LSDN — Manage complex (virtual) networks in cloud environment with Linux kernel facilities

Vojtech Aschenbrenner*, Roman Kapl, Jan Matejek, Adam Vyskovsky
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
*v@asch.cz

Abstract

Contemporary data centers highly rely on SDN (Software Defined Networking) to establish and manage networking among huge number of virtual machines (VM). With a rapid growth of cloud services and their users there is a natural growth of virtual machines providing those services. Therefore reliable SDN solution is a must and all cloud providers depend on it.

There are several open-source solutions providing management of virtual networking for example well-known Open vSwitch. However these solutions depend on running daemons and they also add code to the kernel. This fact may decrease reliability.

In this paper we describe a tool called LSDN. With LSDN you can easily manage (not only) virtual networks and in addition LSDN brings no other code to the kernel. It relies only on Linux Kernel facilities and in most cases it does not need any running services. In this paper we describe how to properly use LSDN, LSDN internals, its API in C version and DSL (Domain Specific Language) version and also bugs we found in Linux Kernel when using its recent functionality.

Although LSDN is experimentally deployed in non-demanding production with low traffic it is is still a very immature project and there is a huge space for improvements and additional features. We discuss how to tackle some of the most wanted features, e.g. statefull firewall (now we support only stateless version).

Keywords

Linux, Traffic Control, Software Defined Networking, Multi Tenancy, Virtual Network, Cloud, Virtual Machines, Netlink, Data Center

Introduction

In the last decade we can observe rapid inclination towards making the whole computation stack virtual, i.e. being able to create an illusion that we are using real hardware however the hardware is emulated in software. This phenomena began with full virtualization of computer hardware with VMWare[12], Oracle VM Virtual Box[11] or QEMU[1] and evolved into virtualization with lower overhead like containers, e.g. Docker[2], Linux Containers[5] etc.

Naturally most of these virtual machines have to be networked seemingly in the same way as their physical counterparts. This is why Software Defined Networking (SDN) generates such a big interest and why there is a huge effort to fine-tune SDN for the demands of todays cloud environments which are very diverse. Hence the need for high performance, reliable and easily customizable solution which will be easy to integrate (not only) with todays cloud orchestrators and will have minimal installation dependencies.

One of the well known open-source projects in this field is Open vSwitch[7] which suits well for both Linux and BSD environments. It is a high performance, production quality multi-layer switch very often used with open-source orchestrators like Kubernetes[3] or Open Stack[6]. Open vSwitch comes as a kernel module with some code and functionality duplication of the Linux Kernel, especially the TC infrastructure. This leads to parallel development of similar functionality.

Recent activity in the Linux Kernel Traffic Control (TC) subsystem and its maturity lead us to the attempt to create a network management software which uses only Linux Kernel facilities without any additional kernel code.

LSDN Overview

LSDN[4] is a project for complex network setup management especially suited for cloud environment building on top of Linux Kernel TC subsystem. So far it provides the following functionality:

- Support for virtual networks, switches and ports.
- Network overlay.
- Multi tenancy.
- Stateless firewall.
- QoS.
- API for management via C-library.
- DSL for stand-alone management.

LSDN ensures isolation between networks using the existing network tunneling technologies, e.g. VXLAN[10] or Geneve.

Virtual machines never see traffic from devices that are not part of their virtual network, even if they exist on the same host. Multiple virtual machines can even have identical MAC addresses, as long as they are connected to different virtual networks. Thus, it is possible to virtualize multiple existing physical networks and run them without interference in a single hosting location.

Intended Usage

LSDN provides a configuration language, that allows you to describe the desired network configuration (we call it a network model or netmodel for short): the virtual networks, physical machines and virtual machines and their relationships. It can also be driven programmatically using a C API.

You run LSDN on each physical machine and provide it with the same netmodel, either by passing the same configuration file (you can use our dumping mechanism) or calling the same C API calls. LSDN then takes care of the configuration so that the VMs in the same virtual network can correctly talk to each other even if on different computers.

If you run a static ZOO of VMs, you can simply copy over the configuration file to all the physical machines. If you have more complex virtualization setup, you are likely to have an orchestrator on each physical machine. In that case, you can modify your orchestrator to use LSDN as a backend.

More thorough technical description of the API is described in the official documentation on the project home page[4].

Example

Let's use LSDN to configure a simple network: four VMs, running on two physical machines. We will call the physical machines A and B and the virtual machines 1, 2, 3 and 4. The virtual machines 1 and 2 are running on physical machine A, virtual machines 3 and 4 are located on physical machine B. The configuration is illustrated in Figure 1.

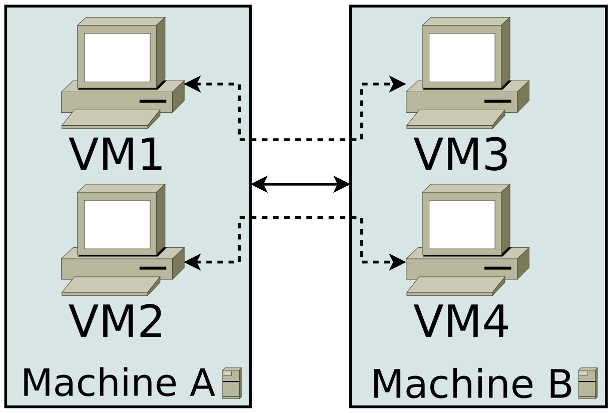

Figure 1: Quickstart example.

C API

See Appendix B for an example of a C program using the C API. Compile it to `quickstart` binary and run as follows:

```
$ ./quickstart A # On Machine A
$ ./quickstart B # On Machine B
```

Configuration File

See Appendix A for an example of a configuration file `config.lsctl` passed to a LSDN control program called `lsctl`. Run it as follows:

```
$ lsctl config.lsctl A # On Machine A
$ lsctl config.lsctl B # On Machine B
```

Network Representation

The public API (either C API or lsctl Configuration Files) gives you tools to build a model of your virtual networks, which LSDN will then realize on top of the physical network, using various tunneling technologies. You will need to tell LSDN both about the virtual networks and the physical network they will be using.

There are three core concepts (objects) LSDN operates with: *virtual machines*, *physical machines* and *virtual networks*. In the rest of the guide (and in the source code) we abbreviate them as *virts*, *physes* and *nets*. If you are wondering if there are any physical networks, then no, LSDN just expects that the physical machines are connected together when needed and that is all.

The terminology is derived from the most common use case, but that does not mean that *virts* really have to be virtual machines and *physes* must really be physical machines. For example the *virts* could be Linux containers and *physes* could be virtual machines running those containers.

The *virts*, *physes* and *nets* have the following relationships:

- *virts* always belong to one *net* (they can not be moved between *nets*).
- *virts* are connected at one of the *physes* (however, they can be reconnected at a different *phys*, in other words, they can migrate).
- *physes* attach to a *net* – this tells LSDN that the *phys* will have *virts* connecting to the network.

Each of these objects can also have attributes – for example *physes* can have an IP address (some network tunneling technologies require this information) and *virts* can have a MAC address (network tunnels not supporting MAC learning require this information).

One of the attributes common to all objects is a name. A name does not have an impact on the functionality of the network, but you can use it to keep track of the object. If you are using lsctl Configuration Files, it is more or less mandatory, because it is the only way to refer to an object if you want to change it at a later point (for example when you want to migrate a *virt*). If you do not specify a name, one will be generated for you. This ensures that the export/dump mechanism will always be able to create cross-references.

Collectively, the model is represented by a LSDN *context*, which contains all the *physes*, *virts* and *nets*. *Context* is a well known concept in C libraries, which essentially replaces global variables and ensures that the library can be safely used by multiple clients in the same process.

Networks

Virtual networks are defined by their virtual network identifier (VID) and the settings for the tunneling technology they should use. The VID is a numeric identifier used to separate one virtual network from another and is mapped to VLAN IDs, VXLAN IDs or similar identifiers. The allowed range of the VID is defined by the used tunneling technology and must be unique among all networks of the same type.

The used networking overlay technology (and any options related to that, like VXLAN port) is encapsulated in the settings object, which serves as a template for the new networks (with only the VID changing each time). A list of supported networking technologies is in the section Supported Tunneling Technologies, including the additional options they support.

Like other objects, networks can have a name. However, they do not have any other attributes, since everything important for their functioning is part of the settings. Settings can have names and lsctl reserves name default for unnamed settings.

Virt

virts are the computers/virtual machines that are going to connect to the virtual network. From LSDN's standpoint, they are just network interfaces that exist on a phys (usually *tap* for a virtual machine or *veth* for a container). LSDN does not care what is on the other end.

When creating a *virt* you have to specify which virtual network it is going to be part of. This can not be changed later. If you remove the network, all its *virts* will be removed as well.

A *virt* also can not be part of multiple virtual networks. The recommended solution in that case is to simply create one *virt* for each virtual network you are going to connect to. In this sense *virt* can be described not as a virtual machine, but as a network interface of a virtual machine.

Once created, you can specify which phys this *virt* will connect at and how is its network interface named on that *phys*. If you are using lsctl, just run *virt* with a new `-phys` argument. In C API use `lsdn_virt_connect()`. If the *virt* was already connected, it will be reconnected (migrated) to the new *phys* (you want to do this in sync with the final stage of the migration of the virtual machine itself).

Like other objects, *virts* can have names for your convenience. The names do not have to be unique globally, but just inside of a single net.

Depending on the networking technology used, you may also need to inform LSDN about the virtual machine's MAC address (currently only one MAC address can be assigned, this may change in future versions). LSDN will use this MAC address for routing network packets to the machine.

Firewall

You can filter out specific packets based on their source/destination IP address range and source/destination MAC address range. The filtering can be done independently on ingress and egress traffic.

The filtering rules are organized by their priority. All rules inside a given priority must match against the same target (a target is a masked part of an IP or MAC address – for example first octet of the IP address) and must be unique. This restriction exists to ensure that only deterministic rules can be defined.

Unfortunately, currently there is no way to ACCEPT packets early, as is common in e.g. iptables.

QoS

You can limit the amount of traffic going in or out of the *virt* for each direction. There are three settings:

- `avg_rate` provides the basic bandwidth limit
- `burst_size` allows the traffic to overshoot the limit for certain number of bytes
- `burst_rate` (optional) absolute bandwidth limit applied even if traffic is allowed to overshoot `avg_rate`

If you do not want to allow any bursting, specify `burst_rate` equal to the maximum size of a single packet (MTU). Setting `burst_rate` to zero will not work.

Phys

physes are used to describe the underlying physical machines that will run your virtual machines.

You will tell LSDN which machine it is currently running on (using `claimLocal` or `lsdn_phys_claim_local()`). LSDN will then make sure that the *virts* running on this machine are connected to the rest of the *virts* running on the other machines.

If your machine has multiple separate network interfaces (not bonded), you will want to create a new *phys* for each network interface on that machine and claim all such *physes* as local. In this sense, a *phys* is not a physical machine but a network interface of a physical machine.

This use-case is not meant for a case where both network interfaces are connected to the same physical network and you just want to choose which one will be used. LSDN does not support two *physes* claimed as local connecting to the same virtual network for technical reasons, so it will not work.

Like other objects, *physes* can have names. They can also have an ip attribute, specifying an IP address for the network overlay technologies that require it.

Validation

The validation step in LSDN serves to validate the network model. There are several reasons why the validation step is present in LSDN. One reason is that when a network model is being gradually built up using the C API the user does not have to worry too much about the order in which network objects are being created as long as the final *netmodel* is valid. The intermediate steps are not being checked on-the-fly. For example when creating a virtual machine its MAC attribute may be specified just before committing the network model even though for a particular network type this information may be mandatory (this is specified for each network type in networking technology).

Another advantage of this approach is that when there are problems detected during the validation phase they will all get reported one by one. LSDN conveniently provides a `lsdn_problem_stderr_handler()` function which will report every detected problem on the standard error output. It is also possible to invoke the `lsdn_validate()` step with a different error handler. This error handler must have the same function signature as `lsdn_problem_stderr_handler()`.

This way you can try some network scenario and if the validation reports to you some problems it has detected in the network model you may fix all these issues at once and perhaps the next network validation phase will succeed.

Every host participating in a network must share a compatible network representation. This usually means that all hosts have the same model, presumably read from a common configuration file or installed through a single orchestrator. It is then necessary to `claimLocal` (or `lsdn_phys_claim_local()`) a *phys* as local, so that LSDN knows on which machine it is running. Several restrictions also apply to the creation of networks in LSDN.

Fixing all the issues present in your network model in the validation step greatly reduces the risk of creating inconsistent network models in the kernel and it also alleviates the complexity of the creation of the individual network objects in the right order inside the kernel.

The validation phase will ensure the network model does not violate any of the restrictions listed in Network Restrictions.

Commit

Committing a network model means telling LSDN to actually set-up the network inside Linux kernel.

When we commit a network model the first thing LSDN does it validates the whole network model. Only if the validation phase succeeds, the commit phase may proceed. This way the user does not even need to be aware of the validation phase involved and can only commit the *netmodel* when appropriate. This often eliminates the possibility of getting the network in some undesirable state.

We need to be able to distinguish among network objects already created and committed in the kernel and network objects newly created, but not yet committed. LSDN will keep track of the state of each network object. Basically what we need to do is to remember which objects are already present in the kernel in their most up-to-date state and which objects have been newly created or updated since the last time they have been committed (if ever) and which objects have been deleted. Each attribute you add, remove from or change on a network object is considered as an update of this object.

If you want to know more about LSDN state management and also to view a diagram of all states and transitions between these states have a look at the Netmodel Implementation section.

It is important to note that any updates exercised on the kernel data structures representing our network objects are only performed on local objects, where:

- *phys* is local if and only if it has been claimed local (either with `claimLocal` or `lsdn_phys_claim_local()`),
- *virt* is local if and only if it is connected at a local *phys*.

However, local objects may sometimes need to be updated as a result of a non local network object being added, updated or removed. E.g. when a MAC address of a non local *virt* changes inside a network where this information is mandatory (such as in static VXLAN networks) then local routing information in the kernel must be updated.

Also, there are transitive dependencies among the network objects. In particular, when:

- *virt* is deleted then all its Firewall rules and QoS are deleted as well,
- *net* is deleted then all its *virts* are deleted as well,
- *phys* is deleted then all *virts* attached to this *phys* are deleted as well,
- *settings* are deleted then all *nets* of this type are deleted as well.

After the initial validation step is completed, LSDN will then proceed with the actual commit phase which is further subdivided into two subphases:

- decommit and
- recommit.

In the decommit subphase LSDN will consider all the network objects that need to be either updated or deleted and it will delete both of these objects from the kernel data structures. However, LSDN will keep track of those objects which have been initially updated, but not deleted, as they will need to be committed back again in the next subphase.

The second subphase is the recommit phase in which LSDN will iterate over all local *phys* objects and commit any new or updated *virts* residing on this *phys*.

You can perhaps think of the whole commit phase as finding the smallest possible delta between the objects ready to be committed and those already committed. In the special case of committing for the very first time we can imagine we have only committed an empty network model (which, by the way, is also possible to do).

Unfortunately, things can go wrong in the commit phase even when the network model passes the validation phase. Depending on the phase at which an error occurred we may or may not be able to keep the network model consistent.

If an error occurs in the recommit phase, a limited rollback is performed and the kernel rules remain in mixed state. Some objects may have been successfully committed, others might still be in the old state because the commit failed. In such cases the user can retry the commit to install the remaining objects.

If an error occurs in the decommit phase, however, there is no safe way to recover. Given that kernel rules are not installed atomically and there are usually several rules tied to an object, LSDN can't know what is the installed state after rule removal fails. In this case the model is considered to be in an inconsistent state. The only way to proceed is to tear down the whole model and reconstruct it from scratch.

Error Handling

During construction of the network model there are several things that can go wrong. LSDN will report these errors to the user of the C API. All the possible error types are grouped in `lsdn_err_t`.

A successful operation will return the `LSDNE_OK` error code.

When parsing an IP address of a phys or when parsing a MAC attribute of a *virt* the operation may fail if the provided address is invalid. In that case LSDN will report this as a `LSDNE_PARSE` error.

When assigning a name to a network object (such as *virt*, *phys* or *net*) the assignment may fail with the `LSDNE_DUPLICATE` error code if an object of the same type with this name already exists.

A `LSDNE_NOIF` error code will be returned when querying the recommended MTU for a *virt* if the given *virt* has no locally assigned interface (see `lsdn_virt_get_recommended_mtu()`).

A `LSDNE_NETLINK` error code is returned when LSDN is unable to establish a netlink socket for communicating with the kernel.

`LSDNE_VALIDATE` is returned when the network model validation failed. This can happen while validating the network with validate or `lsdn_validate()`. It can also happen when committing the network model with commit or `lsdn_commit()`, because the network model is always validated first. In the latter case of committing the network model, the current network model will stay in effect.

The `LSDNE_COMMIT` error code means a network model commit failed and a mix of old, new and dysfunctional objects are in effect. You may retry the commit and see if the error was only temporary.

`LSDNE_INCONSISTENT` is more serious than the `LSDNE_COMMIT` failure, since the commit operation can not be successfully retried. The only operation possible is to rebuild the whole model again.

You may also encounter a `LSDNE_NOMEM` error. LSDN deals with out-of-memory errors in the following fashion: whenever it fails to allocate dynamic memory it will call a registered callback (if any) that may deal with this error as it sees fit. The callback is registered with the `lsdn_context_set_nomem_callback()` function. It is possible to set a default handler using `lsdn_context_abort_on_nomem()` function provided by LSDN. This error handler will simply print an error message on the standard error output and will immediately abort the program should any dynamic memory allocation fail. Of course, you may register your own out-of-memory callback as long as the function signature of the callback is that of `lsdn_context_abort_on_nomem()`. You can also use the callback to implement a setjmp/longjmp error handling scheme.

If no `nomem` callback is registered (the default), the `LSDNE_NOMEM` error is simply returned to the caller.

Debugging

The LSDN library and the lsctl tool both respect the `LSDN_DEBUG` environment variable. If you have any problem when committing a model, try setting `LSDN_DEBUG=nlerr` to print extended netlink messages. Alternatively, you can try `LSDN_DEBUG=all` for very verbose output.

`LSDN_DEBUG` accepts a comma separated list of the following message categories:

- `netops` — High-level network commit operations (add *virt*, *phys* etc.)
- `rules` — Creation and deletion of TC flower rules.
- `nlerr` — Errors returned from kernel (mostly netlink).
- `all` — All of the above

Supported Tunneling Technologies

Currently LSDN supports three network tunneling technologies: VLAN, VXLAN (in three variants) and Geneve. They are all configured the same in LSDN (only the settings differ), but it is important to realize what technology you are using and what restrictions it has.

Theoretically, you should be able to define your network model once and then switch the networking technologies as you wish. But in practice some technologies may need more detailed network models than others. For example, `ovl_vxlan_mcast` does not need to known the MAC addresses of the virtual machines and `ovl_vlan` does not need to know the IP addresses of the physical machines nor the MAC addresses of the virtual machines.

VLAN

Also known as 802.1Q, VLAN is a Layer-2 tagging technology, that extends the Ethernet frame with a 12-bit VLAN tag. LSDN needs no additional information to setup this type of network, as it relies on the networking equipment along the way to route packets (typically using MAC learning).

If either the physical network already uses VLAN tagging (the physical computers are connected to a VLAN segment) or the virtual network will be using tagging, then the networking equipment along the way must support this. The support is called 802.1ad or sometimes QinQ.

Restrictions are in 12-bit VID, need of having physical nodes on the same L2 segment and you have to take care when doing a nesting into another VLAN.

VXLAN

VXLAN is a Layer-3 UDP-based tunneling protocol. It is available in three variants in LSDN, depending on the routing method used. All of the variants need the connected participating physical machines to have the IP attribute set and they must all see each other on the IP network directly (no NAT).

VXLAN tags have 24 bits (16 million networks). VXLANs by default use UDP port 4789, but this is configurable and could in theory be used to expand the vid space. LSDN currently does not do this.

VXLANs support IPv6 addresses, but they can not be mixed with IPv4. All physical nodes must use the same IP version and the version of multicast address for Multicast VXLAN must be the same. This does not prevent you from using both IPv6 and IPv4 on the same physical node for other purposes than LSDN, you just have to choose one version for the phys IP attribute.

Multicast This is a self configuring variant of VXLANs. No further information for any machine needs to be provided, because the VXLAN routes all unknown and broadcast packets to a designated multicast IP address and the VXLAN iteratively learns the source IP addresses. Hence the only additional information is the multicast group IP address.

End-to-End Partially self-configuring variant of VXLANs. LSDN must be informed about the IP address of each physical machine participating in the network using the IP attribute. All unknown and broadcast packets are sent to all the physical machines and the VXLAN iteratively learns the IP address - MAC address mapping.

Fully Static VXLAN with fully static packet routing. LSDN must be informed about the IP address of each physical machine and the MAC address of each virtual machine participating in the network. LSDN then constructs a routing table from this information. Broadcast packets are duplicated and sent to all machines.

Geneve

Geneve is a Layer-3 UDP-based tunneling protocol. All participating physical machines must see each other on the IP network directly (no NAT).

Geneve uses fully static routing. LSDN must be informed about the IP address of each physical machine (using IP attribute) and MAC address of each virtual machine participating in the network.

No Tunneling

No separation between the networks. You can use this type of network for corner cases, like connecting a VM serving as an internet gateway to a dedicated interface. In this case no separation is needed nor desired.

Network Restrictions

Certain restrictions apply to the set of possible networks and their configurations that can be created using LSDN. Anywhere where the keyword mandatory is written in the following list with regards to a network type, please refer to Supported tunneling technologies to see if the rule applies to a given network type:

- You can not assign the same MAC address to two different *virts* that are part of the same *net*.
- Any two *nets* of the same network type must not be assigned the same virtual network identifier.
- Any two VXLAN networks sharing the same *phys*, where one network is of type Fully static and the other is either of type Endpoint-to-Endpoint or Multicast, must use different UDP ports.
- A *virt* must be explicitly assigned a MAC address when this is mandatory for a given network type.

- IP address has been specified for a *phys* if it hosts a *net* where this information is mandatory.
- No duplicate IP addresses were specified for any two *physes*.
- All *phys* attached to the same *net* have the same IP versions of their IP addresses.
- While trying to connect a *virt* to a *net* on *phys*, the *phys* is attached to *net*.
- Interface specified for *virt* exists.
- No duplicate MAC addresses were specified for any two *virts* connected to the same *net* if this attribute is mandatory for a given network type.
- Any two *nets* created on the same *phys* have compatible network types.
- The virtual network identifier is within the allowed range for a given network type where this is mandatory.
- No two nets of the same network type have the same virtual network identifier.
- No two rules on the same *virt* sharing the same priority have different match targets or masks.
- Two rules on the same *virt* sharing the same priority are not equal.
- QoS rates specified for a *virt* are within the allowed range (rate).

Netmodel Implementation

The network model (in lsdn.c) provides functions that are not specific to any network type. This includes QoS, firewall rules and basic validation.

Importantly, it also provides the state management needed for implementing the commit functionality, which is important for the overall ease-of-use of the C API. The network model layer must keep track of both the current state of the network model and what is committed. Also it tracks which objects have changed attributes and need to be updated. Finally, it keeps track of objects that were deleted by the user, but are still committed.

For this, it is important to understand a life-cycle of an object, illustrated in Figure 2.

The objects always start in the *NEW* state, indicating that they will be actually created with the nearest commit. If they are freed, the free call is done immediately. Any update leaves them in the *NEW* state, since there is nothing to update yet.

Once a *NEW* object is successfully committed, it moves to the *OK* state. A commit has no effect on an *OK* object, since it is up-to-date.

If a *OK* object is freed, it is moved to the *DELETE* state, but its memory is retained until commit is called and the object is deleted from kernel. The objects in *DELETE* state can not

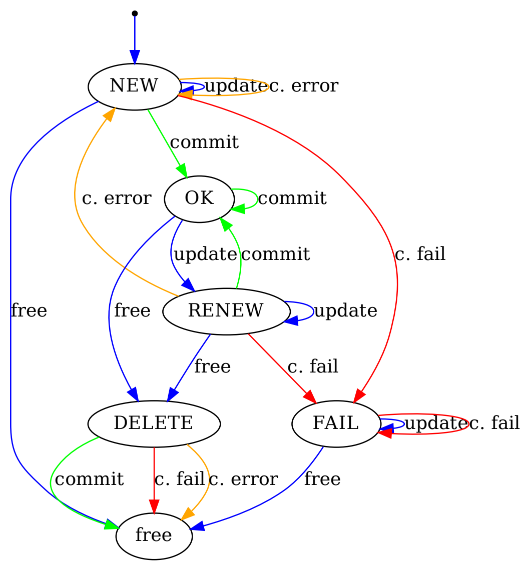

Figure 2: Object life-cycle.

be updated, since they are no longer visible and should not be used by the user of the API. Also, they can not be found by their name.

If an *OK* object is updated, it is moved to the *RENEW* state. This means that on the next update, it is removed from the kernel, moved to *NEW* state, and in the same commit added back to the kernel and moved once again to the *OK* state. Updating the *RENEW* object again does nothing and freeing it moves it to the *DELETE* state, since that takes precedence.

If a commit for some reason fails, LSDN tries to unroll all operations for that object and returns the object to a temporary *ERR* state. After the commit has ended, it moves all objects from *ERR* state to the *NEW* state. This means that on the next commit, the operations will be retried again, unless the user decides to delete the object.

If even the unrolling fails, the object is moved to the *FAIL* state. The only possibility for the user is to release its memory. If the object was originally already deleted, it bypasses the *FAIL* state.

If validation fails, commit is not performed at all and object states do not change at all.

How does it translate to TC?

The supported network overlays can be divided into roughly three types:

- Static forwarding based on pre-configured MAC addresses.
- Learning forwarding using a standard Linux bridge.

- Learning forwarding using the native ability of the network tunnel (e.g. Linux VXLANs, VLANs).

Static forwarding

Static forwarding is handled entirely by TC rules that forward the packets to appropriate interfaces and set appropriate tunnel meta-data. These rules are attached to the VM (*virt*) interfaces provided by the user and to the tunnel interface created by LSDN. To avoid blow-up of the number of rules, we use a dummy interface hold most of the forwarding rules.

When a packet enters the system from the *virt* side, it is first classified as broadcast, multicast or unicast. Broadcast and multicast packets are mirrored (using the *mirred* action) to all interface except the originating *virt*. They are mirrored multiple times to the tunnel interface, but each time with different tunnel meta-data. Unicast packets are redirected to the dummy interface, which sends them to the appropriate *virt* or tunnel depending on the destination MAC address.

When a packet enters the system from the tunnel side, it is classified based on the virtual network identifier and destination MAC address. Unicast packets are redirected to the dummy interface (same as for the *virts*) and broadcast packets are mirrored to all local *virts*.

Learning forwarding

Learning forwarding is handled using a standard Linux bridge and a set of dummy interfaces, each representing one remote physical machine. The dummy interfaces are responsible for setting the correct tunnel meta-data for the outgoing packets. Packets entering from the tunnel side are redirected based on the virtual network identifier and IP address of the remote *phys* to the correct dummy interface.

The setup described so far has the problem that the virtual ethernet network we have created contains cycles — in particular, the *physes* form a complete graph, since they are all connected with each other. To rectify this, we add special TC rules on the dummy interfaces that block re-broadcasting of packets coming through the tunnel back to other physical machines.

Tunnels that support forwarding natively

Some tunnel technologies are able to forward packets to the correct recipient themselves. This includes VLANs, where the forwarding behavior is provided by the underlying Ethernet network. Another case is VXLAN with either multicast support or with explicitly pre-filled forwarding database (fdb), where the Linux VXLAN driver provides the forwarding behavior.

We have decided to use the native ability of VXLAN, since it is already there. But the same behaviour could be replicated in the non-multicast VXLAN by using Linux bridge.

TC Abilities

We have to conclude that TC is steadily gaining in expressiveness. One of the recent additions, which was helpful, was support for goto chains [8]. This allows users to efficiently implement processing DAGs (Direct Acyclic Graphs). The expressiveness was already pointed out by [9], but at that time it had to be either implemented in non-efficient manner (linear matching) or entirely in the u32 filter, which supported nested hash-tables.

Still, we feel that some of the rules we generate could be more efficient or redundant with some changes to the TC infrastructure. Naturally the question remains how to do that simply and generically. Please take the following paragraphs more as a basis for discussion.

For example, we use the dummy interface to share the forwarding rules. But using interface feels quiet heavyweight. Recently a support for sharing the set of filter chains between interfaces was added. This feature is not currently usable for our purpose, since our broadcast processing is different depending on the ingress device and only after that we use the shared rules. Maybe the starting chain could be different for each device sharing the set of chains? Since the feature was introduced (as far as we know), to better match the capabilities of the hardware, it must be considered if this model still matches the capabilities of the hardware.

But maybe there is no reason to have different broadcast actions for each interface? The semantics of TC *mirred* action could be extended (or a new action introduced) so that *mirred* mirrors the packet only if the original ingress interface of the packet is not the same as the interface we are mirroring to. Alternatively, to make this scheme more extensible, the mirroring condition could be based not on ingress interface, but on *skbmark* set by the user. We would like to hear how programmable switches and network card expose the broadcast capability to the user.

Technically, this set-up could be implemented today, using filters, but in quite cumbersome way. Existing filter only provide “if something then action” behavior, but we need “if something not then action” behavior. The negation has to be emulated using the “goto chain” capability.

The last major comment is related to the integration of Linux bridge with TC. Since all the forwarding done by the Linux bridge is based on Linux interfaces, we were forced to create dummy interfaces for each remote physical machine, even if the packet goes through the same tunnel in the end.

Maybe the bridge could be modified to recognize multiple forwarding destinations (bridge ports) going through one Linux interface. The destinations could be identified using *skbmark*, or directly using tunnel meta-data. In the long run, this could make the forwarding code present in VXLAN driver (which is similar to the bridge code) obsolete and could be reused for other types of tunnels.

Bugs in Linux Kernel

During the development of LSDN several bugs in the kernel were found. Here is the list with the first 7 digits of the commit corresponding to the provided patch and short description.

- f15ca72 (not fixed, just reported), some `dst_ops` do not set `update_pmtu`, patched by conditional call.
- a60b3f5, bad RCU implementation on blocks with goto chain action.
- d7aa04a, use after free when deleting filter chain.
- 5ae437a, report if filter is too large to dump.

Conclusion and Future Work

In this paper we described the motivation behind LSDN. It was mainly the fact, that TC subsystem in Linux Kernel is powerful enough to build a tool on top of it without any additional kernel code and providing enough features for management of complex networking setups.

We summarized the main requirements which are needed for tool of this kind and described it's design. We have written it in C and designed our own domain specific language for configuration file style of running the application in addition to the usage of the C library liblsdn via the C API.

Next we described the network model and all the internals of LSDN. We have shown some examples how to run LSDN and presented its features showing the suitability for deploying it in a cloud environment. Furthermore we mentioned various kernel bugs we fixed or reported.

LSDN is in a very early stage and there are a lot of plans for future development. For example we would like to stabilize its API and try to spread the word and convince the community that it is an elegant way how to configure complex network setups. Furthermore there are some features that we consider useful and could be improved upon straight away. Some of them rely on things that the kernel learned to do in the last months of the project, or that we have discovered recently - the egress qdisc or better default disciplines (CoDEL was suggested). We would also like to improve the firewall (rewrite the rule engine and add support for ACCEPT actions).

References

[1] Bellard, F. 2005. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 41–41. USENIX Association.

[2] Docker Home Page. https://www.docker.com/. Accessed: 2018-06-25.

[3] Kubernetes Home Page. https://kubernetes.io/. Accessed: 2018-06-25.

[4] LSDN Github Project Page. http://www.github.com/asch/lsdn. Accessed: 2018-06-25.

[5] Linux Containers Home Page. https://linuxcontainers.org/. Accessed: 2018-06-25.

[6] Kubernetes Home Page. https://www.openstack.org/. Accessed: 2018-06-25.

[7] Open vSwitch Home Page. https://www.openvswitch.org/. Accessed: 2018-06-25.

[8] net: sched: introduce multichain support for filters. https://www.mail-archive.com/netdev@vger.kernel.org/msg168662.html. Accessed: 2018-06-25.

[9] Salim, J. H. 2015. Linux traffic control classifier-action subsystem architecture.

[10] Sridhar, T.; Kreeger, L.; Wright, C.; Dutt, D. G.; Bursell, M.; Mahalingam, M.; Agarwal, P.; and Duda, K. 2014. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks.

[11] Oracle VM VirtualBox Home Page. https://www.virtualbox.org/. Accessed: 2018-06-25.

[12] VMWare Home Page. https://www.vmware.com/. Accessed: 2018-06-25.

Appendix A

```
# Boilerplate
namespace import lsdn::*
# Choose the network tunneling technology
settings geneve

# Define the two virtual networks we have mentioned
net 1
net 2

# Describe the network
phys -name A -if eth0 -ip "192.168.10.1" {
    attach 1 2
    virt -name 1 -if tap0 -mac "14:9b:dd:6b:81:71" -net 1
    virt -name 2 -if tap1 -mac "92:89:90:93:61:75" -net 2
}

phys -name B -if eth0 -ip "192.168.10.2" {
    attach 1 2
    virt -name 3 -if tap0 -mac "42:94:a5:f9:69:c6" -net 1
    virt -name 4 -if tap1 -mac "f2:9b:4f:48:2d:d1" -net 2
}

# Tell LSDN what machine we are configuring right now
# (first commandline argument must contain the phys. machine name)
claimLocal [lindex $argv 0]
# Activate everything
commit
```

Appendix B

```
#include <assert.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#include <lsdn/lsdn.h>

/* Use the default GENEVE port */
static uint16_t geneve_port = 6081;

static struct lsdn_context *ctx;
static struct lsdn_settings *settings;
static struct lsdn_net *net1, *net2;
static struct lsdn_phys *machine1, *machine2;
static struct lsdn_virt *VM1, *VM2, *VM3, *VM4;

int main(int argc, const char* argv[])
{
    /* On the command line pass in the machine name on which the
     * program is being run. In our case the names will be either
     * A or B. */
    assert(argc == 2);

    /* Create a new LSDN context */
    ctx = lsdn_context_new("quickstart");
    lsdn_context_abort_on_nomem(ctx);

    /* Create new GENEVE network settings */
    settings = lsdn_settings_new_geneve(ctx, geneve_port);

    /* Create Machine 1 */
    machine1 = lsdn_phys_new(ctx);
    lsdn_phys_set_ip(machine1, LSDN_MK_IPV4(192, 168, 10, 1));
    lsdn_phys_set_iface(machine1, "eth0");
    lsdn_phys_set_name(machine1, "A");

    /* Create Machine 2 */
    machine2 = lsdn_phys_new(ctx);
    lsdn_phys_set_ip(machine2, LSDN_MK_IPV4(192, 168, 10, 2));
    lsdn_phys_set_iface(machine2, "eth0");
    lsdn_phys_set_name(machine2, "B");

    /* Create net1 */
    net1 = lsdn_net_new(settings, 1);

    /* Attach net1 */
    lsdn_phys_attach(machine1, net1);
    lsdn_phys_attach(machine2, net1);

    /* Create net2 */
    net2 = lsdn_net_new(settings, 2);

    /* Attach net2 */
    lsdn_phys_attach(machine1, net2);
    lsdn_phys_attach(machine2, net2);

    /* Create VM1 */
    VM1 = lsdn_virt_new(net1);
    lsdn_virt_connect(VM1, machine1, "tap0");
    lsdn_virt_set_mac(VM1, LSDN_MK_MAC(0x14,0x9b,0xdd,0x6b,0x81,0x71));
    lsdn_virt_set_name(VM1, "1");

    /* Create VM2 */
    VM2 = lsdn_virt_new(net2);
    lsdn_virt_connect(VM2, machine1, "tap1");
    lsdn_virt_set_mac(VM2, LSDN_MK_MAC(0x92,0x89,0x90,0x93,0x61,0x75));
    lsdn_virt_set_name(VM2, "2");

    /* Create VM3 */
    VM3 = lsdn_virt_new(net1);
    lsdn_virt_connect(VM3, machine2, "tap0");
    lsdn_virt_set_mac(VM3, LSDN_MK_MAC(0x42,0x94,0xa5,0xf9,0x69,0xc6));
    lsdn_virt_set_name(VM3, "3");

    /* Create VM4 */
    VM4 = lsdn_virt_new(net2);
    lsdn_virt_connect(VM4, machine2, "tap1");
    lsdn_virt_set_mac(VM4, LSDN_MK_MAC(0xf2,0x9b,0x4f,0x48,0x2d,0xd1));
    lsdn_virt_set_name(VM4, "4");

    /* Claim local A or B */
    struct lsdn_phys *local = lsdn_phys_by_name(ctx, argv[1]);
    assert(local != NULL);
    lsdn_phys_claim_local(local);

    /* Commit the created netmodel */
    lsdn_commit(ctx, lsdn_problem_stderr_handler, NULL);

    lsdn_context_free(ctx);
    return 0;
}
```