UDP Encapsulation in Linux

Tom Herbert

Google
USA
therbert@google.com

Abstract

UDP encapsulation encompasses the techniques and protocols to encapsulate and decapsulate networking packets of various protocols inside UDP. UDP encapsulation has become prevalent in data centers, and in fact nearly all the solutions for network virtualization currently being proposed in IETF are based on UDP encapsulation.

In this paper we present much of the recent work done by the Linux networking community to make UDP encapsulation a first class citizen. This cumulative work has resulted in greatly improved performance, robustness, and scalability. We begin by describing the basic support and model for UDP encapsulation. Next, we look at performance enhancements in the areas of load balancing, checksum offload, and segmentation offload. Finally, we examine two generic methods of UDP encapsulation: Foo-over-UDP and Generic UDP Encapsulation.

Keywords

UDP, encapsulation, Linux, GRO, GSO, checksum, GUE, FOU

Introduction

UDP encapsulation is becoming ubiquitous in data centers, not just for virtualization use cases, but also for non-virtualization. The reason for this is simple: it is a low overhead protocol that allows several UDP specific optimizations commonly supported by networking hardware to be leveraged. UDP is a very simple and flexible transport protocol that offers a great deal of interoperability and compatibility with legacy hardware.

In this paper we focus on the recent work done in the Linux networking stack to support UDP encapsulation. First, we describe the basics of UDP encapsulation and its support in Linux. Secondly, we discuss use of common networking optimizations with UDP encapsulation for load balancing, checksum offload, and segmentation offload. We present novel techniques of source port flow identifiers, checksum-unnecessary conversion, and remote checksum offload. Finally, we examine support for some specific UDP encapsulation methods; in particular we look at *Foo-over-UDP (FOU)* and *Generic UDP Encapsulation (GUE)*. FOU provides the simplest no frills model of UDP encapsulation, it simply encapsulates packets directly in the UDP payload. GUE is a generic and extensible encapsulation, it allows encapsulation of packets for any IP protocol and optional data as part of the encapsulation.

Basics of UDP encapsulation

Encapsulation is the technique of adding network headers to a fully formed packet for the purposes of transit across a network. *UDP encapsulation* includes the techniques and protocols to encapsulate networking packets within User Datagram Protocol [1]. Packets are contained in the UDP payload, and are said to be *encapsulated* in UDP packets.

Tunneling, overlay networks, and network virtualization, are terms often associated with encapsulation. *Tunneling* refers to the use of a high level transport service to carry packets or messages from another service. Encapsulation is often the lower level mechanism that implements a tunnel. An *overlay network* is a computer network which is built on the top of another network. An overlay network may be composed of links which are implemented by tunneling. *Network virtualization* creates logical, virtual networks that are decoupled from the underlying network hardware. A virtual network is often implemented as an overlay network which provides the illusion of being a physical network to the user.

Encapsulation does not require UDP, in fact there are several methods for encapsulation of packets within IP not using UDP; these include IPIP (IP over IP), SIT (IPv6 over IPv4), GRE (Generic Routing Encapsulation), L2TP (Layer Two Tunneling Protocol) and EtherIP (Ethernet over IP) [2,3,4,5,6]. However, encapsulating using UDP provides some distinct advantages:

- Hardware optimizations for scaling, such as *RSS* (Receive Side Scaling) and *ECMP* (Equal Cost Multipath) routing, can be leveraged. These can provide significant performance benefits.
- The UDP checksum provides protection against packet mis-delivery. This especially relevant if a packet is being encapsulated in IPv6 which does not include a header checksum.
- Hardware support for UDP checksum can be leveraged. NIC support for UDP checksum offload is ubiquitous and can be used to offload inner checksum calculation.
- The destination UDP port provides a demux for different encapsulation methods or encapsulation protocols.
- UDP allows extensible encapsulation protocols. For instance, some proposed protocols include sending optional data with encapsulated packets.

Model of UDP Encapsulation

Conceptually, UDP encapsulation is simple. Encapsulation is performed by an *encapsulator*. An encapsulator starts with a packet which could be for layer 2, layer 3, or layer 4. IP and UDP headers are prepended to the packet. The IP header addresses the endpoints of the encapsulation, the destination being the node that will perform decapsulation. The destination port of the UDP header is set to a specific port number for the encapsulation method. An additional encapsulation header may be inserted after the UDP header which can indicate the protocol of the encapsulated packet or other data related to the encapsulation. Once the encapsulated packet is created it is transmitted to the destination IP address.

At the destination of the encapsulated packet a *decapsulator* performs decapsulation. This involves verifying and removing the IP and UDP headers as well as any additional encapsulation headers. After removing these headers, the resultant packet is now the same one that was originally encapsulated. This packet is then processed by the networking stack based on its protocol.

In an encapsulated packet, the encapsulating headers are known as *outer headers*. The headers of the encapsulated packet are known as *inner headers*.

Figure 1 illustrates UDP encapsulation being used to create tunnels for network virtualization.

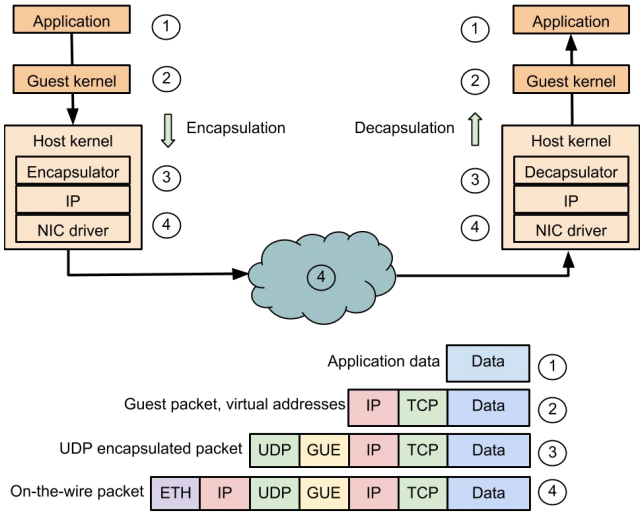

Figure 1. UDP and GUE encapsulation for network virtualization. The diagram at the top illustrates the flow of a packet from an application in one Virtual Machine (VM) to a peer application in another VM on another host. The bottom portion shows the packet encapsulations and protocol headers for the various protocol layers.

UDP Encapsulation Support in Linux

The Linux stack includes various facilities for supporting UDP encapsulation. An encapsulation method is usually implemented as part of a specialized kernel module. Method specific configuration specifies use and parameters of encapsulation for transmit as well as receive including the UDP port number for the encapsulation. In the case that UDP encapsulation is being used for implementing network tunnels (i.e. encapsulation of Layer 2 or Layer 3 packets) configuration includes the source and destination addresses of the tunnel endpoints which are set in the outer IP header.

The facilities and APIs described in this paper are based on the 3.18 version of Linux unless otherwise noted [7].

Encapsulated packet representation. In the Linux kernel, control information and data pointers for a packet are contained in the `sk_buff` data structure [8]. Components of an encapsulated packet are represented by fields in the `sk_buff` structure. The `sk_buff` has references to both the outer headers and inner headers of encapsulation.

The fields for the outer headers are also just references to the headers of a packet without encapsulation. These are:

```
transport_header
     Transport layer header
network_header
     Network layer header
mac_header
     Link layer header
```

The fields referring to the inner headers of an encapsulated packet are:

```
inner_protocol
     Protocol of encapsulated packet
inner_transport_header
     Inner transport layer header
inner_network_header
     Inner network layer header
inner_mac_header
     Inner link layer header
```

The inner header fields are only valid if the `encapsulation` bit is set in an `sk_buff`, and they are only relevant in the transmit path. In the case of multiple nested encapsulations, the outer header fields always refer to the outermost headers, and the inner header fields refer to the innermost headers. Note that the number of nested encapsulations in a packet is only bounded by the MTU (maximum size of a packet), however some kernel mechanisms are optimized to handle up to three nested encapsulations.

Receive path. To implement the receive path, an implementation creates an in-kernel UDP socket and binds the local port to the port number specified for encapsulation. The Linux stack defines an `encap_rcv` function for sockets which is set by an encapsulation method to receive packets. When `encap_rcv` is set, the UDP layer calls this function in lieu of normal receive processing for a socket. Up to the point that the

`encap_rcv` function is called, the UDP stack processes packets with encapsulation no differently than other UDP packets; this includes validating the UDP headers and verifying the UDP checksum.

Transmit path. On the transmit side, an encapsulation method builds encapsulated packets. This typically entails prepending a UDP header and encapsulation header if needed to the packet being encapsulated. Helper functions are called to set the UDP source port and initialize the UDP checksum field. The assembled packet is then sent using the IP packet transmit functions. Note that the transmit path does not require a socket as the receive side does, and there is no required relationship between the send and receive paths.

Offload mechanisms

Much of the work in enhancing the Linux stack to support UDP encapsulation is focused on making existing offload mechanisms "encapsulation aware". Offload mechanisms are techniques that are implemented separately from the normal protocol implementation of the stack and are intended to optimize or speed up protocol processing. Hardware offload is performed within a NIC device on behalf of a host. Software offload mechanisms are implemented in a lower layer than protocol processing, typically near or in NIC drivers.

There are thee basic offload techniques of interest:

- Load balancing
- Checksum offload
- Segmentation offload

Load Balancing

Both networking hardware and software stacks implement a variety of mechanisms to perform load balancing (statistical multiplexing) of packets across a set of networking resources. Switches often implement Equal Cost Multipath routing (ECMP) which distributes packets over multiple network paths to improve utilization [9]. Most Network Interface Cards (NICs) implement Receive Side Scaling (RSS) which is a technique to distribute packets over a number of receive queues to promote parallelism and reduce latency in host processing [10]. The Linux stack implements Receive Packet Steering (RPS) which is a software analogue for RSS, and Receive Flow Steering (RFS) which steers packets to the CPU where they are being received by an application [11]. In most cases, load balancing is done at the granularity of packet flows. A flow is a sequence of packets that belong to the same the logical communication happening between a pair of hosts (packets for a TCP connection for instance). Usually, the packets for a flow should follow the same path through load balancing mechanisms so that they are delivered in order.

Hardware devices commonly perform hash computations on packet headers to classify packets into flows or flow buckets. Packets are classified into flows by computing a *flow hash*. Flow hashes are usually either a three-tuple hash over the source address, destination address, and protocol number; or a five-tuple hash over the source address, destination address, source port, destination port, and protocol number. Some devices and the Linux stack in its flow hash calculation (`skb_get_hash`) omit the protocol number to produce a two-tuple or four-tuple hash which doesn't appreciably reduce the quality of the flow hash value. Typically, networking hardware will compute five-tuple hashes for TCP and UDP, but only three-tuple hashes for other protocols. Since the five-tuple hash provides more granularity, load balancing can be finer grained with better distribution.

In the case of UDP encapsulation, the computed flow hash of a packet should be representative of the flow for the encapsulated packet. To provide for this, the source port of the outer UDP header can be set to a value that maps to the inner flow. This is referred to as the *inner flow identifier*. The inner flow identifier is set by the encapsulator; it can be computed on the fly based on packet contents or retrieved from state maintained for the inner flow. A device that computes the flow hash of a UDP packet will include the source port in its calculation, so in turn the flow hash will correspond to the inner flow.

Examples of deriving an inner flow identifier are:

- If the encapsulated packet is a layer 4 packet, TCP/IPv4 for instance, the inner flow identifier could be based on the canonical five-tuple hash of the inner packet.
- If the encapsulated packet is an AH (IPsec Authentication Header) transport mode packet with TCP as next header, the inner flow identifier could be based on the two-tuple hash of the source and destination TCP ports.
- If a node is encrypting a packet using ESP (IPsec Encapsulating Security Payload) tunnel mode, the inner flow identifier could be based on the contents of clear-text packet. For instance, a canonical five-tuple hash for a TCP/IP packet could be used.

The five-tuple hash commonly used to identify a flow in UDP will cover the outer source address, destination address, source port (inner flow identifier), and destination port. These values are expected to be mostly persistent for the lifetime of an encapsulated flow, only changing infrequently (at most once every thirty seconds).

NIC support for UDP hash. Most NICs that support UDP flow hash calculation (for UDP RSS) disable it by default. This was done to ensure that fragments of a UDP packet are received in order when using RSS. If a UDP packet is fragmented, a five-tuple hash can only be calculated for the first fragment which contains the UDP headers. For the rest of the fragments, only a three-tuple hash can be procured. In UDP encapsulation, fragmentation of the outer UDP packet is avoided so a five-tuple hash should always be calculable.

The UDP flow hash in the NIC should be enabled to optimize for UDP encapsulation. This can be done per device using the `ethtool` command [12]. For example:

```
ethtool -N eth0 rx-flow-hash udp4 sdfn
```

This command configures the NIC device for `eth0` to include the UDP ports and IP addresses when computing the flow hash for UDP over IPv4 packets.

Function to set UDP source port for encapsulation. In the Linux stack a common function is called to create a source port value for UDP encapsulation [13]. This function is:

```
be16 udp_flow_src_port(struct net *net,
      struct sk_buff *skb,
      in mint, int max, bool use_eth)
```

This function returns a hash value for the packet passed in the `sk_buff` (before encapsulation). The value is limited to the range provided by the `min` and `max` arguments. By default (when `min` and `max` are zero) the range for the return value is the local port range in the system which defaults to the ephemeral port range 32768 to 65535. `skb_get_hash` is called to determine the flow hash for the packet. If a flow hash has already been set in the `sk_buff` (`hash` field is nonzero) that value is used, else the packet will be parsed to determine the flow hash (`skb_flow_dissect` function is called).

Flow label in IPv6. An alternative to setting the UDP source port with a flow identifier is to set the IPv6 flow label to correspond to the inner flow [14]. Some devices may be configured to use the flow label in hash computation, and the Linux stack will compute a flow hash using flow label and the IP addresses of a received packet if the flow label is non-zero.

A common function exists to create a flow label for transmit based on a packet's flow hash [15]:

```
be32 ip6_make_flowlabel(
   struct net *net, struct sk_buff *skb
   be32 flowlabel,bool autolabel)
```

If `autolabel` is true, `flowlabel` is zero, and the `auto_flowlabels` IPv6 networking sysctl is set then `skb_get_hash` is called to determine the hash for the flow and it is returned. The caller will mask the value to twenty bits for setting the IPv6 flow label in a packet.

Checksum offload

IP checksum calculation is known to be an expensive operation to perform in a host CPU. Most deployed NICs provide capabilities to offload checksum calculations for both transmit and receive. If the Linux stack must calculate a packet checksum, it is done at most once per packet using simple arithmetic properties of the checksum to validate or set multiple checksums in a packet as necessary.

When encapsulating using UDP, there are at least two checksums within a packet to be considered: the checksum of the encapsulated transport packet and the checksum of the encapsulating UDP header. For IPv4, the UDP checksum is optional by setting the checksum field to zero. For IPv6, the UDP checksum was originally required to be used, however this is relaxed by RFC6936 which allows the IPv6 UDP checksum to be zero for UDP tunneling under certain conditions [16].

Transmit checksum offload. There are two methods of transmit hardware checksum offload supported by NICS: `NETIF_F_HW_CSUM` and `NETIF_F_IP_CSUM` [8].

`NETIF_F_HW_CSUM` is a protocol agnostic method to offload the transmit checksum. In this method the host provides checksum related parameters in a transmit descriptor for a packet. These parameters include the starting offset of data to checksum and the offset in the packet where the computed checksum is to be written. The length of data to checksum is implicitly the length of the packet minus the starting offset. The host initializes the checksum field to the complement (bitwise *not*) of the pseudo header checksum for the transport protocol. In the case of UDP encapsulation, the checksum for an encapsulated transport layer packet, a TCP checksum for instance, can be offloaded by setting the appropriate checksum parameters. NICs typically can offload only one transmit checksum per packet, so simultaneously offloading both an inner transport packet's checksum and the outer UDP checksum is likely not possible. In that case setting the UDP checksum to zero and offloading the inner transport packet checksum might be acceptable.

To request checksum offload, a transport layer sets checksum related fields in the `sk_buff` for a packet. The starting offset for the checksum calculation is set in `csum_start`, and the offset where the checksum is to be written (relative to `csum_start`) is set in `csum_offset`. The checksum status field `ip_summed` is set to `CHECKSUM_PARTIAL`. The checksum field in the transport header is initialized to the complement of the pseudo header checksum for the transport protocol.

Many legacy devices implement `NETIF_F_IP_CSUM` instead of `NETIF_F_HW_CSUM`. This is a limited form of checksum offload where a device can only perform transmit checksum offload for certain protocol combinations. Originally, `NET_F_IP_CSUM` was only applicable to simple TCP/IP and UDP/IP packets, but some newer devices have extended this to handle certain instances of VXLAN or NVGRE encapsulation. Because `NETIF_F_HW_CSUM` is protocol agnostic and generic, it is generally preferred over `NETIF_F_IP_CSUM`.

Functions to initialize UDP checksum. When building a packet with UDP encapsulation, an encapsulation method can call common functions to initialize the UDP checksum field [17].

The function used when encapsulating in IPv4 is:

```
void udp_set_csum(bool nocheck,
   struct sk_buff *skb, be32 saddr,
   be32 daddr, int len)
```

The function used when encapsulating within IPv6 is:

```
void udp6_set_csum(bool nocheck,
   struct sk_buff *skb,
   const struct in6_addr *saddr,
   const struct in6_addr *daddr,
   int len)
```

`skb` is the `sk_buff` for the UDP packet. `saddr` and `daddr` are the source and destination IP addresses. `len` is the UDP length which includes eight bytes for the UDP header and length of the UDP payload.

These functions do one of the following:

- If `nocheck` is set then zero is written in the UDP checksum field. UDP checksum is not enabled.
- Else, if the transmit device supports checksum offload of UDP, then checksum offload is set up for the UDP checksum.
- Otherwise, the full packet checksum is calculated and the proper UDP checksum is written in the UDP checksum field.

Receive checksum offload. Similar to transmit checksum offload, there are two methods that NICs may implement for receive checksum offload: `CHECKSUM_COMPLETE` and `CHECKSUM_UNNECESSARY` [8].

`CHECKSUM_COMPLETE` is a technique where a NIC computes the ones complement checksum over all (or some predefined portion) of a packet. The computed value is provided to the host in the packet's receive descriptor and saved in the `csum` field in the packet's `sk_buff`. The host stack uses this checksum to verify any transport checksums in the packet (both in inner and outer headers).

As a packet is processed by different protocol layers the saved checksum value is adjusted to correspond to the packet seen at each protocol layer. Adjusting the checksum is facilitated by a utility function [18]:

```
void skb_postpull_rcsum(
   struct sk_buff *skb,
   const void *start, unsigned int len)
```

This function is called by a protocol layer to adjust the saved `csum` before passing the packet to the next layer. It "subtracts out" the checksum for the current layer's protocol headers starting from `start` pointer for `len` bytes. For example, in IPv6 `skb_postpull_rcsum` is called to subtract out the checksum of the IPv6 header from the saved checksum value before passing the packet to transport layer processing. Interestingly, this function is not called in the IPv4 input processing since it is assumed that the IPv4 header already has a zero checksum value due to the use of IPv4 header checksum.

Many legacy NICs don't provide the complete checksum but instead may explicitly verify checksums within the packet. The device returns an indication to the host that a checksum is verified, and the network driver marks the `sk_buff` to indicate `CHECKSUM_UNNECESSARY`. A device may validate more than one checksum per packet, for instance the outer UDP checksum in encapsulation and an inner transport checksum. The `csum_level` field in the `sk_buff` indicates the number of checksums validated by `CHECKSUM_UNNECESSARY` (the number validated is `csum_level` plus one).

`CHECKSUM_UNNECESSARY` only works for specific protocol combinations that a device is capable of parsing. For instance, if a new encapsulation protocol were created, a device supporting `CHECKSUM_UNNECESSARY` might need to be updated, whereas a device supporting `CHECKSUM_COMPLETE` should continue to work without change. For this reason, `CHECKSUM_COMPLETE` is generally the recommend approach for new devices.

Checksum-unnecessary conversion. Checksum-unnecessary conversion is a technique in the Linux stack to deduce the complete checksum of a received packet when a non-zero UDP checksum has been verified [19]. This is useful in cases where a NIC is only capable of providing `CHECKSUM_UNNECESSARY` for simple UDP/IP packets. If a UDP checksum has been verified, the ones complement checksum of the packet starting from the UDP header equals the complement of the pseudo header checksum used in UDP checksum calculation. This is used to convert the `CHECKSUM_UNNECESSARY` indication for the UDP checksum to `CHECKSUM_COMPLETE` with a checksum value. Any inner checksums in the packet can then be verified by the stack without performing a checksum calculation over the packet. Most of the work for checksum-unnecessary conversion is implemented in the `skb_checksum_try_convert` function.

The actions of checksum-unnecessary conversion are:

1. NIC reports in receive descriptor that the transport (UDP) checksum for a packet has been verified.
2. Host driver sets `CHECKSUM_UNNECESSARY` in the `sk_buff` for the packet.
3. UDP layer accepts that the UDP checksum is valid based on `CHECKSUM_UNNECESSARY`.
4. UDP socket is matched. If checksum-unnecessary conversion is configured for the socket and the UDP checksum is non-zero perform conversion.
5. Calculate the checksum of the pseudo header that is used in calculating the standard UDP checksum. This is a checksum calculation over the IP addresses and UDP ports in the packet.
6. Set `csum` in the `sk_buff` to the complement of the pseudo header checksum calculated in step #5. Set `ip_summed` to `CHECKSUM_COMPLETE`.

7. Continue processing the packet. Inner protocol layers call `skb_postpull_rcsum` so that `csum` reflects the checksum of the packet for the current layer being processed. If an encapsulated packet has a checksum (e.g. TCP), the checksum complete value is used to validate it.

Remote Checksum Offload. Remote checksum offload is a mechanism that provides checksum offload on transmit of encapsulated packets using only rudimentary NIC offload capabilities [20]. This technique leverages UDP transmit checksum offload which is supported by most NICs including those that only support `NETIF_IP_CSUM`. The outer UDP checksum is enabled in packets and, with some additional meta data, a receiver is able to deduce the checksum to be set for an inner encapsulated packet. Effectively this offloads the computation of the inner checksum to the remote host. The UDP checksum covers the whole packet so there is no loss of protection for the inner packet.

Remote checksum offload requires an encapsulation header that allows optional data in the encapsulation. This has been implemented for Generic UDP Encapsulation and VXLAN (in Linux version 3.19) [21,22,23]. The remote checksum data set in an encapsulation header is comprised of a pair of checksum start and checksum offset values. More than one offloaded checksum could be supported if multiple pairs are represented. *Checksum start* is the starting offset for checksum computation relative to the start of the encapsulated payload. This is typically the offset of a transport header (e.g. UDP or TCP). *Checksum offset* is the offset where the derived checksum value is to be written relative to the start of encapsulated payload. This typically is the offset of the checksum field in the transport header (e.g. UDP or TCP checksum).

The typical actions to set up remote checksum offload on transmit are:

1. Transport layer creates a packet and indicates in the `sk_buff` that its checksum is to be offloaded to the NIC for normal transport checksum offload.
2. Encapsulation layer adds its headers to the packet including the optional data for remote checksum offload. The start offset and checksum offset are set per `csum_start` and `csum_offset` in the `sk_buff.`
3. Encapsulation layer arranges for hardware checksum offload of the outer UDP checksum, this overrides the offload parameters set in the `sk_buff` for the transport layer checksum.
4. Packet is sent to the NIC. The NIC will perform transmit checksum offload and set the checksum field in the outer UDP header. The inner headers and rest of the packet are transmitted without modification.

The typical actions a host receiver does to support remote checksum offload are:

1. Receive packet and validate outer checksum following normal processing (ie. validate non-zero UDP checksum).
2. Deduce complete checksum for the packet. This is directly provided if the device returns the packet checksum in `CHECKSUM_COMPLETE`. If the device returned `CHECKSUM_UNNECESSARY`, checksum-unnecessary conversion can be done to deduce the checksum
3. From the packet checksum, subtract the checksum computed from the start of the packet (outer IP header) to the offset in the packet indicated by checksum start in the optional data. The result is the deduced checksum to set in the checksum field of the encapsulated transport packet.
4. Write the resultant checksum value into the packet at the offset provided by checksum offset in the optional data.
5. Checksum is verified at the transport layer using normal processing. This should not require any checksum computation over the packet since the complete checksum has already been deduced.

Segmentation Offload

Segmentation offload refers to techniques that attempt to reduce CPU utilization on hosts by having the transport layers of the stack operate on large packets. In transmit segmentation offload, a transport layer creates large packets greater than MTU size (Maximum Transmission Unit). It is only at much lower point in the stack, or possibly the NIC, that these large packets are broken up into MTU sized packet for transmission on the wire. Similarly, in receive segmentation offload, small packets are coalesced into large, greater than MTU size packets at a point low in the stack receive path or possibly in a device. The effect of segmentation offload is that the number of packets that need to be processed in various layers of the stack is reduced, and hence CPU utilization is reduced.

The Linux stack supports TCP segmentation offload and UDP fragmentation offload. For UDP encapsulation, TCP segmentation is the primary interest.

Generic Segmentation Offload. *Generic Segmentation Offload*, or *GSO*, is software feature of the Linux networking stack which allows the stack to create large, greater than MTU size TCP packets [24]. Immediately before handing off a packet to a driver for transmission, GSO splits the packet into separate smaller packets of size less than or equal to the MTU.

The packets created by GSO need to have their own headers, and certain fields in these headers need to be explicitly set on a per packet basis. For each created segment the general process is:

1. Replicate the TCP header and all preceding headers of the original packet.

2. Set payload length fields in any headers to reflect the length of each new segment.
3. Set the TCP sequence number to correctly reflect the offset of the TCP data in the stream.
4. Recompute and set any checksums that either cover the payload of the packet or a cover header which was changed by setting a payload length.

In Linux, `skb_gso_segment` is called by the networking stack to perform segmentation of a large packet before transmission. `skb_gso_segment` calls a series of chained callbacks for each protocol layer of the packet, usually starting from the Ethernet layer. Each protocol that participates in GSO implements a `gso_segment` function which returns the created packets in a list of `sk_buff`s. Each GSO function does two things. First `gso_segment` is called for the next layer protocol. The lowest layer GSO function (e.g. `tcp_gso_segment`) calls `skb_segment` to actually create the packets. For each packet created, protocol headers are simply copied from the large packet. Upon return from calling `gso_segment` for the next layer (or `skb_segment`), a protocol layer sets header fields in each new packet to reflect the segmentation. For instance, the IP layer needs to set the total length field in each packet before returning.

The `sk_buff` structure includes a field `gso_type` which contains a set of flags describing the affected protocol layers and parameters for a large GSO packet. These flags have the form `SKB_GSO_*`. For example, an IPIP packet containing an encapsulated TCP packet would include types `SKB_GSO_IPIP` and `SKB_GSO_TCP`. A packet with UDP encapsulation would include `SKB_GSO_UDP_TUNNEL` if zero UDP checksum is to be set on transmission, or `SKB_GSO_UDP_TUNNEL_CSUM` if UDP checksums are enabled.

In the case of GSO with encapsulation, the stack initially performs encapsulation on the large packet. The encapsulation layer sets the inner headers appropriately in the `sk_buff` and also sets the `inner_protocol` value to correspond to the encapsulated packet (this can be an IP protocol or Ethertype). The UDP GSO handler will detect that a packet is UDP encapsulated when the `encapsulation` bit is set in the `sk_buff` and the `gso_type` includes `SKB_GSO_UDP_TUNNEL_*`. The UDP GSO handler will call the `gso_segment` function for the encapsulated protocol based on the value in `inner_protocol`. Any additional encapsulation headers between the start of the UDP payload and the start of the encapsulated packet (indicated by the offset in `inner_mac_header`) are treated as being opaque and just copied into each segment. This allows a generic encapsulation that works with any UDP encapsulation protocol as long it does not have fields that need to be explicitly set on a per segment bases.

Large Segmentation Offload. Many NICs provide *Large Segment Offload* (*LSO*) for performing transmit segmentation offload in hardware [25]. This is called *TCP Segmentation Offload* (*TSO*) when applied to TCP segmentation. Since LSO happens in the device, it can provide better performance compared to GSO.

The process of LSO is similar to that of GSO. The hardware is provided a large packet for transmission. It splits the large packet into smaller packets properly setting lengths, checksums, sequence numbers in each packet's headers.

Devices typically only support a subset of protocols for LSO that the stack supports for GSO. Drivers indicate supported protocols for LSO by setting flags in their advertised feature flags. These flags typically correspond to equivalent `SKB_GSO_*` flags. For instance, `NETIF_F_GSO_UDP_TUNNEL` would indicate that a device is capable of performing LSO on a UDP tunnel. Some devices may support an offload with constraints, for instance the encapsulation headers might need to be less than a certain length. In these cases, drivers may define `ndo_features_checks` which is called from the core transmit path to determine if a device is capable of performing offload operations on a given packet [26]. This function gives the driver an opportunity to implement any restrictions that cannot be otherwise expressed by feature flags. If it is determined that a device cannot perform LSO for a packet, the stack will always fall back to doing GSO.

To implement LSO with UDP encapsulation, it is desirable that the implementation is agnostic to the particular method of UDP encapsulation. This is possible assuming that encapsulation headers don't include fields that need to be updated for each segment. In this case the NIC would be provided with the offset of the inner MAC header (or inner network header), so that for each segment the bytes from the start of the UDP payload to the inner MAC header are just copied from the large packet.

Generic Receive Offload. *Generic Receive Offload*, or *GRO*, is feature of the Linux networking stack which coalesces packets for a flow (usually for TCP) into large packets before being subjected to higher layer protocol processing [27]. The GRO functions are called as early as possible in the network receive path in order to maximize the benefits.

In Linux, network drivers call `napi_gro_receive` to handoff received packets to the stack. If the GRO feature is enabled on the receiving device, the stack will attempt to perform GRO coalescing on the packet. For each networking device, a list of flows being coalesced is maintained. The structure for each flow holds a list of packets that have been matched to belong to the same network flow and are being coalesced. The GRO operation is to try to match a received packet with one of the flows. If a flow is matched and the packet is next in sequence for the flow, it can be coalesced. The list of flows being coalesced is created on demand, so if a packet does not match an existing flow a new one is added to the device's list.

To match a packet, a series of chained callbacks (`gro_receive` functions) is called for each protocol layer of the packet usually starting at Ethernet. Before the callbacks, the stack marks each recorded flow as a candidate for matching the packet (`same_flow` is set for each flow in the list). At each callback, a protocol layer considers whether the packet matches the flows based on characteristics of the particular protocol layer (for instance IP addresses are compared in the IP layer). For flows that do not match they are marked to not be a candidate (`same_flow` is cleared). A protocol callback can also mark a packet to be flushed so that no further attempt is made to match the packet and it will be received directly by the stack.

Upon return from the gro_receive calls, if there is a flow in the device list that matches the packet (`same_flow` is set) then the packet is coalesced into the flow. Otherwise, the packet is either received normally (when marked to flush), or a new flow is created for the received packet.

Coalesced packets are sent into the stack for processing when `napi_gro_flush` is called. This is normally called at NAPI completion (NAPI is the soft interrupt processing for a receive network interrupt). For each large packet created, another series of chained callbacks (`gro_complete` functions) are called to finalize the GRO coalescing.

To support GRO with UDP encapsulation, `gro_receive` and `gro_complete` need to be called for the specific encapsulation method which essentially means that the functions need to be associated with a UDP port. To provide for this, a facility to register offload callback functions per UDP port was introduced [28]. In the UDP GRO functions (`udp_gro_receive` and `udp_gro_complete`), the destination port of a packet is looked up in the list of registered offloads. If a match is found, the corresponding `gro_receive` or `gro_complete` function for the UDP port is called. `udp_add_offload` and `udp_del_offload` are used to register and unregister the per port offload functions.

The `gro_receive` and `gro_complete` functions for an encapsulation method need to call the functions associated with the protocol of the encapsulated packet. If the protocol is carried in an additional encapsulation header (like in GUE), the packet can be parsed to retrieve this. If the protocol is saved in the receive socket (like in FOU), this value is passed by the UDP layer in the control buffer (`cb`) of the `sk_buff` for a packet. If the encapsulation method includes an encapsulation header, its fields should be considered when matching flows; the most straightforward implementation is to compare all the bytes in the encapsulation header.

Large Receive Offload. *Large Receive Offload* (*LRO*) is a NIC feature where packets of a TCP connection are coalesced in the NIC and delivered to the host as one large packet [29]. LRO is analogous to GRO, requires significant protocol awareness to be implemented correctly, and is difficult to generalize. The NIC must be informed of the port number for a supported UDP encapsulation method. Packets in the same flow need to be unambiguously identified. In the presence of tunnels or network virtualization, this may require more than a five-tuple match (packets for flows in two different virtual networks may have identical five-tuples). Additionally, a NIC needs to perform validation over packets that are being coalesced, and needs to fabricate a single meaningful header from all the coalesced packets.

The conservative approach to supporting LRO for UDP encapsulation would be to match packets to the same flow only if they were encapsulated exactly the same way and both the outer and inner headers match. That is the outer IP addresses, outer ports, inner protocol, inner IP addresses or Ethernet addresses, inner transport layer ports, and any additional encapsulation headers are all identical.

UDP Encapsulation protocols

Several protocols can be encapsulated over UDP in the Linux networking stack. L2TP (Layer two tunneling protocol) and ESP (Encapsulating Security Payload) may be configured to be encapsulated directly within UDP. VXLAN (Virtual Extensible LAN) and Geneve (Generic Network Virtualization Encapsulation) are relatively new encapsulation techniques targeted towards carrying Layer 2 packets for network virtualization. [30,31] Two generic encapsulation techniques of the Linux stack are Foo-over-UDP and Generic UDP Encapsulation; these are discussed in more detail below.

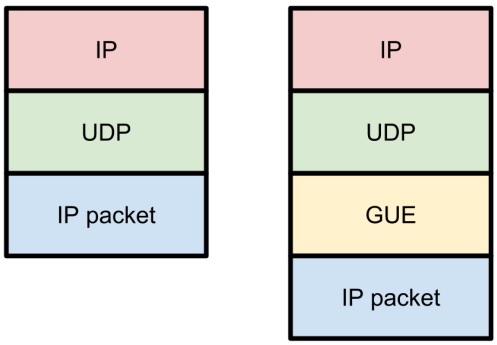

Figure 2. Protocol headers for encapsulation of an IP packet. The figure on the left depicts FOU encapsulation, the destination port in the UDP header implies that the UDP payload is an IP packet. The figure on the right illustrates GUE encapsulation, the proto/ctype field in the GUE header is set to 4 indicating IPv4 encapsulation.

Foo-over-UDP

Foo-over-UDP (*FOU*) is a feature of Linux which allows packets of any IP protocol to be directly encapsulated in UDP [32]. As depicted in figure 2, FOU does not define an

encapsulation header, so the protocol of an encapsulated packet is inferred from the destination UDP port. For instance, a server may open port 5555 which receives IP over UDP (IP packets encapsulated within UDP). A peer may send an encapsulated IP packet in UDP to this port with no additional encapsulation headers. Logically, the UDP header for FOU encapsulation is inserted in a packet on encapsulation, and removed on decapsulation.

The receive side of FOU is implemented in the `fou` kernel module. Configuration is really just a matter of setting up a UDP port to be the recipient of encapsulated packets. The "`fou`" subcommand of "`ip`" is intended for this purpose:

```
ip fou add port 5555 ipproto 4
```

The **port** keyword indicates the port number for the encapsulation and **ipproto** indicates the associated IP protocol. So this command sets aside port 5555, saying that packets arriving there will have IP protocol 4, which is IP encapsulation. Packets received on that port will have the encapsulating UDP header removed; and are then fed back into the network stack for IP layer processing of the inner packet.

Upon opening a FOU socket, the `encap_rcv` callback is set to `fou_udp_recv` which is the FOU UDP receive function. The configured IP protocol is kept in the private data for the socket. When receiving a packet the stack calls `fou_udp_recv` via `encap_rcv`. This function logically removes the UDP header in the packet by setting the `sk_buff`'s `transport_header` to refer to the UDP payload (encapsulated packet). The `total_length` field in the IP header is reduced by eight bytes for the UDP header, so that to the stack the resulting packet is an IP header followed by the encapsulated packet. `fou_udp_recv` returns the negative of the protocol number stored in the socket; this serves as an indication to the stack the packet should be re-injected for transport protocol processing with the protocol layer referred to by the returned number.

On the transmit side, IPIP, SIT, and GRE tunnels have been updated to allow FOU encapsulation. FOU is effectively treated as another attribute of a tunnel. A typical command to configure a tunnel with FOU encapsulation might look like:

```
ip link add name tun1 type ipip \
 remote 192.168.1.1 \
 local 192.168.1.2 \
 ttl 225 \
 encap fou \
       encap-sport auto \
       encap-dport 5555
```

This command will set up a new virtual interface (`tun1`) configured for IPIP encapsulation over UDP. The **encap** keyword indicates that the IP tunnel is being encapsulated (choices currently are **fou** and **gue**). **encap-sport** indicates the source port where **auto** as the argument means that the source port is is automatically set by the stack based on the inner flow identifier by calling `udp_flow_src_port`. **Encap-dport** indicates the destination port to use. In the this example the destination UDP port is 5555, and the source port is automatically set by the stack.

Generic UDP encapsulation

Generic UDP Encapsulation is a general method for encapsulating packets of arbitrary IP protocols within UDP [33,34]. GUE defines an encapsulation header which immediately follows the UDP header as shown in figure 2. The GUE header has an extensible format to allow carrying of optional data. This optional data potentially covers items such as virtual networking identifier, security data for validating or authenticating the GUE header, congestion control data, etc. GUE also allows private optional data in the encapsulation header; this can be used by a site or implementation to define custom optional data.

GUE Header Format. The header format for version 0x0 of GUE in UDP diagrammed in figure 3.

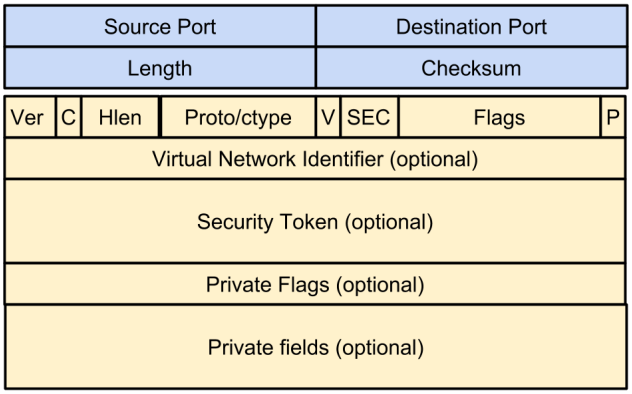

Figure 3. UDP and GUE headers in Generic UDP Encapsulation. The UDP header is always eight bytes, the GUE header is variable length composed of a fixed four byte header followed by optional data fields.

The contents of the UDP header are:

- **Source port**: (inner flow identifier): This should be set to a value that represents the encapsulated flow.
- **Destination port**: Set to port number for GUE.
- **Length**: Canonical length of the UDP packet (length of UDP header and payload).
- **Checksum**: Standard UDP checksum.

The GUE header consists of:

- **Version number**: Version number of the GUE protocol.

- **C**: Control bit. When this bit is set the payload is a control message, when not set the payload is a data message (encapsulated packet of an IP protocol).
- **Hlen**: Length in 32-bit words of the GUE header, including optional fields but not the first four bytes of the header. Note that each field in GUE has a size which is a multiple of thirty-two bits so there is no need for padding.
- **Proto/ctype**: When the C bit is set, this field contains a control message type in the payload. When C bit is not set, the field holds the IP protocol number for the encapsulated packet in the payload. The control message or encapsulated packet begins at the offset provided by Hlen.
- **Flags**: Header flags that may be allocated for various purposes and may indicate presence of optional fields. Undefined header flag bits must be set to zero on transmission.
- **P**: Private flag. Indicates presence of private flags field in the optional fields.
- **Fields**: Optional fields whose presence is indicated by corresponding flags.
- **Private flags**: An optional field indicated by the P bit. This field is a set of private flags which may in turn indicate presence of private fields.
- **Private fields**: Optional fields that are present when a corresponding bit in the private flags is set.

GUE characteristics. The protocol type in a GUE data message allows the use of any IP protocol number. This includes Layer 2 encapsulation (EtherIP), Layer 3 encapsulations of IPv4 and IPv6 packets, as well as encapsulation of layer 4 packets such as TCP or UDP. In the latter case, also referred to as transport mode encapsulation, the outer IP header is the header for the both the outer UDP and the encapsulated transport protocol packet; if the inner transport protocol has a checksum which includes an IP pseudo header, the pseudo header is based on the outer (only) IP header.

Flags and associated optional fields are the primary mechanism of extensibility in GUE. There are sixteen flag bits in the GUE header, one of which is reserved to indicate the presence of a private flags optional field.

A flag may indicate presence of optional fields. Fields contain optional data. Field sizes are multiples of thirty-two bytes, and the size of an optional field indicated by a flag must be fixed. Fields are processed in a manner similar to GRE processing. They are arranged in the packet in the order of the flags that indicate their presence, the offset of a particular field is determined by the sum of all the sizes of preceding fields that are present in the header.

Flags and fields had been defined for network virtualization identifiers, security data, GUE header checksum, and remote checksum offload.

GUE implementation. Similar to FOU, the Linux GUE implementation separates the transmit and receive path.

The receive side of GUE is implemented in the `fou` module. Upon opening a GUE socket, the `encap_rcv` callback is set to `gue_udp_recv` to receive packets encapsulated using GUE. Configuration is performed by using the "`fou`" subcommand of "`ip`" with **gue** as a parameter:

```
ip fou add port 7777 gue
```

This command sets aside port 7777, saying that packets arriving there are encapsulated with GUE as indicated by the **gue** keyword. Packets received on that port are processed by `gue_udp_recv`. They are verified to be valid GUE packets, and if the GUE header contains options these are processed also. If the packet cannot be verified, required security credentials are not present, or the encapsulation is otherwise malformed then the packet is dropped. For an acceptable packet, the encapsulating UDP and GUE headers are removed by adjusting `transport_header` in the `sk_buff` to refer to the encapsulated packet, and the packet is then fed back into the network stack for processing the inner packet similar to FOU handling. The IP protocol of the encapsulated packet is taken directly from the `proto/ctype` field in GUE header, this can conceptually be any legal IP protocol number.

Similar to FOU, on the transmit side the IPIP, SIT, and GRE tunnels have been updated for GUE encapsulation.

A typical configuration command might look like:

```
ip link add name tun1 type ipip \
     remote 192.168.1.1 \
     local 192.168.1.2 \
     ttl 225 \
     encap gue \
           encap-sport auto \
           encap-dport 7777 \
           encap-udp-csum \
           encap-remcsum
```

This command will set up a new virtual interface (`tun1`) configured for IPIP encapsulation using Generic UDP Encapsulation. The destination UDP port is 7777, and the source port is automatically set by the stack. Packets sent on this tunnel are encapsulated in a UDP and GUE header where the next protocol in the GUE header is set to 4 (for IPIP). In this example the UDP checksum and remote checksum offload are also enabled by the **encap-udp-csum** and **encap-remcsum** keywords

Conclusion

Support for UDP encapsulation is an impressive achievement in the Linux networking stack and is the result of a broad community effort with many contributors.

Further development, new encapsulation features and protocols, and new use cases for encapsulation will continue to contribute to the importance and utility of UDP encapsulation in the data center.

Acknowledgements

We would like to thank Willem de Bruin and Nandita Dukkipati for their valuable feedback.

References

1. Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980
2. Perkins, C., "IP Encapsulation within IP", RFC 2003, October 1996
3. Gilligan, R. and E. Nordmark, "Transition Mechanisms for IPv6 Hosts and Routers", RFC 1933, April 1996
4. Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000
5. Lau, J., Ed., Townsley, M., Ed., and I. Goyret, Ed., "Layer Two Tunneling Protocol - Version 3 (L2TPv3)", RFC 3931, March 2005
6. Housley, R. and S. Hollenbeck, "EtherIP: Tunneling Ethernet Frames in IP Datagrams", RFC 3378, September 2002
7. “Linux 3.18", *Linux Kernel Newbies*, December 2014, http://kernelnewbies.org/Linux_3.18
8. “skbuff.h”, Linux source file (version 3.18 of Linux) http://lxr.free-electrons.com/source/include/linux/skbuff.h
9."Equal-cost multi-path routing", *Wikipedia*, http://en.wikipedia.org/wiki/Equal-cost_multi-path_routing
10. “Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS”, *Microsoft*, http://download.microsoft.com/download/5/D/6/5D6EAF2B-7DDF-476B-93DC-7CF0072878E6/NDIS_RSS.doc
11. de Bruijn, W., Herbert, T., "Scaling in the Linux Networking Stack", https://www.kernel.org/doc/Documentation/networking/scaling.txt
12. "ethtool(8) - Linux man page", *Linux man pages* http://linux.die.net/man/8/ethtool
13. Herbert, T., “udp: Add function to make source port for UDP tunnels”, *Linux commit*, July 2014, https://patchwork.ozlabs.org/patch/366245/
14. Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, November 2011
15. Herbert, T., “ipv6: Implement automatic flow label generation on transmit”, *Linux commit*, July 2014, https://patchwork.ozlabs.org/patch/366248/
16. Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, April 2013
17. Herbert, T., “udp: Generic functions to set checksum”, *Linux commit*, June 2014, https://patchwork.ozlabs.org/patch/356132/
18. “skb_postpull_rcsum (9)”, *Linux man pages*, http://dev.man-online.org/man9/skb_postpull_rcsum/
19. Herbert, T., *Linux commit*, “net: Infrastructure for checksum unnecessary conversions”, *Linux commit*, August 2014, https://patchwork.ozlabs.org/patch/384592/
20. Herbert, T., “Remote checksum offload for encapsulation”, *Internet-draft*, November 2014, https://tools.ietf.org/html/draft-herbert-remotecsumoffload-01
21. Herbert, T., “gue: Remote checksum offload” http://permalink.gmane.org/gmane.linux.network/336556
22. Herbert, T., “Remote checksum offload for VXLAN”, *Internet-draft*, December 2014, https://tools.ietf.org/html/draft-herbert-vxlan-rco-00
23. Herbert, T., “vxlan: Remote checksum offload”, *Linux commit*, January 2015, https://patchwork.ozlabs.org/patch/428200/
24. Xu, H., “GSO: Generic Segmentation Offload”, June 2006 http://lwn.net/Articles/188489/
25. “Large Segment Offload”, *Wikipedia*, http://en.wikipedia.org/wiki/Large_segment_offload
26. Gross, J., “net: Generalize ndo_gso_check to ndo_features_check”, *Linux* commit, December 2014, https://patchwork.ozlabs.org/patch/423854/
27. Corbet, J, “JLS2009: Generic receive offload”, October 2009, https://lwn.net/Articles/358910/
28. Gerlitz, O., “net: Add GRO support for UDP encapsulating protocols”, *Linux* commit, January 2014, http://patchwork.ozlabs.org/patch/312516/
29. “Large Receive Offload”, *Wikipedia*, http://en.wikipedia.org/wiki/Large_receive_offload
30. Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", RFC 7348, August 2014
31. Gross, J., Sridhar, P., Wright, C., Ganga, I., Agarwal, P., Duda, K., Dutt, D., Hudson, J., “Geneve: Generic Network Virtualization Encapsulation”, *Internet-draft*, October 2014, https://datatracker.ietf.org/doc/draft-gross-geneve/
32. Corbet, J., “Foo over UDP”, October 2014, http://lwn.net/Articles/614348/
33. Herbert, T., and Yong, L., “Generic UDP Encapsulation”, *Internet*-draft, October 2014, https://tools.ietf.org/html/draft-herbert-gue-02
34. Herbert, T. “net: Generic UDP Encapsulation”, October 2014, http://lwn.net/Articles/615044/

Author Biography

Tom Herbert is a software engineer at Google working on content ads indexing, Linux kernel networking, and networking protocol

development. He is an active contributor to Linux netdev as well as a participant in the IETF.