Encapsulation Offloads LCO, GSO_PARTIAL, TSO_MANGLEID, and Why Less is More

Alexander Duyck

Intel United States alexander.duyck@gmail.com

Abstract

The last several kernel releases have seen a number of new features introduced that enable significant performance improvements with tunnels, and more specifically tunnels with outer checksums enabled.

Local checksum offload (LCO) is a checksum offloading approach that was introduced early this year to resolve the issue of providing an outer checksum while offloading the inner checksum of a given tunneled frame.

TCP offload with IPv4 ID mangling (TSO_MANGLEID) is a feature that was introduced to allow devices to repeat the same IPv4 ID field for each packet.

Partial GSO (GSO_PARTIAL) is an approach by which a device that cannot support tunnel offloads is enabled to do so by only providing the device with information on some subset of the packet headers while allowing GSO itself to populate the remaining headers with the correct values for the segmented frames.

With these features it becomes possible for drivers that didn't previously support encapsulation offloads such as the igb and ixgbe to now support encapsulation offloads. In addition they can support offloads for more tunnel types than supported by most other NICs with pure hardware support simply because they don't have to rely on it and instead allow software to perform most of the header configuration while handling the crucial piece which is segmenting the actual frame. When combined with an outer checksum for the tunnel it allows for full offloads including generic receive offload (GRO) which in turn allows us to send a single thread of encapsulated frames at rates exceeding 10Gbps with test tools such as netperf.

This paper will go over the kernel work that has been done to enable these features, demonstrate the benefits of the features, explain what settings need to be configured in order to get the most out of these features, and provide a look at what hardware vendors can do in order to make use of these features in future devices.

Keywords

checksum, encapsulation, FOU, GENEVE, GRO, GSO, GUE, Kernel, LCO, Linux, TCP, TSO, UDP, VXLAN

Introduction

UDP encapsulation based tunnels, and more specifically approaches such as VXLAN, have become the de-facto standard in data centers that are making use of virtualization. One issue with this is that in many cases VXLAN is not supported with offloads such as TSO, Tx Checksum, Rx checksum, or Receive Side Scaling due to the extra header overhead involved in processing. Network device vendors are working to enable support in new devices but there are already a number of devices in use that do not support such features.

In this paper we will go over recent work done to allow us to make use of these offloads on devices that do not actually support a given tunnel type, but can be adapted to do so with a few minor tweaks. First we will need to describe the basics of UDP encapsulation and how the headers for encapsulated frames are structured. Second we can discuss how it is possible to offload the checksum for a tunneled frame that might require an outer checksum via local checksum offload (LCO). Then we will discuss how this allows us to perform TSO on hardware that may not support these tunnel types. Finally we discuss how making use of these features and enabling Tx checksums in the outer headers of encapsulated frames allows us to greatly increase the throughput for such tunnels as it allows for robust offload support across multiple devices.

Basics of UDP Encapsulation

UDP encapsulation is a mechanism that allows us to take a fully formed Ethernet frame and package it in a UDP datagram. This allows for transport across another network or the Internet itself.. In the case of virtualization it is often used to hide the structure of the underlying network from the virtualized guests that may be resident in one ore more data centers. An example of the header layout for a typical TCP frame encapsulated inside of a VXLAN tunnel is shown in figure 1 below.



Figure 1: Typical TCP Frame and Typical VXLAN Encapsulated Frame

The initial design of many tunnel interfaces strongly encouraged using a checksum of 0 in the outer UDP header [1]. This is primarily due to the fact that many switches do not normally support generating checksum so adding and/or validating a checksum on encapsulation/ decapsulation would result in additional overhead. In addition most Ethernet adapters do not provide a mechanism for providing more than one transport checksum. This resulted in many tunnel protocols being non-compliant with things such as IPv6, which prior to the introduction of UDP based tunnels had required non-zero checksums so that the source and destination address could be validated a part of the transport checksum [2].

In addition the lack of an outer header checksum meant that identifying cases where the packet might be eligible for aggregation via mechanisms such as Generic Receive Offload (GRO) would require additional processing overhead in the case of CHECKSUM_COMPLETE not being supported as it would be necessary to process outer headers before the inner packet checksum could be validated.

Basics of One's Complement Checksum

The idea behind a one's complement checksum is to perform a sum of the data to be validated using one's complement addition.

To generate a checksum we must first clear the checksum field and then perform the one's complement sum over the data to be checksummed. Once that value is generated we then place the complement of that value in the checksum field. To validate the checksum when the packet is received we must take a one's complement sum over the checksummed region. If the result is all ones, otherwise known as negative zero in one's complement arithmetic, then the checksum has been validated to be correct.

There are a number of mathematical properties to a one's complement checksum that can be exploited to allow us to make better use of it when offloading.

The first is that as long as we respect the even/odd assignment of the bytes for the checksum we can order the additions any way we want [4]. So if we want to work from the beginning of the data to the end, or from the end to the beginning either ordering will provide the same result.

The second is that we can perform incremental updates to the checksum. To do this all we need to do is add the difference between the original and the new value.

In the case of transport checksums they will also typically include a pseudo header that is based on the network layer source and destination addresses, length, and protocol value. When offloading a transport checksum the sum for the pseudo header is stored in the location of the checksum so that it is included in the resultant checksum.

Remote Checksum Offload

An initial approach to resolving the problem of having to populate two transport checksums was Remote Checksum Offload (RCO). RCO stores optional metadata with the encapsulated packet that indicates the location of the outer checksum. Using this information it is then possible for the contents of the inner frame to be verified as the outer checksum will include the contents of the inner frame as a part of its own checksum calculation.

While this approach works to resolve the problem of having 2 checksums. It results in a number of complications as it requires additional metadata to be included with the frame [3]. This often results in hardware parsers not being able to parse the frames that use RCO as they expect a certain layout to the frame and do not know how to handle the frame when it contains this extra metadata.

Since many network adapters do support a UDP checksum offload one advantage to this approach is that the Rx checksum verification can be performed by hardware. This in turn allows for offloads such as GRO which can greatly improve performance by aggregating multiple frames into a single frame which reduces the number of trips needed through the stack.

Local Checksum Offload

Making use of the known properties of a one's complement checksum an alternate approach was found that allows us to simply compute the outer checksum for a relatively low CPU cost while offloading the inner transport checksum.

An assertion we must accept in order to be able to do this is that the resultant one's complement sum for the frame from the start of the inner transport header to the end of the packet will be equal to the checksum for the inner network pseudo header [5]. The assumption here is that we will be offloading the inner transport checksum and since the sum of all bytes including the pseudo header is supposed to be negative zero, if we exclude the pseudo header as it exists outside of the transport data then the resultant checksum for all bytes from the transport header onward must be equal to the checksum of just the pseudo header.

Once we know that the one's complement sum for all of the inner transport header and data is the checksum for the inner pseudo header it becomes much easier to compute as we only need to compute the checksum from the outer transport header to the inner transport header. In the case of VXLAN this is typically somewhere between 50 to 70 octets depending on if the inner network protocol is IPv4 or IPv6.

This approach has many advantages. Among them are the fact that using this approach gets the benefits of RCO in that the outer checksum is populated so older devices that do not know how to parse a given encapsulation type can still perform UDP checksum validation which in turn allows for the use of GRO. However it doesn't suffer from the RCO parsing penalties as it is not having to package any additional metadata with the frame. So hardware parsers that expect a specific encapsulation layout can make use of the outer checksum without negatively impacting their ability to parse the frame headers.

Using this approach it becomes possible to make use of any device that can support the Tx checksum via the NETIF_F_HW_CSUM feature. As a result many legacy devices become capable of supporting providing both an inner and outer checksum.

Partial GSO

One limitation of LCO is that it only applies to a single send. In order to extend it beyond this limitation it was necessary to explore what limitations needed to be addressed to apply this to segmentation cases. In addition there were already a number of devices and drivers that supported segmentation of encapsulated frames, however none of these supported segmentation when an outer checksum was present. To address this it is necessary to find cases where we can provide a checksum via a mechanism similar to LCO that could be applied to a segmentation offload.

When a frame is segmented typically only a few fields are updated. In the case of a IPv4/UDP encapsulated IPv4/TCP frame the following fields vary between packets:

- Outer IPv4 ID
- Outer IPv4 Checksum
- Outer IPv4 Length / Outer IPv6 Payload Length
- Outer UDP Checksum
- Outer UDP Length
- Inner IPv4 ID
- Inner IPv4 Checksum
- Inner IPv4 Length / Inner IPv6 Payload Length
- Inner TCP Sequence Number
- Inner TCP Checksum
- Inner TCP Flags

Most tunnel segmentation offloads can update all of the above fields with the exception of the outer UDP checksum. On investigating it turns out that most devices actually ignore the value and instead just replicate it, so if it is populated that populated value is replicated to all segments.

The outer UDP checksum for a given sequence of segments is actually the same value for all segments but the last if it varies in size. This is a result of the Inner IPv4 checksum canceling out the changes in the inner IPv4 ID field, and the inner TCP checksum canceling out changes in the sequence number and flags fields. This leaves us with only the outer IPv4 length, outer UDP length, and inner IPv4 length fields actually effecting the UDP checksum. Similar logic applies in the case of IPv6 as it doesn't include either an ID or checksum field. As a result if we can guarantee that all of the segments of a given frame are exactly the same length we could provide one outer UDP checksum value and that value would be correct for all of the frames in a given sequence.

To force all of the segments of a given frame to be the same size Partial Generic Segmentation Offload was introduced. The idea behind Partial GSO is to force a frame requesting segmentation into a size that is an even multiple of the maximum segment size and to populate all of the fields prior to the inner transport header with the values as though they are a single frame.

This leaves us with a frame ready to be segmented assuming that the device performing the segmentation will update the following fields:

- Outer IPv4 ID
- Outer IPv4 Checksum
- Inner IPv4 ID
- Inner IPv4 Checksum
- Inner TCP Sequence Number
- Inner TCP Checksum

Inner TCP Flags

The resulting fields needed to be updated are greatly reduced. In addition the fields only really differ from traditional TSO without tunnels in that we have to update the inner header IPv4 ID and IPv4 checksum as a result of the ID changing. If we could leave these two fields fixed it might be possible for us to extend TSO even further.

TSO with IPv4 ID Mangling

The concept behind TSO with IPv4 with ID mangling is meant to provide a way to take GSO Partial further. Specifically what we do is find cases where we can avoid having to update the IPv4 ID field and as a result we can leave that value and the IPv4 checksum value static.

In the case of TCP we normally require that the "do not fragment" flag is set. This means that the IPv4 frame should not be fragmented. If we reference RFC 6864 it clearly states that "The IPv4 ID field MUST NOT be used for purposes other than fragmentation and reassembly" [6]. With these two facts one would assume it is safe to leave the IPv4 ID field static in the case of TSO as a correctly implemented protocol stack should not be evaluating the IPv4 ID as we are not fragmenting nor reassembly fragmented frames.

One limitation on this is that GRO and GSO are meant to provide a non-destructive means for aggregating and segmenting frames [7]. This means that we should not be losing data. Unfortunately if we aggregate a frame with an increasing IPv4 ID via GRO and we are attempting to segment it via a device that can only do so via TSO with IPv4 ID mangling then we would be altering the frame data. In addition there are some compression schemes, such as those used for PPP, for which attempting to transmit frames with a fixed IPv4 ID value can be less efficient. For this reason we currently are leaving this feature disabled by default and it must be enabled by the user to make use of it.

Why Less is More

Many network drivers currently implement a mechanism for tracking the port numbers that are used for protocols such as VXLAN and GENEVE. These allow for functionality such as parsing of frames for Tx segmentation offload, Tx checksum offload, Rx checksum offload, and Receive Side Scaling (RSS). One limitation of this is that such functionality is normally limited to only a certain number of ports on a PF as in the case of i40e, or limited to one port for each protocol type as in the case of fm10k.

An additional limitation is that normally only a few tunnel types are supported such as VXLAN or possibly GENEVE, however there are additional UDP based tunnel types available such as VXLAN-GPE, FOU, or GUE. Hardware often can not be extended to support other protocols, or if it can it requires significant firmware changes. These limitations make it difficult to use these offloads when you have an environment that consists of things such as multiple namespaces, multiple ports, multiple VFs, and situations where you might need to support things such as tunnels in tunnels, or new tunnel extensions that were released after the hardware was released.

One of the main goals of GSO partial was to support an environment in which devices without tunnel offloads would be able to make use of some form of tunnel offload. To this end I make use of a common Ethernet silicon, specifically the Intel X540, that had no support for VXLAN offloads so that I could use that for my initial proof of concept. What I did is determined what environment variables and changes would be needed in order to allow for ideal throughput when sending messages between VFs on the same silicon. This allowed me to exceed the 10Gb/s limitations of the external port on the device.

The first major ingredient involved in enabling better performance for VXLAN tunnels is to find a way to overcome the lack of an inner checksum offload. In order to overcome this it was necessary to enable the outer checksum on the tunnel as I could then use this value to validate the inner checksum using software techniques. However this places a limitation on our Tx path as we must then also support providing inner checksum offload and LCO. In order to support inner checksums on the ixgbe and ixgbevf drivers it was necessary to modify the driver so that we could support NETIF F HW CSUM as this allows for a fully generic Tx checksum offload. If the device supported notifications of Rx checksums via CHECKSUM_COMPLETE it would be possible to support offloading the checksums without even needing to include the outer checksum in the packet. However with that not being the case it was necessary to enable outer tunnel checksums. By enabling the outer checksum and adding support for NETIF_F_HW_CSUM I saw the throughput double for tunnels with the bottleneck being moved from the Rx path to the Tx path as the segmentation workload consumed a significant amount of CPU time.

The second piece to improving VXLAN tunnel performance is providing the ability to scale traffic between two tunnel endpoints. One of the pieces of data lost when data is encapsulated inside of a tunnel is that the inner L3 and L4 headers are no longer accessible unless you know that a given UDP port number is a tunnel and the format of tunnel used. To work around this many UDP based tunnels provide a hash of the inner header data in the source port for the tunnel. As such we can work around this limitation by enabling hashing on the UDP source and destination ports for all UDP traffic. One limitation this introduces though is possible packet reordering for UDP flows that experience fragmentation as fragments are normally hashed only on the IP header source and destination address, whereas a non-fragmented frame will include the UDP source and destination port numbers in the resultant hash.

The final piece to improving the performance was GSO partial. To fully support GSO Partial for ixgbe and ixgbevf it was necessary to add support for TCP Segmentation with

IPv4 ID mangling as the TSO support in hardware was only capable of updating one IP header. Enabling support for GSO partial reduced the Tx overhead for a simple netperf test by several fold. As a result the bottleneck had moved back from the Tx to the Rx side. Adding additional threads I was able to approach nearly 15Gb/s but began to encounter the limitations of the PCIe Gen2 x8 link for the device instead of saturating the Tx CPU which made it difficult to determine the full gain of this change.

In the case of the Intel drivers igb, igbvf, ixgbe, ixgbevf, i40e, and i40evf it is possible to fully support all tunnel types currently supported by the stack using these approaches. However there are many drivers where we cannot support segmentation in this way. Examples include the fm10k driver where the ability to parse the packet is required to perform checksum offload and/or segmentation. In the case of such devices our functionality becomes limited and we have to resort to performing all of the segmentation and checksum work in software. In addition if devices future in the supported CHECKSUM COMPLETE and reported the packet checksum instead of the current approach that normally only validates it we might be able in the future to support tunnels much more generically.

Conclusion

With the introduction of GSO Partial and LCO it becomes quite easy to support offloading of tunnels on devices that may not support the actual tunnel protocol, but provide the basic tools needed to support minimal offloads.

Acknowledgements

I would like to thank Edward Cree and Tom Herbert for their input and feedback during the work on GSO Partial.

References

1. Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreer, L., Sridhar, T., Bursell, M., and Wright, C., "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014 2. Eubanks, M., Chimento, P., and Westerlund, M., "IPv6

and UDP Checksums for Tunneled Packets", RFC 6935, April 2013

3. Herbert, T., "UDP Encapsulation in Linux", Proceedings of netdev 0.1, February 2015,

https://people.netfilter.org/pablo/netdev0.1/papers/UDP-Encapsulation-in-Linux.pdf

4. Braden, R., Borman, D., and Partridge, C., "Computing the Internet Checksum", RFC 1071, September 1988

5. Cree, E, "Checksum Offloads in the Linux Networking Stack",

https://www.kernel.org/doc/Documentation/networking/ch ecksum-offloads.txt

6. Touch, J., "Update Specification of the IPv4 ID Field", RFC 6864, February 2013

7. Duyck, A., "Segmentation Offloads in the Linux Networking Stack",

https://www.kernel.org/doc/Documentation/networking/seg mentation-offloads.txt

Author Biography

Alexander Duyck is an Open Source Technologist at Intel. He has worked on Linux and kernel networking over the last 10 years contributing to many areas of kernel networking including drivers such as igb, igbvf, ixgbe, ixgbevf, and fm10k, features such as DCB and SR-IOV, routing, general driver performance, and most recently focusing on tunnel offloads.