LSDN Documentation

LSDN Collective

Jul 07, 2018

Contents

CHAPTER 1

User's Documentation

1.1 Introduction

LSDN is a tool which allows you to easily configure networks with virtual machines (or containers) in Linux. It let's you configure network tunnels (*VXLAN*, *Geneve* ...) for separating groups of VMs into their own virtual networks.

Each virtual network behaves (from the perspective of a VM) as if all the computers were connected to a simple switch and were on the same LAN.

LSDN ensures isolation between networks using the existing network tunneling technologies. 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.

1.1.1 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.

Open vSwitch

LSDN intentionally does not use Open vSwitch to configure the tunnels, but only basic Linux networking (TC + flower classifier) to show that this is possible and can be made reasonably convenient.

Configuring let's say VLANs in this way is not very difficult, but it can be daunting if done for Geneve or static VXLANs.

1.2 Installation

1.2.1 System requirements

The following libraries are needed for compiling LSDN:

- tcl >= 8.6.0
- uthash
- libmnl
- json-c
- libdaemon
- linux-headers >= 4.14
- libvirt (optional, only for libvirt demos)

You will also need CMake and (naturally) GCC for building the packages.

If you are running **Ubuntu** or **Debain**, run:

```
apt install tcl-dev uthash-dev libmnl-dev libjson-c-dev libdaemon-dev libvirt-dev␣
↪cmake build-essential
```

If you are running **Arch Linux**, run:

```
pacman -S tcl uthash libmnl json-c libdaemon libvirt cmake
```

If you are running **CentOS 7**, run:

```
yum install gcc cmake uthash-devel libmnl-devel libdaemon-devel json-c-devel
```

However, you will need the `EPEL` repository for the `uthash` package and you will need to install `tcl-devel` package from a different source (for example Psychotic Ninja).

If you are running **openSUSE**, run:

```
zypper install gcc cmake uthash libmnl-devel libdaemon-devel linux-glibc-devel tcl-
↪devel libjson-c-devel
```

You will also need fairly recent Linux Kernel headers (at least 4.14) to build LSDN. To actually run LSDN, we recommend 4.15, as 4.14 still has some bugs in the used networking technologies and you might encounter crashes. This means you will either need to run a recent version of your distribution or install the kernel manually.

The exception is RHEL and CentOS – they backport features into their kernels very agressively and you migh be get lucky even with an 3.x CentOS/RHEL kernel.

If you do not plan on running LSDN on your machine, it is also possible to install just the kernel headers by running:

```
make headers_install INSTALL_HDR_PATH=$header_dir
```

1.2.2 Building from source

Simply install all the required software listed above and run these commands in the directory where you put the downloaded sources:

```
mkdir build
cd build
cmake ..
make
sudo make install
```

Now try running `lsctl` to see if the package was installed correctly.

If `lsctl` does not run correctly, check if your distribution looks for libraries in `/usr/local/lib` or `/usr/local/lib64`. Some distributions do not look for libraries in user-installed paths by default. To check the effective linker path, use `ldconfig -v 2>/dev/null | grep -v ^$'\t'`.

If you have installed kernel headers manually (see previous section), instead of running `cmake ..`, run:

```
cmake -DKERNEL_HEADERS=$header_dir/include ..
```

1.2.3 Building packages

This project contains instruction files for building packages for various distributions of Linux.

Arch

The PKGBUILD file for Arch Linux is located in `dist/arch/` and the package can be built and installed as follows:

```
cd dist/arch/
makepkg
pacman -U lsdn*.tar.xz
```

If you do not want to build the package on your own, you can install lsdn with all it's dependencies directly from Arch User Repository (lsdn-git package):

```
pacaur -S lsdn-git # pacaur is AUR helper of our choice
```

Debian

See https://wiki.archlinux.org/index.php/installation_guide :)

Jokes aside, run:

```
ln -s dist/debian .
dpkg-buildpackage
```

The packages `lsdn` and `lsdn-dev` will be available in the parent folder.

RPM-based distributions

Spec file is located in `dist/rpm` directory. In addition, a convenience build script is provided. Make sure your system has all the dependencies (see above) and also `rpm-build`. Then, run:

```
sh dist/rpm/rpmbuild.sh
```

Resulting rpms will be placed in the `dist/rpm` directory.

1.2.4 Running tests

LSCTL includes a test-suite that constructs various virtual networks and tries pinging VMs inside those networks. `sudo make test` starts these tests.

If you plan on developing LSDN, you might want to run the tests inside another level of VM. There is already a testing environment ready for those purposes, built on QEMU and minimal Arch root file system. More information can be found in the Developer documentation section *Test Environment*.

1.3 Quick-Start

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 Fig. 1.1.

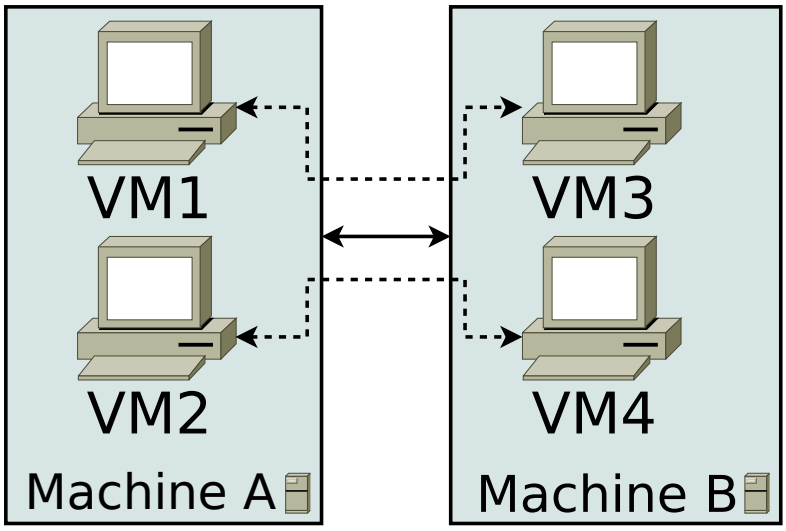


Fig. 1.1: Network setup. Solid lines are physical machine connections, dashed lines denote communication between virtual machines.

VMs *1* and *3* can communicate with each other and so can VMs *2* and *4*. This means we will create two virtual networks, one for VM *1* and *3*, second for VM *2* and *4*.

As mentioned in the *Introduction*, there are two major ways to use LSDN – *configuration files* and *C API*. We will look at both possibilities.

1.3.1 Setting up virtual machines

You are free to use any Virtual Machine Manager you like: bare Qemu/KVM, libvirt or VirtualBox or run containers (via LXC for example). The only thing LSDN needs to know is which network interfaces on the host are assigned to the virtual machines. Typically, this will be a *tap* interface for a VM and a *veth* interface for a container.

Qemu

If you are just trying out LSDN, we suggest you download some live distro (like Alpine Linux) and run Qemu/KVM. First make sure QEMU is installed and then on physical machine *A* run:

```
sudo qemu-system-x86_64 -enable-kvm -m 256 \
    -cdrom $iso_path.iso \
    -netdev type=tap,ifname=tap0,script=no,downscript=no,id=net0 \
    -device virtio-net-pci,netdev=net0,mac=14:9b:dd:6b:81:71
```

This will start up the Live ISO. Now login into the VM and setup a simple IP configuration:

```
ip addr change dev eth0 192.168.0.1/24
ip link set eth0 up
```

Do the same for the remaining virtual machines, but each time with a different MAC and TAP interface name. There is no need to change the `net0` strings:

- on *A* create VM using `ifname=tap0`, `mac=14:9b:dd:6b:81:71` and set up IP address as `192.168.0.1` (we just did that in example above).
- on *A* create VM using `ifname=tap1`, `mac=92:89:90:93:61:75` and set up IP address as `192.168.0.2`
- on *B* create VM using `ifname=tap0`, `mac=42:94:a5:f9:69:c6` and set up IP address as `192.168.0.3`
- on *B* create VM using `ifname=tap1`, `mac=f2:9b:4f:48:2d:d1` and set up IP address as `192.168.0.4`

Libvirt

If you are using Libvirt, set up the virtual machines as usual. Unfortunately, `virt-manager` can not be told to leave the VM's networking alone. It will try to connect it to a network, but that's what LSDN will be used for! It can also not change an interface MAC address. Instead, use `virsh edit` to manually change the VM's XML. Change the `interface` tag of VM *1* on *A* to look like this:

```
<interface type='ethernet'>
  <mac address='14:9b:dd:6b:81:71'/>
  <script path='/usr/bin/true'/>
  <target dev='tap0'/>
  <!-- original <model> and <address> -->
 </interface>
```

Also change the other virtual machines but with different MAC and TAP interface names (look at the *Qemu* section for correct values).

1.3.2 Using configuration files

Now that we have set-up the virtual machines, we can use LSDN to connect them. We will start with an example using the configuration files (as opposed to the *C API*), as it is simpler.

First, create the file `config.lsctl` with the following contents:

```
# 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
```

Naturally, if you are using different IP addresses for your physical machines, change the configuration file. Also pay attention to the `-if eth0` arguments – they tell LSDN what interface you use for connecting machines *A* and *B* together and you may also need to change the interface to reflect your physical setup.

Then make sure the file is available on both physical machines *A* and *B* and run the following commands:

- on *A*: `$ lsctl config.lsctl A`
- on *B*: `$ lsctl config.lsctl B`

Congratulations, your network is set-up. Try it:

- in VM *1*: `$ ping 192.168.0.3`
- in VM *2*: `$ ping 192.168.0.4`

And they are correctly isolated too, since `$ ping 192.168.0.2` won't work in VM *1*.

1.3.3 Using the C API

The equivalent network setup created using the LSDN *C API*:

```c
#include <assert.h>
#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);
```

(continues on next page)

(continued from previous page)

```
    /* 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;
}
```

Afterwards compile the program for machines A and B and link them together with the LSDN library. Call the resulting executables `quickstart` and run the respective executables on the two machines:

- on A: `$ ./quickstart A`
- on B: `$ ./quickstart B`

Your network is now set-up using the C API. Try:

- in VM *1*: `$ ping 192.168.0.3`
- in VM *2*: `$ ping 192.168.0.4`

And they are correctly isolated too, since `$ ping 192.168.0.2` won't work in VM *1*.

1.4 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[1].

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 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 *the export/dump mechanism* will always be able to create cross-references.

[1] In theory LSDN could figure out if a *phys* should be attached to a *net* just by checking if any of its *virts* are attached to that *net*. But we have decided to make this explicit. LSDN checks if *physes* connected to the same *net* have certain properties (for example their IP address must use the same IP version) and we did not want to make these checks implicit. A switch may be provided in future versions, though.

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.

Note: LSDN is not thread-safe. It assumes that a given context is never accessed concurrently. Different context can be accessed concurrently.

1.4.1 Networks and their settings

LSCTL: `net`, `settings`

C API: `lsdn_net_new()` and various `lsdn_settings_*`.

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[2].

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 chapter *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 a the name `default` for unnamed settings.

1.4.2 Virts

LSCTL: `virt`

C API: `lsdn_virt_new()`, `lsdn_virt_connect()`, `lsdn_virt_set_mac()`

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 it's *virts* will be removed as well.

[2] In theory, they could overlap if the *nets* are always connected to different *physes* (and so there are is no ambiguity), but LSDN still checks that they are globally unique.

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 rules

LSCTL: `rule`

C API: `lsdn_vr_new()` and other functions (see *Rules engine*)

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

LSCTL: `rate`

C API: `lsdn_virt_set_rate_in()`, `lsdn_virt_set_rate_out()`

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 (the MTU). Setting *burst_rate* to zero will not work.

1.4.3 Physes

LSCTL: `phys`, `attach`, `claimLocal`

C API: `lsdn_phys_new()`, `lsdn_phys_set_ip()`, `lsdn_phys_claim_local()`

physes are used to described 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 and *ip* attribute, specifying IP address for the network overlay technologies that require it.

1.4.4 Validation

LSCTL: `validate`

C API: `lsdn_validate()`

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 `claim` (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*.

1.4.5 Commit

LSCTL: *`commit`*

C API: *`lsdn_commit()`*

Commiting 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 iff it has been claimed local (either with *`claimLocal`* or *`lsdn_phys_claim_local()`*),
- *virt* is local iff 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*
- *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 occured 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.

1.4.6 Error Handling

C API: *lsdn_context_set_nomem_callback()*, *lsdn_context_abort_on_nomem()*, lsdn_err_t

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.

1.4.7 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:

Category	Description
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

1.4.8 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

Available as: `settings vlan` (lsctl), `lsdn_settings_new_vlan()` (C API).

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:

- 12 bit *vid*
- Physical nodes in the same virtual network must by located on the same Ethernet network

- Care must be taken when nesting

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.

Note: 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

Available as: `settings vxlan/mcast` (lsctl), `lsdn_settings_new_vxlan_mcast()` (C API).

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.

Restrictions:

- 24 bit *vid*
- Physical nodes in the same virtual network must be reachable on the IP layer
- UDP and IP header overhead
- Requires multicast support

Endpoint-to-Endpoint

Available as: `settings vxlan/e2e` (lsctl), `lsdn_settings_new_vxlan_e2e()` (C API).

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.

Restrictions:

- 24 bit *vid*
- Physical nodes in the same virtual network must be reachable on the IP layer
- UDP and IP header overhead
- Unknown and broadcast packets are duplicated for each physical machine

Fully static

Available as: `settings vxlan/static` (lsctl), `lsdn_settings_new_vxlan_static()` (C API).

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.

Restrictions:

- 24 bit *vid*
- Physical nodes in the same virtual network must be reachable on the IP layer
- UDP and IP header overhead
- Unknown and broadcast packets are duplicated for each physical machine
- The virtual network is not fully opaque (MAC addresses of virtual machines must be known).

Geneve

Available as: `settings geneve` (lsctl), `lsdn_settings_new_geneve()` (C API).

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.

Restrictions:

- 24 bit *vid*
- Physical nodes in the same virtual network must be reachable on the IP layer
- UDP and IP header overhead
- Unknown and broadcast packets are duplicated for each physical machine
- The virtual network is not fully opaque (MAC addresses of virtual machines must be known).

No tunneling

Available as: `settings direct` (lsctl), `lsdn_settings_new_direct()` (C API).

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 *phys*,
- 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`).

1.5 Lsctl Configuration Files

LSDN has its own configuration language for describing the network topology. If you saw the *Quick-Start* or *Network representation* sections, you have already seen examples of the configuration files.

1.5.1 Syntax

The configuration syntax is actually based on the TCL language – but you do not have to be afraid, this guide is self-contained. You might not even guess TCL is there if we did not tell you. However, it is good to know TCL is there if you need more advanced stuff, like variable substitution or loops. And if you know TCL, you will recognize some of the conventions and feel at home.

The only downside is that you have to include a short boilerplate at the top of each configuration file to tell TCL that you don't want to prefix everything with `lsdn::`. Start your configuration file with this line:

```
namespace import lsdn::*
```

The configuration file itself is a list of directives that tell LSDN what objects the network consists of, how they are configured and how they connect. The directives are terminated by a newline. Other white-space is not significant. All available directives are listed in section *Directive reference*.

All LSDN directives follow the same basic patterns. They start with the directive name (for example `net` or `settings`) and are followed with argument for that directive. Directives and their arguments are separated by white-space. Some directives go without an argument. Other directives make use of named arguments (or as they are called in some languages "keyword arguments"):

```
directiveWithNamedArgs -opt1 value1 -opt2 value2 -opt3WithoutAnyValue
```

The directives may combine named and regular arguments. In that case, consult the documentation for the particular directive, if the regular (non-keyword) arguments should be placed before or after the keyword ones.

If you need the directive to span multiple lines, use the backslash `\` continuation character as you do in shell:

```
virt -name critical_server_with_unknown_purpose -if enps0 \
    -mac F9:9B:81:C2:66:F9 -net service_net
```

1.5.2 Names

Objects (physical machines, virtual machines etc.) in LSCTL are given names, to allow you to refer to them later. The name must be given when a configuration directive is

used to create an object. You are free to choose any name you like, as the names do not have any direct impact on the network.

Please note that forward references are not allowed, because in its core LSCTL is a procedural language. For example, this snippet will not work:

```
virt -net test
net -vid 1 test {}
```

Any directive referencing an undefined object will return an error like this:

```
can not find network
```

The order must be swapped like this:

```
net -vid 1 test {}
virt -net test
```

If a directive references an undefined object, it will print a stack-trace and the script execution will end.

1.5.3 Nesting

Some directives may also contain other directives. In that case, the nested directives are enclosed in curly-braces `{}` following the parent directive like this[1]:

```
phys -name p
net 42 {
    virt -name a -phys p
    attach p
}
```

Nested directives are used to simplify definition of related elements. The example above specifies a virtual machine *a* connected to network *42*. The connection is implied from the nesting. The nesting is quite flexible and you can decide to use a style that most suits you and that reflects organization of your network. An equivalent way to write the example above would be:

```
phys -name p
net 42
virt -name a -net 42 -phys p
attach -phys p -net 42
```

Or:

```
net 42
phys -name p {
    virt -name a -net 42
```

(continues on next page)

[1] If you are familiar with TCL, you will recognize this is how TCL control-flow commands work.

(continued from previous page)

```
}
attach -phys p -net 42
```

Or:

```
net 42 {
    phys -name p {
        virt -name a
    }
}
```

Note that there is no need for `attach` in the last example, since nesting took care of it for us.

In general, nesting can be used anywhere you would otherwise have to specify a relationship using arguments. Other nestings are disallowed. The supported nestings are:

- *virt* in *net* = *virt* will be connected to the *net*
- *virt* in *phys* = *virt* will be connected at this *phys*
- *net* in *phys* = *phys* will be attached to the *net*
- *phys* in *net* = *phys* will be attached to the *net*
- *attach* in *net* = *net* will be attached to phys given as argument
- *attach* in *phys* = nets given as arguments will be attached to *phys*
- *detach* follows the same rules
- *claimLocal* in *phys* = *phys* will be claimed as local

Some directives are only settings for one object (and do not imply any relationship). These are the `rate` (for virt QoS) and `rules` (for virt firewall) directives. They **must** be nested under a `virt` directive.

1.5.4 Argument types

`int`
: An integer number, given as string of digits optionally prefixed with a sign. LSCTL recognizes the `0x` prefix for hexadecimal and `0` for octal integers.

`string`
: String arguments in LSCTL are given the same way as in shell - they don't need to be quoted. Mostly they are used for names, so there is no need to give string arguments containing spaces.

  If you want to give a directive an argument containing spaces, newlines or curly brackets, simply enclose the argument in double-quotes. If you want the argument to contain double-quotes, backslash or dollar sign, precede the character with backslash:

```
virt -name "really\$bad\\idea
on so many levels"
```

If you need the full syntax definition, refer to `man tcl.n` on your system.

`direction`
: Either `in` or `out`. `in` is for packets entering the virtual machine, `out` is for packets leaving the virtual machine.

`ip`
: IP address, either IPv6 or IPv4. Common IPv6 and IPv4 formats are supported.

  For exact specification, refer to `inet_pton` function in C library.

  Examples:

```
2a00:1028:8380:f86::2
192.168.56.1
```

`subNet`
: IP address optionally followed by `/` and prefix size. If the prefix size is not given, it is equivalent to 128 for IPv6 and 32 for IPv4, that is subnet containing the single IP address.

  Instead of writing a prefix after the `/`, a network mask can be given, using the same format as for the IP address.

  Examples:

```
2a00:1028:8380:f86::2
2a00:1028:8380:f86::0/64
192.168.56.0/24
192.168.56.0/255.255.255.0
```

`mac`
: MAC address in octal format. Both addresses with colons and without colons are supported, as long as the colons are consistent. Case-insensitive

```
9F:1A:C1:4C:EE:0B
9f1ac14cee0b
```

`size`
: An unsigned decimal integer specifying a number of bytes. Suffices `kb`, `mb`, `gb` and `bit`, `kbit`, `mbit`, `gbit` can be given to change the unit. All units are 1024-base (not 1000), despite their SI names. This is for compatibility with the `tc` tool from `iproute` package, which uses the same units.

`speed`
: An unsigned decimal integer specifying a number of bytes per second.

  Supported units are the same as for *`size`*.

1.5.5 Directive reference

net name -vid -settings -phys -remove { … }
Define new virtual network or change an existing one.

C API equivalents: *lsdn_net_new()*, *lsdn_net_by_name()*.

Parameters

- **vid** (int) – Virtual network identifier. Network technologies like VXLANs or VLANs use these numbers to separate different networks. The ID must be unique among all networks of the same network type. The parameter is forbidden if network already exists. The permissible range of network identifiers differs for individual network types (see *Network representation*).
- **name** (string) – Name of the network. Does not change network behavior, only used by the configuration to refer to the network. However, if the `-vid` argument is not specified, this `name` argument will also specify the `vid`.
- **phys** (string) – Optional name of a *phys* you want to attach to this network. Shorthand for using the *attach* directive. Can not be used when nested inside *phys* directive.
- **settings** (string) – Optional name of a previously defined *settings*, specifying the network overlay type (VLAN, VXLAN etc.). If not given, the `default` settings will be used. Settings of existing net can not be changed.
- **remove** – Optional, remove the network. This will effectively also remove any child object (e.g. any *virt* inside this network).

Scopes

- **none** – This directive can appear at root level.
- ***phys*** – Automatically attaches the parent phys to this network. Shorthand for using the *attach* directive.

phys -name -if -ip -net -remove -ifClear -ipClear
Define a new physical machine or change an existing one.

C API equivalents: *lsdn_phys_new()*, *lsdn_phys_by_name()*.

Parameters

- **name** (string) – Optional, name of the physical machine. Does not change network behavior, only used by the configuration to refer to the phys.
- **if** (string) – Optional, set the network interface name this phys uses to communicate with the physical network.
- **ip** (ip) – Optional, set the IP address assigned to the phys on the physical network.

- **net** (string) – Optional, name of a *net* you want this phys to attach to. Shorthand for using the *attach* directive. Can not be used when nested inside *net* directive.
- **remove** – Optional, remove the physical machine. This will effectively also disconnect any *virt* residing on this machine.
- **ifClear** – Optional, clear the machine's interface name, if any.
- **ipClear** – Optional, clear the IP address of the *phys*, if any.

Scopes

- **none** – This directive can appear at root level.
- ***net*** – Automatically attaches this phys to the parent network. Shorthand for using the *attach* directive.

virt -net -name -mac -phys -if -remove -macClear

Define a new virtual machine or change an existing one.

C API equivalents: *lsdn_virt_new()*, *lsdn_virt_by_name()*.

Parameters

- **net** (string) – The virtual network this virt should be part of. Mandatory if creating new virt, forbidden if changing an existing one. Forbidden if nested inside *net*.
- **name** (string) – Optional, name of the virtual machine. Does not change network behavior, only used by the configuration to refer to this virt.
- **mac** (mac) – Optional, MAC address used by the virtual machine.
- **phys** (string) – Optional, connect (or migrate, if already connected) at a given *phys*.
- **if** (string) – Set the network interface used by the virtual machine to connect at the phys. Mandatory, if `-phys` argument was used.
- **remove** – Optional, remove the virtual machine.
- **macClear** – Optional, clear the virtual machine's MAC address, if any.

Scopes

- **none** – This directive can appear at root level.
- ***net*** – Equivalent with giving the `-net` parameter and thus mutually exclusive.
- ***phys*** – Equivalent with giving the `-phys` parameter and thus mutually exclusive

attach -phys -net

attach -phys netlist

attach -net physlist

Attaches a given physical machine(s) to a virtual network(s). The command can either attach a single phys to a single net (using the `-phys` and `-net` arguments) or to multiple nets at once (using the `-phys` argument and positional arguments) or attach multiple physes to a single network (using the `-net` argument and positional arguments).

If scoped, the `-net` or `-phys` arguments are implicit, so you can easily attach a phys to multiple nets like this:

```
phys test {
    attach net1 net2
}
```

Scopes

- **root** – This directive can appear at root level.
- ***net*** – Equivalent with giving the `-net` parameter and thus mutually exclusive.
- ***phys*** – Equivalent with giving the `-phys` parameter and thus mutually exclusive

detach -phys -net

detach -phys netlist

detach -net physlist

Detaches the virtual networks from physical machines. See *attach* for syntax of the command.

rule direction prio action -srcIp -dstIp -srcMac -dstMac

Add a new firewall rule for a given virt. The rule applies if all the matches specified by the arguments are satisfied.

C API equivalents: *lsdn_vr_new()* and other functions (see *Rules engine*)

Parameters

- **direction** (direction) – Direction of the packets.
- **prio** (int) – Priority of the rule. Rules with lower numbers are matched first.
- **action** (string) – Currently only drop action is supported.
- **srcIp** (subNet) – Match if the source IP address of the packet is in the given subnet.
- **dstIp** (subNet) – Match if the destination IP address of the packet is in the given subnet.

- **srcMac** (mac) – Match if the source MAC address of the packet is equal to the given one.
- **dstMac** (mac) – Match if the source MAC address of the packet is equal to the given one.

Scopes

- ***virt*** – Only allowed in a virt scope.

flushVr
Remove all virt firewall rules defined by *rule* previously.

Scopes

- ***virt*** – Only allowed in a virt scope.

rate direction -avg -burst -burstRate
Limit bandwidth in a given direction. If no arguments are given, all limits are lifted.

C API equivalents: *lsdn_virt_set_rate_in()*, *lsdn_virt_set_rate_out()*, *lsdn_virt_clear_rate_in()*, *lsdn_virt_clear_rate_out()*.

Parameters

- **direction** (direction) – Direction to limit.
- **avg** (speed) – Average speed allowed.
- **burstRate** (speed) – Higher speed allowed during short bursts.
- **burst** (size) – Size of the burst during which higher speeds are allowed.

Scopes

- ***virt*** – Only allowed in a virt scope.

claimLocal -phys
Inform LSDN that it is running on this physical machine.

You might want to distribute the same configuration file to all physical machines, just with different physical machines claimed as local. You can use the following command to allow the control of the local phys using the first command-line argument to the script:

```
claimLocal [lindex $argv 0]
```

After that, invoke *lsctl* like this:

```
lsctl <your script> <local phys>
```

C API equivalents: *lsdn_phys_claim_local()*.

Parameters

- **phys** (string) – The phys to mark as local.

Scopes

- **none** – This directive can appear at root level.
- ***phys*** – Equivalent to specifying the `-phys` parameter.

settings type
Set a network overlay type for newly defined networks. Use one of the concrete overloads below.

settings direct -name
Do not use any network separation.

See *No tunneling* for more details.

Parameters

- **name** (`string`) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.

Scopes

- **none** – This directive can only appear at root level.

settings vlan -name
Use VLAN tagging to separate networks.

See *VLAN* for more details.

Parameters

- **name** (`string`) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.

Scopes

- **none** – This directive can only appear at root level.

settings vxlan/mcast -name -mcastIp -port
Use VXLAN tunnelling with automatic setup using multicast.

See *Multicast* VXLAN for more details.

Parameters

- **name** (`string`) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.
- **mcastIp** (`ip`) – Mandatory, the IP address used for VXLAN broadcast communication. Must be a valid multicast IP address.
- **port** (`int`) – Optional, the UDP port used for VXLAN communication.

Scopes

- **none** – This directive can only appear at root level.

settings vxlan/e2e -name -port
Use VXLAN tunnelling with endpoint-to-endpoint communication and MAC learning.

See *Endpoint-to-Endpoint* VXLAN for more details.

Parameters

- **name** (string) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.
- **port** (int) – Optional, the UDP port used for VXLAN communication.

Scopes

- **none** – This directive can only appear at root level.

settings vxlan/static -name -port
Use VXLAN tunnelling with fully static setup.

See *Fully static* VXLAN for more details.

Parameters

- **name** (string) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.
- **port** (int) – Optional, the UDP port used for VXLAN communication.

Scopes

- **none** – This directive can only appear at root level.

settings geneve -name -port
Use Geneve tunnelling with fully static setup.

See *Geneve* for more details.

Parameters

- **name** (string) – Optional, creates a non-default named setting. Use the *net* `-setting` argument to select.
- **port** (int) – Optional, the UDP port used for Geneve communication.

Scopes

- **none** – This directive can only appear at root level.

commit
Apply all changes done so far. This will usually be located at the end of each LSCTL script.

If the validation or commit fails, the errors will be printed to stderr and the directive will end with an error. The script will be terminated.

C API equivalents: *lsdn_commit()*

Scopes

- **none** – This directive can only appear at root level.

validate
Check the network model for errors.

If the validation fails, the errors will be printed to stderr and the directive will end with an error. The script will be terminated.

C API equivalents: *lsdn_validate()*

Scopes

- **none** – This directive can only appear at root level.

cleanup
Revert all changes done so far.

If the cleanup fails, the errors will be printed to stderr and the directive will end with an error. The script will be terminated.

C API equivalents: *lsdn_context_cleanup()*

Scopes

- **none** – This directive can only appear at root level.

show -tcl -json
Show the network model so far. Shows even changes that are not yet commited.

Parameters

- **tcl** – Dump the network model in LSCTL format.
- **json** – Dump the network model in JSON format.

C API equivalents: *lsdn_dump_context_tcl()*, *lsdn_dump_context_json()*.

Scopes

- **none** – This directive can only appear at root level.

free
Free all the resources used by LSDN, but do not revert the changes. This is useful for memory leak debugging (Valgrind etc.).

C API equivalents: *lsdn_context_free()*

Scopes

- **none** – This directive can only appear at root level.

1.5.6 Command-line tools

The LSCTL configuration language is accepted by the command-line tools *lsctl* and *lsctld*. The one you should choose depends on your use-case. *lsctl* is used for simple

run-and-forget configuration, while *lsctld* runs in the background and supports virtual machine migration and other types of network evolution.

Using lsctl

Run `lsctl` with the name of your configuration script like this:

```
lsctl my_configuration.lsctl
```

You can also pass additional arguments to lsctl, which will be all available in the `$argv` variable. See *`claimLocal`* for an example use.

If you run `lsctl` without arguments, you will receive an interactive shell, where you can enter directives one after another.

Using lsctld and lsctlc

If you want to migrate machines, you have to keep a `lsctld` daemon running in the background, so that it can remember the current state of the network and make changes appropriately. You can send new configuration directives to the daemon using the `lsctlc` command.

First, let's decide on the location of the control socket for `lsctld`. `lsctld` uses a regular Unix socket that can be located anywhere on the file-system, so let's use `/var/run/lsdn`:

```
lsctld -s /var/run/lsdn
```

Afterwards, commands can be sent to `lsctld` using `lsctlc`. Either pass them on standard input:

```
cat my_configuration.lsctl | lsctlc /var/run/lsdn
```

Or directly on the command-line:

```
lsctlc /var/run/lsdn virt -name vm1 -phys b -net customer
lsctlc /var/run/lsdn commit
```

`lsctld` can be controlled with the following options:

`--socket, -s`
: Specify the location of the Unix control socket (mandatory).

`--pidfile, -p`
: Specify the location of the PID file. `lsctld` will use the PID file to prevent multiple instances from running and it can be used for daemon management.

  If the option is not specified, no PID file will be created.

`-f`
: Run in foreground, do not daemonize.

TCL extension (tclsh)

Instead of using the *lsctl* command-line tool, you can use TCL directly and load LSDN as an extension. This will allow you to combine LSDN with larger TCL programs and run it using `tclsh`. This can be done using the regular TCL means:

```
package require lsdn
namespace import lsdn::*

net test { ... }
```

1.6 Examples

Now is the right time to describe through a couple of examples how LSDN can actually be used. The following are very simple (but complete nonetheless) examples of virtual networks that can be set up through LSDN. They should be descriptive enough to get you started with your own use-cases.

1.6.1 Example 1 - Basic Principles

In the first example let's imagine we have three computers, **A**, **B** and **C**, and that we are managing the networking infrastructure of two local businesses - a bookstore and a bakery. The local businesses have their own software running inside virtual machines (VMs) hosted by our three computers **A**, **B** and **C**. It is expected that the virtual machines of the bookstore will be able to communicate with each other on the network and likewise the VMs of the bakery.

It is also very desirable for the network traffic sent between the VMs of the bookstore not to be seen by the VMs of the bakery and vice versa.

The three computers hosting the two companies are perhaps connected to the same LAN, but it may not necessarily be the case as they quite as well might each be on a different continent. We don't really care. We only assume that **A**, **B** and **C** are able to send messages to each other. Without any further ado let's present the first complete LSDN network configuration file. Afterwards, we will split the example into smaller chunks and explain each section in detail.

```
namespace import lsdn::*

settings vxlan/static -name vxlan

phys -if eth0 -name A -ip 172.16.0.1
phys -if eth0 -name B -ip 172.16.0.2
phys -if eth0 -name C -ip 172.16.0.3

net -vid 1 Bookstore -settings vxlan {
```

(continues on next page)

(continued from previous page)

```
    attach A B C
    virt -phys A -if 1 -mac 00:00:00:00:00:a1
    virt -phys A -if 2 -mac 00:00:00:00:00:a2
    virt -phys B -if 1 -mac 00:00:00:00:00:b1
    virt -phys C -if 1 -mac 00:00:00:00:00:c1
}

net -vid 2 Bakery -settings vxlan {
    attach A B
    virt -phys A -if 3 -mac 00:00:00:00:00:a3
    virt -phys B -if 2 -mac 00:00:00:00:00:b2
}

claimLocal [lindex $argv 0]
commit
free
```

It might also be handy to actually see in picture how our networking infrastructure will look like once we're done:

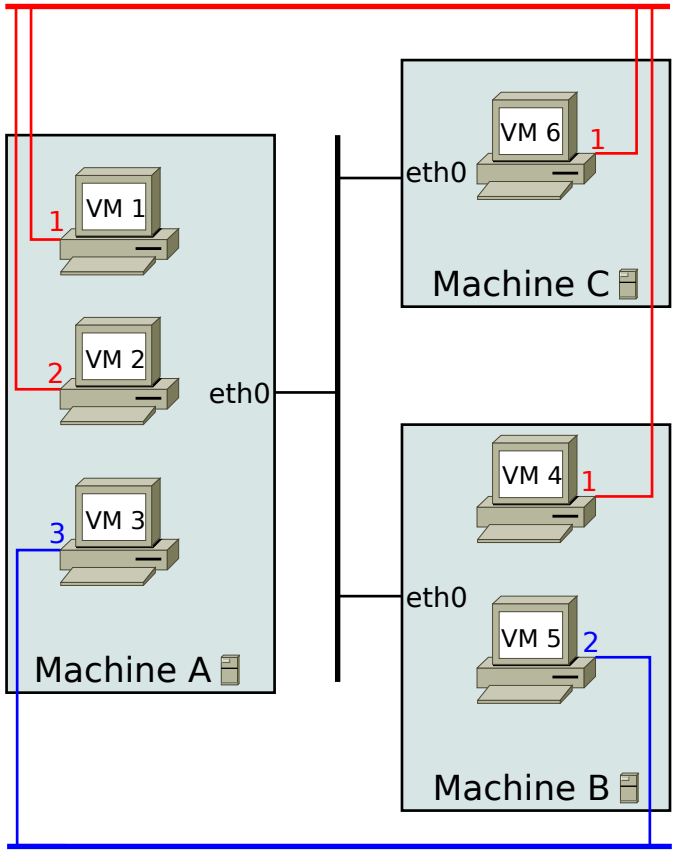


Fig. 1.2: Virtual networks in Example 1
On three physical machines we host 6 VMs, pertaining to two different virtual networks.

Now let's step through our configuration file, starting with:

```
settings vxlan/static -name vxlan
```

This line will create a VXLAN static virtual network settings, named *vxlan*. As we haven't specified any port for this network settings, the default VXLAN UDP port will be used (*VXLAN*).

The following lines

```
phys -if eth0 -name A -ip 172.16.0.1
phys -if eth0 -name B -ip 172.16.0.2
phys -if eth0 -name C -ip 172.16.0.3
```

describe three physical machines, or rather just three physical interfaces present on our three machines. We have given each interface a name, which corresponds to the name of the interface on the respective machines. And finally we have also set an IPv4 address for those three interfaces. It should be noted that it is expected that the physical interfaces have already been assigned an IP address and that they have been brought up as well. Maybe you're asking yourself why did we bother to specify the IP addresses of the interfaces in the configuration file then? That's because this information is needed when we are using the VXLAN tunnels.

```
net -vid 1 Bookstore -settings vxlan {
    attach A B C
    virt -phys A -if 1 -mac 00:00:00:00:00:a1
    virt -phys A -if 2 -mac 00:00:00:00:00:a2
    virt -phys B -if 1 -mac 00:00:00:00:00:b1
    virt -phys C -if 1 -mac 00:00:00:00:00:c1
}
```

Afterwards we describe a virtual network we are going to set up for the bookstore. We will call this virtual network conveniently just **Bookstore**. The **Bookstore** network will be tunneled through the VXLAN tunnels. We have assigned the network a virtual network identifier *1*. The network will span all the machines **A**, **B** and **C** - that's what we have written with the `attach` statement. The next line describes a virtual machine that will reside on machine **A**. It will connect via an interface which is simply called *1* (yes, interface can have arbitrary names). We have also assigned a MAC address to this virtual machine. Again, LSDN expects that an interface called *1* is already present on the physical machine **A** and that it is assigned the same MAC address we have given it in the configuration file. Similarly, the next three lines describe three other virtual machines inside the **Bookstore** network.

In a very similar fashion we have created a *Bakery* virtual network:

```
net -vid 2 Bakery -settings vxlan {
    attach A B
    virt -phys A -if 3 -mac 00:00:00:00:00:a3
    virt -phys B -if 2 -mac 00:00:00:00:00:b2
}
```

It has two virtual machines, but this time the virtual network spans only the physical machines **A** and **B**. Note that the **Bakery** virtual network is again going to be tunneled

inside a VXLAN tunnel, only with a different network identifier *2*.

This line:

```
claimLocal [lindex $argv 0]
```

will instruct LSDN which machine it should consider as being local. How this command exactly works is described in *claimLocal*.

If we don't want to perform just a dry run then we'd better tell LSDN to take the network model it has constructed up to this point parsing the configuration file and write (or *commit* in LSDN terminology) the model into the appropriate kernel data structures. That's exactly what's being done with the single command:

```
commit
```

The last line:

```
free
```

instructs LSDN to clean up it's internal network model stored in memory. For details consult *free*. Especially note this does not delete the networks stored in the kernel.

That was our first complete example. Now it remains to distribute this configuration file (let's name it *example1.lsctl*) to our three computers **A**, **B** and **C**. You may be wondering whether we didn't forget to show you two other configuration files so that we would have three files that we could then distribute to our three machines. In a moment you will see why it's not actually needed.

On machine **A** type:

```
lsctl example1.lsctl A
```

Similarly on machine **B**:

```
lsctl example1.lsctl B
```

and on machine **C**:

```
lsctl example1.lsctl C
```

By passing the command line parameter *A*, *B* or *C* to *lsctl* on the appropriate nodes, LSDN will be able to distinguish which machines are local.

That's it. Now your customers should be able to communicate inside the virtual networks we have just created.

Keeping all our networking configuration in a single file will hopefully make it easier for us to keep the networks in sync. But it is by no means the only way how to configure your networks using LSDN. You may perhaps prefer to keep and edit a configuration file on each physical machine separately; or you may have a separate configuration file for each virtual network. The possibilities are plentiful.

1.6.2 Example 2 - VM Migration

In the second example we will focus on one very important aspect of virtual networking - the problem of virtual machine migration. There are many reasons why we might want to migrate virtual machines between physical machines hosting them. For example we would like to do some planned maintenance on one of the physical machines so we need to take all the VMs hosted on this machine and migrate them (seamlessly if possible) to a different host in our infrastructure.

Let's jump right in and list the contents of the second configuration file which we're going to name *example2-1.lsctl*:

```
namespace import lsdn::*

settings vxlan/static

phys -if eth0 -name A -ip 172.16.0.1
phys -if eth0 -name B -ip 172.16.0.2
phys -if eth0 -name C -ip 172.16.0.3

net 1 {
    attach A B C
    virt -phys A -if 1 -mac 00:00:00:00:00:a1 -name migrator
    virt -phys A -if 2 -mac 00:00:00:00:00:a2
    virt -phys B -if 1 -mac 00:00:00:00:00:b1
    virt -phys C -if 1 -mac 00:00:00:00:00:c1
}
```

If you're not recognizing any of the syntax used in this configuration file, please refer to *Example 1 - Basic Principles*.

We will run the following commands on node **A**:

```
lsctld -s /var/run/lsdn/example2.sock
lsctlc /var/run/lsdn/example2.sock < example2-1.lsctl
lsctlc /var/run/lsdn/example2.sock claimLocal A
lsctlc /var/run/lsdn/example2.sock commit
```

and similarly on node **B**:

```
lsctld -s /var/run/lsdn/example2.sock
lsctlc /var/run/lsdn/example2.sock < example2-1.lsctl
lsctlc /var/run/lsdn/example2.sock claimLocal B
lsctlc /var/run/lsdn/example2.sock commit
```

and node **C**:

```
lsctld -s /var/run/lsdn/example2.sock
lsctlc /var/run/lsdn/example2.sock < example2-1.lsctl
lsctlc /var/run/lsdn/example2.sock claimLocal C
lsctlc /var/run/lsdn/example2.sock commit
```

Again, the VMs inside the virtual network should now be able to reach each other on the network.

Maybe after some time we realize it would be better to move the *migrator* VM from node **A** to node **B**. We instruct LSDN to migrate this virtual machine with the following commands run on each of the machines **A**, **B** and **C**:

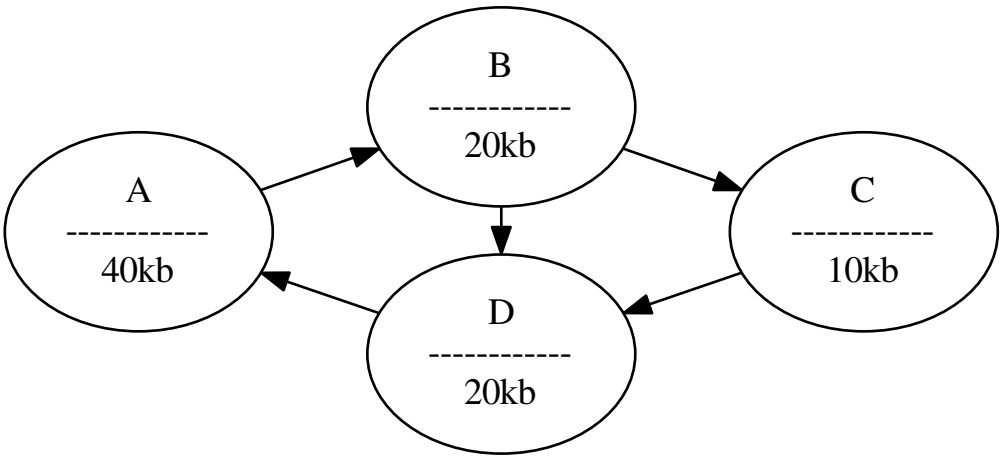
```
lsctlc /var/run/lsdn/example2.sock virt -phys B -if 2 -name migrator -net 1
lsctlc /var/run/lsdn/example2.sock commit
```

What effectively happened is the *migrator* VM was disconnected from the virtual network on node **A** and reconnected back again on node **B**.

It is important to note we have to perform this update on all nodes **A**, **B** and **C**. Had we decided to create for example a VLAN virtual network then we would not have to update the LSDN netmodel on machine **C**. Regardless of the network settings type (e.g. VXLAN, GENEVE) created for out virtual networks, it is always safe to run the same updates on all physical machines hosting the virtual networks even if some nodes might not be impacted by any of the performed change.

1.6.3 Example 3 - Traffic Shaping

In this example we are going to build on the *Example 1 - Basic Principles*, but this time we are going to demonstrate ways how we can shape the network traffic inside out virtual networks. We will shape the traffic with firewall and Quality of Service (QoS for short) rules. These rules will be specified for individual VMs. It will be somewhat of a contrived example, but it will demonstrate the concepts well. There will be just one virtual network with four VMs (**A**, **B**, **C** and **D**). Schematically the scenario will look like this:



The VMs will be able to send network packets only along the edges in the figure above.

The virtual network is also shaping the outgoing network bandwidth of each VM (allocated bandwidth is depicted inside each node).

A transcription of this network setup with LSDN:

```
namespace import lsdn::*

settings vlan

phys -if eth0 -name a

net 1 {
    attach a
    virt -phys a -if 1 -name A {
        rule out 1 drop -dstIp 192.168.0.3
        rule out 2 drop -dstIp 192.168.0.4
        rule in 3 drop -srcIp 192.168.0.2
        rule in 4 drop -srcIp 192.168.0.3

        rate out -avg 40kb -burstRate 40kb -burst 40kb
    }
    virt -phys a -if 2 -name B {
        rule out 1 drop -dstIp 192.168.0.1
        rule in 2 drop -srcIp 192.168.0.3
        rule in 3 drop -srcIp 192.168.0.4

        rate out -avg 20kb -burstRate 20kb -burst 20kb
    }
    virt -phys a -if 3 -name C {
        rule out 1 drop -dstIp 192.168.0.1
        rule out 2 drop -dstIp 192.168.0.2
        rule in 3 drop -srcIp 192.168.0.1
        rule in 4 drop -srcIp 192.168.0.4

        rate out -avg 10kb -burstRate 10kb -burst 10kb
    }
    virt -phys a -if 4 -name D {
        rule out 1 drop -dstIp 192.168.0.2
        rule out 2 drop -dstIp 192.168.0.3
        rule in 1 drop -srcIp 192.168.0.1

        rate out -avg 20kb -burstRate 20kb -burst 20kb
    }
}

claimLocal [lindex $argv 0]
commit
free
```

Let's have a look at all the firewall and QoS rules of one of the virtual machines:

```
virt -phys a -if 2 -name B {
    rule out 1 drop -dstIp 192.168.0.1
    rule in 2 drop -srcIp 192.168.0.3
    rule in 3 drop -srcIp 192.168.0.4

    rate out -avg 20kb -burstRate 20kb -burst 20kb
}
```

The first rule will drop any outgoing traffic with destination IP address 192.168.0.1. The next two rules will drop any traffic incoming from IP addresses 192.168.0.3 or 192.168.0.4. If you take a look at the diagram of our virtual network these are exactly the firewall rules that will ensure that **VM B** will be able to send packets to **VM C** and **VM D**, but not to **VM A** and will be able receive packets only from **VM A**. The last rule installs a QoS rule. It sets the bandwidth for **VM B** with an average rate, burst rate and burst all set to *20kb*. All the rate parameters are described in `rate`.

Similarly you can check the rules for **VM A**, **VM C** and **VM D** and see for yourself they match with our indentation from the sketch above.

You should already be comfortable with the rest of the instructions in the configuration file. If not, please start with *Example 1 - Basic Principles*.

It's a fun exercise to build distributed software that keeps broadcasting a single (UDP) packet with content "A" from within **VM A** to all other VMs in the virtual network at the maximum rate possible. Each other VM upon reception of a packet will append it's own name to the contents of the packet and broadcast this amended packet to all other VMs in the virtual network. **VM A** upon reception of a packet will dump this packet in a log file and drop this packet. What patterns do you expect to see in this log file after some time?

1.7 C API

LSDN is primarily designed as a shared library that can be linked to any program, e.g., a cloud orchestrator service. This guide should give you the necessary information for using LSDN as a C library.

1.7.1 Overview

Networks, physes and virts are each represented by a corresponding struct. In addition, `settings` are its own type of object represented by a struct. This is to allow reusing settings between virtual networks. All these structs are opaque to users of the library and you have to use function calls to modify them, set attributes, etc.

Every LSDN object is also directly or indirectly associated with a *context*. The context basically represents the network model as a whole. It allows you to commit it to kernel tables, configure common behaviors such as handling out-of-memory conditions, and simplify memory management.

Your application will typically have exactly one context. Through it you can create networks, physes and virts, like you would with `lsctl`.

You can modify the network model in memory as much as you like. To apply the changes and install the network model, you need to commit it to kernel tables. After a commit, you can continue modifying the model; subsequent commits will apply the changes.

1.7.2 Object life-cycle

Objects are created by calling `lsdn_<type>_new`. This requires an argument through which the new object is linked to the model: context (for settings and physes), settings (for networks), or network (for virts).

These functions return a pointer to a newly allocated struct of the appropriate type. You can use this pointer to set attributes (see below), construct child objects, perform actions like attaching a virt to a phys, etc.

You can also destroy the object by calling `lsdn_<type>_free`. This ensures that the object is deinitialized properly and the network model remains in a consistent state. All child objects are also destroyed. Because of this behavior, you don't need to keep track of all created objects in your program. Specifically, due to everything being ultimately associated with a context, `lsdn_context_free` will safely deallocate all memory.

Note that in the current version, it is impossible to walk the context to find all child objects. If you want to modify an object later, you need keep track of its reference – or find it by name.

1.7.3 Attributes

Many objects have configurable attributes, such as IP addresses, MAC addresses and similar. For each attribute, you get a collection of functions:

```
lsdn_<type>_get_<attr>
lsdn_<type>_set_<attr>
lsdn_<type>_clear_<attr>
```

In addition, there is a special attribute, `name`. You can specify a name for any object, and then look it up by calling `lsdn_<type>_by_name`. Names must be unique for a given type of object.

There is no `clear` method for names. However, if you need to do that for whatever reason, you can set the name to `NULL`.

1.7.4 Network model life-cycle

The network model representation in memory consists of a context and various child objects associated with it. As a whole, it represents virtual network topology with

attached virts and their connections through physes. This is all nice, but in order for the model to *do* something, it must be converted to TC rules and installed into kernel tables.

Once a model is constructed, you must mark a phys as local, by calling *lsdn_phys_claim_local()*. This sets up the viewpoint for rule generation. Afterwards, calling *lsdn_commit()* will walk the model, generate rules and install them into the kernel. There is no real-time connection between memory representation of the network model and the kernel rules. All changes to the model are only reflected in the kernel after a call to *lsdn_commit()*.

After you're done with your program, you have two choices for deleting the model from memory. A call to *lsdn_context_free()* will deallocate the model, but keep rules in kernel as they are. If you want to remove the rules, call *lsdn_context_cleanup()*, which will both delete the model and uninstall the kernel rules – as if you deleted all objects manually and then performed a commit with an empty model.

Commits are not atomic, and `lsdn_commit` can fail in two distinct ways. In the better case, an error prevented installing a particular object, and the kernel rules are a mix of the old and the new network model. This is represented by `LSDNE_COMMIT` error code. You can retry the commit and LSDN will attempt to apply the remaining changes.

In the current version, there is no way to examine the network model and directly find out which changes were applied and which were not. However, the problem callback supplied to `lsdn_commit` will be notified of failed objects.

In the worse case, a rule removal will fail and the kernel rules will remain in an inconsistent state, not corresponding to a valid network model. This is indicated by `LSDNE_INCONSISTENT` error code. It is impossible to recover from this condition, you need to call *lsdn_context_cleanup()* and start over.

1.7.5 Reference

Learn more about individual kinds of objects and their functions.

Context

group **context**

Context, commits and high level network model management.

LSDN context is a core object that manages the network model. It allows the app to keep track of constraints (such as unique names, no two virts using the same interface, etc.), validate the model and commit it to kernel tables.

Context also keeps track of all the child objects (settings, networks, virts, physes, names, etc.) and automatically frees them when it is deleted through *lsdn_context_free* or *lsdn_context_cleanup*.

In practically every conceivable case, a single app should only have one context; in fact, only one context should exist per physical host. The library allows you to

have multiple contexts at the same time (which is equivalent to having multiple instances of an app), but in such case, the user is responsible for conflicting rules on interfaces. In other words: don't do this, things will probably crash and burn if you do.

Typedefs

typedef void**(*lsdn_nomem_cb)**(void *_user_)
Signature for out-of-memory callback.

Functions

struct _lsdn_context_* **lsdn_context_new**(const char * _name_)
Create new LSDN context.

Initialize a _lsdn_context_ struct and set its name to `name`. The returned struct must be freed by _lsdn_context_free_ or _lsdn_context_cleanup_ after use.

Return `NULL` if allocation failed, pointer to new _lsdn_context_ otherwise.

Parameters

- `name`: Context name.

void **lsdn_context_set_nomem_callback**(struct _lsdn_context_ * _ctx_, _lsdn_nomem_cb_ _cb_, void * _user_)
Configure out-of-memory callback.

By default, LSDN will return an error code to indicate that an allocation failed. This function allows you to set a callback that gets called to handle this condition instead.

Parameters

- `ctx`: LSDN context.
- `cb`: Callback function.
- `user`: User data for the callback function.

void **lsdn_context_abort_on_nomem**(struct _lsdn_context_ * _ctx_)
Configure the context to abort on out-of-memory.

This sets the out-of-memory callback to a predefined function that prints an error to `stderr` and aborts the program.

It is recommended to use this, unless you have a specific way to handle out-of-memory conditions.

See lsdn_abort_cb

Parameters

- `ctx`: LSDN context.

void **lsdn_context_free**(struct *lsdn_context* * *ctx*)
Free a LSDN context.

Deletes the context and all its child objects from memory. Does **not** delete TC rules from kernel tables.

Use this before exiting your program.

Parameters

- `ctx`: Context to free.

void **lsdn_context_cleanup**(struct *lsdn_context* * *ctx*, *lsdn_problem_cb* *cb*, void * *user*)
Clear a LSDN context.

Deletes the context and all its child objects from memory. Also deletes configured TC rules from kernel tables.

Use this to deinitialize the LSDN context and tear down the virtual network.

Parameters

- `ctx`: Context to cleanup.
- `cb`: Problem callback for encountered errors.
- `user`: User data for the problem callback.

void **lsdn_context_set_overwrite**(struct *lsdn_context* * *ctx*, bool *overwrite*)
Configure rule overwriting.

By default, LSDN will overwrite existing tc rules and network interfaces. This is to ensure that rules created by previous crashed instances do not cause problems. Set this flag to `false` to prevent overwriting existing rules.

Parameters

- `ctx`: LSDN context.
- `overwrite`: `true` if LSDN should overwrite existing kernel objects. `false` if it should fail if the kernel object already exists.

bool **lsdn_context_get_overwrite**(struct *lsdn_context* * *ctx*)
Query if LSDN should overwrite any of the interfaces or rules.

Return value of overwrite flag.

See *lsdn_context_set_overwrite*

lsdn_err_t **lsdn_validate**(struct *lsdn_context* * *ctx*, *lsdn_problem_cb* *cb*, void * *user*)
Validate network model.

Walks the currently configured in-memory network model and checks for problems. If problems are found, an error code is returned. Problem callback is also invoked for every problem encountered.

Parameters

- `ctx`: LSDN context.
- `cb`: Problem callback.
- `user`: User data for the problem callback.

Return Value

- *`LSDNE_OK`*: No problems detected.
- *`LSDNE_VALIDATE`*: Some problems detected.

lsdn_err_t **`lsdn_commit`**(struct *lsdn_context* * *ctx*, *lsdn_problem_cb* *cb*, void * *user*)

Commit network model to kernel tables.

Calculates tc rules based on the current network model, and its difference from the previously committed network model, and applies the changes. After returning successfully, the current network model is in effect.

Performs a model validation (equivalent to calling *lsdn_validate*) and returns an error if it fails. Afterwards, works through the memory model in two phases:

- In *decommit* phase, rules belonging to modified (or deleted) objects are removed from kernel tables. Deleted objects are also freed from memory.
- In *recommit* phase, new rules are installed that correspond to new objects or new properties of objects that were removed in the previous phase.

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 case, *LSDNE_COMMIT* is returned and 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, *LSDNE_INCONSISTENT* is returned and 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.

Parameters

- `ctx`: LSDN context.
- `cb`: Problem callback.
- `user`: User data for the problem callback.

Return Value

- `LSDNE_OK`: Commit was successful. New network model is now active in kernel.
- `LSDNE_VALIDATE`: Model validation found problems. Old network model remains active in kernel.
- `LSDNE_COMMIT`: Errors were encountered during commit. Kernel is in mixed state, it is possible to retry.
- `LSDNE_INCONSISTENT`: Errors were encountered when decommitting rules. Model state is inconsistent with kernel state. You have to start over.

struct lsdn_context
#include <*lsdn.h*> LSDN Context.

The base object of the LSDN network model. There should be exactly one instance in your program. See *Context*.

Phys (host machine)

group **phys**
Functions for manipulating and configuring phys objects.

Phys is a representation of a physical machine that hosts tenants of virtual networks. Its `interface` attribute specifies the name of the network interface that is connected to the host network. In addition, some network types require IP addresses of physes.

In order to start connecting virts, a phys must be *attached* to a virtual network. That only marks the phys as a participant in that network; a single phys can be attached to any number of networks.

In the network model, all physes must be represented on all machines. To select the current machine and configure network viewpoint, you must call *lsdn_phys_claim_local*. Kernel rules are then generated from the viewpoint of that phys.

It is possible to have multiple physes on the same machine and claimed local. This is useful in situations where the host machine has more than one interface connecting to a host network, or if the machine connects to more than one host network.

Defines

lsdn_mk_phys_name(ctx)
Generate unique name for a phys.

See *lsdn_mk_name*

Parameters

- `ctx`: LSDN context.

Functions

struct *lsdn_phys** **`lsdn_phys_new`**(struct *lsdn_context* * *ctx*)
Create a new phys.

Allocates and initializes a *lsdn_phys* structure.

Return newly allocated *lsdn_phys* structure.

Parameters

- `ctx`: LSDN context.

lsdn_err_t **`lsdn_phys_set_name`**(struct *lsdn_phys* * *phys*, const char * *name*)
Set a name for phys.

Parameters

- `phys`: Phys.
- `name`: New name string. Can be `NULL`.

Return Value

- `LSDNE_OK`: Name set successfully.
- `LSDNE_DUPLICATE`: Phys with the same name already exists.
- `LSDNE_NOMEM`: Failed to allocate memory for name.

const char* **`lsdn_phys_get_name`**(struct *lsdn_phys* * *phys*)
Get the phys's name.

Return pointer to phys's name.

Parameters

- `phys`: Phys.

struct *lsdn_phys** **`lsdn_phys_by_name`**(struct *lsdn_context* * *ctx*, const char * *name*)
Find a phys by name.

Return *lsdn_phys* structure if a phys with this name exists. `NULL` otherwise.

Parameters

- `ctx`: LSDN context.
- `name`: Requested name.

void **lsdn_phys_free**(struct *lsdn_phys* * *phys*)
Free a phys.

Ensures that all virts on this phys are disconnected first.

Parameters

- `phys`: Phys.

lsdn_err_t **lsdn_phys_attach**(struct *lsdn_phys* * *phys*, struct *lsdn_net* * *net*)
Attach phys to a virtual network.

Marks the phys as a participant in virtual network `net`. This must be done before any virts are allowed to connect to `net` through this phys.

You can attach a phys to multiple virtual networks.

Parameters

- `phys`: Phys.
- `net`: Virtual network.

Return Value

- `LSDNE_OK`: Attachment succeeded
- `LSDNE_NOMEM`: Failed to allocate memory for attachment.

void **lsdn_phys_detach**(struct *lsdn_phys* * *phys*, struct *lsdn_net* * *net*)
Detach phys from a virtual network.

After detaching, virts won't be allowed to connect to a given network through this phys.

Warning This will not disconnect currently connected virts. They must be disconnected explicitly. Otherwise, the next commit will fail validation.

Parameters

- `phys`: Phys.
- `net`: Virtual network.

lsdn_err_t **lsdn_phys_claim_local**(struct *lsdn_phys* * *phys*)
Assign a local phys.

All participants in a LSDN network must share a compatible memory model. That means that every host's model contains all the physes in the network. This function configures a particular phys to be the local machine. Only rules related to virts on the local phys are entered into the kernel tables.

lsdn_err_t **lsdn_phys_unclaim_local**(struct *lsdn_phys* * *phys*)
Unassign a local phys.

See *lsdn_phys_claim_local*

lsdn_err_t **lsdn_phys_set_ip**(struct *lsdn_phys* * *object*, *lsdn_ip_t* *value*)
Set IP address of a phys .

Parameters

- `object`: phys to modify.
- `value`: IP address .

const *lsdn_ip_t** **lsdn_phys_get_ip**(struct *lsdn_phys* * *object*)
Get IP address of a phys .

The pointer is valid until the attribute is changed or object freed.

Return value of IP address attribute, or `NULL` if unset.

Parameters

- `object`: phys to query.

void **lsdn_phys_clear_ip**(struct *lsdn_phys* * *object*)
Clear IP address of a phys .

Parameters

- `object`: phys to modify.

lsdn_err_t **lsdn_phys_set_iface**(struct *lsdn_phys* * *object*, const char * *value*)
Set interface of a phys .

Parameters

- `object`: phys to modify.
- `value`: interface .

const char* **lsdn_phys_get_iface**(struct *lsdn_phys* * *object*)
Get interface of a phys .

The pointer is valid until the attribute is changed or object freed.

Return value of interface attribute, or `NULL` if unset.

Parameters

- `object`: phys to query.

void **lsdn_phys_clear_iface**(struct *lsdn_phys* * *object*)
Clear interface of a phys .

Parameters

- `object`: phys to modify.

struct lsdn_phys

#include <lsdn.h> Phys.

Represents a kernel interface for a host node, e.g., eth0 on lsdn1. Physes are attached to network, and then virts can connect through them. See *Phys (host machine)*.

See *Virt (virtual machine)*.

See *Virtual network*.

Virtual network

group **network**

Functions, and related data types, for manipulating network objects and their settings.

Virtual network is a collection of virts that can communicate with each other as if they were on the same LAN. At the same time, they are isolated from other virtual networks, as well as from the host network. Distinct virtual networks can have hosts with same MAC addresses, and it is impossible to read packets belonging to other networks (or the host network), or send packets that travel outside the virtual network.

The *lsdn_net* object represents a network in the sense of "collection of virts". Apart from basic life-cycle and lookup functions, it is only possible to add or remove virts to/from it.

Configuration of network properties is done through separate *lsdn_settings* objects. There is a `lsdn_<kind>_settings_new` function for each kind of network encapsulation, with different required parameters. It is also possible to register *user hooks* for startup and shutdown events.

An exception to this is the `vnet_id` property, which is set on a network directly, as opposed to being a part of settings. It configures the VNET (or encapsulation ID) of the network. That means that several networks can share a common settings object while still being differentiated by `vnet_id`.

Network object management

struct *lsdn_net** **lsdn_net_new**(struct *lsdn_settings* * *settings*, uint32_t *vnet_id*)

Create a new network.

Creates a virtual network object with id `vnet_id`, configured by `s`.

Multiple networks can share the same *lsdn_settings*, as long as they differ by `vnet_id`.

Return newly allocated *lsdn_net* structure.

lsdn_err_t **lsdn_net_set_name**(struct *lsdn_net* * *net*, const char * *name*)
Set a name for the network.

const char* **lsdn_net_get_name**(struct *lsdn_net* * *net*)
Get the network's name.

struct *lsdn_net** **lsdn_net_by_name**(struct *lsdn_context* * *ctx*, const char * *name*)
Find a network by name.

Return *lsdn_net* structure if a network with this name exists.

Return `NULL` otherwise.

void **lsdn_net_free**(struct *lsdn_net* * *net*)
Free a network.

Ensures that all virts in the network are freed and all physes detached.

Network settings

struct *lsdn_settings** **lsdn_settings_new_direct**(struct *lsdn_context* * *ctx*)
Create settings for a new direct network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.

struct *lsdn_settings** **lsdn_settings_new_vlan**(struct *lsdn_context* * *ctx*)
Create settings for a new VLAN network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.

struct *lsdn_settings** **lsdn_settings_new_vxlan_mcast**(struct *lsdn_context* * *ctx*, *lsdn_ip_t* *mcast_ip*, uint16_t *port*)
Create settings for a new VXLAN-multicast network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.
- `mcast_ip`: Multicast group IP address.
- `port`: UDP port for VXLAN tunnel.

struct *lsdn_settings** **lsdn_settings_new_vxlan_e2e**(struct *lsdn_context* * *ctx*, uint16_t *port*)

Create settings for a new VXLAN-e2e network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.
- `port`: UDP port for VXLAN tunnel.

struct *lsdn_settings** **lsdn_settings_new_vxlan_static**(struct *lsdn_context* * *ctx*, uint16_t *port*)

Create settings for a new VXLAN-static network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.
- `port`: UDP port for VXLAN tunnel.

struct *lsdn_settings** **lsdn_settings_new_geneve**(struct *lsdn_context* * *ctx*, uint16_t *port*)

Create settings for a new GENEVE network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.
- `port`: UDP port for GENEVE tunnel.

struct *lsdn_settings** **lsdn_settings_new_geneve_e2e**(struct *lsdn_context* * *ctx*, uint16_t *port*)

Create settings for a new GENEVE network.

Return new *lsdn_settings* instance.

Parameters

- `ctx`: LSDN context.
- `port`: UDP port for GENEVE tunnel.

void **lsdn_settings_free**(struct *lsdn_settings* * *settings*)

Free settings object.

Deletes the settings object and all *lsdn_net* objects that use it.

void **lsdn_settings_register_user_hooks**(struct *lsdn_settings* * *settings*, struct *lsdn_user_hooks* * *user_hooks*)

Configure user hooks.

Associates a *lsdn_user_hooks* structure with `settings`.

lsdn_err_t **lsdn_settings_set_name**(struct *lsdn_settings* * *s*, const char * *name*)

Assign a name to settings.

Return Value

- *LSDNE_OK*: if the name is successfully set.
- *LSDNE_DUPLICATE*: if this name is already in use.

const char* **lsdn_settings_get_name**(struct *lsdn_settings* * *s*)

Get settings name.

Return name of the settings struct.

struct *lsdn_settings** **lsdn_settings_by_name**(struct *lsdn_context* * *ctx*, const char * *name*)

Find settings by name.

Searches the context for a named *lsdn_settings* object and returns it.

Return Pointer to *lsdn_settings* with this name.

Return `NULL` if no settings with that name exist in the context.

Parameters

- `ctx`: LSDN context.
- `name`: Requested name

Defines

lsdn_mk_net_name(ctx)

Generate unique name for a net.

See *lsdn_mk_name*

Parameters

- `ctx`: LSDN context.

lsdn_mk_settings_name(ctx)

Generate unique name for a settings object.

See *lsdn_mk_name*

Parameters

- `ctx`: LSDN context.

struct lsdn_user_hooks

#include <lsdn.h> User callback hooks.

Configured as part of *lsdn_settings*, this structure holds the callback hooks for startup and shutdown, and their custom data.

Public Members

void **(*lsdn_startup_hook)** (struct *lsdn_net* *net, struct *lsdn_phys* *phys, void *user)

Startup hook.

Called at commit time for every local phys and every network to which it is attached.

Parameters

- `net`: network.
- `phys`: attached phys.
- `user`: receives the value of *lsdn_startup_hook_user*.

void* **lsdn_startup_hook_user**

Custom value for *lsdn_startup_hook*.

void **(*lsdn_shutdown_hook)** (struct *lsdn_net* *net, struct *lsdn_phys* *phys, void *user)

Shutdown hook.

void* **lsdn_shutdown_hook_user**

Custom value for *lsdn_shutdown_hook*.

struct lsdn_net

#include <lsdn.h> Virtual network.

Network is a collection of virts that can communicate with each other. See *Virtual network*.

See *Virt (virtual machine)*.

See *Phys (host machine)*.

struct lsdn_settings

#include <lsdn.h> Configuration structure for a virtual network.

Multiple networks can share the same settings (e.g. VXLAN with static routing on port 1234) and only differ by their identifier (VLAN id, VNI…). See *Virtual network*.

Virt (virtual machine)

group **virt**

Functions for manipulating and configuring virt objects.

Virt is a representation of a tenant in the virtual network. By default, it does not need any attributes. However, you can configure its MAC address on the virtual network (required for some network types), and set inbound and outbound QoS rates.

Virt is created as part of a network, but to participate in the network, it must first be connected, through a phys and a network interface on that phys. It is possible to disconnect a virt and reconnect it on a different phys, e.g., when the VM is migrated to a different physical host. The migration is transparent to the virtual network, but obviously, the virt is unreachable while disconnected.

Defines

lsdn_mk_virt_name(ctx)

Generate unique name for a virt.

See *lsdn_mk_name*

Parameters

- `ctx`: LSDN context.

Functions

struct *lsdn_virt** **lsdn_virt_new**(struct *lsdn_net* * *net*)

Create a new virt.

Creates a virt as part of `net`.

Return newly allocated *lsdn_virt* structure.

void **lsdn_virt_free**(struct *lsdn_virt* * *vsirt*)

Free a virt.

struct *lsdn_net** **lsdn_virt_get_net**(struct *lsdn_virt* * *virt*)

Get the virt's network.

Return *lsdn_net* object of the network that this virt is part of.

Parameters

- `virt`: Virt object.

lsdn_err_t **lsdn_virt_set_name**(struct *lsdn_virt* * *virt*, const char * *name*)

Set a name for the virt.

const char* **lsdn_virt_get_name**(struct *lsdn_virt* * *virt*)
Get the virt's name.

struct *lsdn_virt** **lsdn_virt_by_name**(struct *lsdn_net* * *net*, const char * *name*)
Find a virt by name.

Return *lsdn_virt* structure if a network with this name exists.

Return `NULL` otherwise.

lsdn_err_t **lsdn_virt_connect**(struct *lsdn_virt* * *virt*, struct *lsdn_phys* * *phys*, const char * *iface*)
Connect a virt to its network.

Associates a virt with a given phys and a network interface, and ensures that this interface will receive the network's traffic.

See lsdn-public

Parameters

- `virt`: virt to connect.
- `phys`: phys on which the virt exists.
- `iface`: name of Linux network interface on the given phys, which will receive the virt's traffic.

void **lsdn_virt_disconnect**(struct *lsdn_virt* * *virt*)
Disconnects a virt from its network.

Disconnected virt will no longer be able to send and receive traffic.

lsdn_err_t **lsdn_virt_get_recommended_mtu**(struct *lsdn_virt* * *virt*, unsigned int * *mtu*)
Get recommended MTU for a given virt.

Calculates the appropriate MTU value, taking into account the network's tunneling method overhead.

The MTU is based on the current state and connection port of the virt (it is not based on the committed state). The phys interface must already exist.

Parameters

- `virt`: Virt object.
- `mtu`: Pointer into which the MTU is stored.

Return Value

- `LSDNE_OK`: Operation was successful.
- `LSDNE_NETLINK`: Netlink communication error.
- `LSDNE_NOIF`: Virt's connected interface does not exist.

lsdn_err_t **lsdn_virt_set_mac**(struct *lsdn_virt* * *object*, *lsdn_mac_t* *value*)
Set MAC address of a virt .

Parameters

- `object`: virt to modify.
- `value`: MAC address .

const *lsdn_mac_t** **lsdn_virt_get_mac**(struct *lsdn_virt* * *object*)
Get MAC address of a virt .

The pointer is valid until the attribute is changed or object freed.

Return value of MAC address attribute, or `NULL` if unset.

Parameters

- `object`: virt to query.

void **lsdn_virt_clear_mac**(struct *lsdn_virt* * *object*)
Clear MAC address of a virt .

Parameters

- `object`: virt to modify.

lsdn_err_t **lsdn_virt_set_rate_in**(struct *lsdn_virt* * *object*, *lsdn_qos_rate_t* *value*)
Set inbound bandwidth limit of a virt .

Parameters

- `object`: virt to modify.
- `value`: inbound bandwidth limit .

const *lsdn_qos_rate_t** **lsdn_virt_get_rate_in**(struct *lsdn_virt* * *object*)
Get inbound bandwidth limit of a virt .

The pointer is valid until the attribute is changed or object freed.

Return value of inbound bandwidth limit attribute, or `NULL` if unset.

Parameters

- `object`: virt to query.

void **lsdn_virt_clear_rate_in**(struct *lsdn_virt* * *object*)
Clear inbound bandwidth limit of a virt .

Parameters

- `object`: virt to modify.

lsdn_err_t **lsdn_virt_set_rate_out**(struct *lsdn_virt* * *object*, *lsdn_qos_rate_t* *value*)
Set outbound bandwidth limit of a virt .

Parameters

- `object`: virt to modify.
- `value`: outbound bandwidth limit .

const *lsdn_qos_rate_t** **lsdn_virt_get_rate_out**(struct *lsdn_virt* * *object*)
Get outbound bandwidth limit of a virt .

The pointer is valid until the attribute is changed or object freed.

Return value of outbound bandwidth limit attribute, or `NULL` if unset.

Parameters

- `object`: virt to query.

void **lsdn_virt_clear_rate_out**(struct *lsdn_virt* * *object*)
Clear outbound bandwidth limit of a virt .

Parameters

- `object`: virt to modify.

struct lsdn_qos_rate_t
#include <lsdn.h> Bandwidth limit for virt's interface (for one direction).

See *lsdn_virt_set_rate_out*

See *lsdn_virt_set_rate_in*

Public Members

float **avg_rate**
Bandwidth restriction in bytes per second.

uint32_t **burst_size**
A size of data burst that is allowed to exceed the *avg_rate*.

It is not possible to leave this field zero, because no packets would go through. Since each packet is considered as a short burst, the burst rate must be at least as big as your MTU.

float **burst_rate**
An absolute restriction on the bandwidth in bytes per second, even during bursting.

If zero is given, the peak rate is unrestricted.

struct lsdn_virt
#include <lsdn.h> Virt.

A virtual machine (typically it may be any Linux interface).

Virts are tenants in networks. They must be connected through a phys. They can be migrated between physes at runtime. See *Virt (virtual machine)*.

See *Phys (host machine)*.

See *Virtual network*.

Rules engine

group **rules**
Virt rules engine configuration.

LSDN supports a basic firewall filtering. It is possible to set up packet rules matching several criteria (source or destination addresses or ranges, tunnel key ID) and assign them to inbound or outbound queues of a particular virt. Currently, the firewall can only drop matching packets. There is no support for creating custom firewall actions.

lsdn_vr is its own kind of object, tied to a virt. It can either be created preconfigured to match something, or set as empty and configured later.

Rules are evaluated in order of increasing priority. The lower the priority value, the higher the actual priority.

Defines

LSDN_MAX_MATCHES
Maximum number of match targets per rule.

In this implementation, a rule can match on at most two simultaneous objects (e.g. MAC address and IPv4 address).

LSDN_VR_PRIO_MIN
Minimum Virt Rule priority.

LSDN_VR_PRIO_MAX
Upper limit for Virt Rule priority.

Actual priority must be strictly lower than this.

LSDN_PRIO_FORWARD_DST_MAC
Use this priority if you want your rule to take place during forwarding decisions.

Enums

enum lsdn_direction
Virt rule direction.

Values:

LSDN_IN
Inbound rule.

LSDN_OUT
Outbound rule.

enum lsdn_rule_target
Rule target.

Values:

LSDN_MATCH_NONE
Do not match.

LSDN_MATCH_SRC_MAC
Match source MAC.

LSDN_MATCH_DST_MAC
Match destination MAC.

LSDN_MATCH_SRC_IPV4
Match source IPv4 address.

LSDN_MATCH_DST_IPV4
Match destination IPv4 address.

LSDN_MATCH_SRC_IPV6
Match source IPv6 address.

LSDN_MATCH_DST_IPV6
Match destination IPv6 address.

LSDN_MATCH_ENC_KEY_ID
Match tunnel key ID.

LSDN_MATCH_ENC_KEY_SRC_IPV4
Match tunnel source IP address .

LSDN_MATCH_ENC_KEY_SRC_IPV6
Match tunnel source IP address .

LSDN_MATCH_ENC_KEY_DST_IPV4
Match tunnel source IP address .

LSDN_MATCH_ENC_KEY_DST_IPV6
Match tunnel source IP address .

LSDN_MATCH_COUNT
Guard value.

See LSDN_ENUM for details.

Functions

struct *lsdn_vr** **lsdn_vr_new**(struct *lsdn_virt* * *virt*, uint16_t *prio*, enum lsdn_direction *dir*, struct *lsdn_vr_action* * *a*)

Create a virt rule.

Creates a rule with a given priority, to match packets to or from a given virt, and assigns an action when the rule is matched.

Rule created with this function does not match anything. It must be configured through one or more of the `lsdn_vr_add_<match>` functions.

Return New *lsdn_vr* struct.

Parameters

- `virt`: Virt to which the rule applies.
- `prio_num`: Rule priority. Lower number = higher priority.
- `dir`: Inbound or outbound rule.
- `a`: Assigned action when rule matches.

void **lsdn_vr_free**(struct *lsdn_vr* * *vr*)

Deallocate a rule.

Parameters

- `vr`: Rule to deallocate.

void **lsdn_vrs_free_all**(struct *lsdn_virt* * *virt*)

Deallocate all rules for a virt.

Parameters

- `virt`: Virt whose rules will be removed.

void **lsdn_vr_add_masked_src_mac**(struct *lsdn_vr* * *rule*, *lsdn_mac_t* *mask*, *lsdn_mac_t* *value*)

Configure virt rule to match source MAC with a mask.

Parameters

- `rule`: Virt rule.
- `mask`: Mask value.
- `value`: Match value.

static void **lsdn_vr_add_src_mac**(struct *lsdn_vr* * *rule*, *lsdn_mac_t* *value*)

Configure virt rule to match a specified source MAC .

Parameters

- `rule`: Pointer to virt rule.
- `value`: Match value.

static struct *lsdn_vr** **`lsdn_vr_new_masked_src_mac`**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_mac_t* *value*, *lsdn_mac_t* *mask*, struct *lsdn_vr_action* * *action*)

Create virt rule matching source MAC with a mask.

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `mask`: Mask value.
- `action`: Rule action.

static struct *lsdn_vr** **`lsdn_vr_new_src_mac`**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_mac_t* *value*, struct *lsdn_vr_action* * *action*)

Create virt rule matching a specified source MAC .

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `action`: Rule action.

void **`lsdn_vr_add_masked_dst_mac`**(struct *lsdn_vr* * *rule*, *lsdn_mac_t* *mask*, *lsdn_mac_t* *value*)

Configure virt rule to match destination MAC with a mask.

Parameters

- `rule`: Virt rule.

- `mask`: Mask value.
- `value`: Match value.

static void **lsdn_vr_add_dst_mac**(struct *lsdn_vr* * *rule*, *lsdn_mac_t* *value*)
Configure virt rule to match a specified destination MAC .

Parameters

- `rule`: Pointer to virt rule.
- `value`: Match value.

static struct *lsdn_vr** **lsdn_vr_new_masked_dst_mac**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_mac_t* *value*, *lsdn_mac_t* *mask*, struct *lsdn_vr_action* * *action*)
Create virt rule matching destination MAC with a mask.

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `mask`: Mask value.
- `action`: Rule action.

static struct *lsdn_vr** **lsdn_vr_new_dst_mac**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_mac_t* *value*, struct *lsdn_vr_action* * *action*)
Create virt rule matching a specified destination MAC .

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `action`: Rule action.

void **lsdn_vr_add_masked_src_ip**(struct *lsdn_vr* * *rule*, *lsdn_ip_t* *mask*, *lsdn_ip_t* *value*)

Configure virt rule to match source IP address with a mask.

Parameters

- `rule`: Virt rule.
- `mask`: Mask value.
- `value`: Match value.

static void **lsdn_vr_add_src_ip**(struct *lsdn_vr* * *rule*, *lsdn_ip_t* *value*)

Configure virt rule to match a specified source IP .

Parameters

- `rule`: Pointer to virt rule.
- `value`: Match value.

static struct *lsdn_vr** **lsdn_vr_new_masked_src_ip**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_ip_t* *value*, *lsdn_ip_t* *mask*, struct *lsdn_vr_action* * *action*)

Create virt rule matching source IP with a mask.

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `mask`: Mask value.
- `action`: Rule action.

static struct *lsdn_vr** **lsdn_vr_new_src_ip**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_ip_t* *value*, struct *lsdn_vr_action* * *action*)

Create virt rule matching a specified source IP .

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.

- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `action`: Rule action.

void **lsdn_vr_add_masked_dst_ip**(struct *lsdn_vr* * *rule*, *lsdn_ip_t* *mask*, *lsdn_ip_t* *value*)

Configure virt rule to match destination IP address with a mask.

Parameters

- `rule`: Virt rule.
- `mask`: Mask value.
- `value`: Match value.

static void **lsdn_vr_add_dst_ip**(struct *lsdn_vr* * *rule*, *lsdn_ip_t* *value*)

Configure virt rule to match a specified destination IP .

Parameters

- `rule`: Pointer to virt rule.
- `value`: Match value.

static struct *lsdn_vr** **lsdn_vr_new_masked_dst_ip**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_ip_t* *value*, *lsdn_ip_t* *mask*, struct *lsdn_vr_action* * *action*)

Create virt rule matching destination IP with a mask.

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `mask`: Mask value.
- `action`: Rule action.

static struct *lsdn_vr** **lsdn_vr_new_dst_ip**(struct *lsdn_virt* * *virt*, enum lsdn_direction *dir*, uint16_t *prio*, *lsdn_ip_t* *value*, struct *lsdn_vr_action* * *action*)

Create virt rule matching a specified destination IP .

Return New *lsdn_vr* struct.

Parameters

- `virt`: LSDN virt.
- `dir`: Incoming or outgoing rule.
- `prio`: Rule priority.
- `value`: Match value.
- `action`: Rule action.

Variables

struct *lsdn_vr_action* **LSDN_VR_DROP**

DROP rule action.

Packet matching a rule with this action will be dropped.

struct lsdn_vr

#include <rules.h> Virt rule.

Represents a packet rule assigned to a virt. The rule has a priority, assigned direction (affecting incoming or outgoing packets), and an action. Usually, the rule will also have match conditions, such as IP or MAC address mask.

See *lsdn_vr_new* See *Rules engine*.

struct lsdn_vr_action

#include <rules.h> Virt rule action.

Represents an action to be performed on a packet that matches a rule.

In this version, the only possible action is *LSDN_VR_DROP*.

Error codes and error handling

group **errors**

Definitions and descriptions of error codes, and error related functions.

Functions that can fail usually return an error code from *lsdn_err_t*. This is not very specific, so in addition, some functions allow you to specify a *problem callback* function. This is a user-defined function that gets called separately for every error encountered - such as when validating or committing the model.

Convenience functions are provided to dump error strings either to stderr or to a specified `FILE`.

Defines

LSDN_MAX_PROBLEM_REFS
Maximum number of problem items described simultaneously.

Configures size of the problem buffer in *lsdn_context*.

Typedefs

typedef void **(*lsdn_problem_cb)** (const struct *lsdn_problem* *diag, void *user)
Problem handler callback.

Parameters

- `diag`: description of the problem.
- `user`: user-specified data.

Enums

enum lsdn_err_t
Possible LSDN errors.

Values:

LSDNE_OK = 0
No error.

LSDNE_NOMEM
Out of memory.

LSDNE_PARSE
Parsing from string has failed.

Can occur when parsing IPs, MACs etc.

LSDNE_DUPLICATE
Duplicate name.

Can occur when setting name for a network, virt or phys.

LSDNE_NOIF
Interface does not exist.

LSDNE_NETLINK
Netlink error.

LSDNE_VALIDATE
Network model validation failed, and the old model is in effect.

LSDNE_COMMIT
Network model commit failed and a mix of old, new and dysfunctional objects are in effect.

You can retry the commit and it will work if the error was temporary.

The error could also be permanent, if, for example, a user have created a network interface that shares a name with what LSDN was going to use. In that case, you will be getting the error repeatedly. You can either ignore it or delete the failing part of the model.

LSDNE_INCONSISTENT
Cleanup operation has failed and this left an object in state inconsistent with the model.

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

enum lsdn_problem_code
Validation and commit errors.

Values:

LSDNP_PHYS_NOATTR
Missing attribute on a phys.

LSDNP_PHYS_DUPATTR
Duplicate attribute on two phys's in the same network.

LSDNP_PHYS_INCOMPATIBLE_IPV
Incompatible IP versions in the same network.

LSDNP_PHYS_NOT_ATTACHED
Connecting a virt from a phys that is not attached to a network.

LSDNP_VIRT_NOIF
Interface specified for a virt does not exist.

LSDNP_VIRT_NOATTR
Missing attribute on a virt.

LSDNP_VIRT_DUPATTR
Duplicate attribute on two virts in the same network.

LSDNP_NET_BAD_NETTYPE
Incompatible networks on the same machine.

LSDNP_NET_BADID
Bad network ID.

LSDNP_NET_DUPID
Duplicate network ID.

LSDNP_VR_INCOMPATIBLE_MATCH
Two incompatible virt rules with the same priority.

LSDNP_VR_DUPLICATE_RULE
Duplicate virt rules.

LSDNP_COMMIT_NETLINK
Committing to netlink failed due to kernel error.

LSDNP_COMMIT_NETLINK_CLEANUP
Decommitting to netlink failed due to kernel error and the state is now inconsistent.

LSDNP_COMMIT_NOMEM
Committing to netlink failed due to memory error.

LSDNP_NO_NLSOCK
Can not establish netlink communication.

LSDNP_RATES_INVALID
QoS has invalid parameters (both rate and burst must be positive).

See *lsdn_qos_rate_t* for correct parameters.

LSDNP_COUNT
Guard value.

See LSDN_ENUM for details.

enum lsdn_problem_ref_type
Problem reference type.

Values:

LSDNS_ATTR
Problem with attribute.

LSDNS_PHYS
Problem with *lsdn_phys*.

LSDNS_PA
Problem with *lsdn_net* and *lsdn_phys* combination.

LSDNS_NET
Problem with *lsdn_net*.

LSDNS_VIRT
Problem with *lsdn_virt*.

LSDNS_IF
Problem with a network interface.

LSDNS_NETID
Problem with `vnet_id`.

LSDNS_VR
Problem with *lsdn_vr*.

LSDNS_END
End of problem list.

Functions

void **lsdn_problem_stderr_handler**(const struct *lsdn_problem* * *problem*, void * *user*)

Problem handler that dumps problem descriptions to `stderr`.

Can be used as a callback in *lsdn_commit*, *lsdn_validate*, and any other place that takes *lsdn_problem_cb*.

When a problem is encountered, this handler will dump a human-readable description to `stderr`.

Parameters

- `problem`: problem description.
- `user`: user data for the callback. Unused.

void **lsdn_problem_format**(FILE * *out*, const struct *lsdn_problem* * *problem*)

Print problem description.

Constructs a string describing the problem and writes it out to `out`.

Parameters

- `out`: output stream.
- `problem`: problem description.

struct lsdn_problem_ref

#include <errors.h> Reference to a problem item.

Consists of a problem type, and a pointer to a struct of the appropriate type.

Public Members

lsdn_problem_ref_type **type**

Problem type.

void* **ptr**

Pointer to the appropriate struct.

struct lsdn_problem

#include <errors.h> Description of encountered problem.

Passed to a *lsdn_problem_cb* callback when an error occurs.

`code` refers to the type of problem encountered. Depending on the type of the problem, this might also indicate any number of related problematic items. Pointers to them are stored in `refs`.

Public Members

lsdn_problem_code **code**
: Problem code.

size_t **refs_count**
: Number of related items.

struct *lsdn_problem_ref** **refs**
: Array of references to related items.

  Note `refs` actually point to a buffer in *lsdn_context*.

Miscellaneous functions

group **misc**
: Miscellaneous functions and definitions.

  This section documents the various odds and ends that didn't fit anywhere else.

  - String dump functions for converting the memory model to JSON or TCL representation
  - Network-related definitions, such as frame lengths, header lengths, minimum and maximum IDs
  - Network address types and enums
  - Address initializer macros
  - Address string parsing / dumping functions
  - Address comparison functions
  - Prefix validation functions
  - Constants for broadcast addresses and full masks.

Dump to various formats

char* **lsdn_dump_context_json**(struct *lsdn_context* * *ctx*)
: Dump the internal LSDN network model in JSON format.

  Return A C string containing the context's representation in JSON format. Caller is responsible for deallocating the string using `free`.

char* **lsdn_dump_context_tcl**(struct *lsdn_context* * *ctx*)
: Dump the internal LSDN network model in TCL format.

  Return A C string containing the context's representation in lsctl-compatible form. Caller is responsible for deallocating the string using `free`.

Defines

lsdn_mk_iface_name(ctx)
Generate unique name for an interface.

See *lsdn_mk_name*

Parameters

- `ctx`: LSDN context.

ETHERNET_FRAME_LEN
Ethernet frame length in bytes.

IPv4_HEADER_LEN
IPv4 header length in bytes.

IPv6_HEADER_LEN
IPv6 header length in bytes.

UDP_HEADER_LEN
UDP header length in bytes.

VXLAN_HEADER_LEN
VXLAN header length in bytes.

GENEVE_HEADER_LEN
GENEVE header length in bytes.

NET_GENEVE_MIN_VNET_ID
Minimum allowed vnet id for GENEVE networks.

NET_GENEVE_MAX_VNET_ID
Maximum allowed vnet id for GENEVE networks.

NET_VXLAN_MIN_VNET_ID
Minimum allowed vnet id for VXLAN networks.

NET_VXLAN_MAX_VNET_ID
Maximum allowed vnet id for VXLAN networks.

NET_VLAN_MIN_VNET_ID
Minimum allowed vnet id for VLAN networks.

NET_VLAN_MAX_VNET_ID
Maximum allowed vnet id for VLAN networks.

LSDN_MAC_LEN
MAC address size in bytes.

LSDN_IPv4_LEN
IPv4 address size in bytes.

LSDN_IPv6_LEN
IPv6 address size in bytes.

LSDN_MK_IPV4(a, b, c, d)
Construct a *lsdn_ip* IPv4 address from a 4-tuple.

LSDN_INITIALIZER_IPV4(a, b, c, d)
struct literal for a *lsdn_ip* IPv4 address constructed from a 4-tuple.

LSDN_MK_IPV6(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p)
Construct a *lsdn_ip* IPv6 address from a 16-tuple.

LSDN_INITIALIZER_IPV6(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p)
struct literal for a *lsdn_ip* IPv6 address constructed from a 16-tuple.

LSDN_MK_MAC(a, b, c, d, e, f)
Construct a *lsdn_mac_t* from a 6-tuple.

LSDN_INITIALIZER_MAC(a, b, c, d, e, f)
struct literal for a *lsdn_mac_t* from a 6-tuple.

LSDN_MAC_STRING_LEN
Maximum length of MAC address string.

Five colons, six hexadecimal octets.

LSDN_IPv4_STRING_LEN
Maximum length of IPv4 address string.

Three dots, four decimal octets.

LSDN_IPv6_STRING_LEN
Maximum length of IPv6 address string.

Seven colons, sixteen hexadecimal octets.

LSDN_IP_STRING_LEN
Maximum length of address string for any IP address.

Typedefs

typedef union *lsdn_mac* **lsdn_mac_t**
MAC address.

typedef union *lsdn_ipv4* **lsdn_ipv4_t**
IPv4 address.

typedef union *lsdn_ipv6* **lsdn_ipv6_t**
IPv6 address.

typedef struct *lsdn_ip* **lsdn_ip_t**
IP address (any version).

Enums

enum lsdn_ipv
IP protocol version.

Values:

LSDN_IPv4 = 4
IPv4.

LSDN_IPv6 = 6
IPv6.

Functions

lsdn_err_t **lsdn_parse_mac**(*lsdn_mac_t* * *mac*, const char * *ascii*)
Parse MAC address string into *lsdn_mac_t*.

Parameters

- `ascii`: MAC address string.
- `mac`: parsed MAC address struct.

Return Value

- `LSDNE_OK`: if parsed successfully.
- `LSDNE_PARSE`: if `ascii` could not be parsed into a MAC address.

bool **lsdn_mac_eq**(*lsdn_mac_t a*, *lsdn_mac_t b*)
Compare two *lsdn_mac_t* for equality.

Return `true` if `a` and `b` are equal, `false` otherwise.

Parameters

- `a`: MAC address.
- `b`: MAC address.

lsdn_err_t **lsdn_parse_ip**(*lsdn_ip_t* * *ip*, const char * *ascii*)
Parse IP address string into *lsdn_ip_t*.

Parameters

- `ascii`: IP address string.
- `ip`: parsed IP address struct.

Return Value

- *LSDNE_OK*: if parsed successfully.
- *LSDNE_PARSE*: if `ascii` could not be parsed into an IP address.

bool **lsdn_ip_eq**(*lsdn_ip_t a*, *lsdn_ip_t b*)
Compare two *lsdn_ip_t* for equality.

Return `true` if `a` and `b` are equal, `false` otherwise.

Parameters

- `a`: IP address.
- `b`: IP address.

bool **`lsdn_ipv_eq`**(*lsdn_ip_t a*, *lsdn_ip_t b*)
Compare two *lsdn_ip_t* for IP version equality.

Return `true` if both `a` and `b` are of the same IP version, `false` otherwise.

Parameters

- `a`: IP address.
- `b`: IP address.

void **`lsdn_mac_to_string`**(const *lsdn_mac_t* * *mac*, char * *buf*)
Format *lsdn_mac_t* as ASCII string.

`buf` must be able to hold at least *LSDN_MAC_STRING_LEN* bytes.

Parameters

- `mac`: MAC address.
- `buf`: destination buffer for the ASCII string.

void **`lsdn_ipv4_to_string`**(const *lsdn_ipv4_t* * *ipv4*, char * *buf*)
Format *lsdn_ipv4_t* as ASCII string.

`buf` must be able to hold at least *LSDN_IPv4_STRING_LEN* bytes.

Parameters

- `ipv4`: IPv4 address.
- `buf`: destination buffer for the ASCII string.

void **`lsdn_ipv6_to_string`**(const *lsdn_ipv6_t* * *ipv6*, char * *buf*)
Format *lsdn_ipv6_t* as ASCII string.

`buf` must be able to hold at least *LSDN_IPv6_STRING_LEN* bytes.

Parameters

- `ipv6`: IPv6 address.
- `buf`: destination buffer for the ASCII string.

void **`lsdn_ip_to_string`**(const *lsdn_ip_t* * *ip*, char * *buf*)
Format *lsdn_ip_t* as ASCII string.

`buf` must be able to hold at least *LSDN_IP_STRING_LEN* bytes.

Parameters

- `ip`: IP address.
- `buf`: destination buffer for the ASCII string.

lsdn_ip_t **lsdn_ip_mask_from_prefix**(enum lsdn_ipv *v*, int *prefix*)
Return an IPv4/6 address mask for the given network prefix.

Sets `prefix` leading bits to 1 and leaves the rest at 0. For example, `lsdn_ipv4_prefix_mask(LSDN_IPV4, 24)` generates address `255.255.255.0`.

Return IP address mask of the appropriate version.

Parameters

- `v`: IP version.
- `prefix`: network prefix number of leading bits to set to 1.

bool **lsdn_is_prefix_valid**(enum lsdn_ipv *ipv*, int *prefix*)
Check if the size of network prefix makes sense for given ip version.

Prefix size must not exceed number of bits of the given address.

Return `true` if `prefix` makes sense for the IP version, `false` otherwise.

Parameters

- `ipv`: IP version.
- `prefix`: network prefix to check.

int **lsdn_ip_prefix_from_mask**(const *lsdn_ip_t* * *mask*)
Calculate length of prefix from a network mask.

Prefix is a number of leading 1 bits in the mask.

Return number of leading 1 bits.

Parameters

- `mask`: network mask to check.

bool **lsdn_ip_mask_is_prefix**(const *lsdn_ip_t* * *mask*)
Check if the given IP address is a valid address mask.

A valid network mask is an IP address which, in bits, is a sequence of 1s followed by a sequence of 0s.

Return `true` if the IP address is a valid mask, `false` otherwise.

Parameters

- `mask`: address to check.

static uint32_t **lsdn_ip4_u32**(const *lsdn_ipv4_t* * *v4*)
Convert *lsdn_ipv4_t* to `uint32_t`.

Return IP address represented as a single `uint32_t` value.

Parameters

- `v4`: address to convert.

const char ***lsdn_mk_name**(**struct** lsdn_context **ctx*, **const** char **type*)
Generate unique name for an object.

The name is based on the context name, type of the object (net, phys, virt, etc.) and a unique object counter on the context. It is in the form `"ctxname-type-12"`.

Results are saved in a reused buffer, so every subsequent call overwrites the previous results. Users need to make a copy of the returned string.

See *lsdn_mk_net_name*, *lsdn_mk_phys_name*, *lsdn_mk_virt_name*, *lsdn_mk_iface_name*, *lsdn_mk_settings_name*

Return pointer to a buffer with a generated unique name.

Parameters

- `ctx`: LSDN context
- `type`: object type (arbitrary string, usually "net", "phys", "virt", "iface" or "settings")

Variables

const *lsdn_mac_t* **lsdn_broadcast_mac**
Broadcast MAC address.

Its value is FF:FF:FF:FF:FF:FF.

const *lsdn_mac_t* **lsdn_all_zeroes_mac**
All zeroes MAC mask.

Matches every MAC address. Its value, obviously, is all zeroes.

const *lsdn_mac_t* **lsdn_multicast_mac_mask**
Multicast MAC address.

Its value is 01:00:00:00:00:00.

const *lsdn_mac_t* **lsdn_single_mac_mask**
Single MAC mask.

Network mask that matches a single MAC address. Its value is all ones.

const *lsdn_ip_t* **lsdn_single_ipv4_mask**
Single IPv4 mask.

Network mask that matches a single IPv4 address. Its value is all ones, or a prefix 32.

const *lsdn_ip_t* **lsdn_single_ipv6_mask**
Single IPv6 mask.

Network mask that matches a single IPv6 address. Its value is all ones, or a prefix 128.

union lsdn_mac
#include <nettypes.h> MAC address.

Public Members

uint8_t lsdn_mac::bytes[LSDN_MAC_LEN]
address as `uint8_t`.

char lsdn_mac::chr[LSDN_MAC_LEN]
address as `char`.

union lsdn_ipv4
#include <nettypes.h> IPv4 address.

Public Members

uint8_t lsdn_ipv4::bytes[LSDN_IPv4_LEN]
address as `uint8_t`.

char lsdn_ipv4::chr[LSDN_IPv4_LEN]
address as `char`.

union lsdn_ipv6
#include <nettypes.h> IPv6 address.

Public Members

uint8_t lsdn_ipv6::bytes[LSDN_IPv6_LEN]
address as `uint8_t`.

char lsdn_ipv6::chr[LSDN_IPv6_LEN]
address as `char`.

struct lsdn_ip
#include <nettypes.h> IP address (any version).

Public Members

lsdn_ipv **v**
IP version.

lsdn_ipv4_t **v4**
IPv4 address.

lsdn_ipv6_t **v6**
IPv6 address.

CHAPTER 2

Programmer's Documentation (Internals)

If you plan on hacking LSCTL, this chapter is for you. It will describe the available internal APIs and how they interact.

2.1 Project organization (components)

The core of LSDN is the **lsdn** library (`liblsdn.so`), which implements all of the C API – the netmodel handling and the individual network types. The library itself relies on *libmnl* library for netlink communication helpers, *libjson* for its ability to dump netmodel into JSON and *uthash* for hash tables.

The command-line tools (*lsctl* and *lsctld*) are built upon our **lsdn-tclext** library, which provides the lsctl language engine and is layered on the C API. For more info, see *Command-line*.

The *lsdn* library itself is composed of several layers/components (see Fig. 2.1 for illustration). At the bottom layer, we have several mostly independent utility components:

- `nettypes.c` manipulates, parses and prints IP addresses and MAC addresses
- `nl.c` provides functions do to more complex netlink tasks than *libmnl* provides - create interfaces, manipulate QDiscs, filters etc.
- `names.c` provides naming tables for netmodel objects, so that we can find physes, virts etc. by name
- `log.c` simple logging to stderr governed by the `LSDN_DEBUG` environment variable
- `errors.c` contains `lsdn_err_t` error codes and infrastructure for reporting commit problems (which do not use simple `lsdn_err_t` errors). The actual problem reporting relies on the netmodel *`lsdn_context`*.

- `list.h` embedded linked-list implementation (every C project needs its own :))

The **netmodel** core (in `net.c` and `lsdn.c`) is responsible for maintaining the network model and managing its life-cycle (more info in *Netmodel implementation*).

For this, it relies on the **rules** (in `rules.c`) system, which helps you manage a chain of TC flower filters and their rules. The system also allows the firewall rules (given by the user) and the routing rules (defined by the virtual network topology) to share the same flower table. However, the sharing is currently not done, because we instead opted to share the routing table among all virts connected through the given phys instead. Since firewall rules are per-virt, they can not live in the shared table. Another function of this module is that it helps us overcome the limit of having at most 32 actions in the kernel for our broadcast rules.

The *netmodel* core only manages the aspects common to all network types – life cycle, firewall rules and QoS, but calls back to a concrete network type plugin for constructing the virtual network. This is done through the *`lsdn_net_ops`* structure and is described more thoroughly in *How to support a new network type*.

The currently supported network types are in `net_direct.c`, `net_vlan.c`, `net_vxlan.c` (all types of VXLANs) and `net_geneve.c`. Depending on the type of the network (learning vs static), the network implementations rely on either `lbridge.c`, a Linux learning bridge, or `sbridge.c`, a static bridge constructed from TC rules. The Linux bridge is pretty self-explanatory, but you can read out more about the TC rule madness in *Static bridge*.

Finally, *liblsdn* also has support for dumping the netmodel in LSCTL and JSON formats, to be either used as configuration files or consumed by other applications (in `dump.c`).

2.2 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 commited.

For this, it is important to understand a life-cycle of an object, illustrated in Fig. 2.2.

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

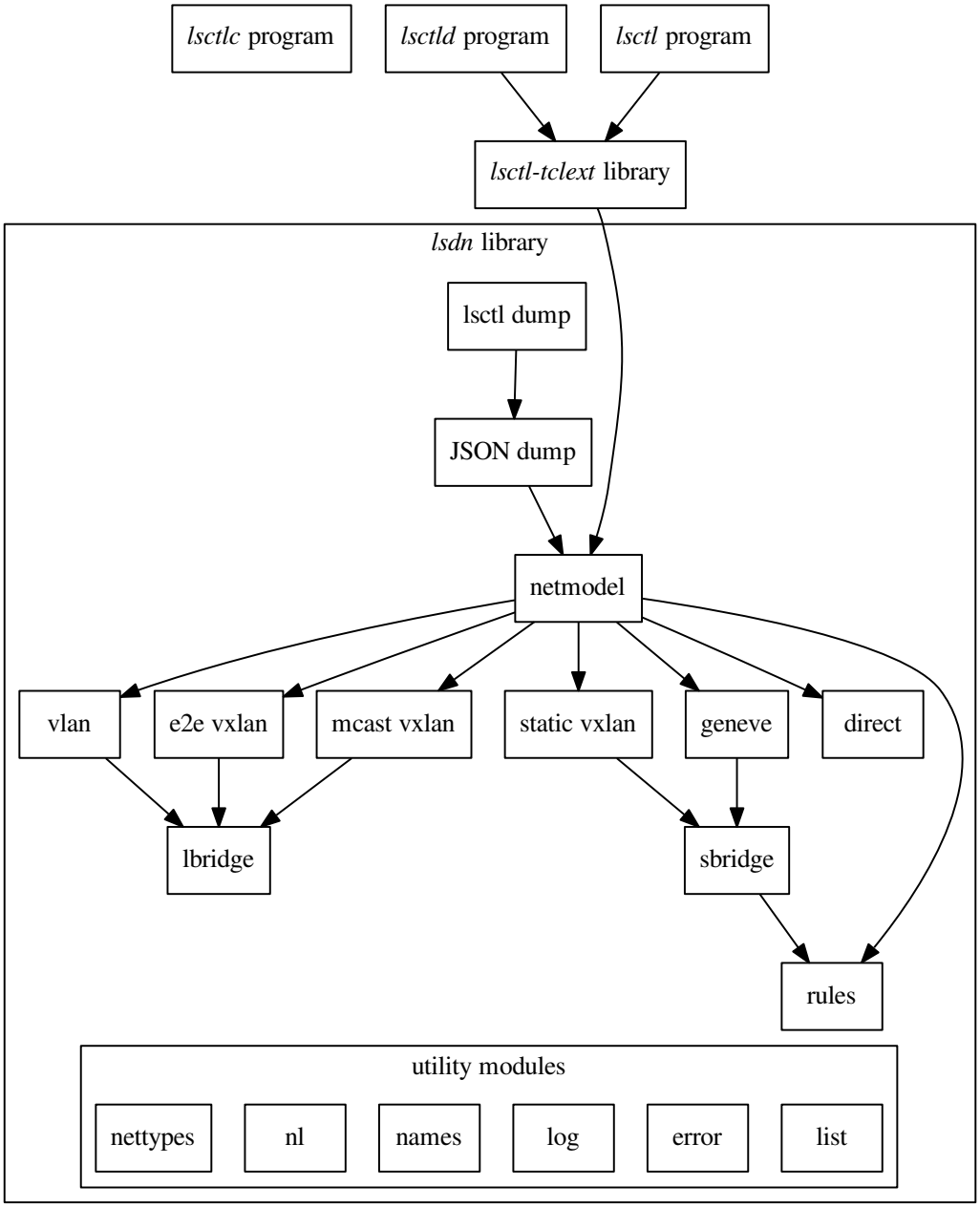


Fig. 2.1: Components and dependencies

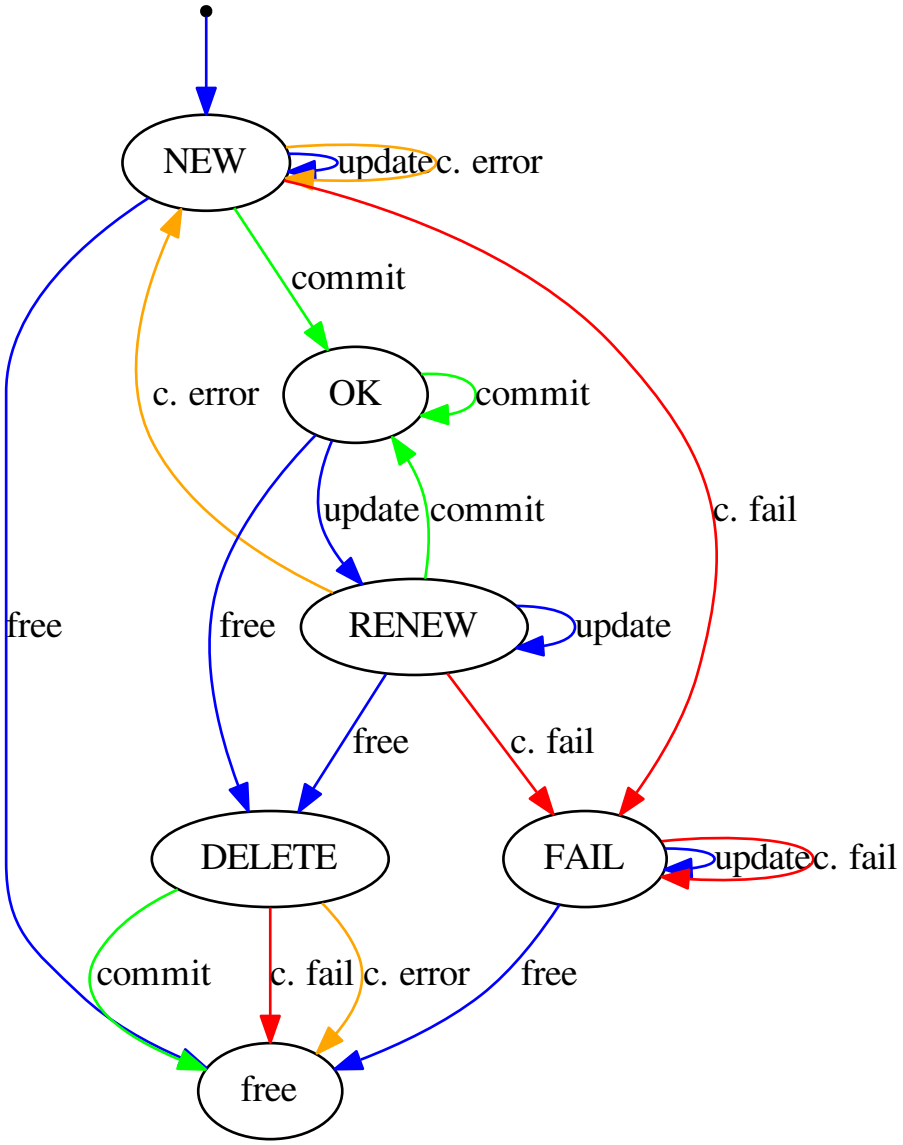


Fig. 2.2: Object states. Blue lines denote update (attribute change, free), green lines commit, orange lines errors during commit, red lines errors where recovery has failed.

commit is called and the object is deleted from kernel. The objects in *DELETE* state can not 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.

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

2.3 How to support a new network type

LSDN does not have an official stable extension API, but the network modules are intended to be mostly separate from the rest of the code. However, there are still a few places you will need to touch.

To support a new type of network :

- add your network to the `lsdn_nettype` enum (in `private/lsdn.h`)
- add the settings for your network to the `lsdn_settings` struct (in `private/lsdn.h`). Place them in the anonymous union, where settings for other types are placed.
- declare a function `lsdn_settings_new_xxx` (in `include/lsdn.h`)
- create a new file `net_xxx.c` for all your code and add it to the `CMakeLists.txt` file

The **settings_new** function will inform LSDN how to use your network type. Do not forget to do the following things in your *settings_new* function:

- allocate new `lsdn_settings` structure via malloc
- initialize the settings using `lsdn_settings_init_common` function
- fill in the:
  - `nettype` (as you have added above)
  - `switch_type` (static, partially static, or learning, purely informational, has no effect)

  - `ops` (*`lsdn_net_ops`* will be described shortly)

- return the new settings

Also note that your function will be part of the C API and should use `ret_err` to return error codes (instead of plain `return`), to provide correct error handling (see *Error codes and error handling*).

However, the most important part of the *settings* is the **lsdn_net_ops** structure – the callbacks invoked by LSDN to let you construct the network. First let us have a quick look at the structure definition (full commented definition is in the source code or *Generated Doxygen Documentation*):

`struct lsdn_net_ops`

Public Members

char* **`type`**

uint16_t**`(*get_port)`**(struct *lsdn_settings* *s)

*lsdn_ip_t***`(*get_ip)`**(struct *lsdn_settings* *s)

lsdn_err_t**`(*create_pa)`**(struct lsdn_phys_attachment *pa)

lsdn_err_t**`(*add_virt)`**(struct *lsdn_virt* *virt)

lsdn_err_t**`(*add_remote_pa)`**(struct lsdn_remote_pa *pa)

lsdn_err_t**`(*add_remote_virt)`**(struct lsdn_remote_virt *virt)

lsdn_err_t**`(*destroy_pa)`**(struct lsdn_phys_attachment *pa)

lsdn_err_t**`(*remove_virt)`**(struct *lsdn_virt* *virt)

lsdn_err_t**`(*remove_remote_pa)`**(struct lsdn_remote_pa *pa)

lsdn_err_t**`(*remove_remote_virt)`**(struct lsdn_remote_virt *virt)

void**`(*validate_net)`**(struct *lsdn_net* *net)

void**`(*validate_pa)`**(struct lsdn_phys_attachment *pa)

void**`(*validate_virt)`**(struct *lsdn_virt* *virt)

unsigned int**`(*compute_tunneling_overhead)`**(struct lsdn_phys_attachment *pa)

The first callback that will be called is *`lsdn_net_ops::create_pa`*. PA is a shorthand for phys attachment and the call means that the physical machine this LSDN is managing has attached to a virtual network. Typically you will need to prepare a tunnel(s) connecting to the virtual network and a bridge connecting the tunnel(s) to the virtual machines (they will be connected later).

If your network does all packet routing by itself, use the `lbridge.c` module. It will create an ordinary Linux bridge and allow you to connect your tunnel interface via that bridge. We assume your tunnel has a Linux network interface. If not, you will have to

come up with some other way of connecting it to the Linux bridge, or use something else than a Linux bridge. In that case, feel free not to use `lbridge.c` and do custom processing in *`lsdn_net_ops::create_pa`*.

If the routing in your network is static, use *Static bridge*. It will allow you to setup a set of flower rules for routing the packets, ending in custom TC actions. In these actions, you will typically set-up the required routing metadata for the packet and send it of.

After the PA is created, you will receive other callbacks.

The *`lsdn_net_ops::add_virt`* callback is called when a new virtual machine has connected on the phys your are managing. Typically, you will add the virtual machine to the bridge you have created previously.

If your network is learning, you are almost done. But if it is static, you will want to handle *`lsdn_net_ops::add_remote_pa`* and *`lsdn_net_ops::add_remote_virt`*. These callbacks inform you about the other physical machines and virtual machines that have joined the virtual network. If the routing is static, you need to be informed about them to correctly set-up the routing information (see *Static bridge*). Depending on the implementation of the tunnels in Linux, you may also need to create tunnels for each other remote machine. In that case, the *`lsdn_net_ops::add_remote_pa`* callback is the right place.

Finally, you need to fill in the *`lsdn_net_ops::type`* with the name of your network type. This will be used as an identifier in the JSON dumps. At this point you might want to decide if your network should be supported in *Lsctl Configuration Files* and modify `lsext.c` accordingly. The network type names in LSCTL and JSON should match.

The other callbacks are mandatory. Naturally, you will want to implement the `remove/destroy` callbacks for all your `add/create` callbacks. There are also validation callbacks, that allow you to reject invalid network configuration early (see *Validation*). Finally, LSDN can check the uniqueness of the listening IP address/port combinations your tunnels use, if you provide them using *`lsdn_net_ops::get_ip`* and *`lsdn_net_ops::get_port`*.

Since an example is the best explanation, we encourage you to look at some of the existing plugins – *VLAN* (`net_vlan.c`) for learning networks, *Geneve* (`net_geneve.c`) for static networks.

2.4 Static bridge

The static-bridge subsystem provides helper functions to help you manage an L2 router built on TC flower rules and actions. Because it is based on TC it can be integrated with the metadata based Linux tunnels.

Metadata-based tunnels (or sometimes called lightweight IP tunnels) are Linux tunnels that can choose their tunnel endpoint by looking at a special packet metadata. This means you do not need to create a new network interface for each endpoint you want to communicate with, but one shared interface can be used, with only the metadata changing. In our case, we use TC actions to set these metadata depending on

the destination MAC address (since we know where a virtual machine with that MAC lives). The setup is illustrated in Fig. 2.3.

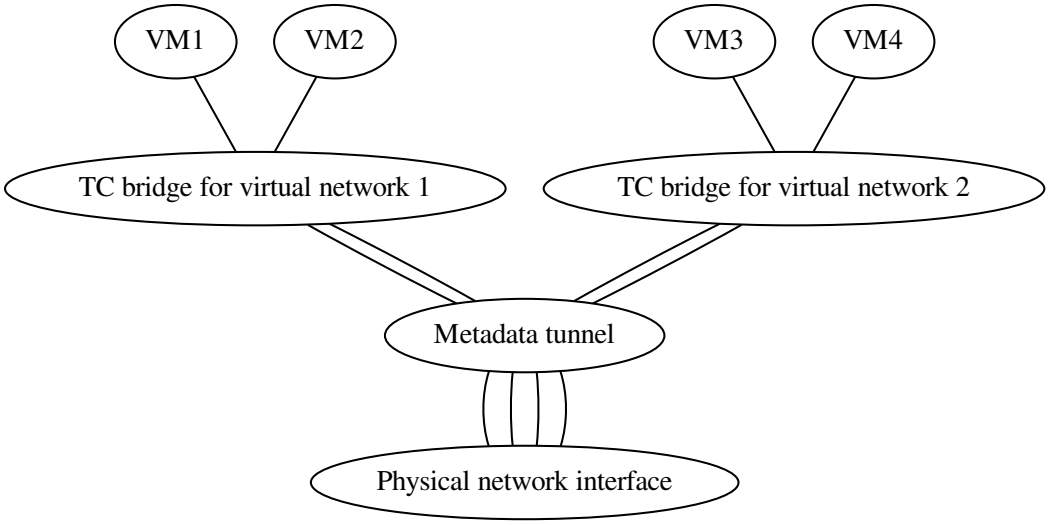


Fig. 2.3: Two virtual networks using a static routing (using TC) and shared metadata tunnel. Lines illustrate a connection of each VM.

The static bridge is not a simple implementation of Linux bridge in TC. A bridge is a virtual interface with multiple enslaved interfaces connected to it. However, the static bridge needs to deal with the tunnel metadata during its routing. For that, it provides the following C structures:

Struct **lsdn_sbridge** represents the bridge as a whole. Internally, it will create a helper interface to hold the routing rules.

Struct **lsdn_sbridge_phys_if** represents a Linux network interface connected to the bridge. This will typically be a virtual machine interface or a tunnel. Unlike with a classic bridge, a single interface may be connected to multiple bridges.

Struct **lsdn_sbridge_if** represents the connection of *sbridge_phys_if* to the bridge. For virtual machines *sbridge_if* and *sbridge_phys_if* will be in a one to one correspondence, since virtual machine can not be connected to multiple bridges. If a sbridge is shared, you have to provide a criteria splitting up the traffic, usually by the *virtual network identifier*.

Struct **lsdn_sbridge_route** represents a route through given *sbridge_if*. For a virtual machine, there will be just a single route, but metadata tunnel interfaces can provide multiple routes, each leading to a different physical machine. The users of the static-bridge module must provide TC actions to set the correct metadata for that route.

Struct **lsdn_sbridge_mac** tells LSDN to use a given route when sending packets to a given MAC address. There will be a *sbridge_mac* for each VM on a physical machine where the route leads.

The structures above need to be created from LSDN callbacks. For a network with static routing, and metadata tunnels, the correspondence will look similar to this:

callback	sbridge
`create_pa` (first call)	create **phys_if** for tunnel
`create_pa`	create **sbridge** and **if** for tunnel
`add_virt`	create **if**, **route** and **mac**
`add_remote_pa`	create **route** for the physical machine
`add_remote_virt`	create **mac** for the route

2.5 Command-line

The *Lsctl Configuration Files* are interpreted by the *lsdn-tclext* library. We have chosen to use the TCL language as a basis for our configuration language. Although it might seem as a strange choice, it provides bigger flexibility for creating DSLs than let's say JSON or YAML. Basically, TCL enforces just a single syntactic rule:`{}` and `[]` parentheses.

Originally, we had a YAML configuration parser, but the project has changed its heading very significantly and the parser was left behind. TCL bindings were done as a quick experiment and have aged quite well since then. The YAML parser was later abandoned altogether.

Naturally, there are advantages to JSON/YAML too. Since our language is Turing complete, it is not as easily analyzed by machines. However, it is always possible to just run the configuration scripts and then examine the network model afterwards. The TCL approach also brings a lot of features for free: conditional compilation, variables, loops etc.

lsdn-tclext library is a collection of TCL commands. One way to use it is in a custom host program (that is *lsctl* and *lsctld*). The program will use *libtcl* to create a TCL interpreter and then call *lsdn-tclext* to register the LSDN specific commands.

lsctld creates the interpreter, registers the LSDN commands, binds to a Unix domain socket and listens for commands. The commands (received as plain strings) are fed to the interpreter and *stdout* and *stderr* is sent back.

lsctlc does not depend on TCL or `lsdn-tclext`, since it is a simple netcat-like program that simply pipes its input to the running `lsctld` instance and receives script output back.

lsctl is just a few lines, since it uses the `Tcl_Main` call. `Tcl_Main` is provided by TCL for building a custom TCL interpreter quickly and does argument parsing and interpreter setup (`tclsh` is actually just `Tcl_Main` call).

The other way to use *lsdn-tclext* is as a regular TCL extension, from `tclsh`. `pkgIndex.tcl` is provided by LSDN and so LSDN can be loaded using the `require` command.

2.6 Test Environment

Our test environment is highly modular, extremely powerful, easy to use and without any complex dependencies. Thus it is easily extensible even for outsiders and people beginning with the project.

2.6.1 CTest

The core of the environment is `CTest` testing tool from `CMake`. It provides a very nice way how to define all the tests in a modular way. We create test parts which can be combined together for one complex test. This means that you can for example say that you want to use `geneve` as a backend for the network, you want to test `migrate` which means that the migration of virtual machines will be tested and as a verifier use `ping`. `CTest` configuration file is called `CMakeLists.txt` and tests composed from parts can be added with `test_parts(...)` command. Examples follow, starting with the example described above:

```
test_parts(geneve migrate ping)
```

For `vlan` and `dhcp` test:

```
test_parts(vlan dhcp)
```

For backend without tunnelling, migration with daemon's help keeping the state in memory and ping:

```
test_parts(direct migrate-daemon ping)
```

For complete list of all tests see `CMakeLists.txt` in the `test` directory and all parts usable to create complex tests are in `test/parts`. To run all the tests inside the `CTest` testing tool just go to `test` folder and run

```
ctest
```

2.6.2 Parts

In the previous section we described the big picture of tests execution. Now we will describe what *part* is and how to define it. *Part* is a simple bash script defining functions according to prescribed API for our test environment.

Function `prepare()` is used for establishing the physical network environment unrelated to the virtual network we would like to manage. These are "wires" we will use for our virtual networking.

`connect()` is the main phase for setting the virtual network environment. LSDN is usually used in this function for configuring all the virtual interfaces and virtual network appliances.

To test if the applied configuration is working, i.e. it has the expected behavior, function `test()` is used. Most often `ping` is used here, but you can use anything for testing the functionality.

If you want to do some special cleanup you can use `cleanup()` function.

Back to the *part* primitive - you can combine various parts together but every rational test should define all the described functions no matter how many parts are used.

`CTest` is pretty good at automated execution of complete tests but if you want to debug the test or execute just part of it there is a `run` script. This script allows you to execute just selected stages and combine parts in a comfortable way. It's usage is self-explanatory:

```
Usage:
        ./run -xpctz [parts]
  -x  trace all commands
  -p  run the prepare stage
  -c  run the connect stage
  -t  run the test stage
  -z  run the cleanup stage
```

Thus for running a test for the example from the beginning, but just using the `connect` and `prepare` stages you can call:

```
./run -pc geneve migrate ping
```

2.6.3 QEMU

Because we are dependent on fairly new versions of the Linux Kernel we provide scripts for executing tests in a virtualized environment. This is useful when you use some traditional Linux distribution like Ubuntu with older kernel and you do not want to compile or install custom recent kernel.

As a hypervisor we use QEMU with Arch Linux user-space. Here are several steps you need to follow for the execution in QEMU:

1. Download actual Linux Kernel to `$linux-path`.
2. Run `./create_kernel.sh $linux-path`. This will generate valid kernel with our custom `.config` file.
3. Run `./create_rootfs.sh` which will create the user-space for virtual machine with all dependencies. Here you need `pacman` for downloading all the packages.
4. Run `./run-qemu $kernel-path $userspace-path all` which will execute all tests and shut down.

`run-qemu` script is much more powerful and you can run all the examples described above together with debugging in the shell inside that virtual machine. The usage is following:

```
usage: run-qemu [--help] [--kvm] [--gdb] kernel rootfs guest-command

Available guest commands: shell, raw-shell, all.
```

`shell` will execute just a shell and leave the test execution up to you and `raw-shell` is just for debugging the virtual machine user-space because it will not mount needed directories for tests. `all` executes all the tests as we have already shown above.

CHAPTER 3

Developmental Documentation

LSDN project focuses on the problem of easily manageable networking setup in thesch environment of virtual machines and cloud environment generally. It perfectly fits to large scale deployment for managing complex virtual networks in data centers as well as small scale deployment for complete control over containers in the software developer's virtual environment. Naturally the network interface providers have to run Linux Kernel as we use it for the real networking work.

Two core goals which LSDN resolved are:

1. Make Linux Kernel Traffic Control (TC) Subsystem usable:
   - LSDN provides library with high-level C API for linking together with recent orchestrators.
   - Domain Specific Language (DSL) for standalone configuration is designed and can be used as is.
2. Audit the TC subsystem and verify that it is viable for management of complex virtual network scenarios as is.
   - Bugs in Linux kernel were found and fixed.
   - TC is viable to be used for complex virtual network management.

3.1 Problem Introduction

The biggest challenge in the cloud industry today is how to manage enormous number of operating system instances together in some feasible and transparent way. No matter if containers or full computer virtualization is used the virtualization of networking brings several challenges which are not present in the world of classical physical

networks, e.g. the isolation customer's networks inside of datacenter (multi-tenancy), sharing the bandwidth on top of physical layer etc. All these problems have to be tackled in a very thoughtful way. Furthermore it would be nice to build high-level domain specific language (DSL) for configuring the standalone network as well as C language API for linking and using with orchestrators.

The networking functionality of such a needed tool is following:

- Support for Virtual Networks, Switches and Ports.
- API for management via library.
- DSL for stand-alone management.
- Network Overlay.
- Multi-tenancy.
- Firewall.
- QoS support.

Most of the requirements above are barely fulfilled with vast majority of recent products which is the main motivation for LSDN project.

3.2 Current Situation

The domination of open-source technologies in the cloud environment is clear. Thus we do not focus on the cloud based on closed, proprietary technologies such as cloud services from Microsoft.

In the open-source world the position of Open vSwitch (OVS) is dominant. It is kernel module providing functionality for managing virtual networks and is used by all big players in the cloud technology, e.g. RedHat. However Linux Kernel provides almost identical functionality via it's traffic control (TC) subsystem. Thus there is code duplicity of TC and OVS and furthermore the code base of OVS is not as clear as the code base of TC. Hence the effort to eliminate OVS in favor of TC and focus to improve just one place in the Linux networking world.

Although TC is super featureful it has no documentation (literally zero) and the error handling of it's calls is most of the time without any additional information. Hence for correct TC usage one has to read bunch of kernel source codes. Add bugs in a rare used places and we have very powerful but almost unusable software. Thus some higher level API is very attractive for everyone who wants to use more advanced networking features from Linux kernel.

3.3 Similar Projects

There is no direct competition among tools building on top of TC to make it much more usable (actually to make it just usable). However there are competitors for TC, which

are not that powerful or they are just modules full of hacks and are taken positively in the Linux mainline.

3.3.1 open vSwitch

- Similar level of functionality to TC.
- Complex, hardly maintainable code and code duplicity with respect to TC.
- External module without any chance to be accepted to kernel mainline.
- Slightly more user convenient configuration than TC.

3.3.2 vSphere Distributed Switch

- Out-of-the game because it is not open-source.
- Not that featureful as TC. E.g. no firewall, geneve etc.
- Closed-source product of VMware.
- Hardly applicable to heterogeneous open-source environment.

3.3.3 Hyper-V Virtual Switch

- Out-of-the game because it is not open-source.
- Not that featureful as TC. E.g. no firewall, geneve etc.
- Closed-source product of Microsoft.
- Hardly applicable to heterogeneous open-source environment.

3.4 Development Environment

In this section we present all the tools used in our project which are worth mentioning.

3.4.1 Development Tools

The platform independent builds with all the dependency and version checks are done thanks to **cmake** in cooperation with **pkgconfig**. This is much nicer and more featureful alternative to autoconf tools.

Furthermore we kept everything since the beginning in the **GIT** repository on GitHub. We used pretty intensely with all it's features like branches etc. VCS is a must for any project and GIT is the most common choice.

When we were developing daemon (for migration support) we found library called **libdaemon** which helps you to write system daemon in a proper way with all the signal handling, sockets management and elimination of code full of race conditions.

As a code editor only **VIM** and **ed** were allowed.

3.4.2 Testing Environment

Our testing environment is based on the highly modular complex of **bash scripts**, where every part which should be tested defines prescribed functions which are further executed together with other parts. Like this we can create complex tests just with combination of several parts.

For automatic test execution and it's simplification we used **ctest** tool which is part of cmake package.

The continuous integration was used through the **Travic-CI** service which after every code commit executed all the tests and provides automatic email notification in case of failure.

We have also extensive support for testing on not supported kernels via **QEMU**. Automatic scripts are able to create minimalistic and up-to-date Arch Linux root filesystem, boot up-to-date kernel and ran all tests. This method is also used on Travis-CI, where only LTS versions of Ubuntu are available.

Of course various networking tools like dhcpd, dhcpcd, dhclient, tcpdump, iproute, ping etc. were used for diagnostics as well as directly in tests.

Note that during tests we were highly dependent on **Linux namespaces**, hence we were able to simulate several virtual machines without any overhead and speed up all the tests.

3.4.3 Communication Tools

Communication among all team members and leaders was performed via old-school mailing lists and IRC combo. We used our own self-hosted **mailman** instance for several mailing lists:

- lsdn-general for general talk, organization, communication with leaders and all important decisions.
- lsdn-travis for automatic reports from Travis-CI notifying us about commits which break the correct functionality.
- lsdn-commits for summary of every commit we made. This was highly motivation element in our setup, because seeing your colleague committing for the whole day can make you feel really bad. Furthermore discussion about particular commit were done in the same thread, which enhances the organization of decisions we made and why.

For real-time communication we used **IRC** channel #lsdn on Freenode. This is useful especially for flame-wars and arguing about future design of the tool.

We have developed a simple bot for our mailing list to remind us of important deadlines and nag people who have not commited to the repository for a long time. This helped us "feel" the schedule and keeped us focused on work.

3.4.4 Documentation Tools

The project has fairly nice documentation architecture. C source codes including API are commented with **Doxygen**, which is a standard way how to this kind of task. Then the Doxygen output is used and enhanced with tons of various documentations (user, developmental…) and processed with Sphinx.

Sphinx is a tool for creating really nice documentations and supports various outputs. Like this we are able to have HTML and PDF documentation synced and both formats look fabulous.

Furthermore we use **readthedocs.io** for automatic generation of documentation after every documentation commit. This also means that we have always up-to-date documentation online in browsable HTML version as well as downloadable and printable PDF version. Note that PDF generation uses LaTeX as a typesetting system, thus the printed documentation looks great.

The whole documentation source is written in **reStructuredText** (rst) markup language which greatly simplified the whole process of creation such a comprehensive documentation.

3.4.5 Open-source contributions

We have identified a few bugs in the Linux kernel during our development. We believe this is mainly because of the unusual setups we exercise and new kernel features (such as goto chain, lwtunnels) we use. Following bugs were patched or at least reported and patched by someone else:

- net: don't call update_pmtu unconditionally (reported)
- net: sched: crash on blocks with goto chain action
- net: sched: fix crash when deleting secondary chains
- v9fs over btrfs (mailing list dead, not merged)
- net: sched: report if filter is too large to dump

We have also identified a bug in iproute2:

- tc: fix an off-by-one error while printing tc actions

Naturally, our tooling also has problems, so we also fixed a bug in sphinx and breathe.

3.5 Project Timeline

The project came from an idea of Jiri Benc (Linux Kernel Networking Developer) from Red Hat Czech who wanted to create a proof-of-concept tool which will try to replace Open vSwitch with purely Linux Kernel functionality and find all the missing functionality or bugs in Linux Kernel which would block or slow down the effort to eliminate Open vSwitch.

These days Vojtech Aschenbrenner was an intern in Jiri's team and also a student who was looking for challenging Software Project topic from Systems field, which was a mandatory part of studies at Charles University. Hence the topic arose.

Formation of the team was not that straightforward. In the beginning the team was composed from 7 people. They were people with Systems interests and also great computer scientists. The property of excellency was actually the biggest problem of the team. In the beginning part of the implementation phase 3 people left the studies and also team because of much better offer. It was two times because of Google and one time because of Showmax. Thus 4 people left in the team which was still manageable.

However another personal problems came with studies in the US and jobs of the remaining members. Vojtech Aschenbrenner left to the University of Rochester and have almost no time to work on a project for a lot of weeks. Similar situation came to Adam Vyškovský who left to Paris because of a dream job in an aviation. Jan Matějek still had full-time job in SUSE and it looked like the project has a huge problems and will most probably fail. However Roman Kápl showed his true determination and saved the project although he has also part-time job in a systems company. It is for sure, that the project would fail without his knowledge, skills in system programming and diligence. When all the remaining members who were still part of the team saw how he is continuously working on the project they came back from abroad and decided to finish the project as well as their master studies instead of continuing they career elsewhere. All of the team members believe that Roman influenced our future life in a positive way.

After this we managed to do hackatons quite often and do the majority of the work in a several months. Because the problematic part of the project where a lot of people left was before the official start the official timeline of the project was according to the plan and we were able to fulfill our deadlines which were following:

- Month 1:
  - Analysis of the requirements of cloud environments for software defined networking.
  - Analysis and introduction to Linux Kernel networking features, especially traffic control framework and networking layer of the Linux Kernel.
  - Description of detailed use-cases which will be implemented.
- Month 2:
  - API design.
- Months 3 - 7:

  - Implementation of the complete functionality of the project. This was the main developing part.
- Months 8 - 9:
  - Finalization.
  - Debugging.
  - Documentation.
  - Presentation preparation.
  - (The most intense part)

3.6 Team Members

The project was originally started with people who are no longer in the team from various of reasons. We would like to honorably mention them, because the initial project topics brainstorming were done with them.

- *Martin Pelikán* left to Google Sydney few weeks after the project was started. Although he is a non-sleeper which can work on several projects together he was not able to find a spare time for this one. This was a big loss because his thesis was about TC.
- *David Krška* left to Google London few weeks after his bachelor studies graduation.
- *David Čepelík* left to Showmax one semester after his bachelor studies graduation.

The rest of the people who started the project were able to stay as a part of the team and finish it.

- *Vojtěch Aschenbrenner* established the team and tried to lead the project. He also created the infrastructure, hosted and managed the communication platform and officially communicated with authorities from the University as well as mediated the communication inside the team. He created the LSDN daemon and the way how it communicates with the client. He also worked on the testing environment's scripts, developmental and testing part of documentation and maintains the Arch Linux package.
- *Roman Kápl* is the main developer of the project. He participated on all parts of the project, most notably on internal parts, which are directly communication with kernel. There is no part of the project, which Roman did not touched. He always solved the most difficult problems, fixed several bugs in Linux kernel and in tools used in the project. He maintains package for distributions based on Debian.
- *Jan Matějek* was mainly involved in writing documentation generated by Doxygen and code reviews. Thanks to it he fixed logical mistakes in the project and commented the whole source code in a great detail. He was partly involved also in non-generated documentation. He maintains package for distributions using RPM. He was the original author of the CMake automated tests.

- *Adam Vyškovský* was together with Roman the main developer of internals and is the main author of the TCL/JSON exporter, however he also wrote big portion of non-generated documentation and most notably periodically revised it and fixed non-trivial amount of mistakes. He spent enormous amount of time during debugging the netlink communication with Linux kernel, which was absolutely crucial for the project.

At this place Jiri Benc, the official leader from Red Hat Czech, should be mentioned because discussion with him was always full of knowledge and his overview of the Linux Kernel and open-source world is enormous. He always found a spare time to arrange a meeting with us and was also willing to help us move forward and motivate us.

3.7 Conclusion, Contribution and Future Work

The project was able to fulfill all the requirements set in the beginning and also follow the plan created in the beginning. This means that all the requested functionality was implemented and properly tested. Furthermore it was documented all through from both programmers view and also from user (API) view. Also detailed use cases with the quick-start guide were described. Especially the quick-start guide showed how easy it is to create complex virtual networking scenario in a few steps with very minimal configuration files.

At the end the whole project was all through tested in both, virtual setups, physical setups as well as hybrid setups. Finally the demo presentation showing the power of LSDN was created. This part of work showed how capable LSDN (and TC framework) is in terms of replacing Open vSwitch – it is capable and the direction of TC framework development goes in the right way of replacing Open vSwitch in the future.

Another big success of the project was patching the upstream of Linux Kernel as well as patching the tooling as Sphinx and Breathe. Also several bugs were reported. This was the secondary and optional target of the project which was also fulfilled.

LSDN has the ambition to become the only tool using the extremely powerful TC framework in Linux Kernel and use it in very user convenient way with very minimal additional dependencies for creation complex virtual network scenarios. Also the core of the tool is written efficiently in C, thus there is no performance impact of using LSDN. Furthermore we were able to push LSDN installation packages to user repositories of Linux distributions or at least create the packages. This means that the comfort of installation is maximal which helps to fulfill the main goal of creating easy to use management tool for complex networks.

Because of the very promising future of the tool, the LSDN team is willing to continue in supporting the project as well as integrate future enhancements in the TC framework, fix bugs found in the production as well as customize the project according to the future needs of virtual networks.

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).

The next challenging step is to integrate LSDN into most popular virtualization orchestrators and eliminate Open vSwitch. This would attract more developers and make the project part of the state of the art cloud ecosystem - this is the real goal!

CHAPTER 4

Generated Doxygen Documentation

4.1 Doxygen (Generated documentation)

Generated documentation by Doxygen for LSDN can be found at https://asch.github.io/lsdn-doxygen . This documentation contains all parts of LSDN (not only the public API).

Index

Symbols

A

C

D

F

G

I

K

L

M

N

P

Q

R

S

V