these threads (which will never happen as the application layer never asks the stack to stop processing packets by calling the stop() method):

```
stack.run();
stack.join();
```

The complete echo application is available in the `app/` directory of the source code repository of *Rusty* (`github.com/RaphaelJ/rusty/`).

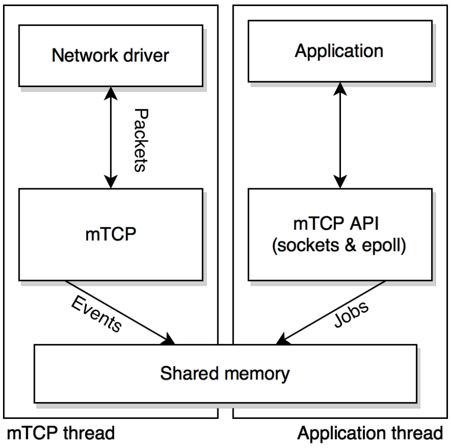
Chapter 2

Similar network stacks

Rusty is not the first user-space *TCP/IP* stack designed for high performances.

I will briefly introduce two existing stacks that I looked at before and while developing *Rusty*: *mTCP* [JWJ+14] and *Seastar* [Clou15]. Both use an user-space driver for some *Intel* 10 Gbps *Ethernet* NICs. *Seastar* has a similar architecture to *Rusty* while the architecture of *mTCP* is quite different.

mTCP



mTCP provides an implementation of *epoll* and *BSD sockets* that runs in user-space (and is thus almost backward compatible with simple applications running on these).

Unlike *Rusty* (and *Seastar*), the network stack does not run on the same threads as the application layer. Each application layer thread is coupled with an *mTCP* thread (that runs the network stack and the network driver in user-space). Both are affinitized to the same *CPU* core. They share a event queue and a job queue. Events (such as the reception of new data) are produced by the *mTCP* thread and consumed by the application layer thread, while jobs (such as new data to transmit) are produced by the application thread and executed by the *mTCP* thread.

Connections are local to a single *mTCP* thread (they are not shared). Because of that, they achieve a better scalability than the *Linux TCP/IP* stack. They chose to run the application layer on a separate thread so it does not affect the behaviour of *TCP* (a long computation in the application layer on *Rusty* can produce retransmissions from the remote as the processing of incoming packets would be delayed). The drawback is that they must carry the overhead of the operating system scheduler, and that they can not provide a zero-copy interface like *Rusty* does.

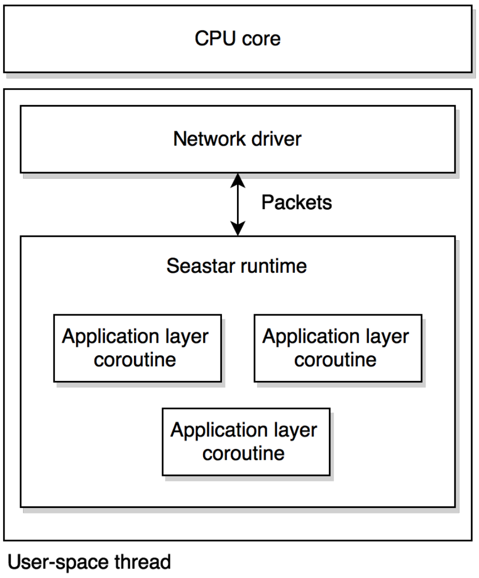
On a 8-core (16 hardware threads) *Intel Xeon*, they achieve a 3× improvement in performances when compared to the Linux stack with reusable sockets. They also point out that the *Linux* stack is not able to scale correctly after more than 4 cores on this hardware.

In the first days of my Master's thesis, I was advised to evaluate if *mTCP* could be modified to be used on the *TILE-Gx36*. But, after going through the source code, I thought that it would be way more painful to modify this poorly documented software than creating a new *TCP/IP* stack myself. Also, the software is strongly coupled to the network driver it uses, making the porting work even harder. *mTCP* was developed more as a proof-of-concept than as a reusable system.

Seastar

Seastar is an open-source framework developed by *Cloudius Systems*. It was released to the public for the first time during the second half of February 2015, and I became aware of it by the end of April (at a time when *Rusty* was already seriously being developed). The project is still actively developed, with 10 different contributors during the month of July 2015, according to the the public code repository.

The framework is slightly more advanced than *Rusty* and *mTCP*. Like


CPU core
Network driver
Packets
Seastar runtime
Application layer coroutine
Application layer coroutine
Application layer coroutine
User-space thread

Rusty, it runs a single thread per core. However, it allows the application layer to run multiple cooperative tasks, or *coroutines*: in addition to its *TCP/IP* stack, the Seastar runtime provides an user-space cooperative multitasking system. The application layer is executed in these coroutines, and must give execution control back to the runtime voluntarily, or by making calls to `send()`, `recv()` or other functions that would be blocking in traditional network stacks. Unlike operating system threads, coroutines are never preempted.

This cooperative model has a small overhead, as coroutines have their own call stacks. Context-switches are however significantly faster than those of an operating system (such as in *mTCP*), as it is only about substituting *CPU* registers. The scheduling algorithm is also simpler, and mostly relies on events from the network.

A such cooperative multitasking system could be implemented on top of *Rusty*. The event-driven model is a kind of low level interface to the network stack.

Unlike *mTCP*, *Seastar* is not backward compatible with software written using traditional *BSD* sockets. Applications must be written with the cooperative multitasking model in mind.

The *Seastar* framework could have been modified to run with the *Tilera*'s network driver, as the network stack is reasonably disassociated from the network driver. It would still be an huge amount of work (network buffers work very differently in the driver they use (*DPDK*), and the framework has become quite huge), but it would have been doable within the context of this Master's thesis. I did not do that because I was already too advanced in my own *TCP/IP* stack at the time I became aware of the *Seastar* project.

Chapter 3

Implementation details

This chapter gives technical details about the implementation of the *Rusty* network stack.

A short introduction of the *TILE-Gx* architecture is given in the first section. The *TILE-Gx36* is a many-core device, with an unique cache hierarchy (memory pages can be cached to a specified core). The first part of this chapter presents how *Rusty* was designed to take profit of this singular architecture.

This chapter then exposes the high-level abstractions the network stack uses to abstract the driver and to provide a zero-copy interface with the application layer.

The last sections give a way to efficiently compute and pre-compute *Internet* checksums.

TILE-Gx architecture

The framework have been developed for the TILE-Gx36 processor.

The *TILE-Gx* processor family has been developed by *Tilera*, until the company was acquired by *EzChip* during the summer 2014.

The family has been declined in 9, 16, 36 and 72-cores models. These use a proprietary 64-bits RISC instruction set and run at a 1.2 Ghz frequency.

TILE-Gx processors have two *DDR3* memory controllers, and a theoretical maximal main memory bandwidth of 25.6 Gbps.

These *CPU*s are designed to run network related software. As a result, they also have:

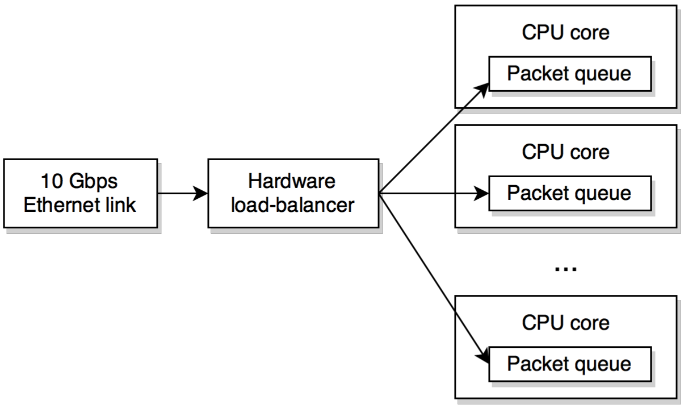
- Co-prosessors for packet analysis, network flow load-balancing, compression and encryption.
- Integrated 10 Gbps *Ethernet* controllers to maximize network throughput, especially on small packets.

The device used to implement this project is engineered around a 36-core TILE-Gx chip, in the form of a *PCI-Express* extension card. It is a computer on its own, having its dedicated *DDR3* memory and four 10 Gbps *Ethernet* links. It runs the *Linux 3.10* operating system.

Parallelisation and scalability

The framework has been designed to be highly parallelizable, benefiting from the hardware network load-balancer included in the *TILE-Gx36* processor.

The hardware load-balancer has the ability to distribute incoming network packets around multiple packet queues. Queue selection is based on a combined hash function of the *IPv4* addresses and *TCP* ports of the incoming packet. Packets belonging to a same TCP flow (source address, source port, destination address and destination port) are always delivered to the same queue. This hardware feature is called Receive Side Scaling (*RSS*), and is also available on other professional *NICs*.



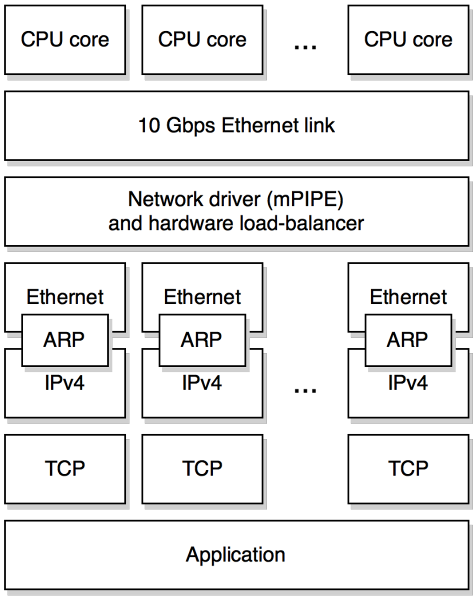
When the framework runs on several *CPU* cores, as much *TCP/IP* stacks and (incoming) packet queues are created. Each network stack instance is continuously polling its queue for a new incoming packet to process. A core is entirely dedicated to the execution of its associated network stack (the operating system can not initiate a context-switch as preemptive multi-tasking is disabled for these cores). The main loop executed by each core looks like this one:

Processes can still enter kernel space, because of some interruption, or a page fault.

```
while (app_is_running) {
    execute_expired_timers();

    if (incomming_queue.is_empty())
        continue;

    packet_t packet = incomming_queue.pop();
    network_stack.receive_packet(packet);
}
```

The timers set is continuously checked for expired timers. Timers are detailed later in this document.

The network driver, provided by the *Tilera development framework*, runs in user space and does not require any system-call when transmitting or receiving packets.



Each running core will be in charge of dealing with its own subset of *TCP* connections (as determined by the load-balancer), and will not share any connection with another core. Cores do not share any mutable state, no mutual exclusion mechanism is required. The *ARP* layer should share a common table between network stacks. As of today, when running on multiple cores, *ARP* layers only share a common static table, and any request to an unknown *IPv4* address fails. Dynamic *ARP* tables (i.e. on which new entries can be added when receiving *ARP* replies) are only implemented when running on a single core.

Event handlers composing the application layer that are related to a same *TCP* connection will always be executed on the same core. The application layer directly reads from and writes to memory buffers used by the network interface.

Advantages

- This architecture makes the framework implementation simpler, as there is no interactions between concurrently executing tasks. The easiest way to deal with multiple tasks concurrently accessing a shared resource, is to not having multiple tasks concurrently accessing a shared resource ...

- It makes the framework very scalable. As there is no mutual exclusion, adding more processing cores should increase performances until the maximal throughput of the memory or of the network hardware is reached.

- The overhead of context-switching, inter-process data sharing and system-calls is removed, as application layer event handlers are executed on the core that processes the connection. The framework is also more *CPU* cache friendly, as temporal locality is higher.

Drawbacks

- The throughput of a single *TCP* connection is upper bounded by the processing power of the core that is handling it. There is no per-connection parallelism. While the framework is able to sustain a several Gbps rate over a large number of connections, the maximum rate which can be reached on a single connection peaks at around 350 Mbps (*TileGx* cores are relatively slow). Similarly, if the load-balancer fails to evenly distribute packets over receiving queues, or if some connections are more resource intensive that others, some cores could be overcharged while some others could be partly idle.

- Programming without preemptive multi-processing could be confusing, and some applications could be harder to reason about.

Client sockets

Rusty does not support the creation of client *TCP* connections by the application layer, as two issues have not yet been resolved:

- To create a new connection, the application layer must know which instance of *TCP* will handle the connection. This is determined by the hashing function that the load-balancer uses. The way this function works is not documented, but could likely be obtained by looking at the source code of the load-balancer (the load-balancer runs on a programmable RISC co-processor).
- A kind communication mechanism between the cores should also be put in place. If one core wants to initiate a connection that will be handled by another core, it should ideally delegate the creation of the connection to the other core.

Cache hierarchy and memory locality

Each *TileGx* core has a 32 KB L1 instruction cache, a 32 KB L1 data cache and a 256 KB unified L2 cache. There is no shared L3 cache but L2 caches can be accessed from others cores, with an added latency. The following table summarizes memory subsystems and their main properties [Tile12]:

Memory	Size	Associativity	Latency (*CPU* cycles)
L1 instruction	32 KB	2-way	2
L1 data	32 KB	2-way	2

Memory	Size	Associativity	Latency (*CPU* cycles)
Local L2	256 KB	8-way	11
Other core L2	36 x 256 KB = 9MB	8-way	32 to 41
RAM (DDR3-1600)	16 GB	Irrelevant	67 if page in TLB, 88 if not

The *TileGx* architecture allows software developers to choose where memory pages are cached in the shared L2 cache. By default pages are cached over the whole set of cores (*hash-homed pages*), the least significant bits of memory addresses determining on which core cache lines are located. The alternative is to link a page to a specific core (*homed pages*), in which case memory entries of this page will all be cached in this core. Ideally, data shared among several cores should be *hash-homed* while memory accessed by a single core should be *homed*. It is still possible to access *homed* memory cached on another core, but the latency is about three to four times higher.

In order to take advantage of this cache hierarchy, data related to each *TCP* connection (connection state, transmission windows, transmission queues, timers ...) are cached on the core that handle the connection. This increased the network throughput by 2.5%.

Abstracting the network stack

One of the main goals while designing *Rusty* was to make the network chunk of the software as loosely coupled as possible from the network driver, while still being as efficient as possible.

The outcome is that the framework can be relatively easily ported to another network driver, without the *Ethernet*, *IPv4*, *ARP* or *TCP* pieces of code requiring any change.

The *Ethernet* layer accepts a '*driver data-type*' as a *C++* template argument. The driver type shall provide some functions, such as one to send packets, and must transfer to the *Ethernet* layer any received packet. The driver must also provide a way to manage network buffers, as described in the following section.

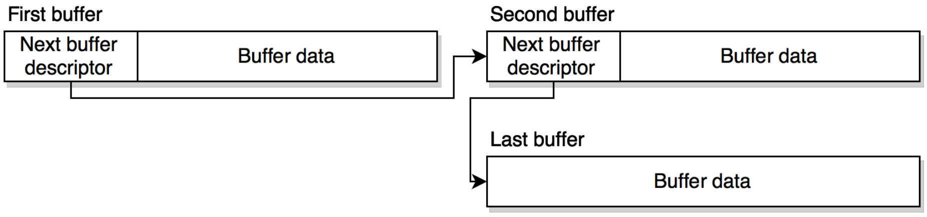
Others layers of the network stack follow the same design. The *IPv4* layer accepts a underlying 'data-link type', the *ARP* layer a 'data-link type' and a 'network type' types, and *TCP* accepts a 'network type'. One could write a new *IPv6* layer and be able to run the existing *ARP* and *TCP* layers as they exist today.

The templates and member types features of the *C++* programming languages were heavily used to put together this kind of genericity, without having any impact on performances.

Interacting with network buffers

The framework interacts with the network hardware by receiving and by passing *Ethernet* frames in network buffers. Network buffers contains the raw bytes that the network interface received or is expected to transmit.

In the case of the network driver of the *Tilera development framework*, network buffers allocation is managed by the network hardware and the buffers reside on a previously allocated memory page that the network controller is aware of. Multiple buffers can be chained together. A buffer chain only contains a single *Ethernet* frame. The following figure shows three chained network buffers:



All the buffers, except the last one, contain a descriptor of the next buffer in the chain (the descriptor tells the memory address and the size of the buffer, and if it is the last buffer in the chain).

The main reason to have chained buffers on *Tilera*'s hardware is because of the way buffers are allocated. For efficiency, network buffers are allocated from pools of pre-allocated 128, 256, 512, 1,024 and 1,664 bytes buffers. Without chained buffers, allocating memory for a 513 bytes packet would cause the allocation of a single 1,024 bytes buffer. With chained buffers, a 512 bytes buffer could be chained with another 128 bytes buffer, saving memory.

Chained buffers can be disabled, in which case an instance of the smallest buffer that can contain the full packet is allocated. This increases memory usage but simplifies how memory buffers are accessed. *Rusty* supports both chained and unchained network buffers.

Some others drivers (such as the one provided by *Intel* for its professional *NICs*) also support some kind of chaining.

Higher level interface

The way buffers are represented is specific to each hardware and driver. To keep the network stack loosely coupled with the driver, it does not directly interact with network buffers, but uses a common higher level interface. An implementation of this higher level interface should be provided by the abstraction layer between the network driver and the network stack.

The goals while designing this common higher level interface were:

- To not produce any (noticiable) performance overhead.
- To provide a way to directly work on the buffer's bytes when possible, without copying data (zero-copy).
- To be easy and simple to use.
- To have slicing operators. Slicing makes it passing sub-sections of a packet to the upper network layer easier.

Rusty uses an interface named *cursors* to abstract network buffers. *Cursors* can be used to read and write data. It is highly influenced by *ByteStrings*, a very simple yet powerful I/O abstraction designed by *Don Stewart* for the *Haskell* programming language [Stew05]. It shares some usage patterns with *C* pointers and iterators of the *C++ STL*.

The application event handlers also read and write transmitted data through cursors.

A cursor is an abstract data-type representing a position in a byte-stream, with a way for the user to know how many bytes remain from this position, and ways to move the cursor:

- `size_t size()` is a method on a cursor that returns the number of bytes that can be read before reaching the end of the stream.
- `cursor_t drop(size_t n)` is another method that returns a new cursor pointing `n` bytes after the cursor on which it was applied.
- `cursor_t take(size_t n)` returns a new cursor containing only the first `n` bytes of the cursor on which the method was applied.

All methods return a new cursor, without modifying the cursor on which they are applied (cursor are *immutables*). This makes cursor usage clearer (cursors can be used in expressions without any side-effect), and makes backtracking effortless.

To read and write data from and into the buffer, two additional methods must be provided by the cursor. `cursor_t read(char *data, size_t n)` and `cursor_t write(const char *data, size_t n)` can be used to copy `n` bytes of data into/from the given memory buffer. Again, both methods return a new cursor, pointing to the first byte after the read/written data.

Zero-copy read and write

The `read()` and `write()` methods do not provide a zero-copy interface. Indeed, data is copied to and from the given user buffer at each call.

One could write a `read()`-like method that returns a pointer to the data directly in the network buffer, but it will not work when the requested data is spread over multiple chained buffers.

A solution could be to provide a function that copies the data that overlaps over multiple buffers in another temporary memory when it happens. There would be zero memory copy except in the rare cases of a such overlap.

The usage would look like:

```
// Reserves some temporary memory required if the data needs to
// be copied, because of being spread over several buffers.
ipv4_header_t hdr_content;

// The method will either return a pointer to the data directly
// in the network buffer, or a pointer to ‘hdr_content’ with the
// data being copied there.
const ipv4_header_t* hdr =
    cursor.read(sizeof (ipv4_header_t), &hdr_content);

// [Processes the IPv4 header at ‘hdr’.]
```

Applying this same idea on a `write()`-like function would be more complex, as the data would need to be copied back into the network buffer after the temporary memory was used:

```
// Reserves some temporary memory.
ipv4_header_t hdr_content;

// The method will either return a pointer to the data directly
// in the network buffer, or a pointer to ‘hdr_content’.
ipv4_header_t* hdr =
    cursor.start_write(sizeof (ipv4_header_t), &hdr_content);

// [Writes something into the IPv4 header at ‘hdr’.]

// This call will do nothing if ‘start_write’ gave us pointer to
// the network buffer. Otherwise, it will copy the content of
// ‘hdr_content’ into the network buffer.
cursor.end_write(sizeof (ipv4_header_t), hdr);
```

Instead of offering this not really convenient pair of `write_` methods, the abstraction layer between the driver and the network stack must provide an higher-order method (i.e. a method that accepts a function as argument) that catches this pattern:

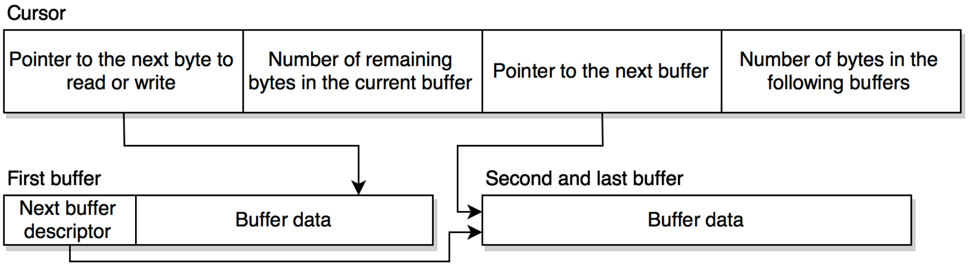
```
cursor.write_with(
    sizeof (ipv4_header_t),
    [](ipv4_header_t* hdr)
    {
        // [Writes something into the IPv4 header at ‘hdr’.]
        }
    );
```

The write_with() and read_with() methods are inlined by the compiler. This usually removes the overhead of using a closure

The method will either directly provide to the given function a pointer to the network buffer, or a pointer to a temporary memory if the data is spread over several network buffers. A similar `read_with()` method exists to read data without copying.

Cursors implementation

Implementing cursors for a new driver should be straightforward. The following diagram shows the four fields of the cursor data-structure used to abstract *Tilera*'s chained buffers:


Cursor
Pointer to the next byte to read or write
Number of remaining bytes in the current buffer
Pointer to the next buffer
Number of bytes in the following buffers
First buffer
Next buffer descriptor
Buffer data
Second and last buffer
Buffer data

The first field is a pointer to the next byte to read/write in the current buffer. The last field, telling how many bytes is remaining in the whole buffer chain (here pictured by two chained buffers), is required to implement slicing efficiently. `take()` is implemented as an `O(1)` operation.

If chained buffers are disabled on the *Tilera*'s hardware, the cursor data-structure only contains the first two fields. It turned out that using or not chained buffers does not significantly affect network throughput. However, it shows that the network stack is able to smoothly work with two different network buffer representations.

Writers

A problem arises when sending data to a remote *TCP* instance. To benefit from the zero-copy interface, the send method used to send data through a *TCP* connection should have access to the network buffers, through a cursor. However, it happens that the network buffer could not always be allocated at the time the `send()` method is called:

- Because the *TCP* transmission window is empty and no data segment can be transmitted immediatly. The size of the data (and thus the size of the network buffer) that it will be possible to send once the transmission window will be updated is not yet known.

- Because the entry of the *ARP* table mapping the recipient *IPv4* address to its *Ethernet* address has expired. An *ARP* transaction must be performed before transmitting the data.

Moreover, the requested data could be too large to be delivered in a single segment, and should therefore be split in multiple network buffers (remember that one network buffer or network buffer chain could only contain one *Ethernet* frame).

To solve this issue, the application layer does not give the data directly when making a call to `send()`. Instead, it gives a function able to write the data into a network buffer. The function should be able to only write a sub-section of the data (in case of multiple segments), and should be able to be called repeatedly on the same data (in case of a retransmission). In the framework, this function is called a writer, and its usage when used with the send function is as follow:

```
static const char http_status[] =
    ‘‘HTTP/1.1 404 Not Found\r\n\r\n’’;

connection.send(
    // Total size of the data to transmit.
    sizeof (http_status) - sizeof (‘\0’),


    // This function writes the content of ‘http_status’ in the
    // network buffer.
    // The ‘offset’ parameter tells how many bytes should be
    // skipped at the beginning of ‘http_status’ (the function
    // will be called with different ‘offset’ values if the data
    // is divided in several segments).
    [](size_t offset, cursor_t out_buffer)
    {
        out.write(http_status + offset, out_buffer.size());
    }
);
```

The *TCP* layer keeps writers in a transmission queue until the data they write have been acknowledged.

Checksums

Two checksums must be computed when receiving or emitting a packet:

- One checksum of the IPv4 header. It is only computed on the header, not on the payload.
- One checksum of the TCP segment. It is not only computed on the TCP header, but also on the segment's payload and on a pseudo-header containing fields of the IPv4 datagram (addresses and segment size).

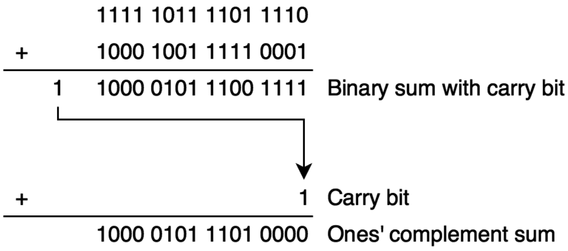
Both are computed the same way, using the standardized *Internet checksum algorithm*.

The Internet checksum

The Internet checksum is the 16 bit ones' complement of the ones' complement sum of all 16-bit words on which the checksum is applied. A virtual zero byte padding is added if the data contain an odd number of bytes. In other words, computing the Internet checksum is a two steps process:

1. First, we compute the ones' complement sum all 16-bit words on which the checksum is applied. The 16-bits ones' complement sum of the bytes `[a, b, c, d, e, f, g]` is `[a, b] +’ [c, d] +’ [e, f] +’ [g, 0]` (where `+’` is the ones' complement addition and the notation `[a, b]` is for a 2 bytes (16-bits) word). A zero byte is added to have an even number of bytes.
2. The checksum is obtained by computing the ones' complement of this sum. The ones' complement of a number is its binary negation (the `NOT` binary operator). The checksum of our 7 bytes is thus `NOT([a, b] +’ [c, d] +’ [e, f] +’ [g, 0])`.

The ones's complement addition (the `+’` operator) is the traditional addition, but with the carry-bit added back to the resulting sum. As an example, here is the 16-bits ones' complement addition of the two binary numbers:


1111 1011 1101 1110
+ 1000 1001 1111 0001
1 1000 0101 1100 1111 Binary sum with carry bit
+ 1 Carry bit
1000 0101 1101 0000 Ones' complement sum

A naive algorithm to compute the Internet checksum could be the following one:

```
uint16_t *p = (uint16_t *) data;

// Computes the ones’ complement sum.

uint32_t sum = 0;
for (int i = 0; i < size / sizeof (uint16_t); i++) {
    sum += p[i];

    // Adds the carry-bit to the sum.
    sum = (sum & 0xFFFF) + (sum >> 16);
}

// If the checksum is computed on an odd number of bytes,
// computes the last pair addition by adding a virtual zero
// byte.
if (size & 0x1) {
    sum += ((uint16_t) data[size - 1]) << 16;
    sum = (sum & 0xFFFF) + (sum >> 16);
}

uint16_t checksum = (uint16_t) ~sum;
```

The algorithm is simplified and is not correct on little-endian computers.

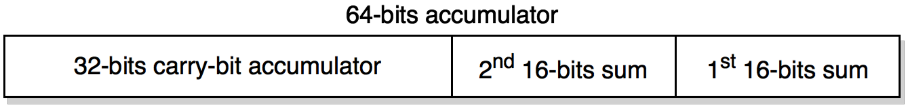
Efficient computation of the Internet checksum

The reduce the cost of computing the checksum, I do not use the algorithm presented in the preceding section. Instead, the checksum is computed using a more efficient algorithm based on techniques and properties of the ones' complement addition featured in [RFC1071].

- First, in place of adding 16-bits of data at a time, I compute two 16-bits sums on 32-bits of data simultaneously. The sum are then added together to produce the final sum.

- Secondly, in place of processing the carry-bits at each iteration, I keep adding them in an accumulator, and I only add them at once to produce the final sum.

Both of these optimisations can be cleverly engineered together by using a single 64-bits integer. The 32 mosts significant bits of the 64-bits integer accumulate carry-bits while the 32 least significant bits contain the two 16-bits sums:



The following (simplified) algorithm computes the ones' complement sum using a single 64-bits integer:

The actual algorithm is slightly more complicated, as this one assumes that the data hold on a multiple of 4 bytes. The actual algorithm also deals with memory alignment and endianness.

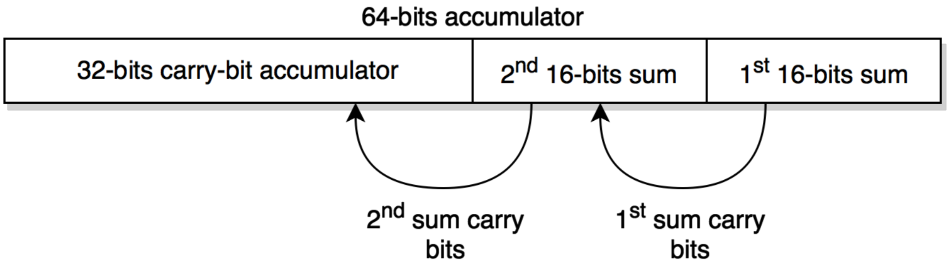
```
uint32_t *p = (uint32_t *) data;
uint64_t sum = 0; // The 64-bits accumulator.

// Sums the data, 32-bits a a time. A each iteration, the first
// 2 bytes of the 32-bits processed data are added to the second
// sum, while the last 2 bytes are added to the first sum.
for (int i = 0; i < size / sizeof (uint32_t); i++)
    sum += p[i];

// Sums the two 16 bits ones' complement sums and the carry-bits
// together.
do
    sum = (sum >> 16) + (sum & 0xFFFF);
while (sum >> 16);
```

It is a little bit puzzling to understand why and how this algorithm works. It has to do with two properties of the ones' complement addition:

1. The ones' complement addition is commutative and associative. That's why you can compute two sums at the same time.

2. The carry-bit addition being a simple addition, it is also commutative and associative. Carries of the first sum are added to the second

sum, while carries of the second sum are added to the carry-bit accumulator. Both will be at some point added to the final sum and no carry will be lost.



Using the optimized routine increases the network throughput of the stack by about 2%.

Pre-computed checksums

While having efficient algorithms is valuable, it appears that the fastest way to compute something is to not having to compute it. The technique presented here shows how checksums of some *TCP* payloads can be pre-computed.

For some applications, it could be interesting to pre-compute the checksums of data that are sent on a regular basis. For example, it would be wise to find a way to pre-compute the checksums of the files served by a web-server.

However, since the size of the remote *TCP* window and *MSS* are not known before the connection is established, any sub-section of the data may need to be checksummed at some point for some connection, e.g. we could only need to compute the checksum of 715 bytes starting at offset 6,287 in the file. A pre-computed table must therefore be able to efficiently bring back the checksum of any sub-section of the data, and not only the checksum of the whole data.

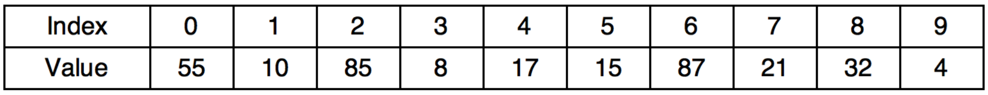
One solution could be to precompute the checksum of every possible sub-section of the data. But that would require an extraordinary amount of memory space (in the order of `O(n²)`, `n` being the size of the data on which the checksum can be computed).

The *TILE-Gx36* has hardware support for checksum computation. However, it can only compute a single checksum per packet, while a *TCP* segment has two. The documentation also tell us that corrupted checksums can be produced under heavy load and when using large frames [Tile12]. But the primary reason to not use this hardware feature is that it was way more exciting to find a way to not having to compute the checksums at all. Thanks to pre-computed checksums and to the optimized checksum routine, less than one percent of the time required to receive or emit a frame is spent to compute checksums.

Rather than doing that, I adapted to ones' complement sums a good old algorithmic technique. The method can find the sum of integers in any contiguous sub-vector of an integer vector in only two memory accesses.

As an example, assume that you have a vector `I` of ten integers:

I



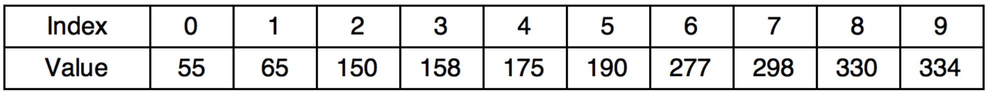
Index	0	1	2	3	4	5	6	7	8	9
Value	55	10	85	8	17	15	87	21	32	4

And that you are asked to find a way to efficiently compute the sum of any contiguous sub-vector of this vector (such as, the sum of integers from index `4` to `8`, which is `172`).

You can construct a second vector `S` containing the sum of all the integers up to the cell's index in the original vector:

All the values in the `S` vector can be computed in a single iteration.

S



Index	0	1	2	3	4	5	6	7	8	9
Value	55	65	150	158	175	190	277	298	330	334

The values in the `S` vector are defined by the following equation:

$$S(i) = \sum_{j=0}^{j \leq i} I(j)$$

Having the `S` vector, it is easy to find the sum of any contiguous sub-vector of the `I` vector. For example, if one wants the sum from index `4` to `8` in the `I` vector, he needs to look at the values at indices 8 and 3 in `S`, and to subtract them (`330 - 158 = 172 = 17 + 15 + 87 + 21 + 32`). Any sum can be computed this way in only two memory accesses.

This trick can also be used as a quick reference table for ones' complement sums, provided some adjustments because of sums being computed over two bytes. The memory space usage of the `S` vector is linear to the number of bytes it was computed from (`O(n)`), and only two memory accesses to the vector are required to compute any ones's complement sum. It is used to precompute ones' complement sums of the files served by the web-server. To avoid cache misses when doing these two memory accesses, the two values are prefetched during the copying of the payload. Using precomped checksums for the web-server instead of computing the optimized routine increases the network throughput by about 20%.

Timers

The *TCP* and the *ARP* protocol operate using timeouts.

These timeouts are kept sorted by expiration date in a binary search tree. The software periodically checks the binary tree for expired timers. As entries are sorted, this check is really fast and takes place every time a packet is received (see the main loop earlier in this chapter).

Calling the system clock to know the current time is too expansive. Instead, the network stack uses the *CPU* cycle count to know if a timer has expired. Getting the cycle count on the *TILE-Gx36* is only a dozen cycles. This way is less precise than what could be achieved with the system clock, but is enough in the context of a network stack.

Scheduling a timer requires the insertion of a node in a tree, and thus a dynamic memory allocation (*Rusty* uses the *C++* `map` container). As it will be shown in the next chapter, dynamic memory allocations are the current performance bottleneck of the network stack.

Chapter 4

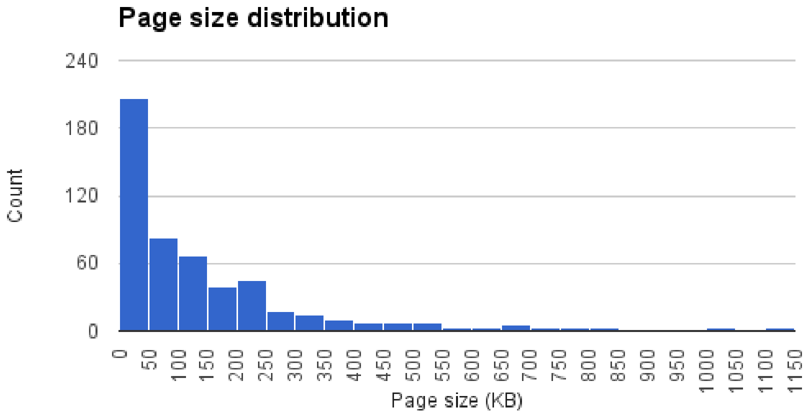
Performance analysis

In this chapter, we will see how the performances of a minimal web-server improve when it uses *Rusty* instead of the *Linux* network stack on the *TILE-Gx36* device. You will see that *Rusty* scales far better than the *Linux* network stack and that performances can be further improved by using *Jumbo Ethernet Frames*, by improving dynamic memory allocations, or by using a faster *CPU* architecture.

Simulated traffic and test conditions

The *Rusty* network stack has been developed to create an application able to generate a large amount of *HTTP* sessions in order to qualify middleboxes for highly loaded production networks. These middleboxes are developed to remove specific content in web pages, such as advertisements.

In order to simulate this use case, the simulated traffic consists of random requests for the front pages of the 500 most popular websites (according to Alexa [Alex15]). The pages have been downloaded locally. Page sizes range from 3.5 KB (`wellsfargo.com`) to 1.1 MB (`xcar.com.cn`). The average page size is 126 KB, the median page size is 68 KB. 70% weight less that 150 KB, 90 % weight less than 300 KB. Only 17 (3 %) weight more that 500 KB. The following graph shows the distribution of page sizes in the 500 pages set:



TCP connections are not used for more than one *HTTP* request. A new *TCP* handshake is initiated for every *HTTP* request. During the following tests, web-servers have been flooded with 1,000 concurrents *TCP* connections for 50 seconds, using *wrk*, a scriptable *HTTP* benchmarking tool [Gloz15]. Network stacks have been configured with an initial *TCP* window size of 29,200 bytes (*Linux*'s default). Turning the congestion control algorithm on the *Linux* stack from *Cubic* (default) to *Reno* does not produce any change in the observed performances: the link capacity is higher than the throughput of the applications and retransmissions do not occur.

Please remember that *HTTP* requests are currently not generated on the *TILE-Gx36*, but on a separate *x86* server. This server fully capable of generating the amount of requests required for the tests to follow. The *TILE-Gx36* only serves the pages via a web-server.

Minimal web-server

To minimize the overhead brought by full featured web servers, I wrote a minimal web-server that runs on top of *Rusty* or on top of *Linux*'s sockets. This server only reads the first line of the *HTTP* requests, and ignores any other *HTTP* header. Because of this, it is not fully compliant with the *HTTP* protocol, but still adequate to simulate web traffic.

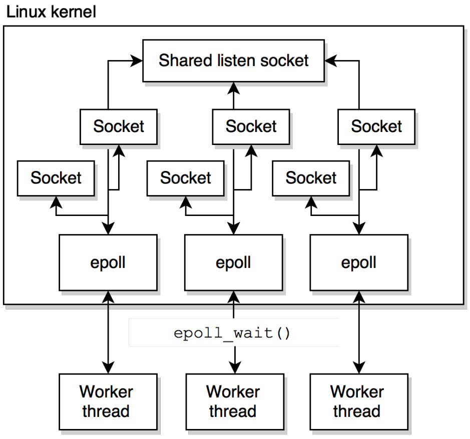
This minimal web-server also preloads in memory all the pages that can be served. It was observed that this slightly increases performances.

Shared sockets

When running on top of the *Linux* stack, the minimal web-server uses the *epoll* system-calls to watch for events on the *TCP* connections it manages. It also uses the shared sockets feature which has been introduced in *Linux 3.9* (April 2013).

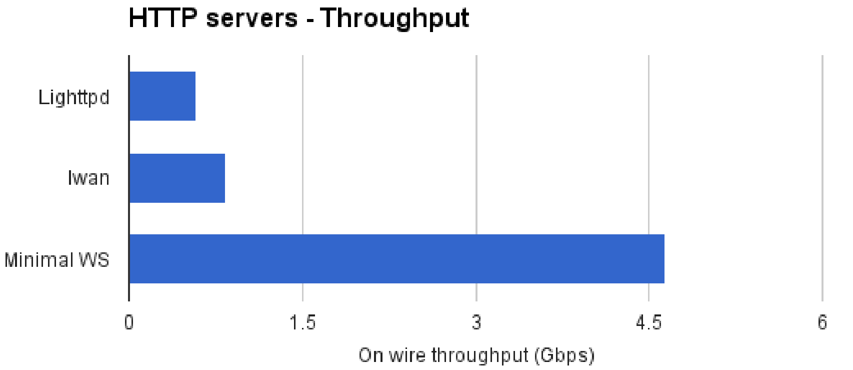
The web-server spwans a defined number of worker threads. Each worker thread has its own *epoll*. An *epoll* is an event queue shared by the operating system and the application. It allows the kernel to notify the application of events on a set of file descriptors (events like a new connection or a new arrival of data). Workers are constantly waiting for events on their queues, and processes them as they arrive. All workers share a single server/listen socket (using the new `SO_REUSEPORT` feature introduced in *Linux 3.9*). On new connections on this shared socket, only one worker is notified, and receives the new client's socket. The worker adds this socket to its event poll and processes any event on it. The number of workers is usually lower or equal to the number of cores on the hardware that runs the software. Without shared sockets, another worker should be dedicated to listening for events on the server socket and to load balance any new connection on the other workers.

This architecture is the advised way of writing efficient network applications on top of the *Linux* kernel [Kerr13].

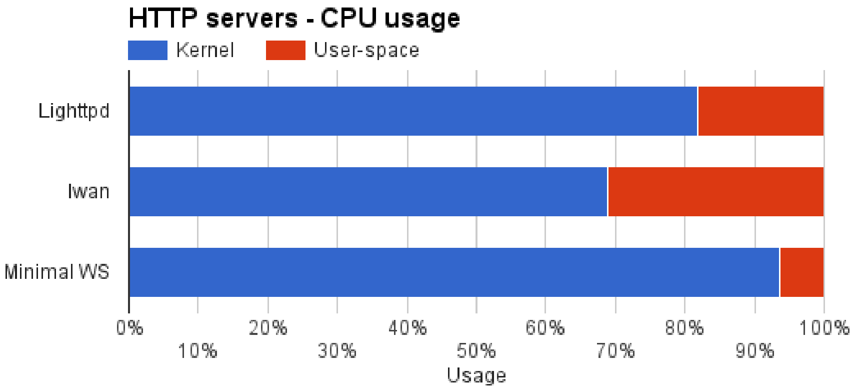


The following graph shows, in terms of *HTTP* throughput, how this minimal web-server (*Mininal WS* in the graphs) compares against full-featured servers on the *TILE-Gx36*. *Lighttpd* (version 1.4.28) is a widely used *HTTP* server which uses *epoll* (but not a shared socket) while lwan is a lightweight server that uses epoll and shared sockets [Pere15].

Nginx (which is build around a shared socket and *epoll*) was also intended to be featured in this comparison, but I did not succeed at compiling it for the *Tilera* architecture.



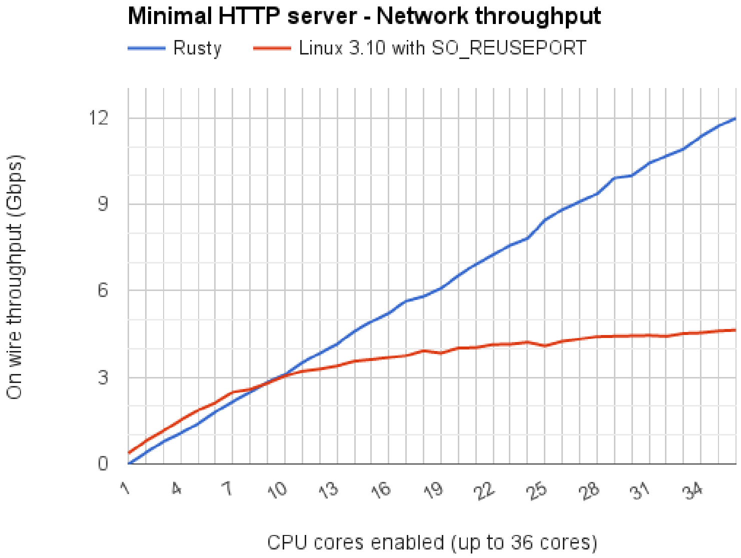
The following graph shows how much *CPU* resources are spent in the kernel and in user-space during the processing of an *HTTP* request:



The minimal web-server only spends 6% of its time in user-space. The only system-calls it does are related to the sockets it handles, as files are preloaded in memory during the initialisation. I think it is fair to say that its performances are very close to the best that one can expect to get from an network application running on top of the *Linux* stack.

Comparing Rusty and the Linux stack

The following graph shows how the minimal web-server performances increase with the number of enabled cores on the *TILE-Gx36*:



Rusty performances scale linearly with the number of cores. It fulfills a single 10 Gbps *Ethernet* link with 29 cores, because of that, starting with 29 cores, the load was evenly balanced over two links. The throughput of a single worker tops at 395 Mbps, while it reaches 12 Gbps with 35 workers.

The *Rusty* throughput on one core is zero because the framework requires at least two cores to operate: one that executes the network stack, and another for the operating system.

The *Linux* stack, unlike *Rusty*, fails to scale correctly on more than seven cores. The problem is a well-known fact and is caused by a thread contention when allocating and dereferencing file descriptors [Boy+08].

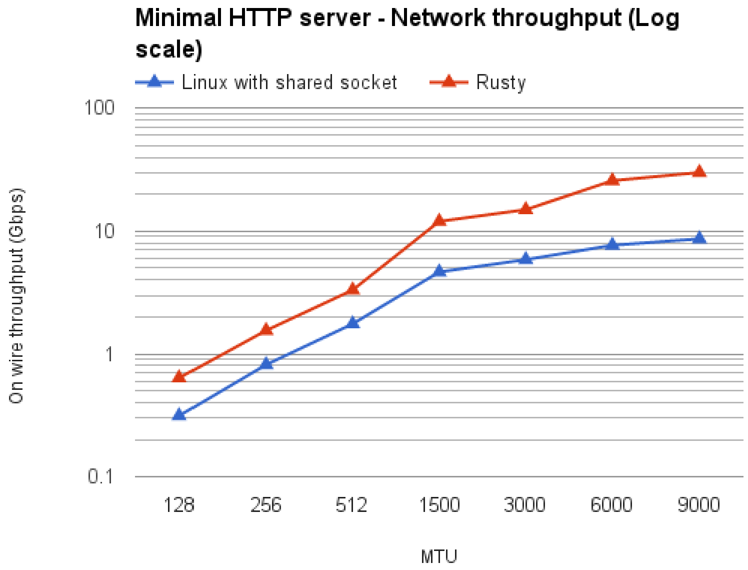
Changing the MTU

In the previous paragraphs of this analysis, the *Maximum Transmission*

Unit (*MTU*) was set to 1,500 bytes, which is the *MTU* of conventional *Ethernet* frames and the default value used by *Linux* (and probably the most frequently experienced on the *Internet*[citation needed], which would justify its usage within a web traffic generator).

The following graph and table show how the two network stacks react when the *MTU* is increased or decreased. Both stacks support *Ethernet Jumbo Frames* and all the cores of the *TILE-Gx36* were used.

Using larger *MTUs* increases performances. This is mostly because, by sending larger segments, we receive less acknowledgement segments (the cost of writing more segments is not as much an overhead as receiving and processing more acknowledgements). However, using *Jumbo Frames* to simulate *HTTP* sessions would not be representative of the typical *HTTP* traffic on the Internet.



Profiling Rusty

The main performance issue while running the minimal *HTTP* server over *Rusty* is linked to dynamic allocations, accounting for about 65% of the consumed *CPU* cycles. Copying the payload into the segment only consumes about 5% of the execution time (with an efficient use of prefetching). About one percent of the execution time is used to interact with the network driver. Thanks to the precomputed checksums, less than one percent of the execution time is spend in computing checksums. The bulk of the remaining execution time is consumed by various procedures of the network stack and of the application layer.

As it is, the framework relies on dynamic allocations to:

- Manage the timers used by the *ARP* and the *TCP* layers. Timers are kept sorted in a binary search tree. The *TCP* layer reschedules the retransmission timer virtually every time an acknowledgement segment is received, which leads to one or more dynamic allocations because of the inability of the standard *C++* binary tree container to move an entry in the tree.

- Simplify the memory management of network buffers. When a network buffer is allocated by the application, a *C++* `shared_ptr` is instancied in order to automatically release the buffer once it is no longer in use. The `shared_ptr` requires the allocation of a structure to maintain its reference counting mechanism. The use of `shared_ptrs` is really penalizing because of this memory allocation, the reference counting machinery does not induce any serious performance overhead.

- Manage the transmission queue of the *TCP* layer.

- Handle closures. The compiler can generate code that allocates memory on the heap to store the variables that closures capture (especially in writers).

- Allocate the structures that store the state of a *TCP* connection (the *TCP Control Block*).

- Store out of order segments.

Dynamic allocations are particularly slow on the *TILE-Gx* architecture. A single allocation takes about 800 ns (1000 cycles) on the *TILE-Gx* while it takes about 100 ns (200 to 300 cycles, depending on the frequency) on a modern *x86* microprocessor.

Except for the closures, all allocations are done through a single standard *C++* allocator instance. It could easily be substituted with another more efficient allocator, that relies on memory polls (*mTCP* and *Seastat* rely on a such allocator). This would probably substantially lessen the cost of memory allocations and increase throughput. It has not been done yet because of a lack of time. The substitute allocator would have to be able to benefit from the homed memory pages provided by the Tilera hardware, like the current one does.

Existing allocations could also be removed (especially by rewriting closures so they do not allocate memory).

Performances of the TILE-Gx36

In the course of the development of this software, I noticed that the *TILE-Gx36* was seriously slower than present-day *x86* microprocessors, even with all its 36 cores used efficiently.

To highlight the relative low performances of the *TILE-Gx36*, I executed a few *CPU*-intensive tasks and compared their execution speed to those of three others *x86* computers:

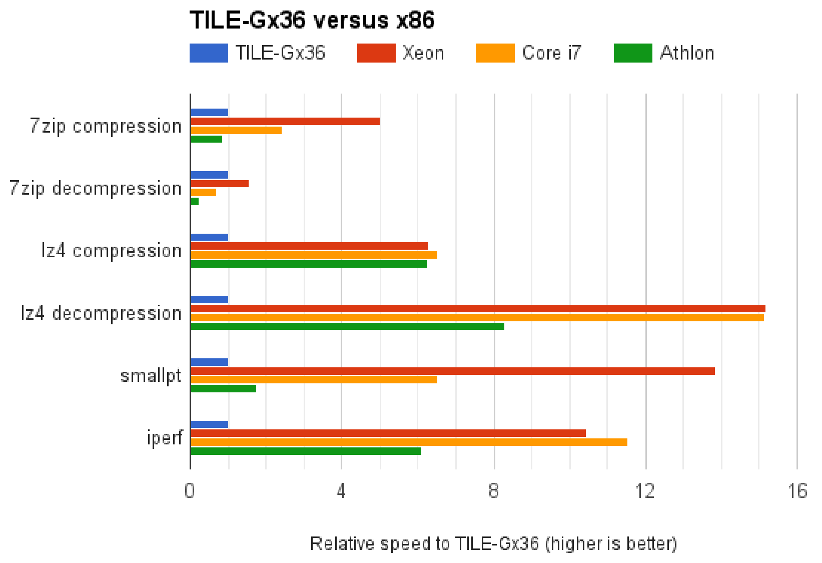
- A server with two 8-core *Intel Xeon E5-2650*, running at 2.6 Ghz each.
- An high-end desktop with a 6-core *Intel Core i7 4930K*, running at 3.4 Ghz.
- A low-end desktop computer orchestrated by a 4-core *AMD Athlon X4 750k*, running at 3.4 Ghz.

Four applications were evaluated:

- *7zip*, an widely used multi-threaded compression/decompression tool.
- *lz4*, a compression/decompression tool written for real-time application. The tool only uses one single core.
- *smallpt*, a small ray-tracer that runs on several cores.

- *iperf*, a system tool to benchmark the performances of the *Linux* network stack. The tool ran on the virtual loopback interface in such a way that the network hardware does not interfere with the results.

The following graph shows the results of this evaluation. Performances are relative the *TILE-Gx36* (i.e. a value of 5 means that the corresponding chip is 5 times faster than the *TILE-Gx36*):



Except for the *7zip* benchmark, the *TILE-Gx36* consistently lags behind other chips, especially on the non-parallelized *lz4* benchmark. On the parallel smallpt test, the *Tilera* chip is almost two times slower than the low-end *x86* CPU.

I think that an significantly higher throughput could potentially be obtained by porting *Rusty* on a many-core *x86* system with a *NIC* supporting the load balancing of *TCP* flows.

Conclusion

This document went through the details of the implementation of an high-throughput *HTTP* server by creating a new highly-scalable *TCP/IP* network stack.

While this solution has managed to improve performances by more than 240% as compared to a solution using the *Linux* network stack, it failed to achieve the initially targeted goal of fulfilling four 10 Gbps links. Performances could be further improved by carrying out some optimisations that have not been put in place.

The development of the *Rusty* network stack took much more time than expected (more than one thousand hours). In particular, the complexity of the *TCP* protocol was vastly undervalued. An efficient implementation of the protocol could actually reach far more states than the eleven usually defined (*LISTEN*, *SYN-SENT*, *ESTABLISHED* ...).

As an example, it is specified that the protocol must move, when the application layer asks to close a connection (i.e. by calling the `close()` function), from the *ESTABLISHED* state into the *FIN-WAIT-1* state while sending a *FIN* segment. But what if the transmission queue is not empty ? In this case, the *FIN* segment can not be delivered immediately and the state machine is actually in a state between those two. It must acts as if it was in the *FIN-WAIT-1* state for the application layer, but still processes incoming segments as if it was in the *ESTABLISHED* state. And what if a *FIN* segment is received while in this not clearly defined state ? In this situation, the state machine must once again move into another unspecified state.

To decision of designing a reusable and portable network stack, instead of a simpler stack directly integrated into the traffic generator, made the software more interesting (in my opinion), but also more challenging and complex.

I am very pleased by the final outcome of this project, but it took me way more time than what I would have ever plan to spend on a Master thesis. I am proud of the result, but never would I do that a second time.

References

[Alex15] Alexa. The top 500 sites on the web. alexa.com/topsites, 2015.

[Boy+08] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao0, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operat- ing system for many cores. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[Clou15] Cloudius Systems. Seastar is an advanced, open-source C++ framework for high-performance server applications on modern hardware. seastar-project.org, 2015.

[Gloz15] Glozer, Will. wrk - a HTTP benchmarking tool. github.com/wg/wrk, 2015.

[JWJ+14] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. *In Proceedings of the USENIX conference on Operating systems design and implementation (OSDI*), April 2014.

[Kerr13] Kerrisk, Michael. The SO_REUSEPORT socket option. lwn.net/Articles/542629/, March 2013.

[Node15] Node.js. Node.js v0.12.7 Manual & Documentation. nodejs.org/docs/latest/api/net.html, 2015.

[Pere15] A. F. Pereira, Leandro. Experimental, scalable, high performance HTTP server. lwan.ws, 2015

[RFC793] Information Sciences Institute (University of Southern California). RFC 793 - Transmission Control Protocol. `tools.ietf.org/html/rfc793`, Sept. 1981.

[RFC1071] Braden, and Borman. RFC 1071 - Computing the Internet Checksum. `tools.ietf.org/html/rfc1071`, Sept. 1988.

[RFC5681] Allman, et al. RFC 5681 - TCP Congestion Control. `tools.ietf.org/html/rfc5681`, Sept. 2009.

[Stew05] Stewart, Don. Bytestring: Fast, Compact, Strict And Lazy Byte Strings With A List Interface. `hackage.haskell.org/package/bytestring`, 2005.

[Tile12] Tilera. *Tile Processor Architecture Overview For The TILE-Gx Series*. 2012.