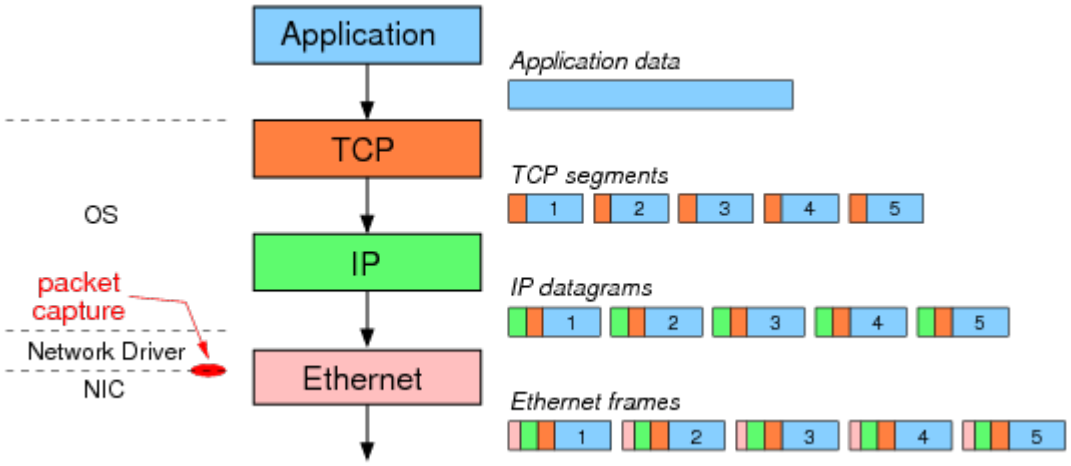
Segmentation and Checksum Offloading: Turning Off with ethtool

By Steven Gordon on Fri, 05/03/2010 - 7:01pm

When introducing data communications concepts and protocols to students I think it is beneficial to demonstrate, and more importantly, allow students to play with real protocols. In the lab I teach (ITS332 [3]), as well as assignments for some lecture courses, we use Wireshark [4] to capture traffic generated by several Internet applications (e.g. ping, Secure Shell, web browsing, iperf). This allows students to see the actual packets being sent across a network, and start to understand the protocol rules and formats used.

Unfortunately sometimes what we see in Wireshark is not what we expect. One case in which this occurs is when TCP/IP operations are offloaded by the operating system to the Network Interface Card (NIC). Common operations for offloading are segmentation and checksum calculations. That is, instead of the OS using the CPU to segment TCP packets, it allows the NIC to use its own processor to perform the segmentation. This saves on the CPU and importantly cuts down on the bus communications to/from the NIC. However offloading doesn't change what is sent over the network. In other words, offloading to the NIC can produce performance gains inside your computer, but not across the network.

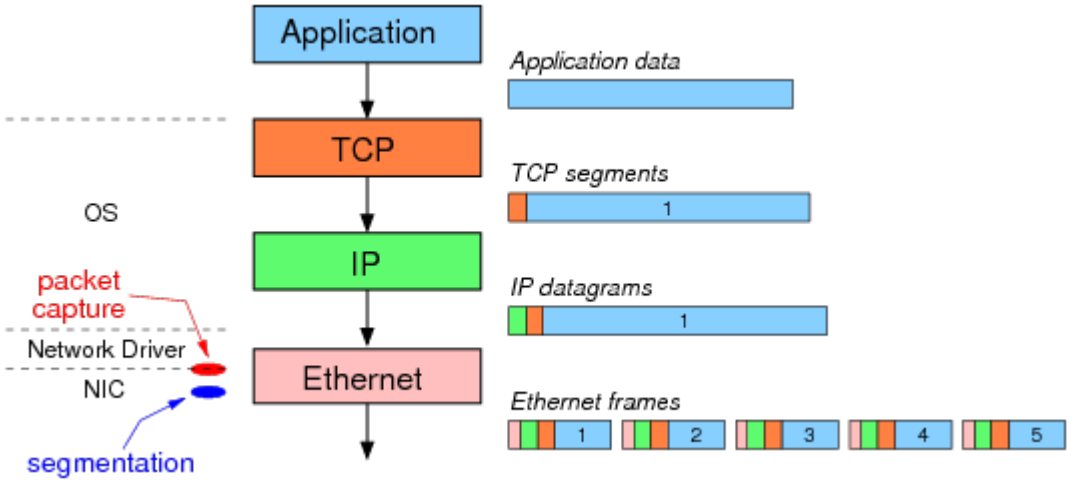
How does this affect what Wireshark captures? Consider the figure below illustrating the normal flow of data through a TCP/IP stack without offloading. Lets assume the application data is 7,300 Bytes. TCP breaks this into five segments. Why five? The Maximum Transmission Unit (MTU [5]) of Ethernet is 1500 Bytes. If we subtract the 20 Byte IP header and 20 Byte TCP header there is 1460 Bytes remaining for data in a TCP segment (this is the TCP Maximum Segment Size (MSS [6])). 7,300 Bytes can be conveniently segmented into five maximum sized TCP segments.



After IP adds a header to the TCP segments the resulting IP datagrams are sent one-by-one to the "Ethernet layer". Note that TCP/IP are part of operating system, while most functionality of Ethernet is implemented on the NIC. However network drivers (lets consider them part of the OS) also perform some of the Ethernet functionality. The network driver creates/receives Ethernet frames. So in the above example, assuming segmentation offloading is not used, the 7,300 Bytes of application data is segmented into 5 TCP/IP packets containing 1460 Bytes of

data each. The network driver encapsulates each IP datagram in an Ethernet frame and sends the frames to the NIC. It is these Ethernet frames that Wireshark (and other packet capture software, like `tcpdump`) captures. The NIC then sends the frames, one-by-one, over the network.

Now consider when segmentation offloading is used (as in the figure below). The OS does not segment the application data, but instead creates one large TCP/IP packet and sends that to the driver. The TCP and IP headers are in fact *template headers*. The driver creates a single Ethernet frame (which is captured by Wireshark) and sends it to the NIC. Now the NIC performs the segmentation. It uses the template headers to create 5 Ethernet frames with real TCP/IP/Ethernet headers. The 5 frames are then sent over the network



The result: although the same 5 Ethernet frames are sent over the network, Wireshark captures different data depending on the use of segmentation offloading. When not used, the 5 Ethernet frames are captured. When offloading is used, Wireshark only captures the single, large frame (containing 7,300 bytes of data).

To further illustrate segmentation offloading, and how to control it in Linux, consider the following tests performed on two Ubuntu computers, `basil` and `ginger`, connected on an Ethernet LAN. On `basil` (which has IP address 10.10.1.22) netcat [7] in server mode is used to receive data:

```
sgordon@basil$ nc -l 5001
```

On `ginger` netcat in client mode is used to send 10,000 Bytes of data (stored in a file) to the server.

```
sgordon@ginger$ nc -p 5002 10.10.1.22 5001 < 10000bytes.txt
```

tcpdump [8] is used to see the captured IP packets, and in particular the size of the TCP segments. I could have used Wireshark, but the text output of `tcpdump>` is easier to include in this page. ethtool [9] is used to view and change the status of segmentation offloading (in this example, generic segmentation offload or GSO).

First note that `ethtool` shows us that generic segmentation offload is on.

```
sgordon@ginger$ sudo ethtool -k eth0
Offload parameters for eth0:
Cannot get device flags: Operation not supported
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp segmentation offload: off
udp fragmentation offload: off
generic segmentation offload: on
large receive offload: off
```

Now, after running the `netcat` client, lets see the output from `tcpdump` (for clarity I have omitted the option fields from selected TCP segments):

```
sgordon@ginger$ sudo tcpdump -i eth0 -n 'not port 22'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
18:30:24.899687 IP 192.168.1.2.5002 > 10.10.1.22.5001: S 679249855:679249855(0) win 58
18:30:24.900583 IP 10.10.1.22.5001 > 192.168.1.2.5002: S 1420594303:1420594303(0) ack
18:30:24.900612 IP 192.168.1.2.5002 > 10.10.1.22.5001: . ack 1 win 92
18:30:24.900713 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 1:2897(2896) ack 1 win 92
18:30:24.900735 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 2897:4345(1448) ack 1 win 92
18:30:24.902575 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 1449 win 68
18:30:24.902591 IP 192.168.1.2.5002 > 10.10.1.22.5001: P 4345:7241(2896) ack 1 win 92
18:30:24.903597 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 2897 win 91
18:30:24.903607 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 7241:8689(1448) ack 1 win 92
18:30:24.903613 IP 192.168.1.2.5002 > 10.10.1.22.5001: P 8689:10001(1312) ack 1 win 9
18:30:24.903617 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 4345 win 114
18:30:24.905573 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 5793 win 136
18:30:24.905587 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 7241 win 159
18:30:24.906628 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 8689 win 181
18:30:24.906637 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 10001 win 204
```

Each line is showing a captured packet. The TCP segments containing data can be identified by the sequence numbers (I've made them **bold**). The number in parentheses indicates the number of bytes in this TCP segment. We can see from the capture that our 10,000 Bytes of data is broken into 5 segments containing: 2896, 1448, 2896, 1448, 1312 Bytes each. But wait ... 2896 Bytes in a TCP segment when the MSS is 1460? (in fact, with TCP header options, like SACK and timestamp, the MSS in this capture is 1448). This is Generic Segmentation Offloading going to work: the OS is sending large segments, as captured above, and letting the NIC do the real segmentation.

So now lets turn Generic Segmentation Offloading off using `ethtool`:

```
sgordon@ginger$ sudo ethtool -K eth0 gso off
```

And run the `netcat` transfer again and look at the `tcpdump` output this time:

```
sgordon@ginger$ sudo tcpdump -i eth0 -n 'not port 22'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
18:33:02.644356 IP 192.168.1.2.5002 > 10.10.1.22.5001: S 3144010294:3144010294(0) win
```

```
18:33:02.645427 IP 10.10.1.22.5001 > 192.168.1.2.5002: S 3901655238:3901655238(0) ack
18:33:02.645471 IP 192.168.1.2.5002 > 10.10.1.22.5001: . ack 1 win 92
18:33:02.645542 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 1:1449(1448) ack 1 win 92
18:33:02.645558 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 1449:2897(1448) ack 1 win 92
18:33:02.645567 IP 192.168.1.2.5002 > 10.10.1.22.5001: P 2897:4345(1448) ack 1 win 92
18:33:02.647415 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 1449 win 68
18:33:02.647433 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 4345:5793(1448) ack 1 win 92
18:33:02.647439 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 5793:7241(1448) ack 1 win 92
18:33:02.648437 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 2897 win 91
18:33:02.648446 IP 192.168.1.2.5002 > 10.10.1.22.5001: . 7241:8689(1448) ack 1 win 92
18:33:02.648451 IP 192.168.1.2.5002 > 10.10.1.22.5001: P 8689:10001(1312) ack 1 win 9
18:33:02.648460 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 4345 win 114
18:33:02.650414 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 5793 win 136
18:33:02.650428 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 7241 win 159
18:33:02.651469 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 8689 win 181
18:33:02.651476 IP 10.10.1.22.5001 > 192.168.1.2.5002: . ack 10001 win 204
```

Now this is what we expect to see - 7 TCP segments each no larger than 1448 Bytes.

Whats the conclusion of all this? What is taught in lectures and textbooks is not always what you see in practice. I suggest turning offloading optimisations off to demonstrate the basic concepts, and then turn them back on again to illustrate the practical performance optimizations applied at the expense of theoretical layering principles.

Interest: Computers and Technology [10]
Networking [11]
Topic: LinuxA [12]
Networking [13]
Content: Howto [14]

Source URL: http://sandilands.info/sgordon/segmentation-offloading-with-wireshark-and-ethtool

Links:
[1] http://sandilands.info/sgordon/segmentation-offloading-with-wireshark-and-ethtool
[2] http://sandilands.info/sgordon/user/2
[3] http://ict.siit.tu.ac.th/~steven/its332/
[4] http://www.wireshark.org/
[5] http://en.wikipedia.org/wiki/Maximum_transmission_unit
[6] http://www.tcpipguide.com/free/t_TCPMaximumSegmentSizeMSSandRelationshiptoIPDatagra.htm
[7] http://en.wikipedia.org/wiki/Netcat
[8] http://www.tcpdump.org/
[9] http://linux.die.net/man/8/ethtool
[10] http://sandilands.info/sgordon/taxonomy/term/158
[11] http://sandilands.info/sgordon/taxonomy/term/168
[12] http://sandilands.info/sgordon/taxonomy/term/200
[13] http://sandilands.info/sgordon/taxonomy/term/7
[14] http://sandilands.info/sgordon/taxonomy/term/212