Stacked Vlan: Performance Improvement and Challenges

Toshiaki Makita

NTT
Tokyo, Japan
makita.toshiaki@lab.ntt.co.jp

Abstract

IEEE 802.1ad vlan protocol type was introduced in kernel 3.10, which has encouraged use of stacked vlan on Linux. With this, people can integrate Linux servers into Ethernet VPN, e.g. Metro Ethernet, which is a typical use case of 802.1ad.

802.1ad and stacked vlan suffer from several problems around performance and interoperability, as discussed in the previous Netdev conference, 0.1.

Since then, performance has been significantly improved by modifying kernel networking core to enable offloading features such as checksum offload, GSO and GRO when handling stacked vlan. However, fixing kernel networking core is not sufficient. There are several important features that require NIC/driver-level support, like RSS, TSO. We introduce how kernel networking core handles offloading features with stacked vlan frames, and clarify what kind of modification is needed for NICs/drivers to enable missing key features.

Solution of the interoperability problem is still a work in progress. The problem is that double tagged frames are dropped by default due to oversize error on NICs that receive them. We propose an approach for this problem, which introduces software implementation of envelope frames defined in IEEE 802.3as, leveraging jumbo frames support of NICs.

Keywords

Stacked vlan, 802.1ad, Ethernet VPN, Metro Ethernet, encapsulation, 802.3as, envelope frames, MTU, hardware acceleration.

Introduction

Stacked vlan is a technique that uses two or more vlan headers in one Ethernet frame. This usage of vlan headers was first introduced by network switch vendors, and later standardized by IEEE 802.1ad[1], which defines the Ethernet type of the outer vlan header, 0x88a8, while the normal Ethernet type of vlan header is 0x8100 (Figure 1). Linux has supported 802.1ad Ethernet type since kernel 3.10, by which users can make use of stacked vlan.

However, 802.1ad and stacked vlan had problems around performance and interoperability. These problems were discussed in the previous edition of Netdev, 0.1[2]. Since then, performance has been gradually improved, while the interoperability issue is yet to be resolved.

In this paper, we describe how the performance problem is being fixed, show its challenges, and propose an approach for the interoperability problem.

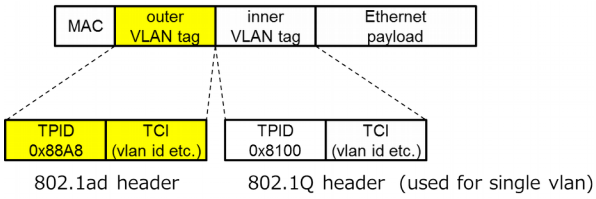

Figure 1:802.1ad and 802.1Q vlan header format

Performance issues

Background

In the early days of 802.1ad in Linux, using stacked vlan required that most hardware and software acceleration had to be disabled and resulted in poor performance. Now many of these issues have been improved and can be enabled with stacked vlan, while some are still unusable. We show the situation of each acceleration technologies with stacked vlan in the following sections.

Acceleration on Tx

Acceleration on Tx includes Tx-checksum, GSO, and TSO. Since these features were not covered in `vlan_features` field of vlan devices, they were disabled on vlan devices on top of other vlan devices, i.e. stacked vlan devices. Also it was not wise to just set those feature bits in kernel due to a number of undiscovered bugs and lack of key infrastructure in kernel.

The situation of each features is as follows.

- **Tx-Checksum** Tx-checksum support is classified into two types: IP_CSUM (and IPV6_CSUM), with which devices are able to compute checksum for only IP/IPv6/TCP/UDP packets, and HW_CSUM, with which devices are able to compute checksum (one's complement sum) regardless of protocols. Stacked vlan did not go well with some NICs with IP_CSUM. Specifically, their drivers were not able to parse encapsulated protocols if packets were tagged with two or more vlan headers, thus failed to set descriptor fields related to checksum, with appropriate values. Most of them depended on a certain library function in kernel named `vlan_get_protocol()`, and this function worked only when packets had single vlan tag.

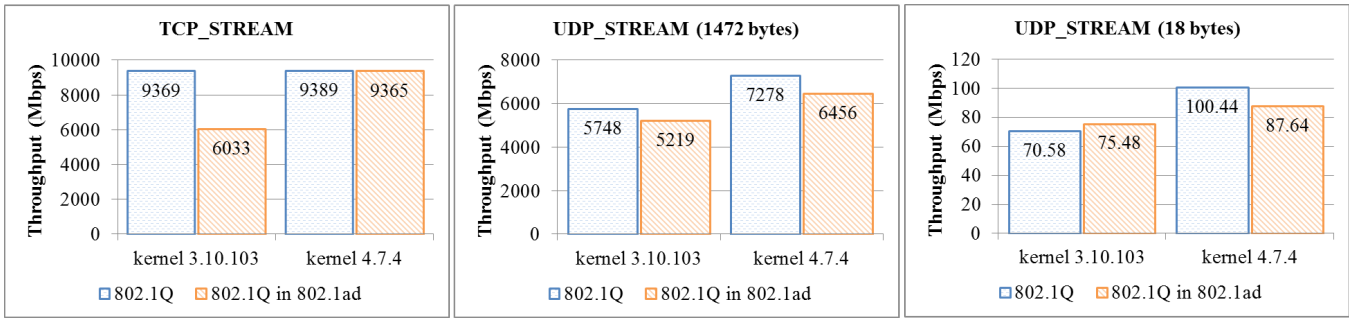

Figure 2: Performance changes of stacked vlan between kernel 3.10.103 and 4.7.4 (single-flow)

Fixing this function has made those malfunctioned drivers work correctly[3].

- **GSO** Network stack in kernel has had an ability to parse and segment multiple vlan tagged packets by software. However, the logic to detect multiple vlan headers was incorrect, due to which those packets were not segmented. By fixing the bogus logic, GSO has come to work correctly[4].
- **TSO** Network stack in kernel did not have infrastructure to check if devices were able to perform TSO for multiple-tagged packets. Thus there was no choice but to segment packets by software (GSO) and TSO was never performed. Now that infrastructure to check if devices can perform TSO for packets with multiple tags has been introduced[5] and some drivers have leveraged this feature, TSO can work with those drivers.

Acceleration on Rx

As opposed to Tx, in which kernel (drivers) can provide certain hints such as IP header offset, Rx depends more on device capabilities to enable offload features. Nevertheless, there are pure software acceleration technologies like RPS and GRO that can work with stacked vlan only by modifying kernel network stack.

The situation of each features are as follows:

- **Rx-Checksum** Rx-checksum works fine if devices have HW_CSUM, while IP_CSUM requires devices to have an ability to parse multiple vlan tags. Any particular work has not been done to harmonize IP_CSUM devices with stacked vlan.
- **GRO** GRO did not support vlan tags that were not stripped by hardware offload, which are common for stacked vlan. A new feature that handles non-offloaded vlan tags in GRO has been introduced[6] and now GRO works with stacked vlan.
- **RSS** RSS is a hardware feature and requires devices to have an ability to parse multiple vlan headers. Any particular work has not been done to enable RSS for stacked vlan.
- **RPS** `skb_flow_dissect()` used by RPS did not support 802.1ad protocol, thus RPS did not work for stacked vlan. This has been fixed[7] and now RPS works with 802.1ad or stacked vlan.

To summarize the situation, at this point pure software acceleration techniques such as GSO, GRO and RPS work well with stacked vlan. Features that cooperate with hardware such as Tx-Checksum, TSO, Rx-Checksum, and RSS work depending on devices.

Performance Changes

We measured performance changes of stacked vlan on Linux, using 2 machines connected back-to-back with pause disabled. Other information on the machines are as follows:

- CPU: Intel Xeon E5-2650 v3, 2.30GHz, 2-socket, 10-core each.
- NIC: Intel 82599ES 10-Gigabit (ixgbe driver).

We used netperf[8] and super_netperf[9] to measure throughput for single-flow and multi-flow respectively. Multi-flow test uses 100 flows. Each test is done on kernel 3.10.103, which is the version 802.1ad was introduced in, and 4.7.4, which is the latest stable kernel as of September 23, 2016.

- **Single-flow** Figure 2 shows performance numbers of single 802.1Q vlan device and stacked vlan device (802.1Q vlan device on 802.1ad vlan device). TCP_STREAM test shows performance improvement of stacked vlan device, from 6033 Mbps to 9365 Mbps, reaching wire speed. This is because the NIC has TSO for stacked vlan in 4.7.4, and GRO for stacked vlan is also enabled in 4.7.4. UDP_STREAM (1472 bytes and 18 bytes) is not that good but improved to some extent. UDP does not benefit from either TSO or GRO, but Tx-checksum is effective.
- **Multi-flow** Figure 3 shows performance numbers of single 802.1Q vlan device and stacked vlan device, comparing single-flow and multi-flow (100 flows) to see scalability for the number of CPU cores. Our machines have 20 cores in total, thus ideally the multi-flow performance should be 20 times better than single-flow. However, the

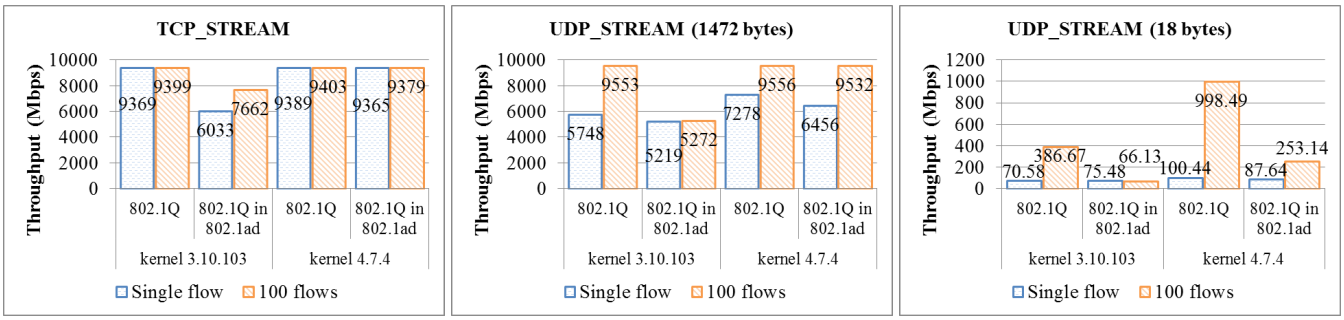

Figure 3: Performance changes of stacked vlan between kernel 3.10.103 and 4.7.4 (multi-flow)

NIC we use does not support RSS for stacked vlan, thus scalability on receiver side is limited. 4.7.4 can leverage RPS as a workaround, while 3.10.103 cannot. TCP_STREAM tests in 3.10.103 shows some scalability for stacked vlan (802.1Q in 802.1ad). UDP_STREAM (1472 bytes and 18 bytes) in 3.10.103 does not scale, while 4.7.4 scales to some extent thanks to RPS. Nevertheless, in any tests, stacked vlan is not comparable to single vlan (802.1Q), which can leverage RSS.

Remaining Work

As described in the previous section, it depends on devices whether Tx-Checksum, TSO, Rx-Checksum, and RSS work with stacked vlan.

- **Tx-Checksum** Kernel networking stack determines that devices can perform Tx-checksum if HW_CSUM or IP_CSUM is set in the `vlan_features`. Drivers with IP_CSUM in `vlan_features` need to inform devices of the correct offset of IP header in multiple-tagged packets, otherwise packets end up with incorrect checksum. Drivers that leverage `vlan_get_protocol()` should do that correctly, but others need to carefully check the header offset. If drivers are not able to inform devices of IP header offset, they need to disable the IP_CSUM feature for stacked vlan in their `ndo_features_check()` callback functions.
- **TSO** Kernel networking stack determines that devices can perform TSO if TSO is turned on in the `vlan_features`. However, by default, TSO for stacked vlan is disabled in `vlan_features_check()` called from within `dflt_features_check()`, which is used when drivers do not implement `ndo_features_check()`. Drivers that are able to perform TSO for stacked vlan (drivers that can handle arbitrary offset of IP header) need to implement `ndo_features_check()` in order not to call `dflt_features_check()` as well as to inform devices of IP header offset correctly. The easiest way to implement `ndo_features_check()` only for enabling TSO for stacked vlan is to set `ndo_features_check()` to `passthru_features_check()`, which essentially does no checking. Figure 4 shows an example implementation of igb driver.

```
static const struct net_device_ops
igb_netdev_ops = {
        ...
        .ndo_features_check
                = passthru_features_check,
};
```

Figure 4: Example implementation of igb driver to enable stacked vlan TSO.

- **Rx-Checksum** Unlike Tx-checksum, kernel networking stack cannot provide a hint such as IP header offset per packet, thus devices need to parse multiple vlan headers of received packets. Some devices have knobs to change their mode into some kind of “Stacked vlan mode”. One example is Intel 10 Gbit devices, which can parse stacked vlan packets when a certain register bit is set (it does not allow hardware acceleration for packets with single or no vlan instead). However, networking stack does not have a feature to enable such a mode.
- **RSS** Similarly to Rx-checksum, devices need to parse multiple vlan headers of received packets. Intel 10 Gbit devices are also able to parse stacked vlan packets when a certain register bit is set, but networking stack does not have a feature to enable such a mode.

Interoperability issue

Problem Statement

Stacked vlan suffers from an interoperability problem, i.e. stacked vlan packets are dropped by NICs due to oversize error.

Vlan is different than other encapsulation protocols. Here packet size is generally larger than normal packets by

vlan header size (4 bytes). Thus, most NICs allow packets the size of which is larger by 4 bytes than MTU. Similarly, when doubly tagged vlan is used leveraging 802.1ad, the packet size will be larger by twice the size of vlan header (8 bytes). This packet size is required to provide Ethernet VPN transparent to the users. Thus, hardware switches support 1508 bytes MTU when using 802.1ad, as suggested in MEF 26.2[10]. Linux stacked vlan devices also have 1500 bytes MTU, which emit 1508 bytes doubly tagged packets. But some NICs do not accept 1508 bytes packets by default, and they are dropped.

Solutions

There are roughly two ways to address this problem. One is to modify MTU of vlan devices or physical devices. The other approach is to introduce software implementation of envelope frames defined in IEEE 802.3as[11] into Linux. Envelope frames are frames which have certain encapsulation headers and have maximum length of 2000 bytes, keeping the original MTU as 1500. Linux can support envelope frames by changing the maximum acceptable packet size on physical devices to greater value than 1518 including Ethernet header and trailers, while not changing 1500 bytes MTU of physical devices in kernel network stack.

We break down these 2 approaches into 5 specific ones.

- **Reduce vlan device MTU** Reduce vlan device MTU from 1500 to 1496 or less according to the number of stacked vlan devices.
- **Increase physical device MTU** Increase physical device MTU from 1500 to 1504 or more according to the number of stacked vlan devices.
- **Accept envelope frames always** Always allow devices to receive packets the maximum size of which is 2000, without changing MTU in kernel network stack.
- **Add API to enable envelope frames** Add an API to allow devices to receive packets the maximum size of which is 2000, without changing MTU in kernel network stack.
- **Add API to set acceptable envelope frames header size** Add an API to set acceptable maximum size of frames, without changing MTU in kernel network stack.

The last four solutions require jumbo frames support of devices, while the last three ones do not allow to send jumbo frames. The requirement comes from the fact that in most cases NICs themselves do not support envelope frames, which requires jumbo frames support of NICs even when handling only envelope frames.

Analysis of Solutions

- **Reduce vlan device MTU** This decreases sending packet size but not receiving packet size, thus does not solve the problem.
- **Increase physical device MTU** This enables devices to receive multiple tagged packets and solves the problem, but has side-effect that devices can send jumbo frames which can be dropped by network elements that receives the frames. Thus, this approach is suboptimal.
- **Accept envelope frames always** This enables devices to receive multiple tagged packets and solves the problem, but can have side-effect. One example is e1000e, which behaves differently when the maximum acceptable packet size is greater than 1522. Thus, this approach is suboptimal.
- **Add API to enable envelope frames** This enables devices to receive multiple tagged packets and solves the problem, but can have the same side-effect as the previous approach only when this feature is enabled. Another possible side-effect happens when a device does not support 2000 bytes maximum packet size but supports certain size less than 2000 and greater than 1518. If we make the API fail in such a case, Linux users cannot use the maximum size. If we make the API succeed, we need an additional API to expose the accepted header size.
- **Add API to set acceptable envelope frames header size** This enables devices to receive multiple tagged packets and solves the problem, but can have side-effect that changes behavior of devices. Nevertheless, the side effect is minimal because it can happen only when the acceptable size is really needed. Thus this is optimal.

To summarize, the last solution (Add API to set acceptable envelope frames header size) is the best way to solve this problem.

Implementation

We describe the envisioned implementation of the best approach below.

- **Kernel** Envelope frames require normal packets to use 1500-sized MTU, while encapsulation headers can be added to the MTU. Thus we need to increase the maximum acceptable frame size of devices without changing `mtu` of struct `net_device`. In order to achieve this, add a new function, `ndo_set_env_hdr_len`, in struct `net_device_ops`, through which kernel can inform device drivers of needed additional header size of envelope frames (`env_hdr_len`). Implementation in device drivers would be as simple as replacing `netdev->mtu` with `netdev->mtu + env_hdr_len`. This makes devices recognize `netdev->mtu + env_hdr_len` as MTU, and allow packets with additional headers up to `env_hdr_len`, while kernel networking stack keeps on recognizing `netdev->mtu` as MTU. Thus no packets larger than MTU will be sent other than those encapsulated by upper devices.

- **Userspace** Since `env_hdr_len` is similar to MTU, it is natural for the user interface to resemble MTU. In this case, userspace would send information of the maximum envelope header size as a netlink attribute. Command would be like figure 5 when we implement the feature in `ip` in the iproute2 utility (the figure shows an example of adding 8 bytes additional header for stacked vlan).

```
# ip link set eth0 envhdrlen 8
```

Figure 5: Setting 8 bytes envelope header length by `ip` command.

Future Work

In the future 802.1ad vlan devices should care this problem in kernel and adjust `env_hdr_len` automatically. This can be achieved by notifying drivers of `env_hdr_len` on creating 802.1ad devices.

Conclusion

Recent work for stacked vlan achieved performance increase. TCP reaches wire speed with 1 core and UDP tests show increase in throughput as well. Scalability for the number of cores are also improved thanks to RPS support. Nevertheless, there still remains work to get even better performance in device drivers.

We also raised an interoperability problem, a solution for it, and future work.

We expect discussion on these topics will be more active and drivers' implementation makes progress, leading to improvement of performance and the interoperability problem.

References

[1] IEEE. 2005. IEEE standard for local and metropolitan area networks – virtual bridged local area networks, amendment 4: Provider bridges. *IEEE Std 802.1ad-2015.*

[2] Toshiaki Makita, "802.1ad HW acceleration and MTU handling.", *Netdev 0.1 BoF session slides, 2015.* http://www.netdevconf.org/0.1/sessions/18.html.

[3] Toshiaki Makita, "Fix checksum error when using stacked vlan", commit 08178e5ac45b, January 31, 2015, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=08178e5ac45b

[4] Toshiaki Makita, "net: Fix stacked vlan offload features computation", commit 796f2da81bea, December 24, 2014, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=796f2da81bea

[5] Toshiaki Makita, "Stacked vlan TSO", commit afb0bc972b52, March 29, 2015, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=afb0bc972b52

[6] Toshiaki Makita and David S. Miller, "vlan: Add GRO support for non hardware accelerated vlan", commit 66e5133f19e9, June 1, 2015, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=66e5133f19e9

[7] Eric Dumazet, "net: flow_dissector: add 802.1ad support", commit e11aada32b39, August 9, 2013, http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e11aada32b39

[8] Rick Jones, "The Netperf Homepage", netperf.org website, accessed September 23, 2016, http://www.netperf.org/netperf/

[9] Daniel Borkmann, "GitHub - borkmann/stuff: Random useful scripts e.g. for netperf", GitHub website, accessed September 23, 2016, https://github.com/borkmann/stuff

[10] The MEF forum 2016. ENNI maximum frame size multilateral attribute [R27], in section 10.3 of External Network Network Interfaces (ENNI) and operator carrier service attributes. *Technical specification MEF 26.2, August 2016.*

[11] IEEE. 2006. IEEE standard for information technology – telecommunications and information exchange between systems – local and metropolitan area networks, specific requirements part 3: Carrier sense multiple access with collision detection (CDMA/CD) access method and Physical layer specifications. *IEEE Std 802.3as-2006.*

Author Biography

Toshiaki Makita works for NTT Open Source Software Center, where he has been providing technical support for Linux kernel for almost 5 years, contributing to Linux network subsystems focusing mainly on bridge and vlan functions. Formerly, he used to be a research and development engineer focusing on Ethernet VPN at NTT West, which is a regional carrier in NTT group.