Advanced Network Programming (ANP)
XB_0048

Networking concepts

Animesh Trivedi
Autumn 2020, Period 1

VU VRIJE UNIVERSITEIT AMSTERDAM

Layout of upcoming lectures - Part 1

Sep 1st, 2020 (today): ~~*Introduction and networking concepts*~~

Sep 3rd, 2020 (this Tuesday): *Networking concepts (continued)*

Sep 8th, 2020 : *Linux networking internals*

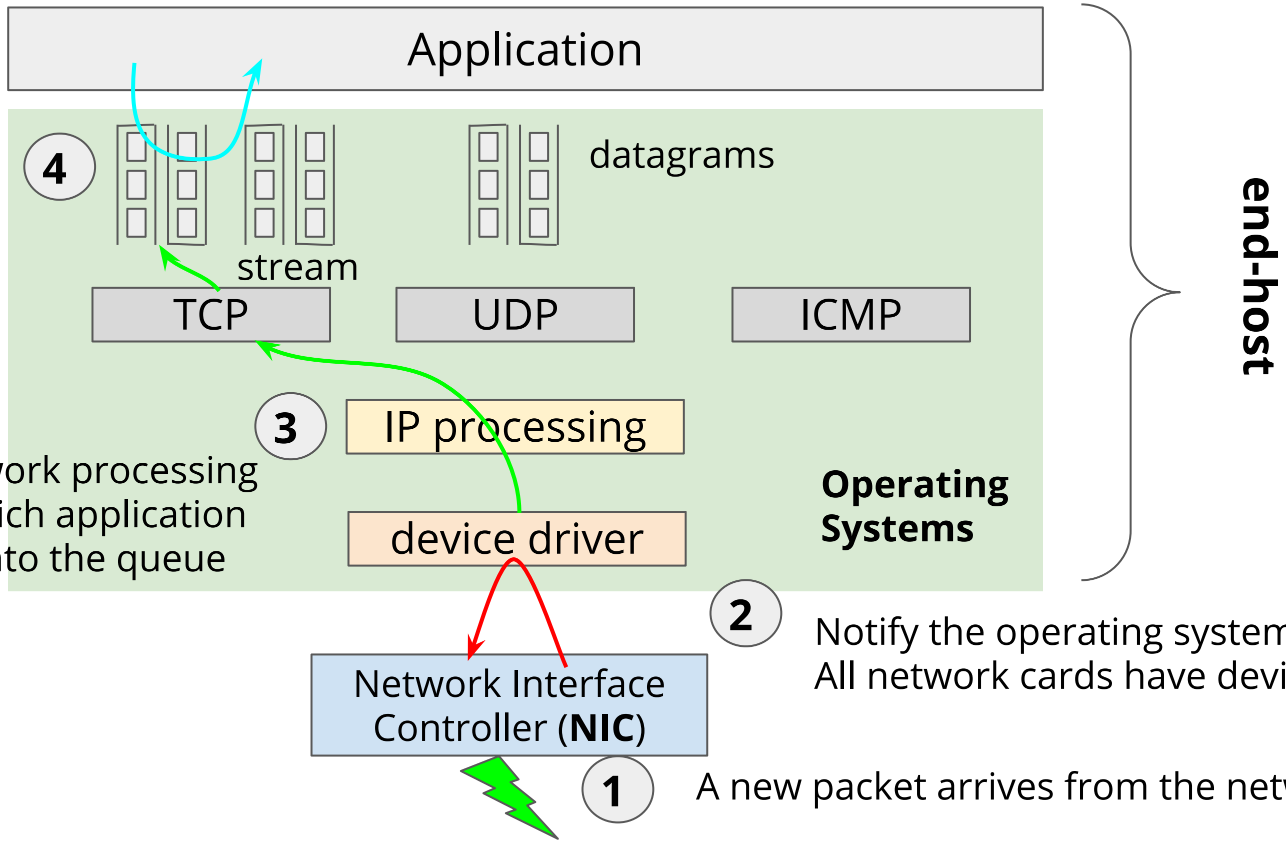
Sep 10th 2020: *Multicore scalability*

Sep 15th 2020: *Userspace networking stacks*

Sep 17th 2020: *Introduction to RDMA networking*

A packet's journey - (simplified) Receiving path

Consume data
when application
calls recv()



Perform network processing
figure out which application
queue data into the queue

Notify the operating system
All network cards have device drivers

A new packet arrives from the network

Still many unanswered questions here

Think of the receive path. This is more complicated than the sending path (can you think of why?)

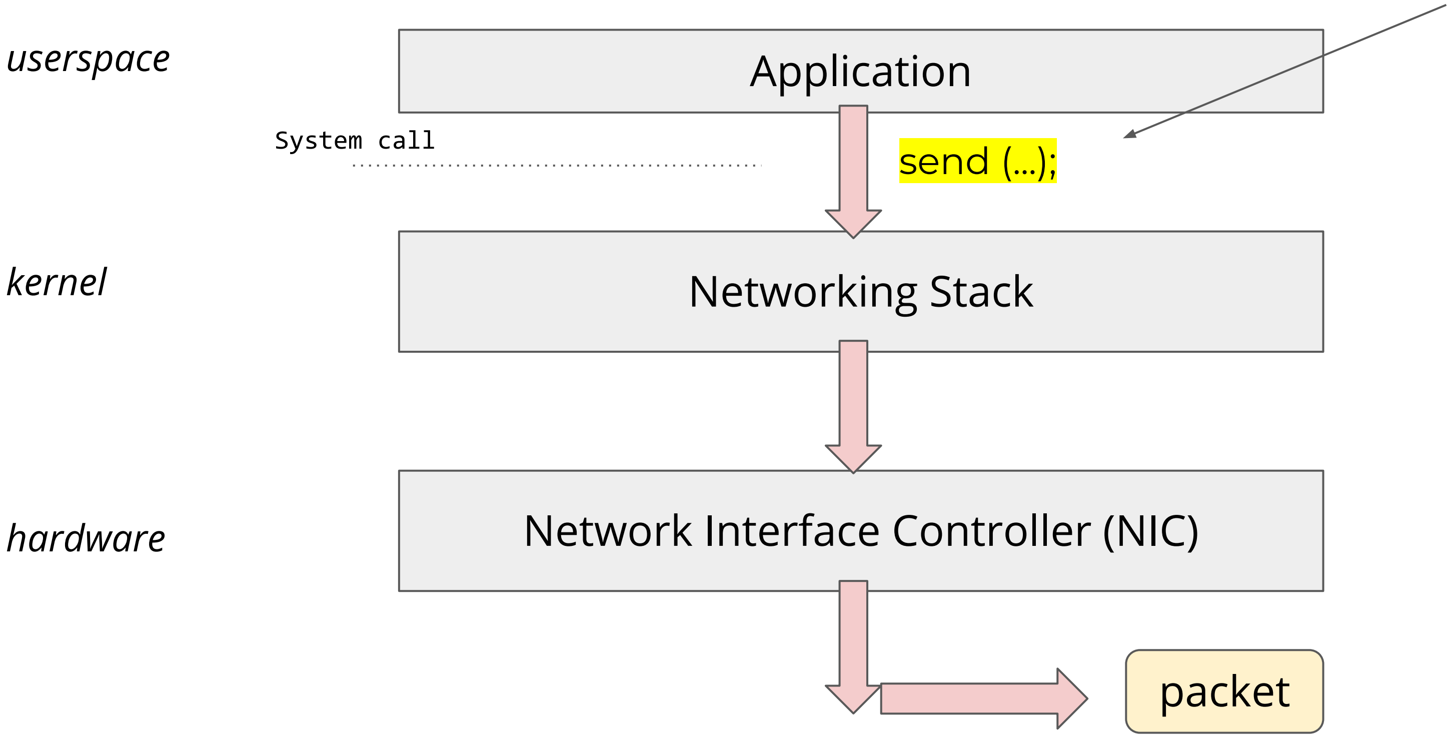
1. ~~How to transfer data between a network controller and the end host~~
2. ~~How to notify the end host about network packet reception~~
    a. ~~Do you need to tell the end host about a packet transmission?~~
3. ~~How to build a packet with multiple protocols and headers~~
4. How much time/steps it takes to receive data? 1 bytes, 1 kB, 1 MB, or 1 GB?
5. ...and many many many more questions.

Lets answer some of them, one by one and introduce the key ideas

What is the unit of data processing, and network I/O?

The Unit of Processing

What is the largest amount of data you can transmit in a single send() call?



$man send

```
SEND(2)                    Linux Programmer's Manual                   SEND(2)

NAME
       send, sendto, sendmsg - send a message on a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       ssize_t send(int sockfd, const void *buf, size_t len, int flags);

       ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                      const struct sockaddr *dest_addr, socklen_t addrlen);

       ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);

DESCRIPTION
       The system calls send(), sendto(), and sendmsg() are used to transmit a
       message to another socket.

       The send() call may be used only when the  socket  is  in  a  connected
       state  (so  that the intended recipient is known).  The only difference
       between send() and write(2) is the presence  of  flags.   With  a  zero
       flags  argument, send() is equivalent to write(2).  Also, the following
       call
```

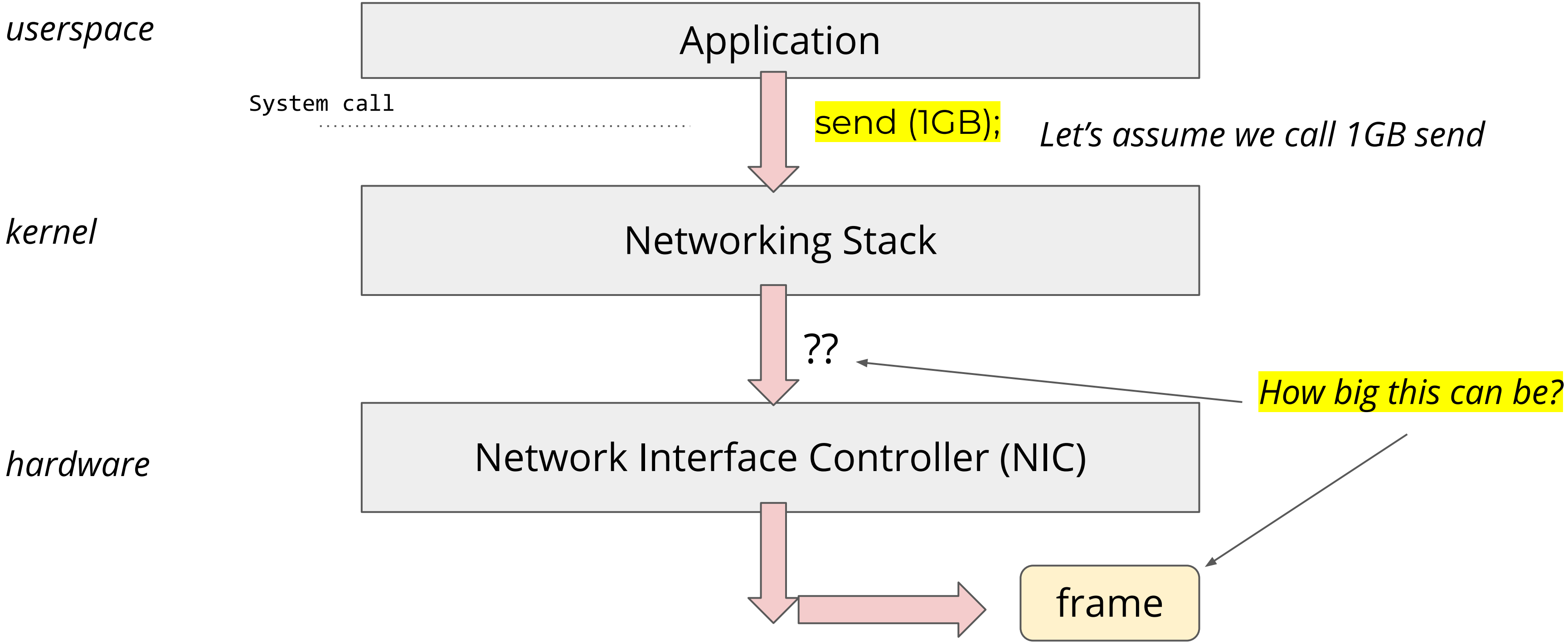
POSIX standard
- unsigned int
- unsigned long

At least 16 bits

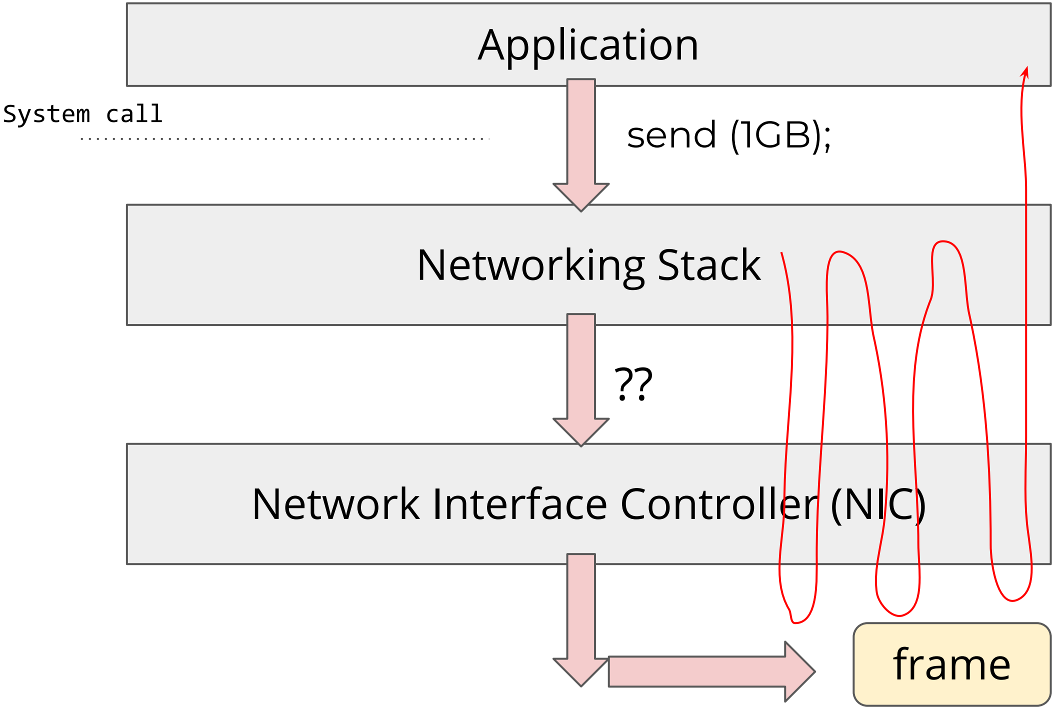
On my x86_64/Linux it is 8 bytes

The unit of network processing

Can you send 1 GB packet on the network?

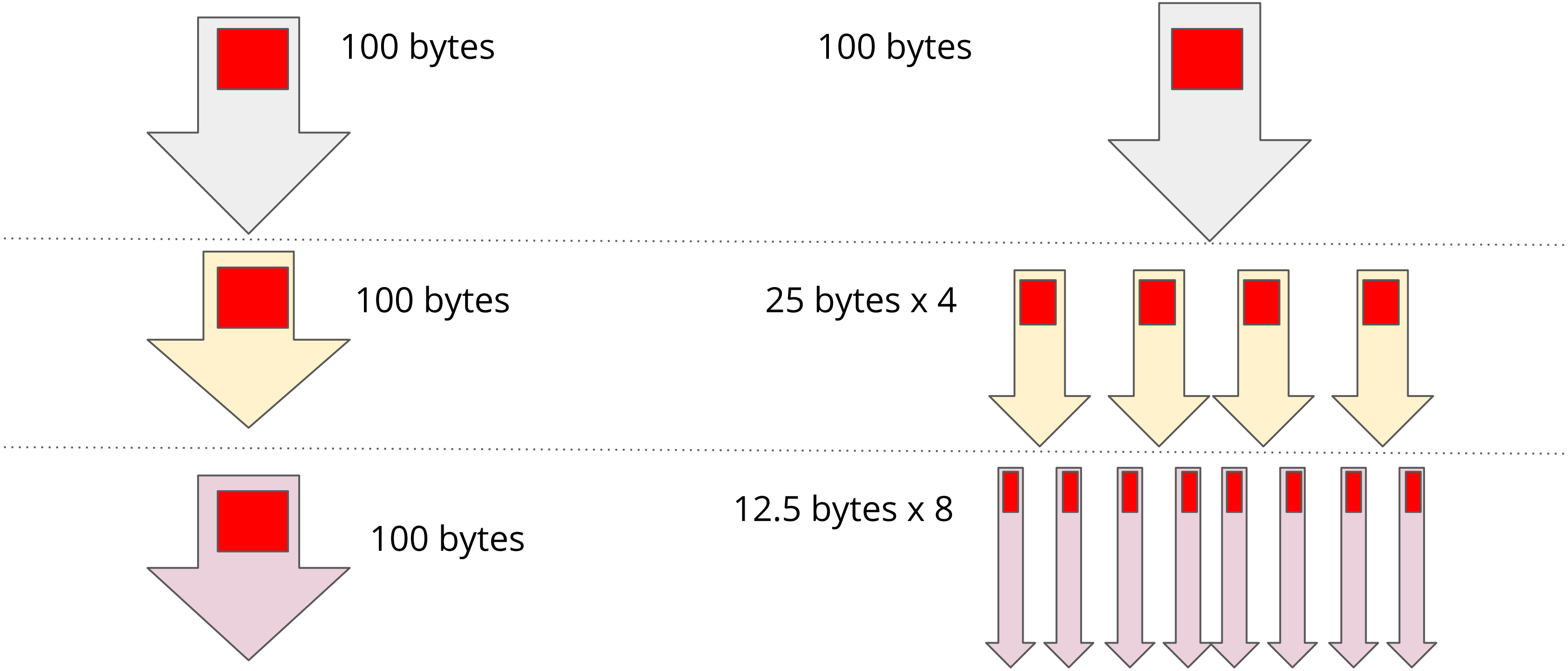


The unit of network processing



Why bother? Why does the loop count matter

The loop count



Each layer 1 iteration,
total **3** iterations done

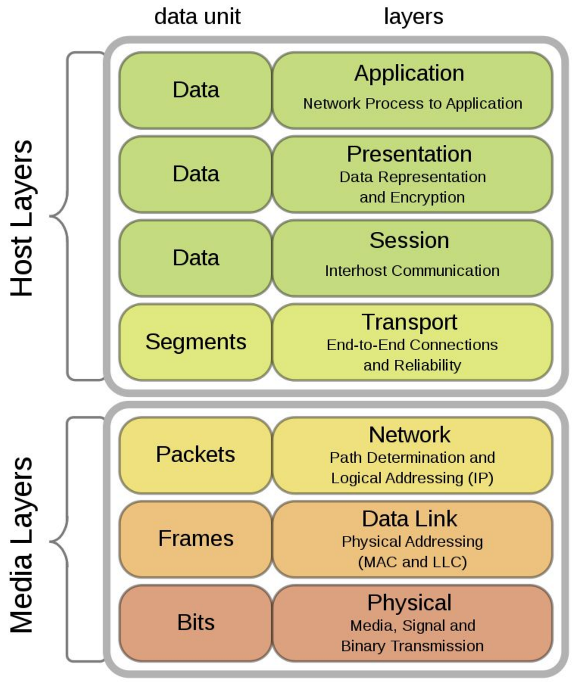
In total **13** iterations

Key challenge: different units of data processing

Multiple different abstractions and units

- Transport **segments**
- Network **packets**
- Link layer **frames**

These three can be off different sizes for different protocols, and yet we need to have a notion of interoperability.

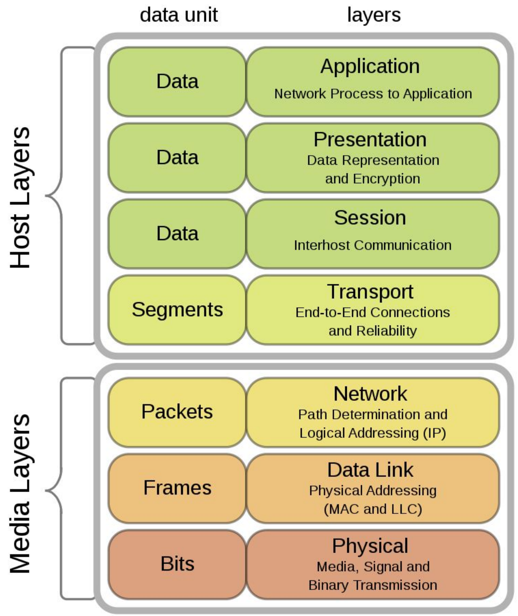


By Offnfopt - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=39917431

Why does "the loop count" matter?

The number of times you execute the "*loop*" depends on the number of "*units*" at every layer

1. Programing hardware for TX/RX is a slow operation, so you want to do it as little as possible ***(frames)***
2. There are per ***packet*** operations such as building packet headers and calculating checksums - you want to do many little times as possible
3. Per ***segment*** overheads (TCP), ACKs, SEQ processing, delivery to userspace, notification management - minimize it as much as possible



By Offnfopt - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=39917431

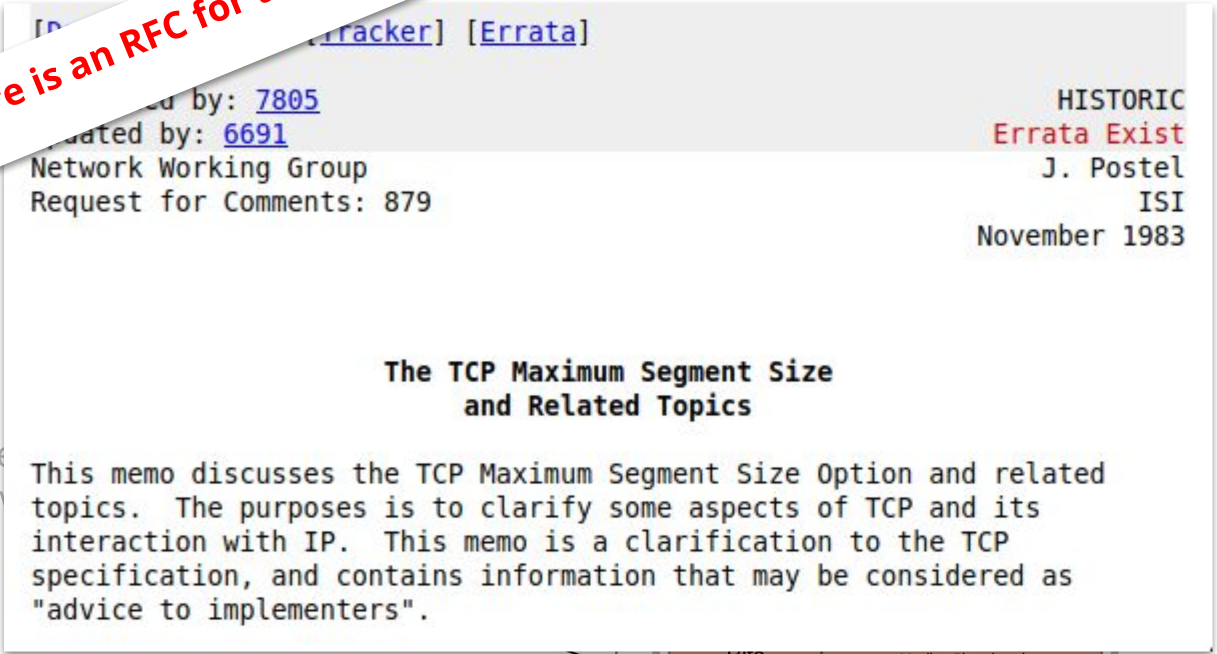
Key challenge: different ...its of data processing

Whenever in doubt check, there is an RFC for that ;)

Multiple different a...

- Tr... ...ents
- ...rk packets
- Link layer frames

These three can be off differe... different protocols, and yet w... notion of interoperability.


[P... ...racker] [Errata]

...d by: 7805 HISTORIC
...dated by: 6691 Errata Exist

Network Working Group J. Postel
Request for Comments: 879 ISI
November 1983

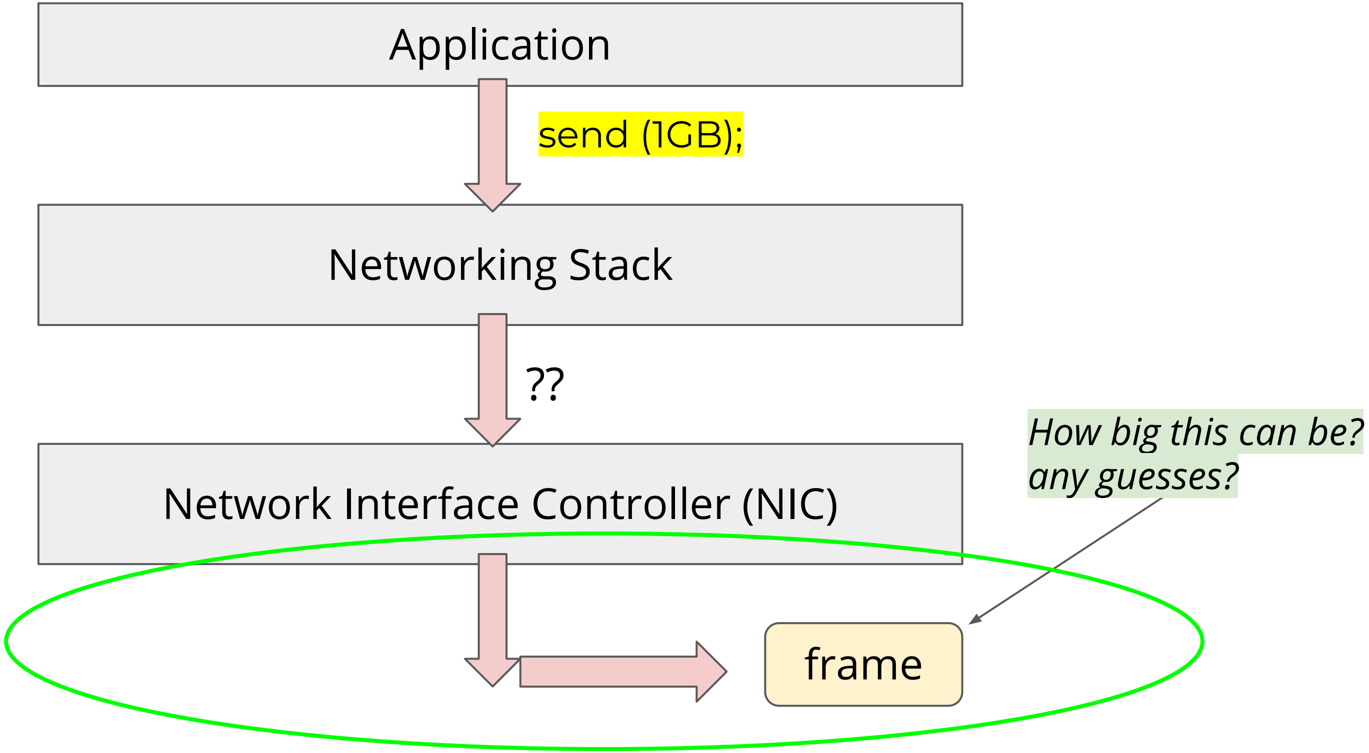
The TCP Maximum Segment Size and Related Topics

This memo discusses the TCP Maximum Segment Size Option and related topics. The purposes is to clarify some aspects of TCP and its interaction with IP. This memo is a clarification to the TCP specification, and contains information that may be considered as "advice to implementers".

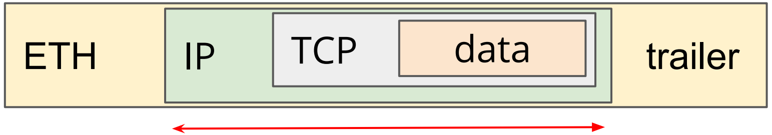
By Offnfopt - Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=39917431

The Unit of Processing

Can you send 1 GB packet on the network?



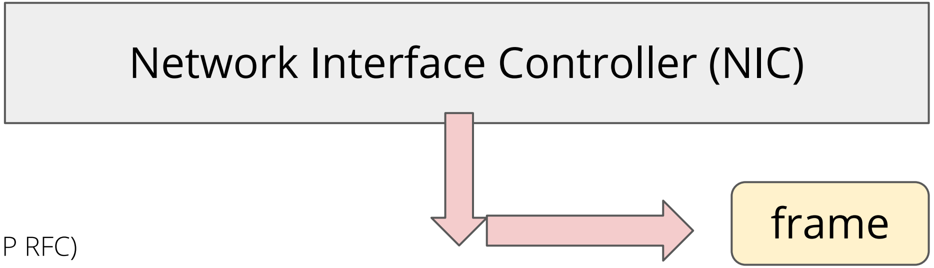
The Unit of Processing - MTU



A **MTU** (closely, but not exactly) defines how big a *frame* on a link layer (L2) can be

MTU is "network-layer concept" that defined what is the largest protocol data unit (PDU) (.e.g., for IP it is the packet) that can be sent/received in a single "network" layer operation (L3)

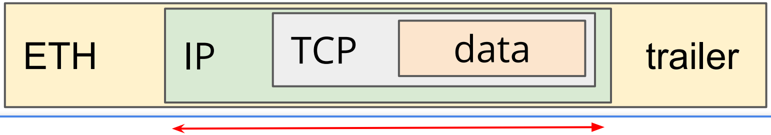
- *IPv4 Specification expect any L2 layer to support at least **576 bytes** of data (old days!)*
- *Anything less than that, IPv4 will not work. Then L2 must then provide its own way of assembly*



Maximum Transmission Unit (MTU)

https://tools.ietf.org/html/rfc791 (IP RFC)

The Unit of Processing - MTU

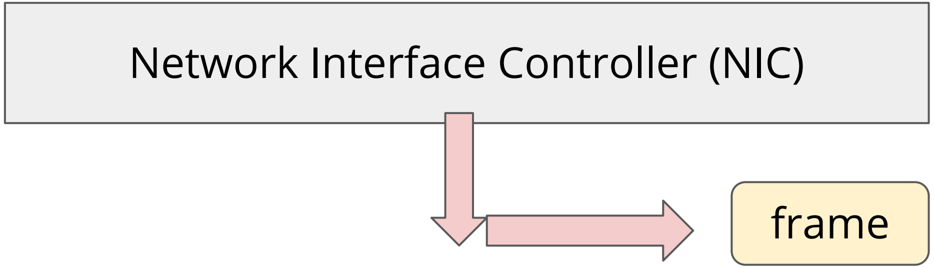


A small MTU:

- \+ more multiplexing, fine grained transmission
- \- inefficiency (see next slides)

A large MTU

- \+ less packets, more data per packet, more efficiency
- \- introduces delay for the next packet, link hogging
- \- if corrupted then a big overhead to retransmit data

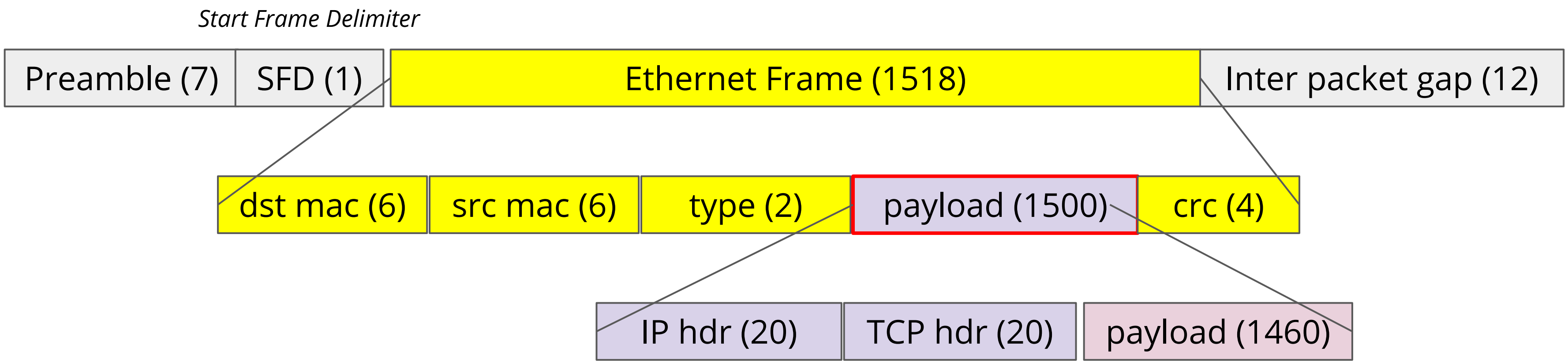


Maximum Transmission Unit (MTU)

Example: Ethernet MTU

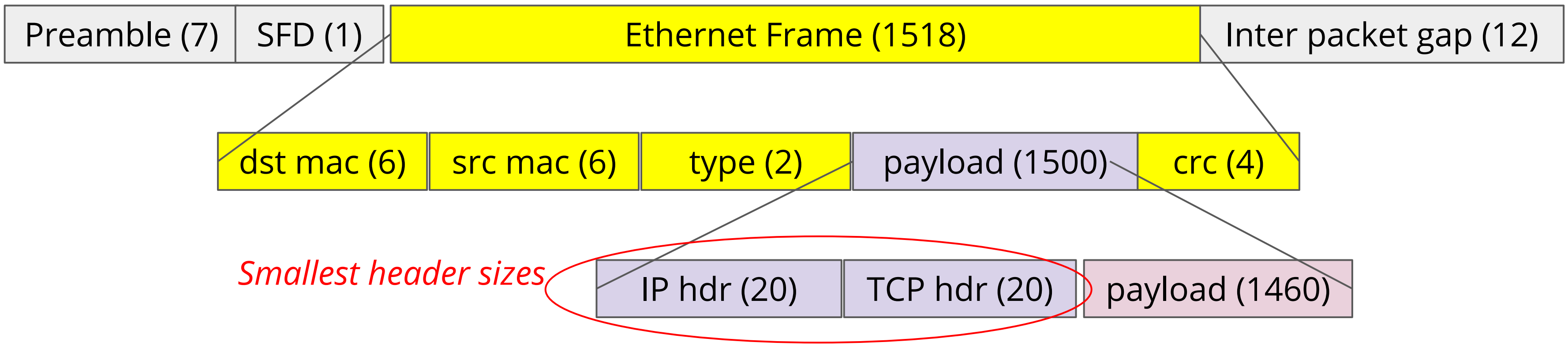
Ethernet has a MTU of **1500 Bytes** (payload, excluding its own headers)

- Historical reasons, trade-off between NIC data buffering capacity (onboard memory) and speed
- This is greater than 576 octet expected for IPv4, hence, OK



https://app.netrounds.com/static/2.33.2/support/defs-notes/theor-thput.html
http://switchpacket.blogspot.com/2014/07/understanding-difference-between-mtu.html

Calculating Ethernet Efficiency for TCP packets



1 Gbps Ethernet link : 10^9 bits per second on the wire, when constructing a maximum MTU packet

- Total bits on the wire : 1500 + 18 (ETH) + 8 (preamble+SFD) + 12 (gap) = 1,538 bytes
- Total actual data payload in the packet : 1500 - IP hdr (20) - TCP hdr (20) = 1,460 bytes

Efficiency = (1500 - 40 / 1500 + 38)*100 = **94.93%** (in reality, TCP and IP have larger headers)

Hence, on a 1 Gbps link you cannot deliver more than a TCP application bandwidth of **949.3 Mbps**

Can we improve it?

Ever heard of JUMBO frames? (https://en.wikipedia.org/wiki/Jumbo_frame)

- Ethernet standard to support larger frames
- Most common **9000** bytes

Let's do the previous calculation again, substituting 1500 by 9000

- Total bits on the wire : 9,000 + 18 (eth) + 8 (preamble) + 12 (gap) = 9,038 bytes
- Total actual data payload in the packet : 9,000 - IP hdr (20) - TCP hdr (20) = 8,960 bytes

Efficiency = (9000 - 40 / 9000 + 38)*100 = **99.14%**

Hence, on a 1 Gbps link your maximum bandwidth improves from **949.3 Mbps** to **991.4 Mbps**

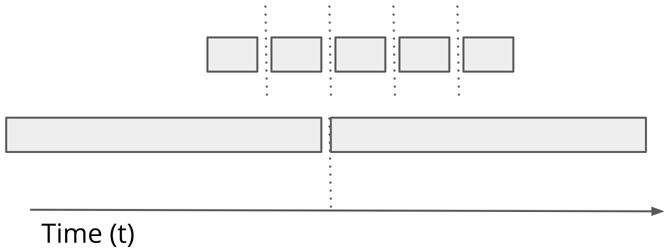
9000 MTU is common inside data centers, where as 1500 common on Internet, why?

https://metalenborden.nl/index.php/webwinkel/postcards-retro/v-amerika/barack-obama-yes-we-can-postcard-retro-metaal-detail

So let's use Jumbo frames everywhere

Advantages:
+ Good throughput
+ Good efficiency

But,
- Needs support from the NIC
- Needs support from the Ethernet switch
- Needs support from the routers
- Can induce delays and multiplexing issues



Inside a data center, we use 9K MTU
Outside, on the Internet??

https://metalenborden.nl/index.php/webwinkel/postcards-retro/v-amerika/barack-obama-yes-we-can-postcard-retro-metaal-detail

So let's use Jumbo frames everywhere

Advantages:

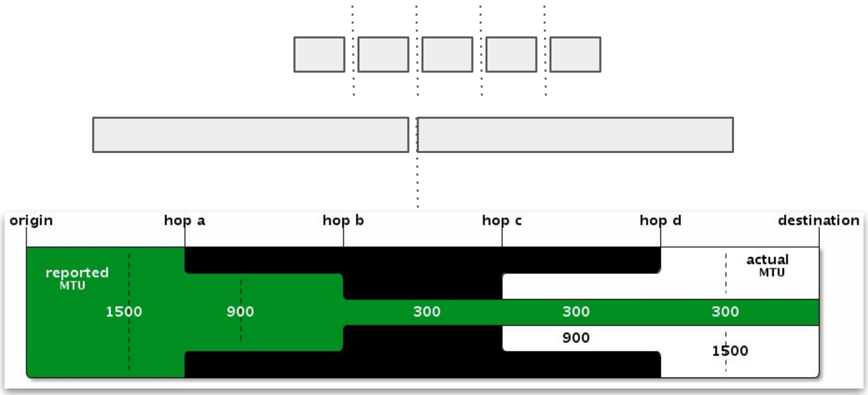
- \+ Good throughput
- \+ Good efficiency

But,

- \- Needs support from the NIC
- \- Needs support from the Ethernet switch
- \- Needs support from the routers
- \- Can induce delays and multiplexing issues

Inside a data center, we use 9K MTU
Outside, on the Internet??

- Path MTU discovery (PMTUD) protocols
  - `ping -s ??? -c 1 -M do 172.217.20.78`
  - `traceroute --mtu 172.217.20.78`



https://elifulkerson.com/projects/mturoute.php

https://metalenborden.nl/index.php/webwinkel/postcards-retro/v-amerika/barack-obama-yes-we-can-postcard-retro-metaal-detail

Linux Tools - ifconfig

```
atr@atr-XPS-13:~$ ifconfig
enx9cebe8cd8f11: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 9c:eb:e8:cd:8f:11  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3571  bytes 56680
        RX errors 0  dropped 0  over
        TX packets 3571  bytes 56680
        TX errors 0  dropped 0 overr
```

```
atr@atr-XPS-13:~$ sudo ifconfig lo mtu 8192
[sudo] password for atr:
atr@atr-XPS-13:~$ ifconfig lo
lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 8192
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3571  bytes 566808 (566.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3571  bytes 566808 (566.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Linux Tools - MTU shenanigans

I changed the local MTU to 2000 bytes

```
atr@atr-XPS-13:~$ sudo tcpdump -i wlp2s0 icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes
17:27:59.872602 IP atr-XPS-13 > ams15s33-in-f14.1e100.net: ICMP echo request, id 15279, seq 1, length 1468
17:27:59.878737 IP ams15s33-in-f14.1e100.net > atr-XPS-13: ICMP echo reply, id 15279, seq 1, length 76
17:29:07.517540 IP atr-XPS-13 > ams15s33-in-f14.1e100.net: ICMP echo request, id 15337, seq 1, length 1508
17:29:27.921086 IP atr-XPS-13 > ams15s33-in-f14.1e100.net: ICMP echo request, id 15352, seq 1, length 1808
^C
```

```
atr@atr-XPS-13:~$ ping -s 2000 -c 1 -M do 172.217.20.78
PING 172.217.20.78 (172.217.20.78) 2000(2028) bytes of data.
ping: local error: Message too long, mtu=2000

--- 172.217.20.78 ping statistics ---
1 packets transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

atr@atr-XPS-13:~$
```

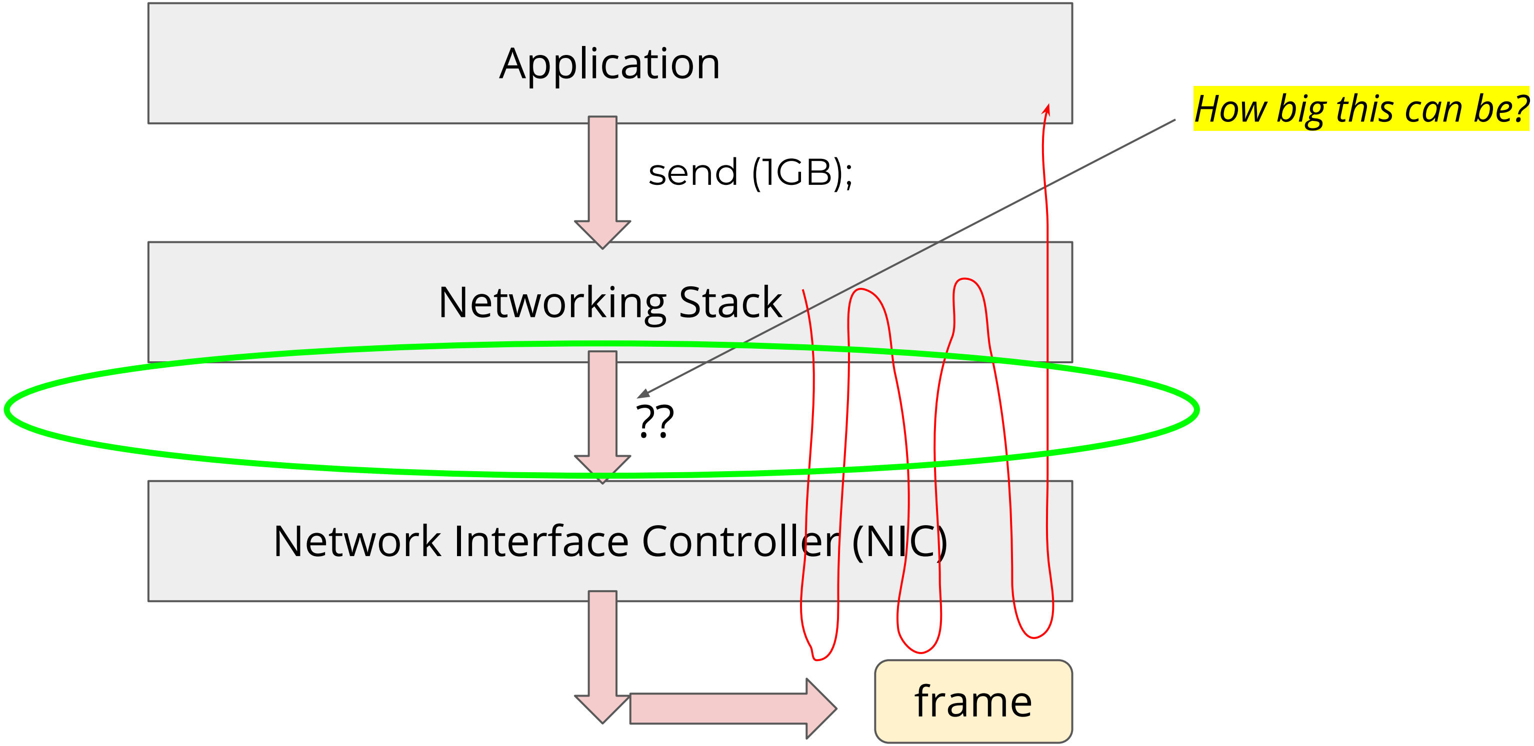
```
atr@atr-XPS-13:~$ ping -s 1800 -c 1 -M do 172.217.20.78
PING 172.217.20.78 (172.217.20.78) 1800(1828) bytes of data.
^C
--- 172.217.20.78 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

atr@atr-XPS-13:~$ ping -s 1460 -c 1 -M do 172.217.20.78
PING 172.217.20.78 (172.217.20.78) 1460(1488) bytes of data.
76 bytes from 172.217.20.78: icmp_seq=1 ttl=119 (truncated)

--- 172.217.20.78 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.174/6.174/6.174/0.000 ms
atr@atr-XPS-13:~$
```

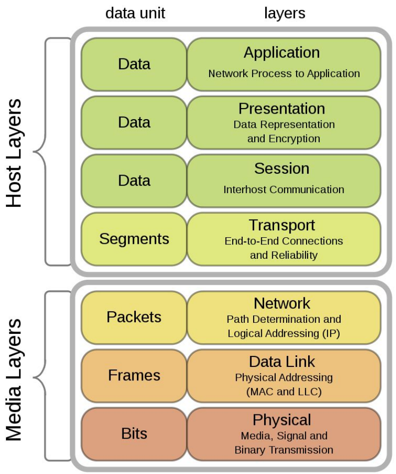
Poing being - 1500 MTU (or 1468B) is the most popular and common type of MTU supported on the internet

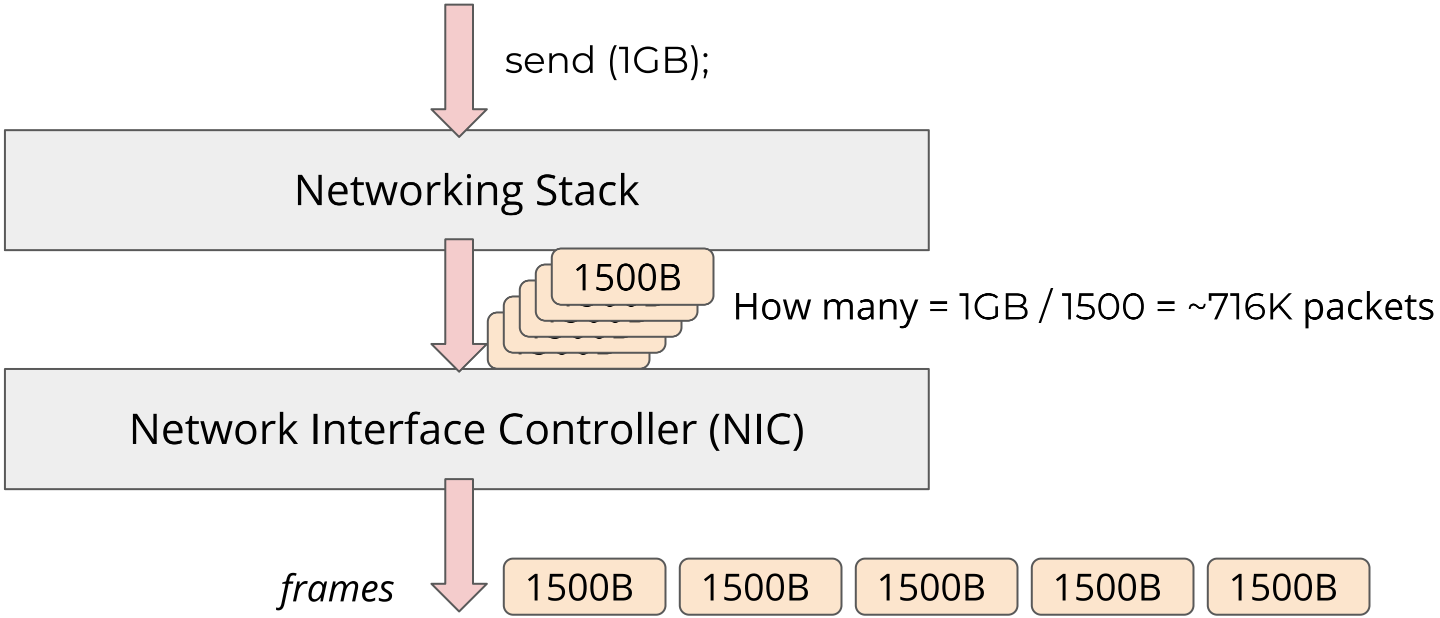
The Unit of Processing



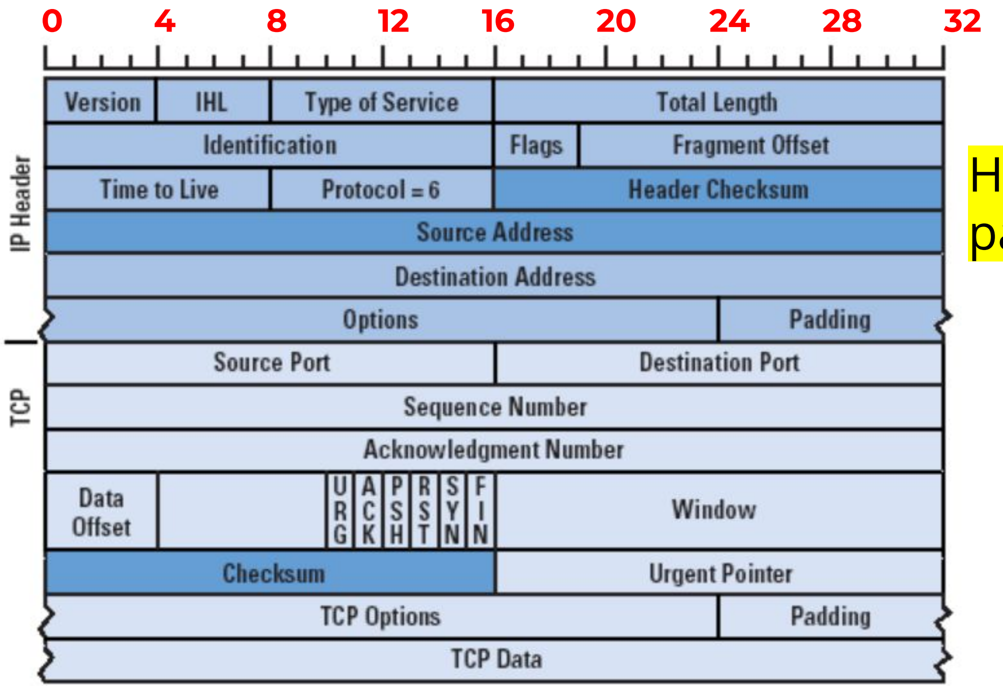
A Simple Solution

Just keep it as the MTU size, and create MTU size ***segments*** ***(L4)*** that become the MTU size ***packets*** (***L3***) and ***frames*** ***(L2):*** so essentially 1500 bytes





But for a Moment, consider the TCP IP Packet



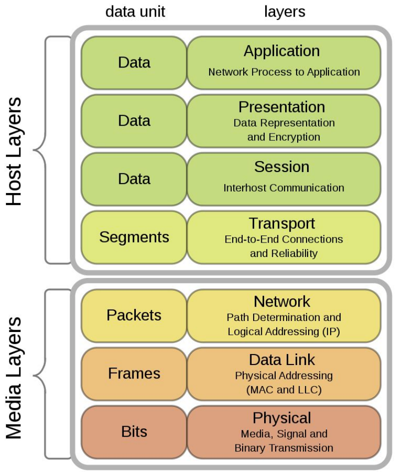
How big a TCP/IP packet can be?

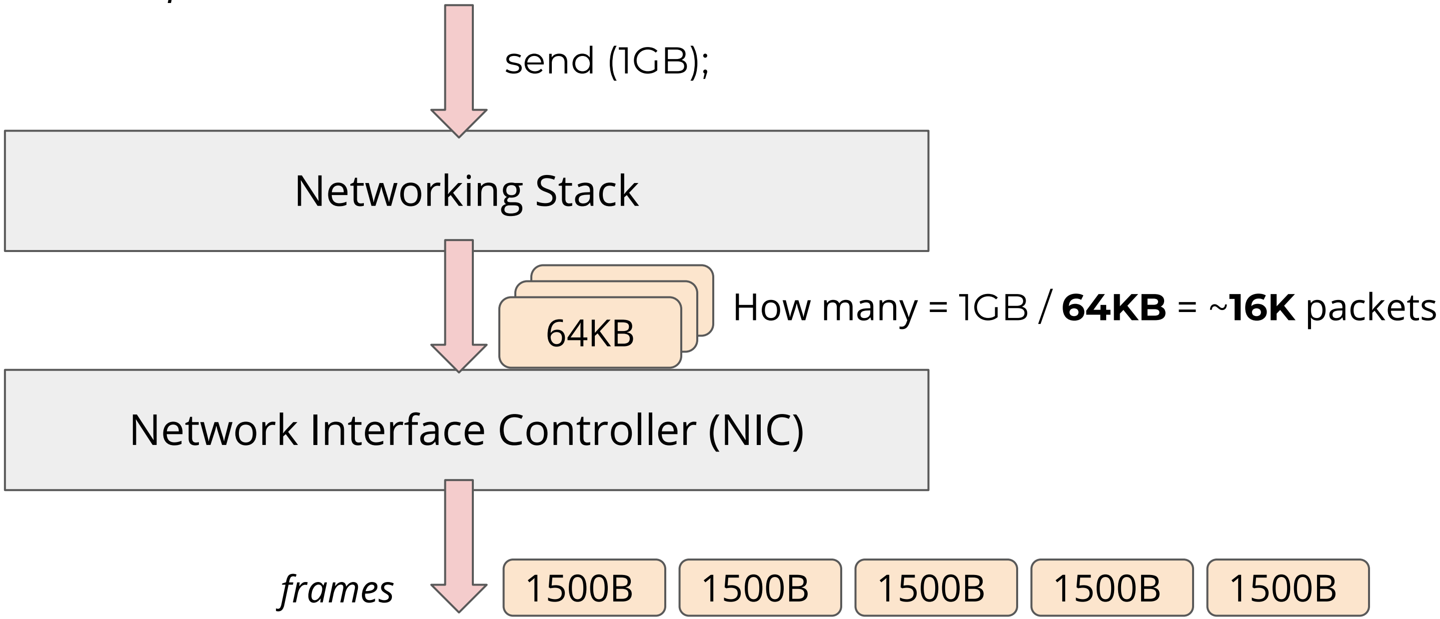
https://erlerobotics.gitbooks.io/erle-robotics-introduction-to-linux-networking/content/introduction_to_network/img/packet.gif

Large Packets

Build large TCP/IP *__packets__* (***L3***), so essentially 64kB (more efficient, less I/O programming)

But we still can not sent 64kB packets with 1500 MTU?





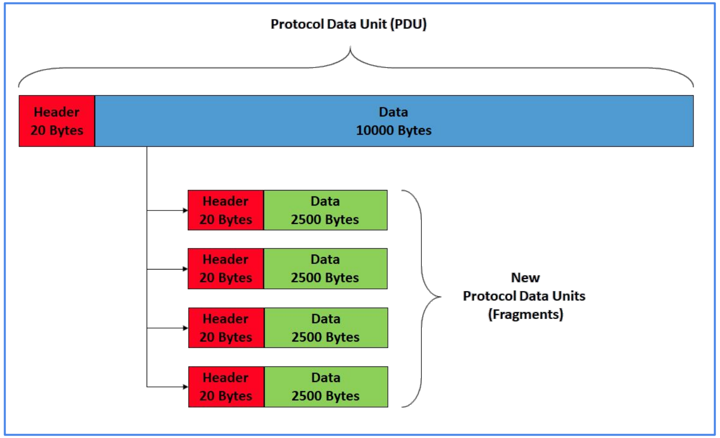
How do we cut / segment this packet?

Packet segmentation (network layer) into smaller link layer frames (e.g., 1500 Bytes on Ethernet)

Is it a difficult job?

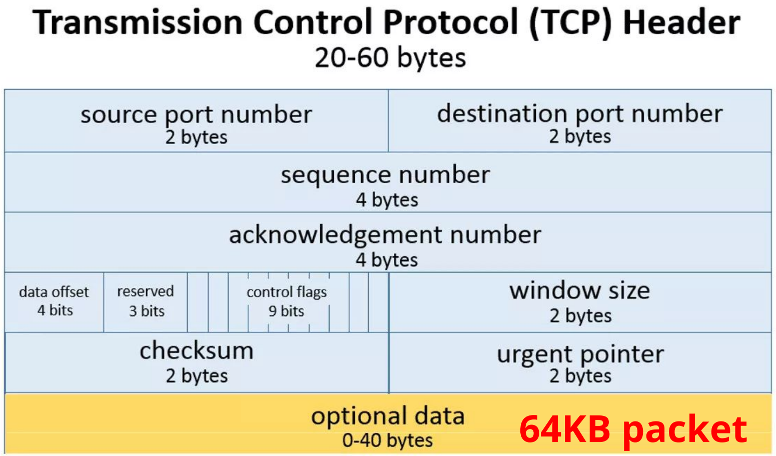
- IP already has “fragmentation” support
    - Flags, and fragment offset in the header
    - All routers and switches support it
    - IP packet can be (de)assembled in hw/sw at end host (keep track of state)
    - See, https://tools.ietf.org/html/rfc815

What about TCP?



len=2500	fragflag = 1	fragoffset = 0
len=2500	fragflag = 1	fragoffset = 2500
len=2500	fragflag = 0	fragoffset = 7500

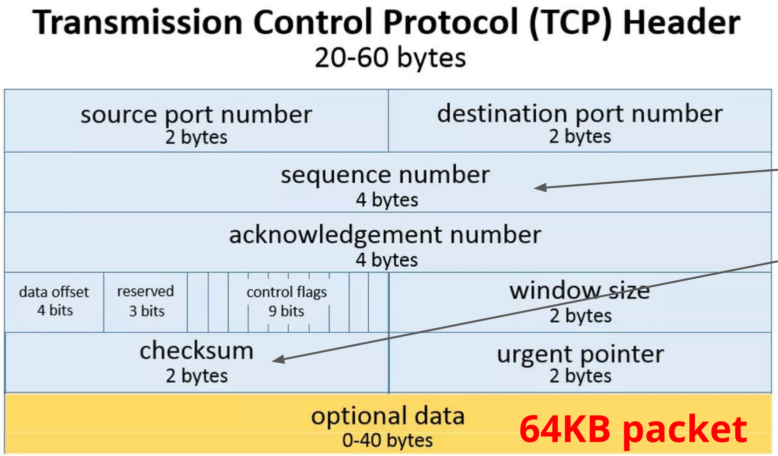
TCP Packet Segmentation



What are the fields that will change if a large TCP segment is cut into multiple packets?

https://www.lifewire.com/tcp-headers-and-udp-headers-explained-817970

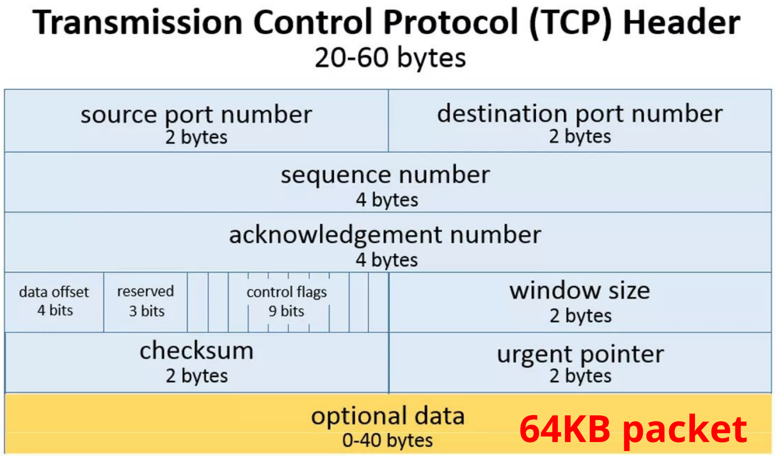
TCP Packet Segmentation



What are the fields that will change if a large TCP segment is cut into multiple packets?

https://www.lifewire.com/tcp-headers-and-udp-headers-explained-817970

TCP Packet Segmentation



How does the sequence number will change?

= 100 (sequence number)

1. TCP packet segmentation

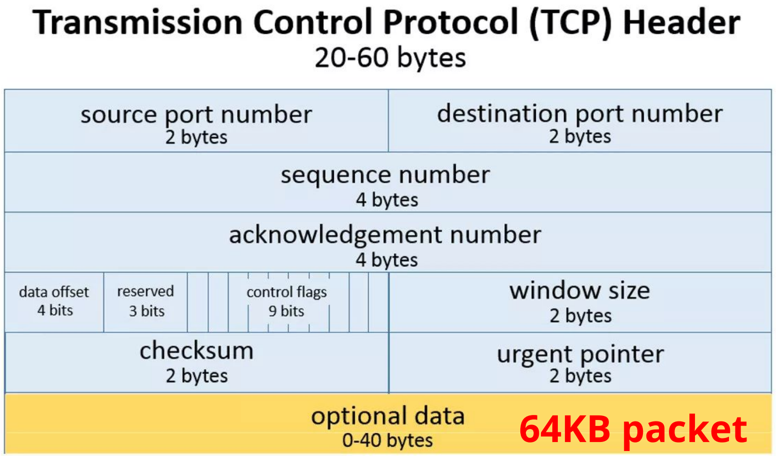
SEQ=100 | SEQ=1600 | SEQ=3100 | SEQ=64136

1500 MTU

2. Redo checksum calculations

https://www.lifewire.com/tcp-headers-and-udp-headers-explained-817970

TCP Packet Segmentation



How does the sequence number will change?

= 100 (sequence number)

1. TCP packet segmentation

2. Redo checksum calculations

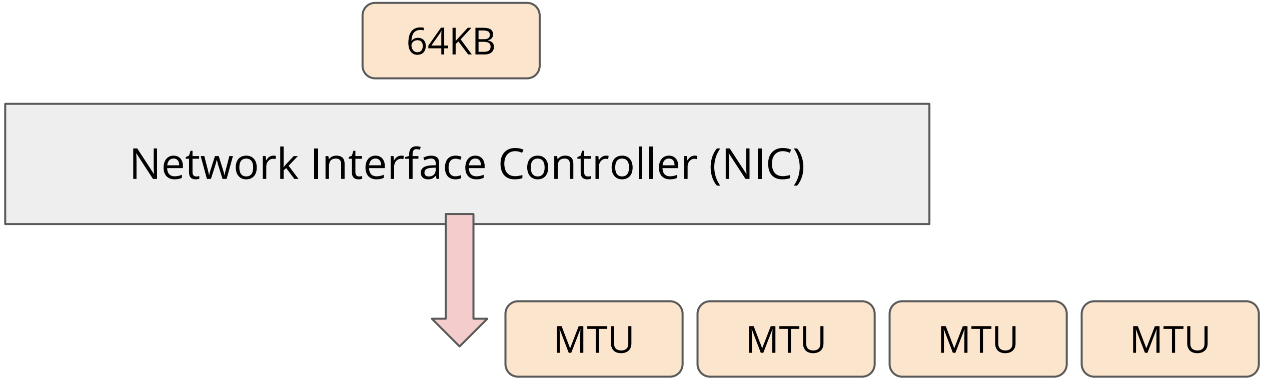
SEQ=100 | SEQ=1600 | SEQ=3100 | SEQ=64136

1500 MTU

Why can we do this: TCP is a byte-stream protocol

https://www.lifewire.com/tcp-headers-and-udp-headers-explained-817970

Who does TCP packet segmentation



- Either in the software, in the NIC device driver
- Or in the hardware, in the NIC device

When moving away work from the CPU to devices (here, the NIC) - it is called ***Offloading*** (reverser is called ***Onloading***)

This particular process is called : TCP Segmentation Offloading or TSO

Linux Tool: ethtool -k

```
atr@evelyn:~$ ethtool -k enp0s25
Features for enp0s25:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: off [fixed]
        tx-checksum-ip-generic: on
        tx-checksum-ipv6: off [fixed]
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp-mangleid-segmentation: off
        tx-tcp6-segmentation: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off [fixed]
rx-vlan-offload: on
tx-vlan-offload: on
```

LRO (Large Receive Offload)

There are different places you can do aggregation

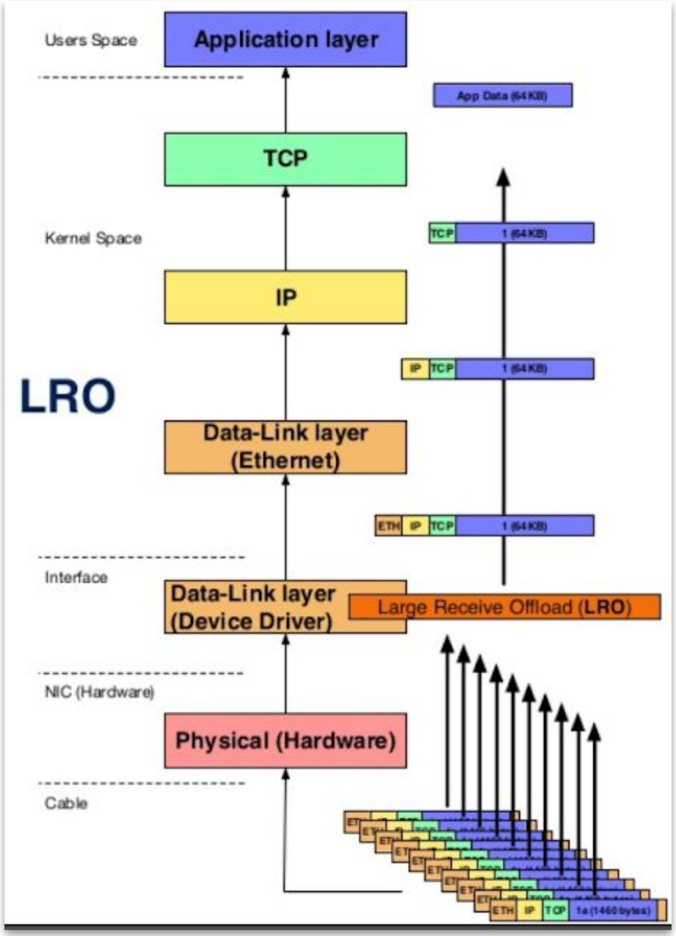
In the device driver (pure software, no hardware support needed)

LRO is TCP/IPv4 specific and quite lenient in merging packets (issues in bridging and/or forwarding setups)

Generic Receive Offload (**GRO**) is more restrictive and supports multiple protocols (is the preferred way of doing packet merging)

But the high-level concept remains the same

https://lwn.net/Articles/358910/
https://sv9rxw.blogspot.com/2020/04/modern-high-speed-networking-techniques.html



Now that we are Adding Further Logic on the NIC

So far we have seen that a NIC can

- transmit and receive link layer packets
- supports doing DMA
- supports doing scatter-gather DMA operations

We can also offload (move from the CPU to the NIC)

- (now) doing TCP segmentation and generate MTU sized packets
- (now) generating checksum
- (now) LRO and GRO

What is next?

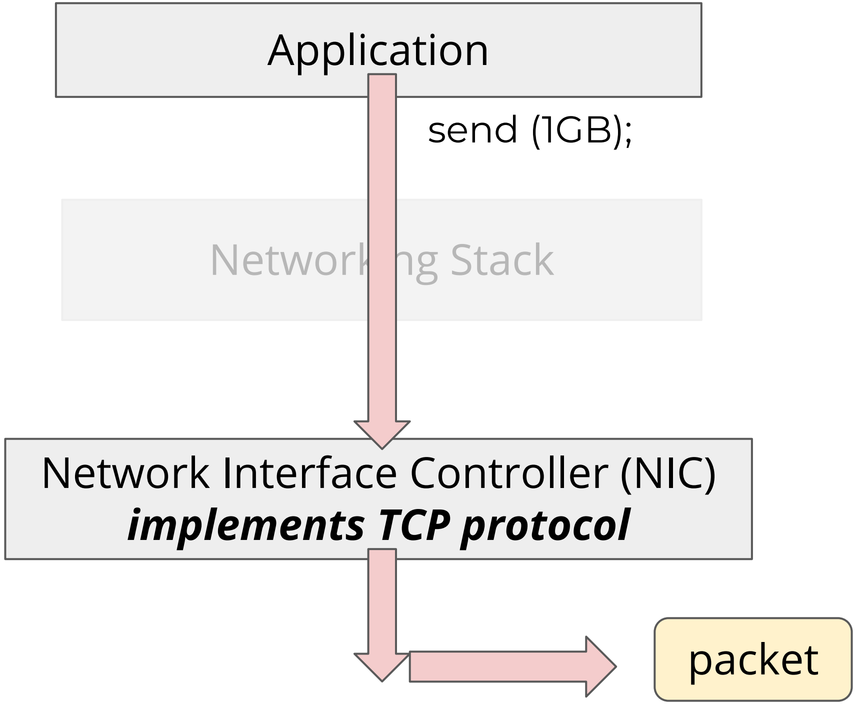
https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html
https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html

Pushing to the Extreme: TCP Offload

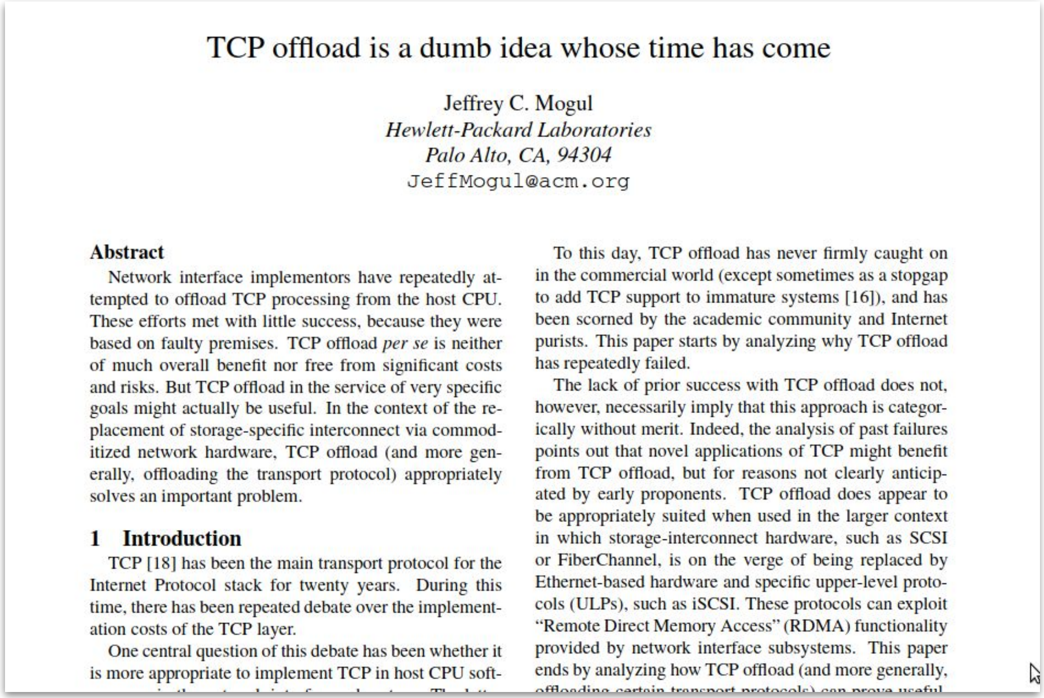
Why not push to the extreme and put everything in the NIC

Yes - It is called **TCP offloading**

*What do you think, is it a **GOOD** idea?*
*Who thinks it is a **BAD** idea?*



TCP offload is a dumb idea whose time has come (2003)



TCP offload is a dumb idea whose time has come

Jeffrey C. Mogul
Hewlett-Packard Laboratories
Palo Alto, CA, 94304
JeffMogul@acm.org

Abstract

Network interface implementors have repeatedly attempted to offload TCP processing from the host CPU. These efforts met with little success, because they were based on faulty premises. TCP offload *per se* is neither of much overall benefit nor free from significant costs and risks. But TCP offload in the service of very specific goals might actually be useful. In the context of the replacement of storage-specific interconnect via commoditized network hardware, TCP offload (and more generally, offloading the transport protocol) appropriately solves an important problem.

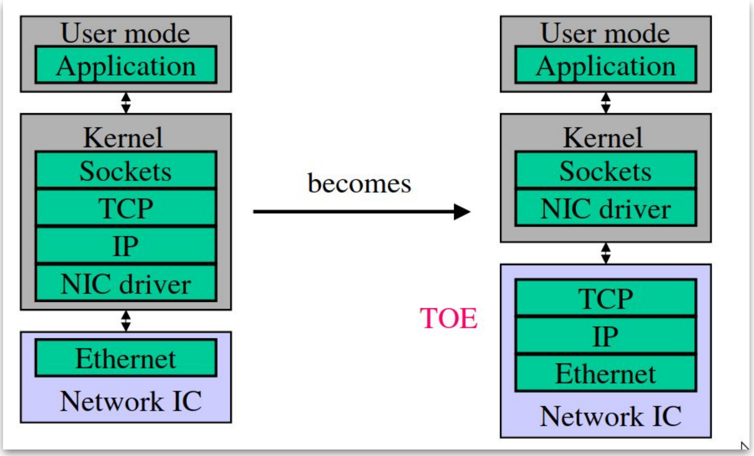
1 Introduction

TCP [18] has been the main transport protocol for the Internet Protocol stack for twenty years. During this time, there has been repeated debate over the implementation costs of the TCP layer.

One central question of this debate has been whether it is more appropriate to implement TCP in host CPU soft-

To this day, TCP offload has never firmly caught on in the commercial world (except sometimes as a stopgap to add TCP support to immature systems [16]), and has been scorned by the academic community and Internet purists. This paper starts by analyzing why TCP offload has repeatedly failed.

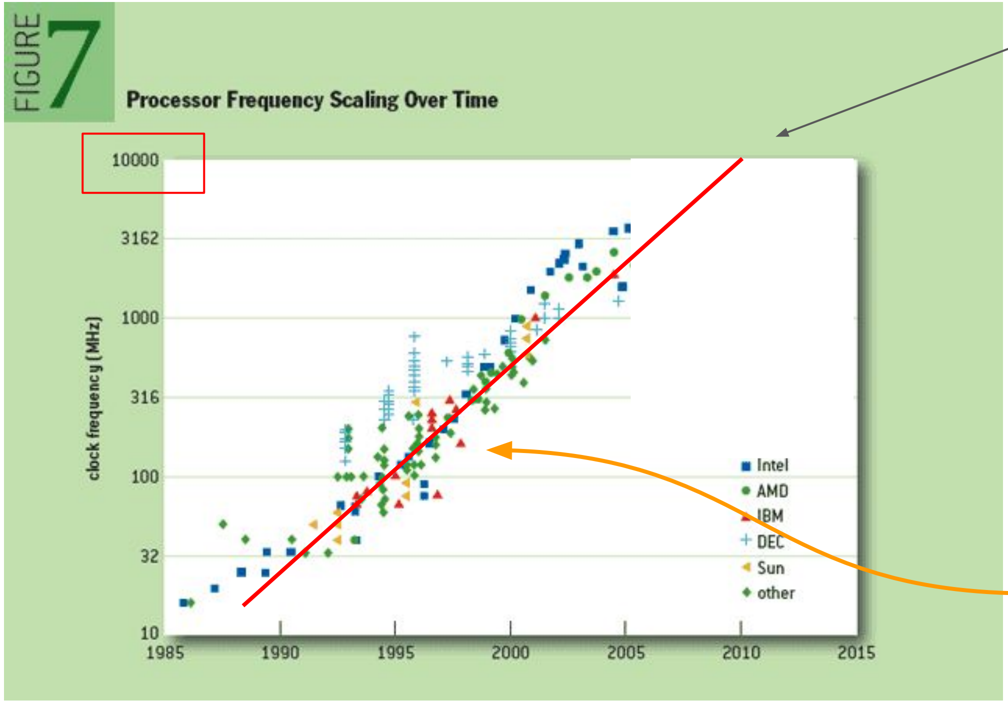
The lack of prior success with TCP offload does not, however, necessarily imply that this approach is categorically without merit. Indeed, the analysis of past failures points out that novel applications of TCP might benefit from TCP offload, but for reasons not clearly anticipated by early proponents. TCP offload does appear to be appropriately suited when used in the larger context in which storage-interconnect hardware, such as SCSI or FiberChannel, is on the verge of being replaced by Ethernet-based hardware and specific upper-level protocols (ULPs), such as iSCSI. These protocols can exploit "Remote Direct Memory Access" (RDMA) functionality provided by network interface subsystems. This paper ends by analyzing how TCP offload (and more generally,



https://www.usenix.org/legacy/events/hotos03/tech/full_papers/mogul/mogul.pdf

The year is 2003

Well on the way for a 10 GHz CPU
(we know how that went)



Ethernet was jumping from

100 Mbps → 1 Gbps → 10 Gbps (late 2010s)

History of computing is littered with failed "advanced NIC" project who failed to take off in this period.

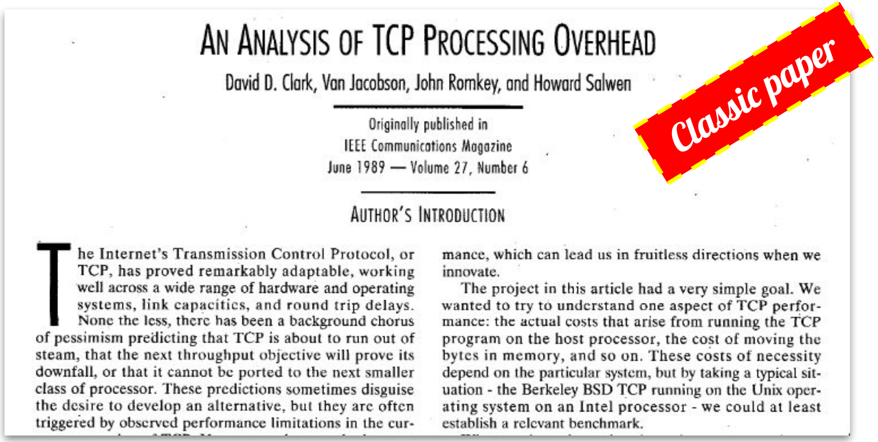
CPU DB: Recording Microprocessor History, https://queue.acm.org/detail.cfm?id=2181798 (really cool read!)
https://www.deviantart.com/darhymes/art/Back-to-the-Future-Icon-276270476

So why was TCP Offload was a dumb idea?

Back in 2003

- Historically it has been shown that TCP “protocol” processing is cheap
    - Means: TCP header processing (only!)
    - But the *devil lives in the socket abstraction* ;)

An Analysis of TCP Processing Overheads (1989)



AN ANALYSIS OF TCP PROCESSING OVERHEAD

David D. Clark, Van Jacobson, John Romkey, and Howard Salwen

Originally published in
IEEE Communications Magazine
June 1989 — Volume 27, Number 6

AUTHOR'S INTRODUCTION

The Internet's Transmission Control Protocol, or TCP, has proved remarkably adaptable, working well across a wide range of hardware and operating systems, link capacities, and round trip delays. None the less, there has been a background chorus of pessimism predicting that TCP is about to run out of steam, that the next throughput objective will prove its downfall, or that it cannot be ported to the next smaller class of processor. These predictions sometimes disguise the desire to develop an alternative, but they are often triggered by observed performance limitations in the cur- mance, which can lead us in fruitless directions when we innovate.

The project in this article had a very simple goal. We wanted to try to understand one aspect of TCP performance: the actual costs that arise from running the TCP program on the host processor, the cost of moving the bytes in memory, and so on. These costs of necessity depend on the particular system, but by taking a typical situation - the Berkeley BSD TCP running on the Unix operating system on an Intel processor - we could at least establish a relevant benchmark.

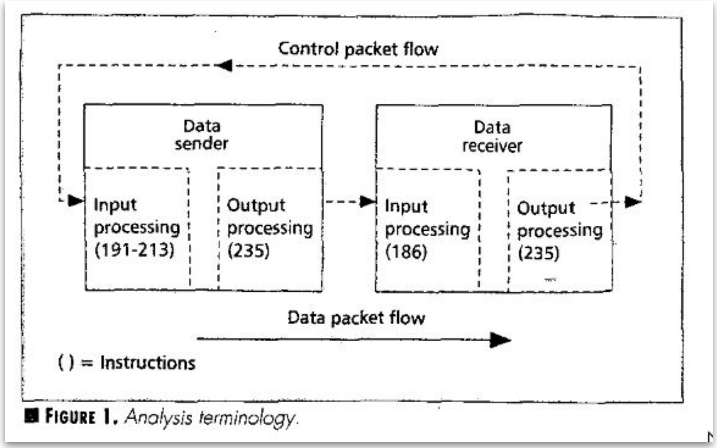


FIGURE 1. Analysis terminology.

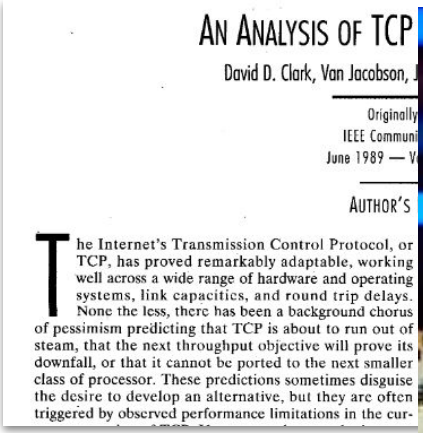
Their findings in BSD

- "*What we showed was that the code necessary to implement TCP* ***was not the major limitation*** *to overall performance. In fact, in this tested system (and many other systems subsequently evaluated by others) the throughput is close to being limited by* ***the memory bandwidth*** *of the system. We are hitting a fundamental limit, not an artifact of poor design. Practically, other* ***parts of the OS had larger overheads than TCP****.* "

- Buffer management, process coordination, signalling, interrupts → none of them will improve with a TOE

https://groups.csail.mit.edu/ana/Publications/PubPDFs/An%20Analysis%20of%20TCP%20Processing%20Overhead.pdf

An Analysis of TCP Processing Overheads (1989)

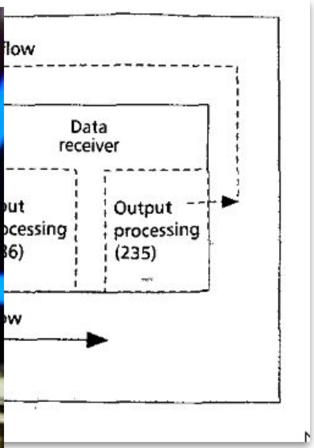

AN ANALYSIS OF TCP

David D. Clark, Van Jacobson, J

Originally
IEEE Communi
June 1989 — V

AUTHOR'S

The Internet's Transmission Control Protocol, or TCP, has proved remarkably adaptable, working well across a wide range of hardware and operating systems, link capacities, and round trip delays. None the less, there has been a background chorus of pessimism predicting that TCP is about to run out of steam, that the next throughput objective will prove its downfall, or that it cannot be ported to the next smaller class of processor. These predictions sometimes disguise the desire to develop an alternative, but they are often triggered by observed performance limitations in the cur-



Their findings in BSD

- *“What we showed was* ... *to overall performance. In fact, i* ... *thers) the throughput is close to* ... *undamental limit, not an artifact of poor design. Practically, other* ***parts of the OS had larger overheads than TCP****.* “
- Buffer management, process coordination, signalling, interrupts → none of them will improve with a TOE

https://groups.csail.mit.edu/ana/Publications/PubPDFs/An%20Analysis%20of%20TCP%20Processing%20Overhead.pdf

So why was TCP Offload was a dumb idea?

Back in 2003

- Historically it has been shown that TCP “protocol” processing is cheap
  - Means: TCP header processing (only!)
  - But the *devil lives in the socket abstraction* ;)
- Moore’s law was working against making intelligent NICs
  - Anything that takes more than 18 months - CPU power will over take it
- What is the TOE (TCP offload engine) interface to the system? Interrupts, polls? How does a TOE reads socket data from application? Does it have enough memory to hold enough packets? What was the main bottleneck TOE was trying optimize?
- Most of the previously discussed techniques: TSO, LRO, checksum offloading, etc. are very effective

Practically speaking

(we will see later as well again)

- Any one who has programmed a hardware/microcontroller - it is pure pain
  - It cannot be better than what you have programming a general purpose CPU
- Quality assurance takes time, for 100s of different combinations
  - As you will see in the ANP code: networking does not work in isolation
- If there is a bug - who should you contact? Linus Torvalds? (ahem, good luck!, anyone nVIDIA fiasco?), NIC hardware manufacturer, or device driver writer
- Limited market - only specific site deployments (data centers were just starting). So, no commodity market at scale

There is a ideological war (still on!)

Content
- Weekly Edition
- Archives
- Search
- Kernel

User: **Password:** Log in | Subscribe | Register

Linux and TCP offload engines

[Posted August 22, 2005 by corbet]

The TCP/IP protocol suite takes a certain amount of CPU power to implement. So it is not surprising that network adapter manufacturers have long been adding protocol support to their cards. This support can vary from the simple (checksumming of packets, for example) through to full TCP/IP implementations. An adapter with full protocol support is often called a TCP offload engine or TOE.

Will it find its way in? Not if David Miller has anything to say on the matter:

> I am still very much against TOE going into the Linux networking stack. There are ways to obtain TOE's performance without necessitating stateful support in the cards, everything that's worthwhile can be done with stateless offloads.

There is essentially zero chance of a networking patch being merged over David's objections, so the TOE developers have an uphill road ahead of them.

https://lwn.net/Articles/148697/

Companies like Google are taking a different step

What is Stateless and Stateful Offloading

Stateless Offloading - there is no (or limited) state that a processing element needs to remember, each packet can be processed independently (self contained)

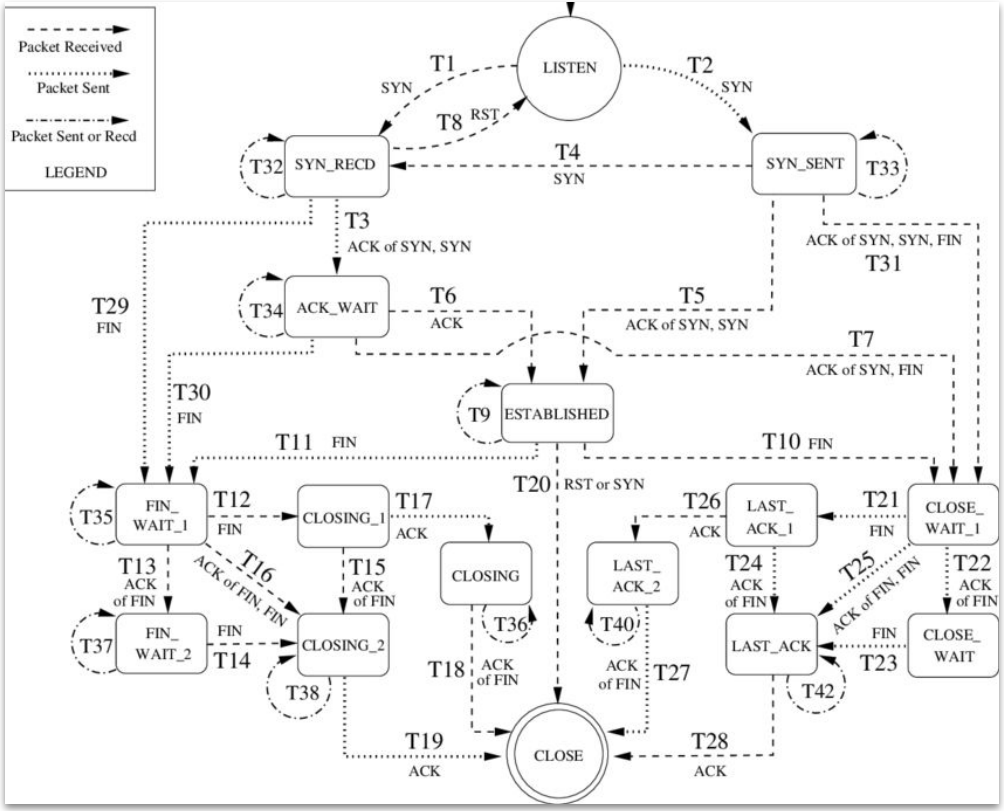
- Checksum offloading, TSO offloading (LRO, GRO offloading)
- Stateless offloading in hardware (or in driver software also possible)
- Often is a performance optimization, than a correctness issue

Stateful Offloading: What you do with the current packet depends upon some state that needs to be maintained. For example, TCP offloading means maintaining the TCP state machine in the hardware ...

- Correctness issue

And in case you have forgotten what the TCP state machine is ...
(you will need it for your project)

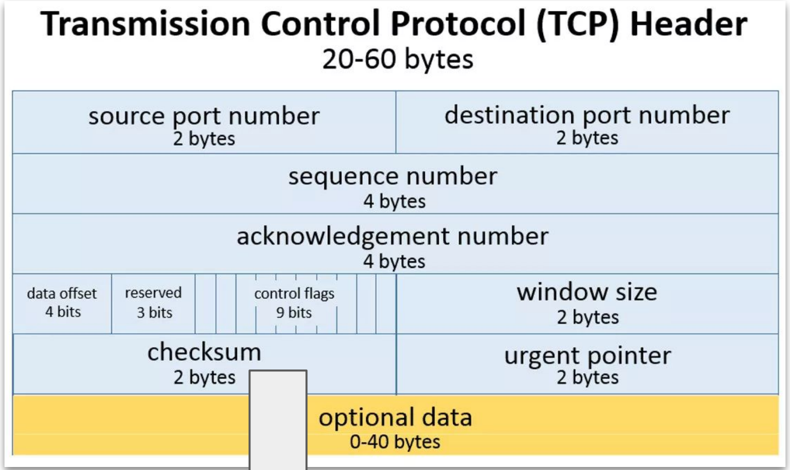
TCP state machine



Hence, what you do with an incoming or outgoing packet depends a lot on what TCP state machine you are in - hence, a stateful packet processing

https://www.researchgate.net/figure/TCP-Protocol-State-Machine_fig1_221609864

A closer look : Stateless offloading



TSO

A given TCP packet can be segmented from a given initial SEQ number (which is already in the packet)

Checksum offloading

Checksum can be generated independently for each packet. No further information needed.

These two are not the only examples of stateless offloading. Linux support many protocols and associated offloading mechanisms - but strongly all “stateless” (because they refuse to let anything else in)

Linux Tool: ethtool -k

What features are supported depends on the Linux kernel version and NIC capabilities

```
ETHTOOL(8)                                   System Manager's Manual

NAME
       ethtool - query or control network driver and hardware settings

SYNOPSIS
       ethtool devname

       ethtool -h|--help

       ethtool --version

       ethtool -a|--show-pause devname

       ethtool -A|--pause devname [autoneg on|off] [rx on|off] [tx on|off]

       ethtool -c|--show-coalesce devname

       ethtool -C|--coalesce devname [adaptive-rx on|off] [adaptive-tx on|off] [rx-usecs N] [rx-frames N] [rx-usecs-irq N] [rx-frames-irq N] [tx-usecs N] [tx-fr
              [tx-usecs-irq N] [tx-frames-irq N] [stats-block-usecs N] [pkt-rate-low N] [rx-usecs-low N] [rx-frames-low N] [tx-usecs-low N] [tx-frames-low N] [
              [rx-usecs-high N] [rx-frames-high N] [tx-usecs-high N] [tx-frames-high N] [sample-interval N]

       ethtool -g|--show-ring devname

       ethtool -G|--set-ring devname [rx N] [rx-mini N] [rx-jumbo N] [tx N]

       ethtool -i|--driver devname

       ethtool -d|--register-dump devname [raw on|off] [hex on|off] [file name]

       ethtool -e|--eeprom-dump devname [raw on|off] [offset N] [length N]

       ethtool -E|--change-eeprom devname [magic N] [offset N] [length N] [value N]

       ethtool -k|--show-features|--show-offload devname

       ethtool -K|--features|--offload devname feature on|off ...
```

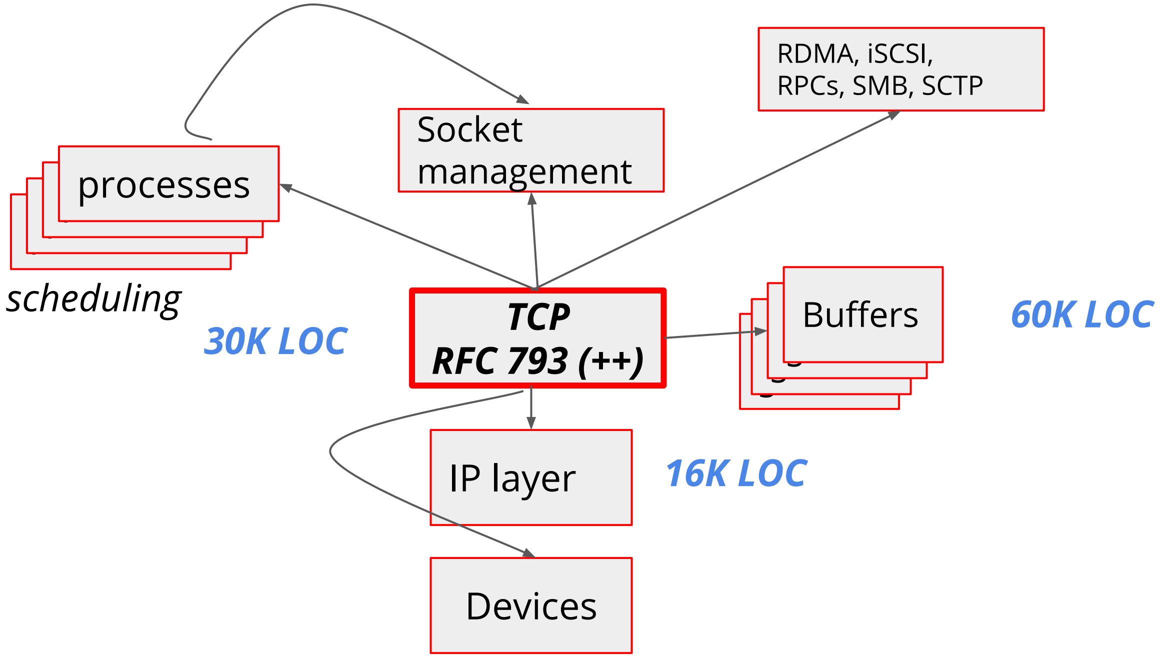
```
atr@evelyn:~$ ethtool -k enp0s25
Features for enp0s25:
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: off [fixed]
        tx-checksum-ip-generic: on
        tx-checksum-ipv6: off [fixed]
        tx-checksum-fcoe-crc: off [fixed]
        tx-checksum-sctp: off [fixed]
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
tcp-segmentation-offload: on
        tx-tcp-segmentation: on
        tx-tcp-ecn-segmentation: off [fixed]
        tx-tcp-mangleid-segmentation: off
        tx-tcp6-segmentation: on
udp-fragmentation-offload: off
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off [fixed]
rx-vlan-offload: on
tx-vlan-offload: on
ntuple-filters: off [fixed]
receive-hashing: on
highdma: on [fixed]
rx-vlan-filter: off [fixed]
vlan-challenged: off [fixed]
tx-lockless: off [fixed]
netns-local: off [fixed]
tx-gso-robust: off [fixed]
tx-fcoe-segmentation: off [fixed]
tx-gre-segmentation: off [fixed]
tx-gre-csum-segmentation: off [fixed]
tx-ipxip4-segmentation: off [fixed]
tx-ipxip6-segmentation: off [fixed]
tx-udp_tnl-segmentation: off [fixed]
tx-udp_tnl-csum-segmentation: off [fixed]
tx-gso-partial: off [fixed]
tx-sctp-segmentation: off [fixed]
tx-esp-segmentation: off [fixed]
fcoe-mtu: off [fixed]
tx-nocache-copy: off
loopback: off [fixed]
rx-fcs: off
rx-all: off
tx-vlan-stag-hw-insert: off [fixed]
rx-vlan-stag-hw-parse: off [fixed]
rx-vlan-stag-filter: off [fixed]
l2-fwd-offload: off [fixed]
hw-tc-offload: off [fixed]
esp-hw-offload: off [fixed]
esp-tx-csum-hw-offload: off [fixed]
rx-udp_tunnel-port-offload: off [fixed]
atr@evelyn:~$
```

Key difference to understand

There is a difference between: (this theme will continue later on)

1. The **TCP protocol** as specified in the RFC 793
2. The BSD **socket implementation** and associated semantics
    a. At no point in time while using socket you need know if you are using TCP

TCP and Sockets (approx. split)



Linux kernel: more than **100,000 lines of code** for networking
(which we are going to cover in the next lecture)

Key difference to understand

There is a difference between: (this theme will continue later on)

1. The **TCP protocol** as specified in the RFC 793
2. The BSD **socket implementation** and associated semantics
    a. At no point in time while using socket you need know if you are using TCP

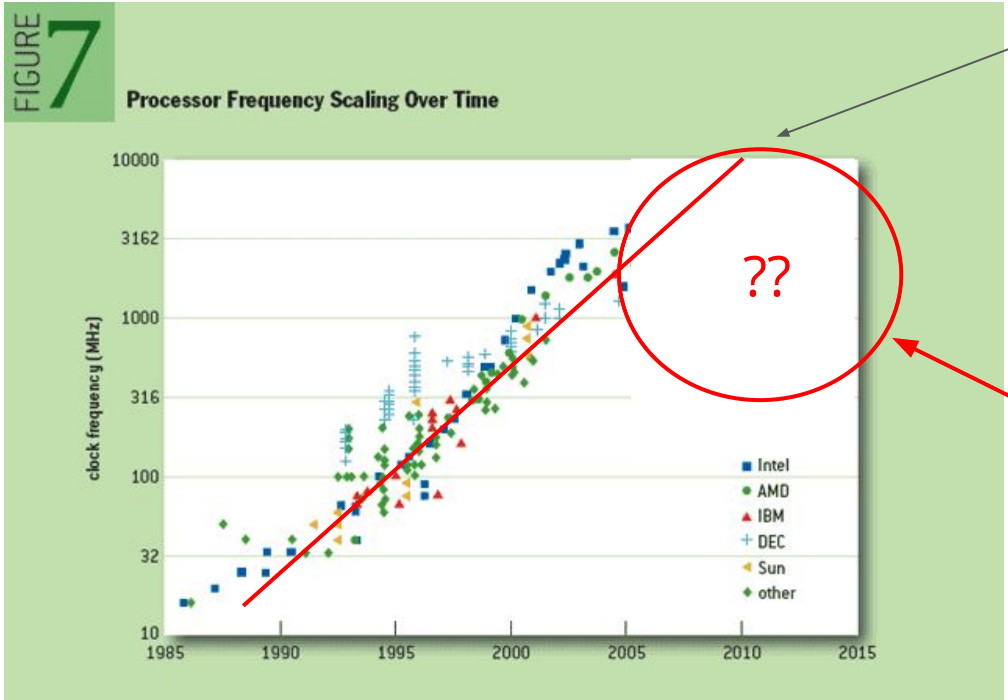
Unfortunately the way currently things are implemented: sockets and TCP semantics are kind of glued together. ***But they don't have to be!***

What Mogul made a case is : *TCP offload "might" be a good idea under certain circumstances with a **different API** than sockets (iSCSI, NFS, MPI, SMB)*

- One such API is RDMA, we will cover at the end of Part 1

The year is 2003

Well on the way for a 10 GHz CPU
(we know how that went)



But then something happened here, and all our dreams of 10 GHz CPU were shattered ;)

CPU DB: Recording Microprocessor History, https://queue.acm.org/detail.cfm?id=2181798 (really cool read!)
https://www.deviantart.com/darhymes/art/Back-to-the-Future-Icon-276270476

Linux Tool: ethtool -S

ethtool -S shows a lot of NIC specific statistics and counters

```
atr@atr-XPS-13:/proc/net$ ethtool -S wlp2s0
NIC statistics:
     rx_packets: 1814031
     rx_bytes: 656139107
     rx_duplicates: 2
     rx_fragments: 1710631
     rx_dropped: 933
     tx_packets: 221975
     tx_bytes: 82353452
     tx_filtered: 0
     tx_retry_failed: 0
     tx_retries: 0
     sta_state: 4
     txrate: 6000000
     rxrate: 234000000
     signal: 198
     channel: 5260
     noise: 160
     ch_time: 149
     ch_time_busy: 5
     ch_time_ext_busy: 18446744073709551615
     ch_time_rx: 18446744073709551615
     ch_time_tx: 18446744073709551615
     tx_pkts_nic: 227554
     tx_bytes_nic: 0
     rx_pkts_nic: 1687653
     rx_bytes_nic: 0
     d_noise_floor: 18446744073709551520
     d_cycle_count: 144895264
```

Linux Tool: netstat

```
NETSTAT(8)                                Linux System Administrator's Manual                                NETSTAT(8)

NAME
       netstat - Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships

SYNOPSIS
       netstat  [address_family_options]  [--tcp|-t]  [--udp|-u] [--udplite|-U] [--sctp|-S] [--raw|-w] [--l2cap|-2] [--rfcomm|-f] [--listening|-l] [--all|-a] [--numeric|-n] [--numeric-hosts] [--numeric-ports] [--numeric-users]
       [--symbolic|-N] [--extend|-e[--extend|-e]] [--timers|-o] [--program|-p] [--verbose|-v] [--continuous|-c] [--wide|-W]

       netstat {--route|-r} [address_family_options] [--extend|-e[--extend|-e]] [--verbose|-v] [--numeric|-n] [--numeric-hosts] [--numeric-ports] [--numeric-users] [--continuous|-c]

       netstat {--interfaces|-i} [--all|-a] [--extend|-e[--extend|-e]] [--verbose|-v] [--program|-p] [--numeric|-n] [--numeric-hosts] [--numeric-ports] [--numeric-users] [--continuous|-c]

       netstat {--groups|-g} [--numeric|-n] [--numeric-hosts] [--numeric-ports] [--numeric-users] [--continuous|-c]

       netstat {--masquerade|-M} [--extend|-e] [--numeric|-n] [--numeric-hosts] [--numeric-ports] [--numeric-users]

       netstat {--statistics|-s} [--tcp|-t] [--udp|-u] [--udplite|-U] [--sctp|-S] [--raw|-w]

       netstat {--version|-V}

       netstat {--help|-h}
```

```
atr@evelyn:~$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 evelyn.home:36464       ec2-3-123-217-208:https ESTABLISHED
tcp        0      0 evelyn.home:34204       whatsapp-cdn-shv-:https ESTABLISHED
tcp        0      0 evelyn.home:58098       ams15s30-in-f3.1e:https ESTABLISHED
tcp        0      0 evelyn.home:60690       lhr26s05-in-f14.1:https ESTABLISHED
tcp        0      0 evelyn.home:41244       ams16s29-in-f42.1:https ESTABLISHED
tcp        0      0 evelyn.home:57354       151.101.37.7:https      ESTABLISHED
tcp        0      0 evelyn.home:36458       ec2-3-123-217-208:https ESTABLISHED
tcp        0      0 evelyn.home:48968       ec2-52-89-164-184:https ESTABLISHED
tcp        0      0 evelyn.home:55238       ams16s29-in-f46.1:https ESTABLISHED
tcp        0      0 evelyn.home:49304       fra02s28-in-f10.1:https ESTABLISHED
tcp        0      0 evelyn.home:53886       108.177.126.189:https   ESTABLISHED
tcp        0      0 evelyn.home:46810       149.154.167.99:https    ESTABLISHED
tcp        0      0 evelyn.home:49090       ams15s32-in-f14.1:https ESTABLISHED
tcp        0      0 evelyn.home:34040       whatsapp-cdn-shv-:https ESTABLISHED
tcp        0    164 evelyn.home:ssh         atr-XPS-13.home:36002   ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ]         DGRAM                    33261    /run/user/1000/systemd/notify
unix  2      [ ]         DGRAM                    17673404 /run/wpa_supplicant/wlp3s0
unix  2      [ ]         DGRAM                    17674353 /run/wpa_supplicant/p2p-dev-wlp3s0
unix  3      [ ]         DGRAM                    782      /run/systemd/notify
unix  2      [ ]         DGRAM                    798      /run/systemd/journal/syslog
unix  22     [ ]         DGRAM                    802      /run/systemd/journal/dev-log
unix  8      [ ]         DGRAM                    809      /run/systemd/journal/socket
unix  3      [ ]         SEQPACKET  CONNECTED     14222800 @00086
```

Linux Tool: tcpdump

Inspection of any arbitrary traffic pattern with any protocol, port, socket, IP, and various other flags...

(tcpdump name is misnomer)

```
atr@atr-XPS-13:~$ sudo tcpdump -i wlp2s0
[sudo] password for atr:
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:47:38.294054 IP ams15s32-in-f14.1e100.net.443 > atr-XPS-13.53455: UDP, length 47
12:47:38.296035 IP atr-XPS-13.53455 > ams15s32-in-f14.1e100.net.443: UDP, length 33
12:47:38.296450 IP atr-XPS-13.37260 > one.one.one.one.domain: 9781+ PTR? 81.1.168.192.in-addr.arpa
12:47:38.304007 IP one.one.one.one.domain > atr-XPS-13.37260: 9781 NXDomain 0/0/0 (43)
12:47:38.305131 IP atr-XPS-13.51536 > one.one.one.one.domain: 62121+ PTR? 110.211.58.216.in-addr.a
```

```
atr@atr-XPS-13:~$ sudo tcpdump -i wlp2s0 tcp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:49:15.452743 IP atr-XPS-13.57962 > ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https: Flags [P.], seq 3526112934:3526112990,
12:49:15.455730 IP atr-XPS-13.58016 > ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https: Flags [P.], seq 1791622675:1791622731,
12:49:15.455968 IP atr-XPS-13.57960 > ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https: Flags [P.], seq 3708215032:3708215088,
12:49:15.456152 IP atr-XPS-13.57964 > ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https: Flags [P.], seq 1113362382:1113362438,
12:49:15.468147 IP ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https > atr-XPS-13.58016: Flags [P.], seq 1:57, ack 56, win 8, op
12:49:15.468232 IP atr-XPS-13.58016 > ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https: Flags [.], ack 57, win 501, options [no
12:49:15.468273 IP ec2-3-123-217-208.eu-central-1.compute.amazonaws.com.https > atr-XPS-13.57962: Flags [P.], seq 1:57, ack 56, win 9, op
```

```
atr@atr-XPS-13:~$ sudo tcpdump -i wlp2s0 'tcp[13] == 2'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:52:21.655038 IP atr-XPS-13.57216 > 142.250.27.105.https: Flags [S], seq 137760916, win 64240, options [mss 1460,sackOK,TS val 822440791 ecr 0,nop,wscale 7], length 0
12:52:22.059653 IP atr-XPS-13.53632 > 142.250.27.106.https: Flags [S], seq 2513723535, win 64240, options [mss 1460,sackOK,TS val 2073482195 ecr 0,nop,wscale 7], length 0
12:52:22.215793 IP atr-XPS-13.58568 > ams16s30-in-f14.1e100.net.https: Flags [S], seq 379245020, win 64240, options [mss 1460,sackOK,TS val 617959276 ecr 0,nop,wscale 7], length 0
```

https://linux.die.net/man/8/tcpdump

Also check out “netcat”
(to generate traffic)

Linux Tool: tcpdump

Inspection of any arbitrary traffic pattern with any protocol, port, socket, IP, and various other flags...

(tcpdump name is misnomer)

https://linux.die.net/man/8/tcpdump

Also check out "netcat"
(to generate traffic)

```
atr@atr:~/home/atr/$ sudo tcpdump -i wlp2s0 port 44441
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp2s0, link-type EN10MB (Ethernet), capture size 262144 bytes

14:03:36.476567 IP 192.168.1.161.41730 > atr-XPS-13.44441: Flags [S], seq 2400912466, win 29200, options
[mss 1460,sackOK,TS val 2019536369 ecr 0,nop,wscale 7], length 0

14:03:36.476648 IP atr-XPS-13.44441 > 192.168.1.161.41730: Flags [S.], seq 330668899, ack 2400912467, win
65160, options [mss 1460,sackOK,TS val 2799286267 ecr 2019536369,nop,wscale 7], length 0

14:03:36.477271 IP 192.168.1.161.41730 > atr-XPS-13.44441: Flags [.], ack 1, win 229, options [nop,nop,TS
val 2019536370 ecr 2799286267], length 0
...
```

Linux Tool: the /proc **file system**

Recall the UNIX philosophy: ***Everything is a file***

```
atr@atr-XPS-13:/proc/net$ ls
anycast6   dev_mcast    icmp      ip6_flowlabel       ip6_tables_targets  ip_tables_targets  netfilter  psched  route      snmp          stat  udplite
arp        dev_snmp6    icmp6     ip6_mr_cache        ip_mr_cache         ipv6_route         netlink    ptype   rt6_stats  snmp6         tcp   udplite6
bnep       fib_trie     if_inet6  ip6_mr_vif          ip_mr_vif           l2cap              netstat    raw     rt_acct    sockstat      tcp6  unix
connector  fib_triestat igmp      ip6_tables_matches  ip_tables_matches   mcfilter           packet     raw6    rt_cache   sockstat6     udp   wireless
dev        hci          igmp6     ip6_tables_names    ip_tables_names     mcfilter6          protocols  rfcomm  sco        softnet_stat  udp6  xfrm_stat
atr@atr-XPS-13:/proc/net$ cat dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls carrier compressed
enx9cebe8cd8f11:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
wlp2s0: 6613994070 7948199    0    7    0     0          0         0 2723641552 4725535    0    0    0     0       0          0
vboxnet0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
    lo:  585009    3835    0    0    0     0          0         0   585009    3835    0    0    0     0       0          0
virbr0-nic:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
virbr0:       0       0    0    0    0     0          0         0        0       0    0    0    0     0       0          0
atr@atr-XPS-13:/proc/net$ cat netstat
TcpExt: SyncookiesSent SyncookiesRecv SyncookiesFailed EmbryonicRsts PruneCalled RcvPruned OfoPruned OutOfWindowIcmps LockDroppedIcmps ArpFilter TW TWRecycled TWKilled PAWSActive PAWSEs
tab DelayedACKs DelayedACKLocked DelayedACKLost ListenOverflows ListenDrops TCPHPHits TCPPureAcks TCPHPAcks TCPRenoRecovery TCPSackRecovery TCPSACKReneging TCPSACKReorder TCPRenoReorder
 TCPTSReorder TCPFullUndo TCPPartialUndo TCPDSACKUndo TCPLossUndo TCPLostRetransmit TCPRenoFailures TCPSackFailures TCPLossFailures TCPFastRetrans TCPSlowStartRetrans TCPTimeouts TCPLos
sProbes TCPLossProbeRecovery TCPRenoRecoveryFail TCPSackRecoveryFail TCPRcvCollapsed TCPDSACKOldSent TCPDSACKOfoSent TCPDSACKRecv TCPDSACKOfoRecv TCPAbortOnData TCPAbortOnClose TCPAbort
OnMemory TCPAbortOnTimeout TCPAbortOnLinger TCPAbortFailed TCPMemoryPressures TCPMemoryPressuresChrono TCPSACKDiscard TCPDSACKIgnoredOld TCPDSACKIgnoredNoUndo TCPSpuriousRTOs TCPMD5NotF
ound TCPMD5Unexpected TCPMD5Failure TCPSackShifted TCPSackMerged TCPSackShiftFallback TCPBacklogDrop PFMemallocDrop TCPMinTTLDrop TCPDeferAcceptDrop IPReversePathFilter TCPTimeWaitOverf
low TCPReqQFullDoCookies TCPReqQFullDrop TCPRetransFail TCPRcvCoalesce TCPOFOQueue TCPOFODrop TCPOFOMerge TCPChallengeACK TCPSYNChallenge TCPFastOpenActive TCPFastOpenActiveFail TCPFast
OpenPassive TCPFastOpenPassiveFail TCPFastOpenListenOverflow TCPFastOpenCookieReqd TCPFastOpenBlackhole TCPSpuriousRtxHostQueues BusyPollRxPackets TCPAutoCorking TCPFromZeroWindowAdv TC
PToZeroWindowAdv TCPWantZeroWindowAdv TCPSynRetrans TCPOrigDataSent TCPHystartTrainDetect TCPHystartTrainCwnd TCPHystartDelayDetect TCPHystartDelayCwnd TCPACKSkippedSynRecv TCPACKSkippe
dPAWS TCPACKSkippedSeq TCPACKSkippedFinWait2 TCPACKSkippedTimeWait TCPACKSkippedChallenge TCPWinProbe TCPKeepAlive TCPMTUPFail TCPMTUPSuccess TCPWqueueTooBig
TcpExt: 0 0 0 0 4 0 0 2 0 0 5474 0 0 0 23 18711 20 5462 0 0 1074912 103468 286362 0 62 0 4 0 0 0 0 7 231 0 0 0 1 63 0 5790 730 443 0 1 0 5412 41 131 1 1947 246 0 526 0 18 0 0 2 0 65 0 0
 0 0 0 0 92 0 0 0 0 4 0 0 0 0 502279 62958 0 41 67 62 0 0 0 0 0 0 0 63 0 46023 24 24 127 3817 426585 65 1451 5 153 0 2 26 0 1 0 50 31641 0 0 0
IpExt: InNoRoutes InTruncatedPkts InMcastPkts OutMcastPkts InBcastPkts OutBcastPkts InOctets OutOctets InMcastOctets OutMcastOctets InBcastOctets OutBcastOctets InCsumErrors InNoECTPkts
 InECT1Pkts InECT0Pkts InCEPkts ReasmOverlaps
IpExt: 52 0 4031 9068 2621 2366 6433883901 2649160234 414862 1401162 494808 434825 0 7962649 19915 0 0 0
```

https://man7.org/linux/man-pages/man5/proc.5.html (very powerful interface, also /sys/)

Recap

From this lecture (+previous) you should know

1. How do network packets are transmitted and received
2. What is a LiveLock? How do you mitigate a livelock?
3. What is a MTU and how to calculate a link efficiency
4. What is a TCP segmentation offloading
5. What is a stateful and stateless offloading (advantages, disadvantages)
6. What is a TCP offload engine
7. Basic tools : ethtool, ifconfig, tcpdump, ifstat, netstate, ss, /proc interface

Don't forget the office hours now 3:30-4:30pm

Useful links

1. A Survey of End-System Optimizations for High-Speed Networks, https://dl.acm.org/doi/pdf/10.1145/3184899, ACM Surveys, 2018.
2. Professional Linux Kernel Architecture, https://www.oreilly.com/library/view/professional-linux-kernel/9780470343432/
3. Modern High-Speed Networking Techniques in Hardware and Software, https://sv9rxw.blogspot.com/2020/04/modern-high-speed-networking-techniques.html