Adventures in TCP/IP

Introduction to TCP Large Receive Offload

BY RANDALL STEWART AND MICHAEL TÜXEN

TCP Large Receive Offload (TCP LRO) is a protocol-specific method to minimize the CPU resources used for receiving TCP segments. It is also implementation specific, and this article describes its implementation in the FreeBSD kernel. At any given time, TCP is often used for unidirectional communication, although TCP provides a bidirectional channel. This is the case, for example, if the application protocol using TCP as its transport protocol is of the request/response type like HTTP.

TCP LRO can reduce the CPU resources required in a number of ways including:

- Combining acknowledgments so that a single large stretch acknowledgment is delivered to the TCP stack instead of multiple smaller acknowledgments. This applies to the case where the TCP endpoint is mostly sending user data.
- Combining multiple inbound data segments into one, big, larger piece of data. This helps if the TCP endpoint is mostly receiving user data.
- Bypassing parts of IP stack processing. Therefore, it is useful for TCP LRO to intercept the packets at the network interface layer.

All of these methods are focused on cutting down the number of times the TCP stack gets called and/or minimizing the number of cache misses that the CPU will have to take by compressing all of the processing into one, or a series of packets, processed together. For most all FreeBSD drivers, a single software TCP LRO process is used, though some specific hardware and its drivers do support hardware TCP LRO. This article will discuss only the software TCP LRO in FreeBSD.

At any given time, TCP is often used for unidirectional communication.

Evolution of TCP LRO

The initial implementation of TCP LRO was implemented by Andrew Gallatin in 2006 and was specific to the mxge(4) driver. It was then made generic to all drivers by Jack Vogel in 2008. It had only two focuses:

1. Collecting and merging together small inbound data segments to present a larger, single, inbound data segment to TCP, or,
2. Collecting a number of acknowledgments and presenting one, single, larger acknowledgment to the TCP stack.

Both methods were implemented to cut down on the number of times the TCP receive path was called to save CPU resources. Its implementation was very careful to only handle consecutive segments and ones without TCP options (the only allowed TCP option was the timestamp option). The initial implementation remained pretty much untouched in FreeBSD for almost a decade, except for the addition of IPv6 support by Bjoern A. Zeeb in 2012.

The Addition of Sorting

By 2016 the TCP LRO code was starting to show its age, with the ever faster NICs being deployed on both clients and servers, more and more packets were arriving on each driver interrupt. The initial implementation only allowed for eight different connections to have data collected and compressed. This worked fine in workloads with only a few connections, but was less effective for workloads with a large number of connections, since a driver was sending in far more packets from different connections on each interrupt. With so many more packets from multiple connections arriving in an interrupt, the chances of a single connection seeing packets with small enough interleaving to fit in the eight connection limit grew less and less to the point where TCP LRO was rarely effective, especially for the server case.

By 2016 the TCP LRO code was starting to show its age.

This is when Hans Petter Selasky had a brilliant idea, he added an optional path for a driver to call that would sort the inbound packets before submitting them to TCP LRO. This meant that all packets arriving from each connection could be processed together. Which then meant that you maximized TCP LRO's effectiveness on each interrupt. This change vastly improved TCP LRO performance while still allowing older drivers to remain unchanged.

Packet Queuing

As TCP LRO became more effective, other problems with this more efficient path began to show up including:

a. TCP's congestion control prefers to see every acknowledgment, since an acknowledgment advances its congestion window. Compressing acknowledgments can hamper the congestion control algorithm.
b. Modern TCP stacks often would like to have precise Round Trip Time (RTT) information, compressing multiple acknowledgments can hide this information from TCP.
c. Implementations of TCP ECN needed to see the IP header bits so that ECN signaling from the network can be monitored and reacted to, compressing data or acknowledgments effectively hides this information.
d. If a TCP stack is pacing packets (We will discuss pacing packets in a future column.), then processing a series of acknowledgments when the stack is prohibited from sending out packets increases overhead. This is because the acknowledgment can't send

and yet results in a number of cache misses on the TCP stack during its processing, which will then have to be repeated when the stack is allowed to output.

This set of problems brought about another optimization where the TCP stack enables the TCP LRO code to directly queue packets to it for processing when it next awakens. This then allows all the data in the IP and TCP headers to be processed at a single time when the stack can send out data and reveals all the information (including the timing due to receive timestamps being added either in hardware by the NIC, or in software in the TCP LRO code) that TCP wants to see.

Compressed Acknowledgments

This new queuing mechanism worked well but also caused an additional set of cache misses when a series of acknowledgments arrived. This is because each packet in queue to the stack results in a cache miss when it is processed. In the old compressed scheme, information was lost but superior optimization was performed, since only one cache miss would occur for some number of arriving acknowledgments.

This brought about another TCP LRO optimization. When consecutive acknowledgments arrive, the TCP LRO code can now compress them into a special packet that holds an array of the arriving packet information. This compression technique allows all the previously lost data (including arrival times) to be presented to the TCP stack in the array structure so that only one cache miss is taken to access the special packet. Note that a TCP stack must signal the TCP LRO code that it supports this special type of processing.

Each packet in queue to the stack results in a cache miss when it is processed.

Inner and Outer Headers

The last set of optimizations to TCP LRO have to do with the way inbound IP packets are examined. Originally, only Ethernet frames containing a TCP segment using IPv4 or IPv6 were supported. To support other encapsulations of TCP segments, for example VXLAN which makes it possible to encapsulate an Ethernet frame into a UDP packet, the packet parsing was generalized to support an inner and outer header. This way, packets with UDP as an outer header and TCP as an inner header can be processed by TCP LRO. This assumes that the NIC can do the checksum offloading for both protocols.

Management of TCP LRO

If a NIC driver supports TCP LRO, it can be enabled or disabled using the `lro` or `-lro` parameter of `ifconfig`.

A NIC driver must contain a `struct lro_ctrl`, which contains in addition to other fields a pointer to:

- An array of pairs consisting of a pointer to `struct mbuf` and a sequence number. The number of these pairs is `lro_mbuf_max`.
- A number of `struct lro_entry`. The number of these entries is `lro_cnt`.

The `struct lro_entry` is used to store the information about one aggregated set of received TCP segments. If such an entry is not used, it is contained in the `lro_free` list. When it is used, it is contained in the `lro_active` list and also accessible via the hashtable `lro_hash`.

These two lists and the hash table are also contained in `struct lro_ctrl`.

There are two ways for a NIC driver to initialize the TCP LRO specific data. The classical way is to call the `tcp_lro_init()` function. The number `lro_cnt` of `struct lro_entry` which should be allocated, is specified by the loader tunable `net.inet.tcp.lro.entries`. When using the classical way of initialization, the array of pairs has no entries. The modern way is to use the function `tcp_lro_init_args()` which allows the caller to specify the `lro_cnt` and `lro_mbuf_max`. This means that the array of pairs might also be allocated.

No matter which way was used for initializing the `struct lro_ctrl`, calling the function `tcp_lro_free()` frees all allocated resources.

Passing TCP Segments to TCP LRO

The NIC driver has a classical and a modern way of trying to pass a TCP segment to TCP LRO. If passing the TCP segment over to TCP LRO fails, the NIC driver must continue the normal processing of the TCP segment. One reason for TCP LRO to fail is if the NIC was not able to verify the checksums on the received IP packet.

There are two ways for a NIC driver to initialize the TCP LRO specific data.

To use the classical way to pass the TCP segment to TCP LRO, the NIC driver calls `tcp_lro_rx()`. Basically this starts the processing done by `tcp_lro_rx_common()`, which is described in the next subsection. The modern way to pass TCP segments to TCP LRO, which also requires the modern way of initialization, is to call `tcp_lro_queue_mbuf()`. This function just computes a sequence number for the TCP segment and stores it in combination with the TCP segment in the next free entry of the array of pairs. If the array becomes full by this operation, `tcp_lro_flush_all()` is called which is also described in the next subsection.

No matter whether the classic or modern way of passing TCP segments to TCP LRO is used, the time when the TCP segment is passed to TCP LRO is saved if there is no hardware receive time from the NIC available.

Processing TCP Segments in TCP LRO

When the modern way of passing TCP segments to TCP LRO is used, one additional initial step is done. `tcp_lro_flush_all()` sorts all entries in the array of pairs based on the sequence number field. This results in all TCP segments for the same TCP connection being most likely located next to each other in the array and in the sequence they were received. Then `tcp_lro_rx_common()` is called for all the entries in the array. From now on, the processing of the TCP segments is the same, no matter whether the classic or modern way of passing them to TCP LRO is used.

`tcp_lro_rx_common()` parses the TCP segment and uses that information to lookup the corresponding entry of type `struct lro_entry` in the hashtable. If such an entry is found, the TCP segment will be added to the packet chain of TCP segments. If no entry is found, a new one is created and the TCP segment is added to the entry. Note that when the TCP LRO code runs out of free entries then an older entry is flushed which then frees up that structure to be reused for the new allocation.

The NIC driver or the TCP LRO code itself can trigger a flush operation, which will result in processing the information in the entries of type `struct lro_entry` such that it is suitable to be processed by the TCP stack as described in the next subsection.

Passing Information from TCP LRO to the TCP Stack

If alternate TCP stacks like the TCP RACK or the TCP BBR stack are used, the High Precision Timer System (HPTS) is employed. If only the FreeBSD base TCP stack is used, this is not the case.

If the HPTS is not loaded in the FreeBSD kernel, the following will happen in case a flush operation is triggered: TCP LRO will combine the packet chain for an entry of type `struct lro_entry` into a single large TCP segment by concatenating all the user data of the individual TCP segments. Of course, this only works if there are no gaps or overlaps. If that happens, TCP LRO might only combine smaller parts. The information about the acknowledged data will also be combined and this large, generated TCP segment will be injected into the interface layer. This results in less packets needing to be processed, but results also in the loss of the information when the individual TCP segments were received, as well as any IP-level ECN bits. Depending on the congestion control or loss recovery, this can have a negative impact.

If the HPTS system is loaded, a flush operation results in a lookup of the TCP endpoint. This information is used to determine if the TCP stack used by the TCP endpoint supports mbuf-queueing. If it does not, the same processing as for the FreeBSD base stack is performed. If the TCP stack supports mbuf-queueing, but not compressed ACKs, the packet chain of the entry is copied over to the TCP endpoint and the TCP endpoint might be triggered to process that packet chain. This is what is done when the TCP BBR stack is used which supports mbuf-queueing but not compressed ACKs. If the TCP RACK stack is used, which also supports compressed ACKs, multiple ACKs, which have been received in sequence, can be stored in a special data structure, which allows passing them in a more memory-efficient way. Please note that when mbuf-queueing and compressed ACKs are used, the information from when the individual packets were received is preserved and passed to the TCP endpoint.

Future Evolution

Accurate ECN for TCP is a TCP feature currently being specified by the Internet Engineering Task Force (IETF) and support for it is under development for FreeBSD. In addition to using two new TCP options, it changes the use of two existing TCP flags and makes use of one additional flag. This requires changes to the TCP LRO code to still allow the aggregation of incoming TCP segments for TCP connections supporting Accurate ECN.

The VXLAN support can also be improved to make use of mbuf-queueing.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.