Contribution ID: **142** Type: **not specified**

DSA switches: domesticating a savage beast

Tuesday, 21 September 2021 10:20 (40 minutes)

The DSA subsystem was originally built around Marvell devices, but has since been extended to cover a wide variety of hardware with even wider views of their management model. This presentation discusses the changes in DSA that took place in the last years for this wide variety of switches to offer more services, and in a more uniform way, to the larger network stack.

Summarized, these changes are:

- Acknowledging switches which only have DSA tags for control plane packets, and modifying the bridge driver to accept termination of data plane packets from these switches.
- Support for unoffloaded upper interfaces.
- Support for more cross-chip topologies than the basic daisy chain, while maintaining the basic principle that network interfaces under the same bridge can forward from one to another, and interfaces under different bridges don't.

The data plane and the control plane

The original DSA architecture of exposing one virtual network interface for each front-facing switch port, and not exposing virtual network interfaces for the ports facing inwards (CPU ports, DSA/cascade ports) has remained unchanged to this day. DSA network interfaces should not only be conduit interfaces for retrieving ethtool statistics and registering with the PHY library, but they should be fully capable of sending and receiving packets. This is accomplished via the DSA tagging protocol drivers, which are hooks in the RX and TX path of the host Ethernet controller (the DSA master) which multiplex and demultiplex packets to/from the correct virtual switch interface based on switch-specific metadata that is placed in the packets.

In this model, the basic function of a network switch from a hardware designer's perspective, which is to switch packets, is an optional feature from the Linux network stack's perspective, and was added years after the original design had been established.

Behind the seemingly uniform implementation of DSA tagging protocols and switch drivers, which are tightly managed by the DSA framework, lie many differences and subtleties that make the feature set exposed by two different DSA switches to the network stack very different.

The majority of network switches capable of management have some sort of distinction between the data plane packets and the control packets.

At the most basic level, control packets, which must be used for link-layer control protocols like STP and PTP, have the ability to target a specific egress port and to override its STP state (inject into a BLOCKING port). These packets typically bypass the forwarding layer of the switch and the frame analysis stage of the ingress (CPU) port and are injected directly into the egress port. The implications are that metadata such as QoS class and VLAN ID must be specified by the operating system driver directly as part of the DSA tag, and that hardware address learning is not performed on the CPU port.

On the opposite side of the spectrum, data plane packets do not perform STP state override, are subject to hardware address learning on the CPU port, but also cannot be steered towards a precise destination port, since they are also subject to the forwarding rules of the switch.

At the extreme, there exists a DSA_TAG_PROTO_NONE tagging protocol, which admits defeat and does not attempt to multiplex/demultiplex virtual switch interfaces from the DSA tag, and all network I/O through such a switch takes place through the DSA master which is seen as a switched endpoint. The network interfaces registered for the switches are only used for control operations (ethtool, PHY library) and are "dead" to the network stack both for control plane and for data plane packets. These are the "unmanaged" switches.

Finally, in some switch designs, injecting a control packet is an expensive operation which cannot be sustained at line rate, and the bulk of the traffic (the data plane packets) should be injected, from the hardware designer's perspective, directly through the DSA master interface, with no DSA tag. These are the "lightly managed" switches, and their virtual DSA interfaces are similarly "dead" to the network stack except for link-local packets.

The most basic and common approach with this type of hardware is to simply set up a user space configuration to perform the traffic termination from the switching domain on the DSA master itself. For some packets to target a single switch port, the user is required to install a bridge VLAN on the switch port which is egress-tagged towards the CPU port, then create an 8021q upper with the same VLAN ID on top of the DSA master, and send/receive traffic through the 8021q upper of the DSA master. This approach is, however, undesirable because bridging DSA interfaces with non-DSA (foreign) interfaces is impossible, which is an important use case for boards with a switch and a Wi-Fi AP (home routers). Interfaces that are DSA masters cannot be added to a bridge either.

A slightly better integrated way of achieving the same result is the relatively new software-defined DSA tagging protocol named tag_8021q, which can bring both the lightly managed and unmanaged switches closer to the user model exposed by DSA switches with hardware support for a DSA tagging protocol.

The tag_8021q protocol is fundamentally still sending data plane packets from the perspective of the hardware, so there are things it cannot accomplish, like STP state override. Additionally, the DSA framework has traditionally not enforced any meaningful distinction between data plane and control plane packets, since originally, the assumption was that all packets injected by the software network stack should be control packets.

To unify the hardware and the software notions, and to use these chips in the way they were meant to, the network stack must be taught about data plane packets. The tag_8021q model breaks down when DSA switch interfaces offload a VLAN-aware bridge, which is in fact their primary use cases. This is because the source port of the switch cannot be retrieved based on the VLAN ID by the tagging protocol driver on RX, because the VLANs are under the control of the bridge driver, not DSA, and there is no guarantee that a VLAN is uniquely installed on a single switch port. So bridging with foreign interfaces becomes equally impossible.

The decisive changes which made these switches correctly offload a VLAN-aware bridge come in the form of not attempting to report a precise source port on RX for data plane packets, just a plausible/imprecise one. As long as some requirements inside the software bridge's ingress path are satisfied (valid STP state, VLAN ID from the packet is in the port's membership list), the bridge is happy to accept the packet as valid, and process it on behalf of the imprecise DSA interface that was reported.

Complications arise due to the fact that the software bridge might learn the MAC SA of these packets on a potentially wrong port, and deliver those packets on the return path towards the wrong port. Additionally, due to bandwidth constraints, DSA interfaces do not synchronize their hardware FDB with the FDB of the software bridge, so the software bridge does not have an opportunity to figure out the real source port of imprecise packets.

To give DSA the chance to right a wrong, the bridge driver was modified to support TX forwarding offload. With this feature, the software bridge avoids cloning an skb which needs to be flooded to multiple ports, and sends only one copy of the packet towards a single network interface from each "hardware domain" that the flooded packet must reach. The port driver is responsible with

looking up its hardware FDB for that packet and replicate the packet as needed. This is a useful feature in itself, because with switches with a large port count, multicast traffic on the bottlenecked link between the DSA master and the CPU port is reduced, and packets are replicated inside the hardware. But with the lightly-managed and unmanaged switches, it makes the imprecise RX work correctly, since the TX is also imprecise. So even though the software bridge did learn the MAC SA of the packets on the wrong source port, that source port is in the same hardware domain with the right port, and even though the software FDB is incorrect, the hardware FDB isn't. So DSA drivers for lightly-managed and unmanaged switches have a chance to properly terminate traffic on behalf of a VLAN-aware bridge, in a way that is compatible with bridging with foreign interfaces, and with a user space interaction procedure that is much more uniform with DSA drivers that always send and receive packets with a DSA tag.

Unoffloaded software upper interfaces

Recently, DSA has also gained support for offloading other virtual network interfaces than the Linux bridge. These are the hsr driver (which supports the HSR and PRP industrial redundancy protocols) and the bonding/team drivers (which support the link aggregation protocol).

Not all switches are capable of offloading hsr and team/bonding, and DSA's policy is to fall back to a software implementation when hardware offload cannot be achieved: the bandwidth to/from the CPU is often times good enough that this is not impractical.

However, DSA's policy could not be enforced right away with the expected results, due to two roadblocks that led to further changes in the kernel code base.

To not offload an upper interface means for DSA that the physical port should behave exactly as it would if it was a standalone interface with no switching to the others except the CPU port, and which is capable of IP termination.

But when the unoffloaded upper interface (the software LAG) is part of a bridge, the bridge driver makes the incorrect assumption that it is capable of hardware forwarding towards all other ports which report the same physical switch ID. Instead, forwarding to/from a software LAG should take place in software. This has led to a redesign of the switchdev API, in that drivers must now explicitly mark to the bridge the network interfaces that are capable of autonomous forwarding; the new default being that they aren't. In the new model, even if two interfaces report the same physical switch ID, they might yet not be part of the same hardware domain for autonomous forwarding as far as the bridge is concerned.

The second roadblock, even after the bridge was taught to allow software forwarding between some interfaces which have the same physical switch ID, was FDB isolation in DSA switches. Up until this point, the vast majority of DSA drivers, as well as the DSA core, have considered that it is enough to offload multiple bridges by enforcing a separation between the ports of one bridge and the ports of another at the forwarding level. This works as long as the same MAC address (or MAC+VLAN pair, in VLAN-aware bridges) is not present in more than one bridging domain at the same time. This is an apparently reasonable restriction that should never be seen in real life, so no precautions have been taken against it in drivers or the core.

The issue is that a DSA switch is still a switch, and for every packet it forwards, regardless of whether it is received on a standalone port, a port under a VLAN-unaware bridge or under a VLAN-aware one, it will attempt to look up the FDB to find the destination. With unoffloaded LAGs on top of a standalone DSA port, where forwarding between the switched domain and the standalone port takes place in software, the expectation that a MAC address is only present in one bridging domain is no longer true. From the perspective of the ports under the hardware bridge, a MAC address might come from the outside world, whereas from the perspective of the standalone ports, the same MAC address might come from the CPU port. So without FDB isolation, the standalone port might look up the FDB for a MAC address and see that it could forward the

packet directly to the port in the hardware bridge domain, where that packet was learned by the bridge port, shortcircuiting the CPU. But the forwarding isolation rules put in place will prevent this from happening, so packets will be dropped instead of being forwarded in software.

Individual drivers have started receiving patches for FDB isolation between standalone ports and bridged ports, but it is possible to conceive real life situations where even FDB isolation between one bridge and another must be maintained. Since the DSA core does not enforce FDB isolation through its API and many drivers already have been written without it in mind, it is to be expected that many years pass until DSA offers a uniform set of services to upper layers in this regard.

Switch topology changes

Traditionally, the cross-chip setups supported by DSA have been daisy chains, where all switches except the top-most one lack a dedicated CPU port, and are simply cascaded towards an upstream switch. There are two new switch topologies supported by DSA now.

The first is the disjoint tree topology. A DSA tree is comprised of all switches directly connected to each other which use a compatible tagging protocol (one switch understands the packets from the other one, and can push/pop them as needed). Disjoint trees are used when DSA switches are connected to each other, but their tagging protocols are not compatible. As opposed to one switch understanding another's, tag stacking takes place, so in software, more than one DSA tagging protocol driver needs to be invoked for the same packet. In such a system, each switch forms its own tree. Disjoint trees were already supported, but the new changes also permit some hardware forwarding to take place between switches belonging to different trees. For example, be there an embedded 5 port DSA switch that has 3 external DSA switches connected to 3 of its ports. Each embedded DSA switch interface is a DSA master for the external DSA switch beneath it, and there are 4 DSA disjoint trees in this system. For a packet to be sent from external switch 1 to external switch 2, it must be forwarded towards the CPU port. In the most basic configuration, forwarding between the two external switches can take place in software. However, it is desirable that the embedded DSA switch that is a master of external switches 1 and 2 can accelerate the forwarding between the two (because the external switches are tagger-compatible, they are just separated by a switch which isn't tagger-compatible with them). Under some conditions, this is possible as long as the embedded DSA switch still has some elementary understanding of the packets, and can still forward them by MAC DA and optionally VLAN ID, even though they are DSA-tagged. With the vast majority of DSA tagging protocols, the MAC DA of the packets is not altered even when a DSA tag is inserted, so the embedded DSA master can sanely forward packets between one external switch and another. This is one of the only special cases where DSA master interfaces can be bridged (they are part of a separate bridge compared to the external switch ports), because in this case, the DSA masters are part of a bridge with no software data plane, just a hardware data plane. The second requirement is for both the embedded and the external switches to have the same understanding of what constitutes a data plane packet, and what constitutes a control plane packet: STP packets received by the external switch should not be flooded by the embedded switch. Due to the same reason that the embedded switch must still preserve an elementary understanding of the MAC DA of packets tagged with the external switch's tagging protocol, this will also be the case, since typical link-layer protocols have unique link-local multicast MAC addresses.

The second is the H tree topology. In such a system, there are multiple switches laterally interconnected through cascade ports, but to reach the CPU, each switch has its own dedicated CPU port. It turns out that to support such a system, there are two distinct issues.

First, with regard to RX filtering, an H tree topology is very similar in challenges to a single switch with multiple CPU ports. Hardware address learning on the CPU port, if at all available, is of no use and leads to addresses bouncing and packet drops. All MAC addresses which need to be

filtered to the host need to be installed on all CPU ports as static FDB entries. This has led to the extension of the bridge switchdev FDB notifiers to cover FDB entries that are local to the bridge, and which should not be forwarded.

Secondly, in an H topology it is actually possible to have packet loops with the TX forwarding offload feature enabled, because TX data plane packets sent by the stack to one switch might also be flooded through the cascade port to the other switch, where they might be again flooded to the second switch's CPU port, where they will be processed as RX packets. Currently, drivers which support this topology need to be individually patched to cut RX from cascade ports that go towards switches that have their own CPU port, because the DSA driver API does not have the necessary insight into driver internals as to be able to cut forwarding between two ports only in a specific direction.

Future changes

One of the most important features still absent from DSA is the support for multiple CPU ports, the ability to dynamically change DSA masters and the option to configure the CPU ports in a link aggregation group. However, with many roadblocks such as basic RX filtering support now out of the way, this functionality will arrive sooner rather than later.

There is also the emerging topic of Ethernet controllers as DSA masters that are aware of the DSA switches beneath them, which is typical when both the switch and the Ethernet controller are made by the same silicon vendor. Right now DSA can freely inherit all master->vlan_features, such as TX checksumming offloads, but this does not work for all switch and DSA master combinations, so it must be refined and only the known-working master and switch combinations inherit the extra features.

On the same topic of DSA-aware masters, SR-IOV capable masters are expected to still work when attached to a DSA switch, but the network stack's model of this use case is unclear. VFs on top of a DSA master should be treated as switched endpoints, but the VF driver's transmit and receive procedures do not go through the DSA tagging protocol hooks, and these packets are therefore DSA-untagged. So hardware manufacturers have the option of inserting DSA tags in hardware for packets sent through a VF that goes through a DSA switch. It is unclear, however, according to which bridging domain are these VFs being forwarded. An effort should be made to standardize the way in which the network stack treats these interfaces. It appears reasonable that DSA switches might have to register virtual network interfaces that are facing each VF of the master, in order to enforce their bridging domain, but this makes the DSA master and switch drivers closely coupled.

On the other hand, letting other code paths than the DSA tagging protocol driver inject packets into the switch risks compromising the integrity of the hardware, which is an issue that currently exists and needs to be addressed.

As a conclusion, taming DSA switches and making them behave completely in accordance with the network stack's expectations proves to be a much more ambitious challenge than initially foreseen, thus the fight continues.

I agree to abide by the anti-harassment policy

I agree

Primary author: OLTEAN, Vladimir (NXP Semiconductors)

Presenter: OLTEAN, Vladimir (NXP Semiconductors)

Session Classification: BPF & Networking Summit

Track Classification: Networking & BPF Summit (Closed)