LXD

LXD contributors

Jul 11, 2024

CONTENTS

LXD is a next generation system container and virtual machine manager. It offers a unified user experience around full Linux systems running inside containers or virtual machines.

It's image based with pre-made images available for a wide number of Linux distributions and is built around a very powerful, yet pretty simple, REST API.

CHAPTER

ONE

SECURITY

Consider the following aspects to ensure that your LXD installation is secure:

- Keep your operating system up-to-date and install all available security patches.
- Use only supported LXD versions (LTS releases or monthly feature releases).
- Restrict access to the LXD daemon and the remote API.
- Do not use privileged containers unless required. If you use privileged containers, put appropriate security measures in place. See the LXC security page for more information.
- Configure your network interfaces to be secure.

See *Security* for detailed information.

Important: Local access to LXD through the UNIX socket always grants full access to LXD. This includes the ability to attach file system paths or devices to any instance as well as tweak the security features on any instance.

Therefore, you should only give such access to users who you'd trust with root access to your system.

CHAPTER

TWO

SUPPORT

See *Support* for information on how to get help.

CHAPTER

THREE

CONTRIBUTING

Fixes and new features are greatly appreciated. See *Contributing* for more information.

3.1 Getting started

In addition to the documentation in this section, see the Getting Started guide on the website.

3.1.1 Requirements

Go

LXD requires Go 1.16 or higher and is only tested with the golang compiler.

We recommend having at least 2GB of RAM to allow the build to complete.

Kernel requirements

The minimum supported kernel version is 3.13.

LXD requires a kernel with support for:

- Namespaces (pid, net, uts, ipc and mount)
- Seccomp

The following optional features also require extra kernel options:

- Namespaces (user and cgroup)
- AppArmor (including Ubuntu patch for mount mediation)
- Control Groups (blkio, cpuset, devices, memory, pids and net_prio)
- CRIU (exact details to be found with CRIU upstream)

As well as any other kernel feature required by the LXC version in use.

LXC

LXD requires LXC 3.0.0 or higher with the following build options:

- apparmor (if using LXD's apparmor support)
- seccomp

To run recent version of various distributions, including Ubuntu, LXCFS should also be installed.

QEMU

For virtual machines, QEMU 4.2 or higher is preferred. Older versions, as far back as QEMU 2.11 have been reported to work properly, but support for those may accidentally regress in future LXD releases.

Additional libraries (and development headers)

LXD uses `dqlite` for its database, to build and setup it, you can run `make deps`.

LXD itself also uses a number of (usually packaged) C libraries:

- libacl1
- libcap2
- libuv1 (for `dqlite`)
- libsqlite3 >= 3.25.0 (for `dqlite`)

Make sure you have all these libraries themselves and their development headers (-dev packages) installed.

3.1.2 Installing LXD

The easiest way to install LXD is to install one of the available packages, but you can also install LXD from the sources.

Installing LXD from packages

The LXD daemon only works on Linux but the client tool (`lxc`) is available on most platforms.

OS	Format	Command
Linux	Snap	snap install lxd
Windows	Chocolatey	choco install lxc
MacOS	Homebrew	brew install lxc

Installing LXD from source

We recommend having the latest versions of liblxc (>= 3.0.0 required) available for LXD development. Additionally, LXD requires Golang 1.13 or later to work. On ubuntu, you can get those with:

```
sudo apt update
sudo apt install acl attr autoconf dnsmasq-base git golang libacl1-dev libcap-dev␣
↪liblxc1 liblxc-dev libsqlite3-dev libtool libudev-dev liblz4-dev libuv1-dev make pkg-
↪config rsync squashfs-tools tar tcl xz-utils ebtables
```

There are a few storage backends for LXD besides the default "directory" backend. Installing these tools adds a bit to initramfs and may slow down your host boot, but are needed if you'd like to use a particular backend:

```
sudo apt install lvm2 thin-provisioning-tools
sudo apt install btrfs-progs
```

To run the testsuite, you'll also need:

```
sudo apt install curl gettext jq sqlite3 socat bind9-dnsutils
```

From Source: Building the latest version

These instructions for building from source are suitable for individual developers who want to build the latest version of LXD, or build a specific release of LXD which may not be offered by their Linux distribution. Source builds for integration into Linux distributions are not covered here and may be covered in detail in a separate document in the future.

```
git clone https://github.com/canonical/lxd
cd lxd
```

This will download the current development tree of LXD and place you in the source tree. Then proceed to the instructions below to actually build and install LXD.

From Source: Building a Release

The LXD release tarballs bundle a complete dependency tree as well as a local copy of libraft and libdqlite for LXD's database setup.

```
tar zxvf lxd-4.18.tar.gz
cd lxd-4.18
```

This will unpack the release tarball and place you inside of the source tree. Then proceed to the instructions below to actually build and install LXD.

Starting the Build

The actual building is done by two separate invocations of the Makefile: `make deps` – which builds libraries required by LXD – and `make`, which builds LXD itself. At the end of `make deps`, a message will be displayed which will specify environment variables that should be set prior to invoking `make`. As new versions of LXD are released, these environment variable settings may change, so be sure to use the ones displayed at the end of the `make deps` process, as the ones below (shown for example purposes) may not exactly match what your version of LXD requires:

We recommend having at least 2GB of RAM to allow the build to complete.

```
make deps
# Follow the instructions from `make deps` to export the required environment variables.
# For example:
#  export CGO_CFLAGS="${CGO_CFLAGS} -I$(go env GOPATH)/deps/dqlite/include/ -I$(go env␣
↪GOPATH)/deps/raft/include/"
#  export CGO_LDFLAGS="${CGO_LDFLAGS} -L$(go env GOPATH)/deps/dqlite/.libs/ -L$(go env␣
↪GOPATH)/deps/raft/.libs/"
```

(continues on next page)

(continued from previous page)

```
#  export LD_LIBRARY_PATH="$(go env GOPATH)/deps/dqlite/.libs/:$(go env GOPATH)/deps/
↪raft/.libs/:${LD_LIBRARY_PATH}"
#  export CGO_LDFLAGS_ALLOW="(-Wl,-wrap,pthread_create)|(-Wl,-z,now)"
make
```

From Source: Installing

Once the build completes, you simply keep the source tree, add the directory referenced by `$(go env GOPATH)/bin` to your shell path, and set the `LD_LIBRARY_PATH` variable printed by `make deps` to your environment. This might look something like this for a `~/.bashrc` file:

```
export PATH="${PATH}:$(go env GOPATH)/bin"
export LD_LIBRARY_PATH="$(go env GOPATH)/deps/dqlite/.libs/:$(go env GOPATH)/deps/raft/.
↪libs/:${LD_LIBRARY_PATH}"
```

Now, the `lxd` and `lxc` binaries will be available to you and can be used to set up LXD. The binaries will automatically find and use the dependencies built in `$(go env GOPATH)/deps` thanks to the `LD_LIBRARY_PATH` environment variable.

Machine Setup

You'll need sub{u,g}ids for root, so that LXD can create the unprivileged containers:

```
echo "root:1000000:1000000000" | sudo tee -a /etc/subuid /etc/subgid
```

Now you can run the daemon (the `--group sudo` bit allows everyone in the `sudo` group to talk to LXD; you can create your own group if you want):

```
sudo -E PATH=${PATH} LD_LIBRARY_PATH=${LD_LIBRARY_PATH} $(go env GOPATH)/bin/lxd --group␣
↪sudo
```

Note: If `newuidmap/newgidmap` tools are present on your system and `/etc/subuid`, `etc/subgid` exist, they must be configured to allow the root user a contiguous range of at least 10M uid/gid.

3.1.3 Frequently asked questions

General issues

How to enable LXD server for remote access?

By default, the LXD server is not accessible from the network as it only listens on a local Unix socket. You can make LXD available from the network by specifying additional addresses to listen to. This is done with the `core.https_address` config variable.

To see the current server configuration, run:

```
lxc config show
```

To set the address to listen to, first find out what addresses are available and then use the `config set` command on the server:

```
ip addr
lxc config set core.https_address 192.168.1.15
```

Also see *Access to the remote API*.

When I do a `lxc remote add` over https, it asks for a password?

By default, LXD has no password for security reasons, so you can't do a remote add this way. To set a password, enter the following command on the host LXD is running on:

```
lxc config set core.trust_password SECRET
```

This will set the remote password that you can then use to do `lxc remote add`.

You can also access the server without setting a password by copying the client certificate from `.config/lxc/client.crt` to the server and adding it with:

```
lxc config trust add client.crt
```

See *Remote API authentication* for detailed information.

How do I configure LXD storage?

LXD supports btrfs, ceph, directory, lvm and zfs based storage.

First make sure you have the relevant tools for your file system of choice installed on the machine (btrfs-progs, lvm2 or zfsutils-linux).

By default, LXD comes with no configured network or storage. You can get a basic configuration done with:

```
    lxd init
```

`lxd init` supports both directory-based storage and ZFS. If you want something else, you'll need to use the `lxc storage` command:

```
lxc storage create default BACKEND [OPTIONS...]
lxc profile device add default root disk path=/ pool=default
```

BACKEND is one of `btrfs`, `ceph`, `dir`, `lvm` or `zfs`.

Unless specified otherwise, LXD will set up loop-based storage with a sane default size.

For production environments, you should be using block-backed storage instead, both for performance and reliability reasons.

How can I live-migrate a container using LXD?

Live migration requires a tool installed on both hosts called CRIU, which is available in Ubuntu via:

```
sudo apt install criu
```

Then, launch your container with the following:

```
lxc launch ubuntu SOME-NAME
sleep 5s # let the container get to an interesting state
lxc move host1:SOME-NAME host2:SOME-NAME
```

This should migrate your container. Be aware though that migration is still in experimental stages and might not work for all workloads. Please report bugs on lxc-devel, and we can escalate to CRIU lists as necessary.

Can I bind-mount my home directory in a container?

Yes. This can be done using a disk device:

```
lxc config device add container-name home disk source=/home/${USER} path=/home/ubuntu
```

For unprivileged containers, you will also need one of:

- Pass `shift=true` to the `lxc config device add` call. This depends on `shiftfs` being supported (see `lxc info`)
- raw.idmap entry (see *Idmaps for user namespace*)
- Recursive POSIX ACLs placed on your home directory

Either of those can be used to allow the user in the container to have working read/write permissions. When not setting one of those, everything will show up as the overflow UID/GID (65536:65536) and access to anything that's not world readable will fail.

Privileged containers do not have this issue because all UID/GID in the container are the same as outside. But that's also the cause of most of the security issues with such privileged containers.

How can I run Docker inside a LXD container?

To run Docker inside a LXD container, the `security.nesting` property of the container should be set to `true`.

```
lxc config set <container> security.nesting true
```

Note that LXD containers cannot load kernel modules, so depending on your Docker configuration you might need to have the needed extra kernel modules loaded by the host.

You can do so by setting a comma-separated list of kernel modules that your container needs with:

```
lxc config set <container> linux.kernel_modules <modules>
```

We have also received some reports that creating a `/.dockerenv` file in your container can help Docker ignore some errors it's getting due to running in a nested environment.

Container startup issues

If your container is not starting, or not behaving as you would expect, the first thing to do is to look at the console logs generated by the container, using the `lxc console --show-log CONTAINERNAME` command.

In this example, we will investigate a RHEL 7 system in which `systemd` cannot start.

```
# lxc console --show-log systemd
Console log:

Failed to insert module 'autofs4'
Failed to insert module 'unix'
Failed to mount sysfs at /sys: Operation not permitted
Failed to mount proc at /proc: Operation not permitted
[!!!!!!] Failed to mount API filesystems, freezing.
```

The errors here say that `/sys` and `/proc` cannot be mounted - which is correct in an unprivileged container. However, LXD does mount these file systems automatically *if it can*.

The *container requirements* specify that every container must come with an empty `/dev`, `/proc` and `/sys` folder, as well as `/sbin/init` existing. If those folders don't exist, LXD will be unable to mount to them, and systemd will then try to. As this is an unprivileged container, systemd does not have the ability to do this, and it then freezes.

So you can see the environment before anything is changed, you can explicitly change the init in a container using the `raw.lxc` configuration parameter. This is equivalent to setting `init=/bin/bash` on the Linux kernel command line.

```
lxc config set systemd raw.lxc 'lxc.init.cmd = /bin/bash'
```

Here is what it looks like:

```
root@lxc-01:~# lxc config set systemd raw.lxc 'lxc.init.cmd = /bin/bash'
root@lxc-01:~# lxc start systemd
root@lxc-01:~# lxc console --show-log systemd

Console log:

[root@systemd /]#
root@lxc-01:~#
```

Now that the container has started, you can check it and see that things are not running as well as expected.

```
root@lxc-01:~# lxc exec systemd bash
[root@systemd ~]# ls
[root@systemd ~]# mount
mount: failed to read mtab: No such file or directory
[root@systemd ~]# cd /
[root@systemd /]# ls /proc/
sys
[root@systemd /]# exit
```

Because LXD tries to auto-heal, it *did* create some of the folders when it was starting up. Shutting down and restarting the container will fix the problem, but the original cause is still there - the **template does not contain the required files**.

Networking issues

In a larger *Production Environment*, it is common to have multiple VLANs and have LXD clients attached directly to those VLANs. Be aware that if you are using netplan and systemd-networkd, you will encounter some bugs that could cause catastrophic issues.

Do not use systemd-networkd with netplan and bridges based on VLANs

At time of writing (2019-03-05), netplan cannot assign a random MAC address to a bridge attached to a VLAN. It always picks the same MAC address, which causes layer2 issues when you have more than one machine on the same network segment. It also has difficulty creating multiple bridges. Make sure you use `network-manager` instead. An example config is below, with a management address of 10.61.0.25, and VLAN102 being used for client traffic.

```
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    eth0:
      dhcp4: no
      accept-ra: no
      # This is the 'Management Address'
      addresses: [ 10.61.0.25/24 ]
      gateway4: 10.61.0.1
      nameservers:
        addresses: [ 1.1.1.1, 8.8.8.8 ]
    eth1:
      dhcp4: no
      accept-ra: no
      # A bogus IP address is required to ensure the link state is up
      addresses: [ 10.254.254.25/32 ]

  vlans:
    vlan102:
      accept-ra: no
      dhcp4: no
      id: 102
      link: eth1

  bridges:
    br102:
      accept-ra: no
      dhcp4: no
      interfaces: [ "vlan102" ]
      # A bogus IP address is required to ensure the link state is up
      addresses: [ 10.254.102.25/32 ]
      parameters:
        stp: false
```

Things to note

- eth0 is the Management interface, with the default gateway.
- vlan102 uses eth1.
- br102 uses vlan102, and *has a bogus /32 IP address assigned to it*

The other important thing is to set `stp: false`, otherwise the bridge will sit in `learning` state for up to 10 seconds, which is longer than most DHCP requests last. As there is no possibility of cross-connecting and causing loops, this is safe to do.

Beware of port security

Many switches do *not* allow MAC address changes, and will either drop traffic with an incorrect MAC or disable the port totally. If you can ping a LXD instance from the host, but are not able to ping it from a *different* host, this could be the cause. The way to diagnose this is to run a tcpdump on the uplink (in this case, eth1), and you will see either "ARP Who has xx.xx.xx.xx tell yy.yy.yy.yy, with you sending responses but them not getting acknowledged, or ICMP packets going in and out successfully, but never being received by the other host.

Do not run privileged containers unless necessary

A privileged container can do things that affect the entire host - for example, it can use things in `/sys` to reset the network card, which will reset it for **the entire host**, causing network blips. Almost everything can be run in an unprivileged container, or - in cases of things that require unusual privileges, like wanting to mount NFS file systems inside the container - you might need to use bind mounts.

3.1.4 Security

Consider the following aspects to ensure that your LXD installation is secure:

- Keep your operating system up-to-date and install all available security patches.
- Use only supported LXD versions (LTS releases or monthly feature releases).
- Restrict access to the LXD daemon and the remote API.
- Do not use privileged containers unless required. If you use privileged containers, put appropriate security measures in place. See the LXC security page for more information.
- Configure your network interfaces to be secure.

See the following sections for detailed information.

If you discover a security issue, see the LXD security policy for information on how to report the issue.

Supported versions

Never use unsupported LXD versions in a production environment.

LXD has two types of releases:

- Monthly feature releases
- LTS releases

For feature releases, only the latest one is supported, and we usually don't do point releases. Instead, users are expected to wait until the next monthly release.

For LTS releases, we do periodic bugfix releases that include an accumulation of bugfixes from the feature releases. Such bugfix releases do not include new features.

Access to the LXD daemon

LXD is a daemon that can be accessed locally over a UNIX socket or, if configured, remotely over a TLS (Transport Layer Security) socket. Anyone with access to the socket can fully control LXD, which includes the ability to attach host devices and file systems or to tweak the security features for all instances.

Therefore, make sure to restrict the access to the daemon to trusted users.

Local access to the LXD daemon

The LXD daemon runs as root and provides a UNIX socket for local communication. Access control for LXD is based on group membership. The root user and all members of the `lxd` group can interact with the local daemon.

Important: Local access to LXD through the UNIX socket always grants full access to LXD. This includes the ability to attach file system paths or devices to any instance as well as tweak the security features on any instance.

Therefore, you should only give such access to users who you'd trust with root access to your system.

Access to the remote API

By default, access to the daemon is only possible locally. By setting the `core.https_address` configuration option (see *Server configuration*), you can expose the same API over the network on a TLS socket. Remote clients can then connect to LXD and access any image that is marked for public use.

There are several ways to authenticate remote clients as trusted clients to allow them to access the API. See *Remote API authentication* for details.

In a production setup, you should set `core.https_address` to the single address where the server should be available (rather than any address on the host). In addition, you should set firewall rules to allow access to the LXD port only from authorized hosts/subnets.

Container security

LXD containers can use a wide range of features for security.

By default, containers are *unprivileged*, meaning that they operate inside a user namespace, restricting the abilities of users in the container to that of regular users on the host with limited privileges on the devices that the container owns.

If data sharing between containers isn't needed, you can enable `security.idmap.isolated` (see *Instance configuration*), which will use non-overlapping uid/gid maps for each container, preventing potential DoS (Denial of Service) attacks on other containers.

LXD can also run *privileged* containers. Note, however, that those aren't root safe, and a user with root access in such a container will be able to DoS the host as well as find ways to escape confinement.

More details on container security and the kernel features we use can be found on the LXC security page.

Network security

Make sure to configure your network interfaces to be secure. Which aspects you should consider depends on the networking mode you decide to use.

Bridged NIC security

The default networking mode in LXD is to provide a "managed" private network bridge that each instance connects to. In this mode, there is an interface on the host called `lxdbr0` that acts as the bridge for the instances.

The host runs an instance of `dnsmasq` for each managed bridge, which is responsible for allocating IP addresses and providing both authoritative and recursive DNS services.

Instances using DHCPv4 will be allocated an IPv4 address, and a DNS record will be created for their instance name. This prevents instances from being able to spoof DNS records by providing false host name information in the DHCP request.

The `dnsmasq` service also provides IPv6 router advertisement capabilities. This means that instances will auto-configure their own IPv6 address using SLAAC, so no allocation is made by `dnsmasq`. However, instances that are also using DHCPv4 will also get an AAAA DNS record created for the equivalent SLAAC IPv6 address. This assumes that the instances are not using any IPv6 privacy extensions when generating IPv6 addresses.

In this default configuration, whilst DNS names cannot not be spoofed, the instance is connected to an Ethernet bridge and can transmit any layer 2 traffic that it wishes, which means an untrusted instance can effectively do MAC or IP spoofing on the bridge.

In the default configuration, it is also possible for instances connected to the bridge to modify the LXD host's IPv6 routing table by sending (potentially malicious) IPv6 router advertisements to the bridge. This is because the `lxdbr0` interface is created with `/proc/sys/net/ipv6/conf/lxdbr0/accept_ra` set to `2`, meaning that the LXD host will accept router advertisements even though `forwarding` is enabled (see /proc/sys/net/ipv4/* Variables for more information).

However, LXD offers several bridged NIC (Network interface controller) security features that can be used to control the type of traffic that an instance is allowed to send onto the network. These NIC settings should be added to the profile that the instance is using, or they can be added to individual instances, as shown below.

The following security features are available for bridged NICs:

Key	Type	Default	Required	Description
security.mac_filtering	boolean	false	no	Prevent the instance from spoofing another's MAC address
security.ipv4_filtering	boolean	false	no	Prevent the instance from spoofing another's IPv4 address (enables mac_filtering)
security.ipv6_filtering	boolean	false	no	Prevent the instance from spoofing another's IPv6 address (enables mac_filtering)

One can override the default bridged NIC settings from the profile on a per-instance basis using:

```
lxc config device override <instance> <NIC> security.mac_filtering=true
```

Used together, these features can prevent an instance connected to a bridge from spoofing MAC and IP addresses. These options are implemented using either `xtables` (iptables, ip6tables and ebtables) or `nftables`, depending on what is available on the host.

It's worth noting that those options effectively prevent nested containers from using the parent network with a different MAC address (i.e using bridged or macvlan NICs).

The IP filtering features block ARP and NDP advertisements that contain a spoofed IP, as well as blocking any packets that contain a spoofed source address.

If `security.ipv4_filtering` or `security.ipv6_filtering` is enabled and the instance cannot be allocated an IP address (because `ipvX.address=none` or there is no DHCP service enabled on the bridge), then all IP traffic for that protocol is blocked from the instance.

When `security.ipv6_filtering` is enabled, IPv6 router advertisements are blocked from the instance.

When `security.ipv4_filtering` or `security.ipv6_filtering` is enabled, any Ethernet frames that are not ARP, IPv4 or IPv6 are dropped. This prevents stacked VLAN QinQ (802.1ad) frames from bypassing the IP filtering.

Routed NIC security

An alternative networking mode is available called "routed". It provides a veth pair between container and host. In this networking mode, the LXD host functions as a router, and static routes are added to the host directing traffic for the container's IPs towards the container's veth interface.

By default, the veth interface created on the host has its `accept_ra` setting disabled to prevent router advertisements from the container modifying the IPv6 routing table on the LXD host. In addition to that, the `rp_filter` on the host is set to 1 to prevent source address spoofing for IPs that the host does not know the container has.

3.1.5 Contributing

Check the following guidelines before contributing to the project.

Pull requests

Changes to this project should be proposed as pull requests on Github at: https://github.com/canonical/lxd

Proposed changes will then go through code review there and once acked, be merged in the main branch.

Commit structure

Separate commits should be used for:

- API extension (`api: Add XYZ extension`, contains `doc/api-extensions.md` and `shared/version.api.go`)
- Documentation (`doc: Update XYZ` for files in `doc/`)
- API structure (`shared/api: Add XYZ` for changes to `shared/api/`)
- Go client package (`client: Add XYZ` for changes to `client/`)
- CLI (`lxc/<command>: Change XYZ` for changes to `lxc/`)
- Scripts (`scripts: Update bash completion for XYZ` for changes to `scripts/`)
- LXD daemon (`lxd/<package>: Add support for XYZ` for changes to `lxd/`)
- Tests (`tests: Add test for XYZ` for changes to `tests/`)

The same kind of pattern extends to the other tools in the LXD code tree and depending on complexity, things may be split into even smaller chunks.

When updating strings in the CLI tool (`lxc/`), you may need a commit to update the templates:

- make i18n
- git commit -a -s -m "i18n: Update translation templates" po/

This structure makes it easier for contributions to be reviewed and also greatly simplifies the process of backporting fixes to stable branches.

License and copyright

By default, any contribution to this project is made under the Apache 2.0 license.

The author of a change remains the copyright holder of their code (no copyright assignment).

Developer Certificate of Origin

To improve tracking of contributions to this project we use the DCO 1.1 and use a "sign-off" procedure for all changes going into the branch.

The sign-off is a simple line at the end of the explanation for the commit which certifies that you wrote it or otherwise have the right to pass it on as an open-source contribution.

> Developer Certificate of Origin Version 1.1
>
> Copyright (C) 2004, 2006 The Linux Foundation and its contributors. 660 York Street, Suite 102, San Francisco, CA 94110 USA
>
> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.
>
> Developer's Certificate of Origin 1.1

> By making a contribution to this project, I certify that:
>
> (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
>
> (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
>
> (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
>
> (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

An example of a valid sign-off line is:

```
Signed-off-by: Random J Developer <random@developer.org>
```

Use your real name and a valid e-mail address. Sorry, no pseudonyms or anonymous contributions are allowed.

We also require each commit be individually signed-off by their author, even when part of a larger set. You may find `git commit -s` useful.

Code of Conduct

When contributing, you must adhere to the Code of Conduct, which is available at: https://github.com/canonical/lxd/blob/main/CODE_OF_CONDUCT.md

Getting Started Developing

Follow the steps below to set up your development environment to get started working on new features for LXD.

Building Dependencies

To build the dependencies, follow the instructions in *Installing LXD from source*.

Adding Your Fork Remote

After building your dependencies, you can now add your GitHub fork as a remote and switch to it:

```
git remote add myfork git@github.com:<your_username>/lxd.git
git remote update
git checkout myfork/master
```

Building LXD

Finally, you should be able to `make` inside the repository and build your fork of the project.

At this point, you would most likely want to create a new branch for your changes on your fork:

```
git checkout -b [name_of_your_new_branch]
git push myfork [name_of_your_new_branch]
```

Important Notes for New LXD Contributors

- Persistent data is stored in the `LXD_DIR` directory which is generated by `lxd init`. The `LXD_DIR` defaults to `/var/lib/lxd` or `/var/snap/lxd/common/lxd` for snap users.
- As you develop, you may want to change the `LXD_DIR` for your fork of LXD so as to avoid version conflicts.
- Binaries compiled from your source will be generated in the `$(go env GOPATH)/bin` directory by default.
  - You will need to explicitly invoke these binaries (not the global `lxd` you may have installed) when testing your changes.
  - You may choose to create an alias in your `~/.bashrc` to call these binaries with the appropriate flags more conveniently.
- If you have a systemd service configured to run the LXD daemon from a previous installation of LXD, you may want to disable it to avoid version conflicts.

3.1.6 Support

You can find information and ask for user support through the following channels.

Support and community

The following channels are available for you to interact with the LXD community.

Bug reports

You can file bug reports and feature requests at: https://github.com/canonical/lxd/issues/new

Forum

A discussion forum is available at: https://discourse.ubuntu.com/c/lxd/

IRC

If you prefer live discussions, you can find us in `#lxd` on `irc.libera.chat`. See Getting started with IRC if needed.

Commercial support

Commercial support for LXD can be obtained through Canonical Ltd.

Documentation

The official documentation is available at: `https://documentation.ubuntu.com/lxd/en/stable-4.0/`

You can find additional resources on the website, on YouTube and in the Tutorials section in the forum.

3.2 Configuration

LXD stores the configuration for the following components:

3.2.1 Containers

Introduction

Containers are the default type for LXD and currently the most featureful and complete implementation of LXD instances.

They are implemented through the use of `liblxc` (LXC).

Configuration

See *instance configuration* for valid configuration options.

Live migration

LXD supports live migration of containers using CRIU. In order to optimize the memory transfer for a container LXD can be instructed to make use of CRIU's pre-copy features by setting the `migration.incremental.memory` property to `true`. This means LXD will request CRIU to perform a series of memory dumps for the container. After each dump LXD will send the memory dump to the specified remote. In an ideal scenario each memory dump will decrease the delta to the previous memory dump thereby increasing the percentage of memory that is already synced. When the percentage of synced memory is equal to or greater than the threshold specified via `migration.incremental.memory.goal` LXD will request CRIU to perform a final memory dump and transfer it. If the threshold is not reached after the maximum number of allowed iterations specified via `migration.incremental.memory.iterations` LXD will request a final memory dump from CRIU and migrate the container.

3.2.2 Instance configuration

Instances

Properties

The following are direct instance properties and can't be part of a profile:

- `name`
- `architecture`

Name is the instance name and can only be changed by renaming the instance.

Valid instance names must:

- Be between 1 and 63 characters long
- Be made up exclusively of letters, numbers and dashes from the ASCII table
- Not start with a digit or a dash
- Not end with a dash

This requirement is so that the instance name may properly be used in DNS records, on the filesystem, in various security profiles as well as the hostname of the instance itself.

Key/value configuration

The key/value configuration is namespaced with the following namespaces currently supported:

- `boot` (boot related options, timing, dependencies, ...)
- `environment` (environment variables)
- `image` (copy of the image properties at time of creation)
- `limits` (resource limits)
- `nvidia` (NVIDIA and CUDA configuration)
- `raw` (raw instance configuration overrides)
- `security` (security policies)
- `user` (storage for user properties, searchable)
- `volatile` (used internally by LXD to store internal data specific to an instance)

The currently supported keys are:

Key	Type	Default	Live update	Condition	Description
boot.autostart	boolean	-	n/a	-	Always start the ins
boot.autostart.delay	integer	0	n/a	-	Number of seconds
boot.autostart.priority	integer	0	n/a	-	What order to start
boot.host_shutdown_timeout	integer	30	yes	-	Seconds to wait for
boot.stop.priority	integer	0	n/a	-	What order to shut
environment.*	string	-	yes (exec)	-	key/value environm
limits.cpu	string	-	yes	-	Number or range o
limits.cpu.allowance	string	100%	yes	container	How much of the C

Key	Type	Default	Live update	Condition	Description
limits.cpu.priority	integer	10 (maximum)	yes	container	CPU scheduling pr
limits.disk.priority	integer	5 (medium)	yes	-	When under load, h
limits.hugepages.64KB	string	-	yes	container	Fixed value in byte
limits.hugepages.1MB	string	-	yes	container	Fixed value in byte
limits.hugepages.2MB	string	-	yes	container	Fixed value in byte
limits.hugepages.1GB	string	-	yes	container	Fixed value in byte
limits.kernel.*	string	-	no	container	This limits kernel r
limits.memory	string	-	yes	-	Percentage of the h
limits.memory.enforce	string	hard	yes	container	If hard, instance ca
limits.memory.hugepages	boolean	false	no	virtual-machine	Controls whether to
limits.memory.swap	boolean	true	yes	container	Controls whether to
limits.memory.swap.priority	integer	10 (maximum)	yes	container	The higher this is s
limits.network.priority	integer	0 (minimum)	yes	-	When under load, h
limits.processes	integer	- (max)	yes	container	Maximum number
linux.kernel_modules	string	-	yes	container	Comma separated l
migration.incremental.memory	boolean	false	yes	container	Incremental memor
migration.incremental.memory.goal	integer	70	yes	container	Percentage of mem
migration.incremental.memory.iterations	integer	10	yes	container	Maximum number
migration.stateful	boolean	false	no	virtual-machine	Allow for stateful s
nvidia.driver.capabilities	string	compute,utility	no	container	What driver capabi
nvidia.runtime	boolean	false	no	container	Pass the host NVID
nvidia.require.cuda	string	-	no	container	Version expression
nvidia.require.driver	string	-	no	container	Version expression
raw.apparmor	blob	-	yes	-	Apparmor profile e
raw.idmap	blob	-	no	unprivileged container	Raw idmap configu
raw.lxc	blob	-	no	container	Raw LXC configur
raw.qemu	blob	-	no	virtual-machine	Raw Qemu configu
raw.seccomp	blob	-	no	container	Raw Seccomp conf
security.devlxd	boolean	true	no	-	Controls the presen
security.devlxd.images	boolean	false	no	container	Controls the availa
security.idmap.base	integer	-	no	unprivileged container	The base host ID to
security.idmap.isolated	boolean	false	no	unprivileged container	Use an idmap for th
security.idmap.size	integer	-	no	unprivileged container	The size of the idm
security.nesting	boolean	false	yes	container	Support running lx
security.privileged	boolean	false	no	container	Runs the instance i
security.protection.delete	boolean	false	yes	-	Prevents the instanc
security.protection.shift	boolean	false	yes	container	Prevents the instanc
security.agent.metrics	boolean	true	no	virtual-machine	Controls whether th
security.secureboot	boolean	true	no	virtual-machine	Controls whether U
security.syscalls.allow	string	-	no	container	A '\n' separated lis
security.syscalls.deny	string	-	no	container	A '\n' separated lis
security.syscalls.deny_compat	boolean	false	no	container	On x86_64 this ena
security.syscalls.deny_default	boolean	true	no	container	Enables the default
security.syscalls.intercept.mknod	boolean	false	no	container	Handles the `mknod`
security.syscalls.intercept.mount	boolean	false	no	container	Handles the `mount`
security.syscalls.intercept.mount.allowed	string	-	yes	container	Specify a comma-s
security.syscalls.intercept.mount.fuse	string	-	yes	container	Whether to redirect
security.syscalls.intercept.mount.shift	boolean	false	yes	container	Whether to mount s
security.syscalls.intercept.setxattr	boolean	false	no	container	Handles the `setxa`

Key	Type	Default	Live update	Condition	Description
snapshots.schedule	string	-	no	-	Cron expression (<
snapshots.schedule.stopped	bool	false	no	-	Controls whether o
snapshots.pattern	string	snap%d	no	-	Pongo2 template st
snapshots.expiry	string	-	no	-	Controls when snap
user.*	string	-	n/a	-	Free form user key/

The following volatile keys are currently internally used by LXD:

Key	Type	Default	Description
volatile.apply_template	string	-	The name of a template hook which should be triggered upon next startup
volatile.base_image	string	-	The hash of the image the instance was created from, if any
volatile.idmap.base	integer	-	The first id in the instance's primary idmap range
volatile.idmap.current	string	-	The idmap currently in use by the instance
volatile.idmap.next	string	-	The idmap to use next time the instance starts
volatile.last_state.idmap	string	-	Serialized instance uid/gid map
volatile.last_state.power	string	-	Instance state as of last host shutdown
volatile.vsock_id	string	-	Instance vsock ID used as of last start
volatile.uuid	string	-	Instance UUID (globally unique across all servers and projects)
volatile.<name>.apply_quota	string	-	Disk quota to be applied on next instance start
volatile.<name>.ceph_rbd	string	-	RBD device path for Ceph disk devices
volatile.<name>.host_name	string	-	Network device name on the host
volatile.<name>.hwaddr	string	-	Network device MAC address (when no hwaddr property is set on the device itself)
volatile.<name>.last_state.created	string	-	Whether or not the network device physical device was created ("true" or "false")
volatile.<name>.last_state.mtu	string	-	Network device original MTU used when moving a physical device into an instance
volatile.<name>.last_state.hwaddr	string	-	Network device original MAC used when moving a physical device into an instance
volatile.<name>.last_state.vf.id	string	-	SR-IOV Virtual function ID used when moving a VF into an instance
volatile.<name>.last_state.vf.hwaddr	string	-	SR-IOV Virtual function original MAC used when moving a VF into an instance
volatile.<name>.last_state.vf.vlan	string	-	SR-IOV Virtual function original VLAN used when moving a VF into an instance
volatile.<name>.last_state.vf.spoofcheck	string	-	SR-IOV Virtual function original spoof check setting used when moving a VF into an instance

Additionally, those user keys have become common with images (support isn't guaranteed):

Key	Type	Default	Description
user.meta-data	string	-	Cloud-init meta-data, content is appended to seed value
user.network-config	string	DHCP on eth0	Cloud-init network-config, content is used as seed value
user.network_mode	string	dhcp	One of "dhcp" or "link-local". Used to configure network in supported images
user.user-data	string	#!cloud-config	Cloud-init user-data, content is used as seed value
user.vendor-data	string	#!cloud-config	Cloud-init vendor-data, content is used as seed value

Note that while a type is defined above as a convenience, all values are stored as strings and should be exported over the REST API as strings (which makes it possible to support any extra values without breaking backward compatibility).

Those keys can be set using the lxc tool with:

```
lxc config set <instance> <key> <value>
```

Volatile keys can't be set by the user and can only be set directly against an instance.

The raw keys allow direct interaction with the backend features that LXD itself uses, setting those may very well break LXD in non-obvious ways and should whenever possible be avoided.

CPU limits

The CPU limits are implemented through a mix of the `cpuset` and `cpu` CGroup controllers.

`limits.cpu` results in CPU pinning through the `cpuset` controller. A set of CPUs (e.g. `1,2,3`) or a CPU range (e.g. `0-3`) can be specified.

When a number of CPUs is specified instead (e.g. `4`), LXD will do dynamic load-balancing of all instances that aren't pinned to specific CPUs, trying to spread the load on the machine. Instances will then be re-balanced every time an instance starts or stops as well as whenever a CPU is added to the system.

To pin to a single CPU, you have to use the range syntax (e.g. `1-1`) to differentiate it from a number of CPUs.

`limits.cpu.allowance` drives either the CFS scheduler quotas when passed a time constraint, or the generic CPU shares mechanism when passed a percentage value.

The time constraint (e.g. `20ms/50ms`) is relative to one CPU worth of time, so to restrict to two CPUs worth of time, something like 100ms/50ms should be used.

When using a percentage value, the limit will only be applied when under load and will be used to calculate the scheduler priority for the instance, relative to any other instance which is using the same CPU(s).

`limits.cpu.priority` is another knob which is used to compute that scheduler priority score when a number of instances sharing a set of CPUs have the same percentage of CPU assigned to them.

VM CPU topology

LXD virtual machines default to having just one vCPU allocated which shows up as matching the host CPU vendor and type but has a single core and no threads.

When `limits.cpu` is set to a single integer, this will cause multiple vCPUs to be allocated and exposed to the guest as full cores. Those vCPUs will not be pinned to specific physical cores on the host.

When `limits.cpu` is set to a range or comma separate list of CPU IDs (as provided by `lxc info --resources`), then the vCPUs will be pinned to those physical cores. In this scenario, LXD will check whether the CPU configuration lines up with a realistic hardware topology and if it does, it will replicate that topology in the guest.

This means that if the pinning configuration includes 8 threads, with each pair of thread coming from the same core and an even number of cores spread across two CPUs, LXD will have the guest show two CPUs, each with two cores and each core with two threads. The NUMA layout is similarly replicated and in this scenario, the guest would most likely end up with two NUMA nodes, one for each CPU socket.

In such an environment with multiple NUMA nodes, the memory will similarly be divided across NUMA nodes and be pinned accordingly on the host and then exposed to the guest.

All this allows for very high performance operations in the guest as the guest scheduler can properly reason about sockets, cores and threads as well as consider NUMA topology when sharing memory or moving processes across NUMA nodes.

Devices configuration

LXD will always provide the instance with the basic devices which are required for a standard POSIX system to work. These aren't visible in instance or profile configuration and may not be overridden.

Those include:

- `/dev/null` (character device)
- `/dev/zero` (character device)
- `/dev/full` (character device)
- `/dev/console` (character device)
- `/dev/tty` (character device)
- `/dev/random` (character device)
- `/dev/urandom` (character device)
- `/dev/net/tun` (character device)
- `/dev/fuse` (character device)
- `lo` (network interface)

Anything else has to be defined in the instance configuration or in one of its profiles. The default profile will typically contain a network interface to become `eth0` in the instance.

To add extra devices to an instance, device entries can be added directly to an instance, or to a profile.

Devices may be added or removed while the instance is running.

Every device entry is identified by a unique name. If the same name is used in a subsequent profile or in the instance's own configuration, the whole entry is overridden by the new definition.

Device entries are added to an instance through:

```
lxc config device add <instance> <name> <type> [key=value]...
```

or to a profile with:

```
lxc profile device add <profile> <name> <type> [key=value]...
```

Device types

LXD supports the following device types:

ID (database)	Name	Condition	Description
0	*none*	-	Inheritance blocker
1	*nic*	-	Network interface
2	*disk*	-	Mountpoint inside the instance
3	*unix-char*	container	Unix character device
4	*unix-block*	container	Unix block device
5	*usb*	-	USB device
6	*gpu*	-	GPU device
7	*infiniband*	container	Infiniband device
8	*proxy*	container	Proxy device
9	*unix-hotplug*	container	Unix hotplug device

Type: none

Supported instance types: container, VM

A none type device doesn't have any property and doesn't create anything inside the instance.

It's only purpose it to stop inheritance of devices coming from profiles.

To do so, just add a none type device with the same name of the one you wish to skip inheriting. It can be added in a profile being applied after the profile it originated from or directly on the instance.

Type: nic

LXD supports several different kinds of network devices (referred to as Network Interface Controller or NIC).

When adding a network device to an instance, there are two ways to specify the type of device you want to add; either by specifying the `nictype` property or using the `network` property.

Specifying a NIC using the `network` property

When specifying the `network` property, the NIC is linked to an existing managed network and the `nictype` is automatically detected based on the network's type.

Some of the NICs properties are inherited from the network rather than being customisable for each NIC.

These are detailed in the "Managed" column in the NIC specific sections below.

NICs Available:

See the NIC's settings below for details about which properties are available.

The following NICs can be specified using the `nictype` or `network` properties:

- *bridged*: Uses an existing bridge on the host and creates a virtual device pair to connect the host bridge to the instance.

The following NICs can be specified using only the `nictype` property:

- *macvlan*: Sets up a new network device based on an existing one but using a different MAC address.
- *sriov*: Passes a virtual function of an SR-IOV enabled physical network device into the instance.
- *physical*: Straight physical device passthrough from the host. The targeted device will vanish from the host and appear in the instance.
- *ipvlan*: Sets up a new network device based on an existing one using the same MAC address but a different IP.
- *p2p*: Creates a virtual device pair, putting one side in the instance and leaving the other side on the host.
- *routed*: Creates a virtual device pair to connect the host to the instance and sets up static routes and proxy ARP/NDP entries to allow the instance to join the network of a designated parent interface.

nic: bridged

Supported instance types: container, VM

Selected using: `nictype`, `network`

Uses an existing bridge on the host and creates a virtual device pair to connect the host bridge to the instance.

Device configuration properties:

Key	Type	Default	Required	Managed	Description
parent	string	-	yes	yes	The name of the host device
network	string	-	yes	no	The LXD network to link device to (instead of parent)
name	string	kernel assigned	no	no	The name of the interface inside the instance
mtu	integer	parent MTU	no	yes	The MTU of the new interface
hwaddr	string	randomly assigned	no	no	The MAC address of the new interface
host_name	string	randomly assigned	no	no	The name of the interface inside the host
limits.ingress	string	-	no	no	I/O limit in bit/s for incoming traffic (various suffixes supported, see below)
limits.egress	string	-	no	no	I/O limit in bit/s for outgoing traffic (various suffixes supported, see below)
limits.max	string	-	no	no	Same as modifying both limits.ingress and limits.egress
ipv4.address	string	-	no	no	An IPv4 address to assign to the instance through DHCP
ipv6.address	string	-	no	no	An IPv6 address to assign to the instance through DHCP
ipv4.routes	string	-	no	no	Comma delimited list of IPv4 static routes to add on host to NIC
ipv6.routes	string	-	no	no	Comma delimited list of IPv6 static routes to add on host to NIC
security.mac_filtering	boolean	false	no	no	Prevent the instance from spoofing another's MAC address
security.ipv4_filtering	boolean	false	no	no	Prevent the instance from spoofing another's IPv4 address (enables mac_filtering)
security.ipv6_filtering	boolean	false	no	no	Prevent the instance from spoofing another's IPv6 address (enables mac_filtering)
maas.subnet.ipv4	string	-	no	yes	MAAS IPv4 subnet to register the instance in
maas.subnet.ipv6	string	-	no	yes	MAAS IPv6 subnet to register the instance in
boot.priority	integer	-	no	no	Boot priority for VMs (higher boots first)

nic: macvlan

Supported instance types: container, VM

Selected using: `nictype`

Sets up a new network device based on an existing one but using a different MAC address.

Device configuration properties:

Key	Type	Default	Required	Description
parent	string	-	yes	The name of the host device
name	string	kernel assigned	no	The name of the interface inside the instance
mtu	integer	parent MTU	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
vlan	integer	-	no	The VLAN ID to attach to
maas.subnet.ipv4	string	-	no	MAAS IPv4 subnet to register the instance in
maas.subnet.ipv6	string	-	no	MAAS IPv6 subnet to register the instance in
boot.priority	integer	-	no	Boot priority for VMs (higher boots first)

nic: sriov

Supported instance types: container, VM

Selected using: `nictype`

Passes a virtual function of an SR-IOV enabled physical network device into the instance.

Device configuration properties:

Key	Type	Default	Required	Description
parent	string	-	yes	The name of the host device
name	string	kernel assigned	no	The name of the interface inside the instance
mtu	integer	kernel assigned	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
security.mac_filtering	boolean	false	no	Prevent the instance from spoofing another's MAC address
vlan	integer	-	no	The VLAN ID to attach to
maas.subnet.ipv4	string	-	no	MAAS IPv4 subnet to register the instance in
maas.subnet.ipv6	string	-	no	MAAS IPv6 subnet to register the instance in
boot.priority	integer	-	no	Boot priority for VMs (higher boots first)

nic: physical

Supported instance types: container, VM

Selected using: `nictype`

Straight physical device passthrough from the host. The targeted device will vanish from the host and appear in the instance.

Device configuration properties:

Key	Type	Default	Required	Description
parent	string	-	yes	The name of the host device
name	string	kernel assigned	no	The name of the interface inside the instance
mtu	integer	parent MTU	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
vlan	integer	-	no	The VLAN ID to attach to
maas.subnet.ipv4	string	-	no	MAAS IPv4 subnet to register the instance in
maas.subnet.ipv6	string	-	no	MAAS IPv6 subnet to register the instance in
boot.priority	integer	-	no	Boot priority for VMs (higher boots first)

nic: ipvlan

Supported instance types: container

Selected using: `nictype`

Sets up a new network device based on an existing one using the same MAC address but a different IP.

LXD currently supports IPVLAN in L2 and L3S mode.

In this mode, the gateway is automatically set by LXD, however IP addresses must be manually specified using either one or both of `ipv4.address` and `ipv6.address` settings before instance is started.

For DNS, the nameservers need to be configured inside the instance, as these will not automatically be set.

It requires the following sysctls to be set:

If using IPv4 addresses:

```
net.ipv4.conf.<parent>.forwarding=1
```

If using IPv6 addresses:

```
net.ipv6.conf.<parent>.forwarding=1
net.ipv6.conf.<parent>.proxy_ndp=1
```

Device configuration properties:

Key	Type	Default	Required	Description
parent	string	-	yes	The name of the host device
name	string	kernel assigned	no	The name of the interface inside the instance
mtu	integer	parent MTU	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
ipv4.address	string	-	no	Comma delimited list of IPv4 static addresses to add to the instance
ipv4.gateway	string	auto	no	Whether to add an automatic default IPv4 gateway, can be "auto" or "none"
ipv6.address	string	-	no	Comma delimited list of IPv6 static addresses to add to the instance
ipv6.gateway	string	auto	no	Whether to add an automatic default IPv6 gateway, can be "auto" or "none"
vlan	integer	-	no	The VLAN ID to attach to

nic: p2p

Supported instance types: container, VM

Selected using: `nictype`

Creates a virtual device pair, putting one side in the instance and leaving the other side on the host.

Device configuration properties:

Key	Type	Default	Required	Description
name	string	kernel assigned	no	The name of the interface inside the instance
mtu	integer	kernel assigned	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
host_name	string	randomly assigned	no	The name of the interface inside the host
limits.ingress	string	-	no	I/O limit in bit/s for incoming traffic (various suffixes supported, see below)
limits.egress	string	-	no	I/O limit in bit/s for outgoing traffic (various suffixes supported, see below)
limits.max	string	-	no	Same as modifying both limits.ingress and limits.egress
ipv4.routes	string	-	no	Comma delimited list of IPv4 static routes to add on host to NIC
ipv6.routes	string	-	no	Comma delimited list of IPv6 static routes to add on host to NIC
boot.priority	integer	-	no	Boot priority for VMs (higher boots first)

nic: routed

Supported instance types: container, VM

Selected using: `nictype`

This NIC type is similar in operation to IPVLAN, in that it allows an instance to join an external network without needing to configure a bridge and shares the host's MAC address.

However it differs from IPVLAN because it does not need IPVLAN support in the kernel and the host and instance can communicate with each other.

It will also respect netfilter rules on the host and will use the host's routing table to route packets which can be useful if the host is connected to multiple networks.

IP addresses must be manually specified using either one or both of `ipv4.address` and `ipv6.address` settings before the instance is started.

For containers it uses a veth pair, and for VMs it uses a TAP device. It then configures the following link-local gateway IPs on the host end which are then set as the default gateways in the instance:

169.254.0.1 fe80::1

For containers these are automatically set as default gateways on the instance NIC interface. But for VMs the IP addresses and gateways will need to be configured manually or via a mechanism like cloud-init.

Note also that if your container image is configured to perform DHCP on the interface it will likely remove the automatically added configuration, and will need to be configured manually or via a mechanism like cloud-init.

It then configures static routes on the host pointing to the instance's veth interface for all of the instance's IPs.

This nic can operate with and without a `parent` network interface set.

With the `parent` network interface set proxy ARP/NDP entries of the instance's IPs are added to the parent interface allowing the instance to join the parent interface's network at layer 2.

For DNS, the nameservers need to be configured inside the instance, as these will not automatically be set.

It requires the following sysctls to be set:

If using IPv4 addresses:

```
net.ipv4.conf.<parent>.forwarding=1
```

If using IPv6 addresses:

```
net.ipv6.conf.all.forwarding=1
net.ipv6.conf.<parent>.forwarding=1
net.ipv6.conf.all.proxy_ndp=1
net.ipv6.conf.<parent>.proxy_ndp=1
```

Each NIC device can have multiple IP addresses added to them. However it may be desirable to utilise multiple `routed` NIC interfaces. In these cases one should set the `ipv4.gateway` and `ipv6.gateway` values to "none" on any subsequent interfaces to avoid default gateway conflicts. It may also be useful to specify a different host-side address for these subsequent interfaces using `ipv4.host_address` and `ipv6.host_address` respectively.

Device configuration properties:

Key	Type	Default	Required	Description
parent	string	-	no	The name of the host device to join the instance to
name	string	kernel assigned	no	The name of the interface inside the instance
host_name	string	randomly assigned	no	The name of the interface inside the host
mtu	integer	parent MTU	no	The MTU of the new interface
hwaddr	string	randomly assigned	no	The MAC address of the new interface
limits.ingress	string	-	no	I/O limit in bit/s for incoming traffic (various suffixes supported, see below)
limits.egress	string	-	no	I/O limit in bit/s for outgoing traffic (various suffixes supported, see below)
limits.max	string	-	no	Same as modifying both limits.ingress and limits.egress
ipv4.address	string	-	no	Comma delimited list of IPv4 static addresses to add to the instance
ipv4.gateway	string	auto	no	Whether to add an automatic default IPv4 gateway, can be "auto" or "none"
ipv4.host_address	string	169.254.0.1	no	The IPv4 address to add to the host-side veth interface
ipv4.host_table	integer	-	no	The custom policy routing table ID to add IPv4 static routes to (in addition to main routing table)
ipv6.address	string	-	no	Comma delimited list of IPv6 static addresses to add to the instance
ipv6.gateway	string	auto	no	Whether to add an automatic default IPv6 gateway, can be "auto" or "none"
ipv6.host_address	string	fe80::1	no	The IPv6 address to add to the host-side veth interface
ipv6.host_table	integer	-	no	The custom policy routing table ID to add IPv6 static routes to (in addition to main routing table)
vlan	integer	-	no	The VLAN ID to attach to

bridged, macvlan or ipvlan for connection to physical network

The `bridged`, `macvlan` and `ipvlan` interface types can be used to connect to an existing physical network.

`macvlan` effectively lets you fork your physical NIC, getting a second interface that's then used by the instance. This saves you from creating a bridge device and veth pairs and usually offers better performance than a bridge.

The downside to this is that macvlan devices while able to communicate between themselves and to the outside, aren't able to talk to their parent device. This means that you can't use macvlan if you ever need your instances to talk to the host itself.

In such case, a bridge is preferable. A bridge will also let you use mac filtering and I/O limits which cannot be applied to a macvlan device.

`ipvlan` is similar to `macvlan`, with the difference being that the forked device has IPs statically assigned to it and inherits the parent's MAC address on the network.

SR-IOV

The `sriov` interface type supports SR-IOV enabled network devices. These devices associate a set of virtual functions (VFs) with the single physical function (PF) of the network device. PFs are standard PCIe functions. VFs on the other hand are very lightweight PCIe functions that are optimized for data movement. They come with a limited set of configuration capabilities to prevent changing properties of the PF. Given that VFs appear as regular PCIe devices to the system they can be passed to instances just like a regular physical device. The `sriov` interface type expects to be passed the name of an SR-IOV enabled network device on the system via the `parent` property. LXD will then check for any available VFs on the system. By default LXD will allocate the first free VF it finds. If it detects that either none are enabled or all currently enabled VFs are in use it will bump the number of supported VFs to the maximum value and use the first free VF. If all possible VFs are in use or the kernel or card doesn't support incrementing the number of VFs LXD will return an error.

To create a `sriov` network device use:

```
lxc config device add <instance> <device-name> nic nictype=sriov parent=<sriov-enabled-
↪device>
```

To tell LXD to use a specific unused VF add the `host_name` property and pass it the name of the enabled VF.

MAAS integration

If you're using MAAS to manage the physical network under your LXD host and want to attach your instances directly to a MAAS managed network, LXD can be configured to interact with MAAS so that it can track your instances.

At the daemon level, you must configure `maas.api.url` and `maas.api.key`, then set the `maas.subnet.ipv4` and/or `maas.subnet.ipv6` keys on the instance or profile's `nic` entry.

This will have LXD register all your instances with MAAS, giving them proper DHCP leases and DNS records.

If you set the `ipv4.address` or `ipv6.address` keys on the nic, then those will be registered as static assignments in MAAS too.

Type: infiniband

Supported instance types: container

LXD supports two different kind of network types for infiniband devices:

- `physical`: Straight physical device passthrough from the host. The targeted device will vanish from the host and appear in the instance.
- `sriov`: Passes a virtual function of an SR-IOV enabled physical network device into the instance.

Different network interface types have different additional properties, the current list is:

Key	Type	Default	Required	Used by	Description
nictype	string	-	yes	all	The device type, one of "physical", or "sriov"
name	string	kernel assigned	no	all	The name of the interface inside the instance
hwaddr	string	randomly assigned	no	all	The MAC address of the new interface. Can be either full 20 byte variant or short 8 byte variant (which will only modify the last 8 bytes of the parent device)
mtu	integer	parent MTU	no	all	The MTU of the new interface
parent	string	-	yes	physical, sriov	The name of the host device or bridge

To create a `physical infiniband` device use:

```
lxc config device add <instance> <device-name> infiniband nictype=physical parent=
↪<device>
```

SR-IOV with infiniband devices

Infiniband devices do support SR-IOV but in contrast to other SR-IOV enabled devices infiniband does not support dynamic device creation in SR-IOV mode. This means users need to pre-configure the number of virtual functions by configuring the corresponding kernel module.

To create a `sriov infiniband` device use:

```
lxc config device add <instance> <device-name> infiniband nictype=sriov parent=<sriov-
↪enabled-device>
```

Type: disk

Supported instance types: container, VM

Disk entries are essentially mountpoints inside the instance. They can either be a bind-mount of an existing file or directory on the host, or if the source is a block device, a regular mount.

LXD supports the following additional source types:

- Ceph-rbd: Mount from existing ceph RBD device that is externally managed. LXD can use ceph to manage an internal file system for the instance, but in the event that a user has a previously existing ceph RBD that they would like use for this instance, they can use this command. Example command

```
lxc config device add <instance> ceph-rbd1 disk source=ceph:<my_pool>/<my-volume> ceph.
↪user_name=<username> ceph.cluster_name=<username> path=/ceph
```

- Ceph-fs: Mount from existing ceph FS device that is externally managed. LXD can use ceph to manage an internal file system for the instance, but in the event that a user has a previously existing ceph file sys that they would like use for this instancer, they can use this command. Example command.

```
lxc config device add <instance> ceph-fs1 disk source=cephfs:<my-fs>/<some-path> ceph.
↪user_name=<username> ceph.cluster_name=<username> path=/cephfs
```

- VM cloud-init: Generate a cloud-init config ISO from the user.vendor-data, user.user-data and user.meta-data config keys and attach to the VM so that cloud-init running inside the VM guest will detect the drive on boot and apply the config. Only applicable to virtual-machine instances. Example command.

```
lxc config device add <instance> config disk source=cloud-init:config
```

Currently only the root disk (path=/) and config drive (source=cloud-init:config) are supported with virtual machines.

The following properties exist:

Key	Type	Default	Required	Description
limits.read	string	-	no	I/O limit in byte/s (various suffixes supported, see below) or in iops (must be suffixed with "iops")
limits.write	string	-	no	I/O limit in byte/s (various suffixes supported, see below) or in iops (must be suffixed with "iops")
limits.max	string	-	no	Same as modifying both limits.read and limits.write
path	string	-	yes	Path inside the instance where the disk will be mounted (only for containers).
source	string	-	yes	Path on the host, either to a file/directory or to a block device
required	boolean	true	no	Controls whether to fail if the source doesn't exist
readonly	boolean	false	no	Controls whether to make the mount read-only
size	string	-	no	Disk size in bytes (various suffixes supported, see below). This is only supported for the rootfs (/)
size.state	string	-	no	Same as size above but applies to the filesystem volume used for saving runtime state in virtual machines.
recursive	boolean	false	no	Whether or not to recursively mount the source path
pool	string	-	no	The storage pool the disk device belongs to. This is only applicable for storage volumes managed by LXD
propagation	string	-	no	Controls how a bind-mount is shared between the instance and the host. (Can be one of `private`, the default, or `shared`, `slave`, `unbindable`, `rshared`, `rslave`, `runbindable`, `rprivate`. Please see the Linux Kernel shared subtree documentation for a full explanation)
shift	boolean	false	no	Setup a shifting overlay to translate the source uid/gid to match the instance (only for containers)
raw.mount.options	string	-	no	Filesystem specific mount options
ceph.user_name	string	admin	no	If source is ceph or cephfs then ceph user_name must be specified by user for proper mount
ceph.cluster_name	string	ceph	no	If source is ceph or cephfs then ceph cluster_name must be specified by user for proper mount
boot.priority	integer	-	no	Boot priority for VMs (higher boots first)

Type: unix-char

Supported instance types: container

Unix character device entries simply make the requested character device appear in the instance's `/dev` and allow read/write operations to it.

The following properties exist:

Key	Type	Default	Required	Description
source	string	-	no	Path on the host
path	string	-	no	Path inside the instance (one of "source" and "path" must be set)
major	int	device on host	no	Device major number
minor	int	device on host	no	Device minor number
uid	int	0	no	UID of the device owner in the instance
gid	int	0	no	GID of the device owner in the instance
mode	int	0660	no	Mode of the device in the instance
required	boolean	true	no	Whether or not this device is required to start the instance

Type: unix-block

Supported instance types: container

Unix block device entries simply make the requested block device appear in the instance's `/dev` and allow read/write operations to it.

The following properties exist:

Key	Type	Default	Required	Description
source	string	-	no	Path on the host
path	string	-	no	Path inside the instance (one of "source" and "path" must be set)
major	int	device on host	no	Device major number
minor	int	device on host	no	Device minor number
uid	int	0	no	UID of the device owner in the instance
gid	int	0	no	GID of the device owner in the instance
mode	int	0660	no	Mode of the device in the instance
required	boolean	true	no	Whether or not this device is required to start the instance

Type: usb

Supported instance types: container, VM

USB device entries simply make the requested USB device appear in the instance.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the USB device
productid	string	-	no	The product id of the USB device
uid	int	0	no	UID of the device owner in the instance
gid	int	0	no	GID of the device owner in the instance
mode	int	0660	no	Mode of the device in the instance
required	boolean	false	no	Whether or not this device is required to start the instance. (The default is false, and all devices are hot-pluggable)

Type: gpu

GPU device entries simply make the requested gpu device appear in the instance.

Note: Container devices may match multiple GPUs at once. However, for virtual machines a device can only match a single GPU.

GPUs Available:

The following GPUs can be specified using the `gputype` property:

- *physical* Passes through an entire GPU. This is the default if `gputype` is unspecified.
- *mdev* Creates and passes through a virtual GPU into the instance.
- *mig* Creates and passes through a MIG (Multi-Instance GPU) device into the instance.
- *sriov* Passes a virtual function of an SR-IOV enabled GPU into the instance.

gpu: physical

Supported instance types: container, VM

Passes through an entire GPU.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the GPU device
productid	string	-	no	The product id of the GPU device
id	string	-	no	The card id of the GPU device
pci	string	-	no	The pci address of the GPU device
uid	int	0	no	UID of the device owner in the instance (container only)
gid	int	0	no	GID of the device owner in the instance (container only)
mode	int	0660	no	Mode of the device in the instance (container only)

gpu: mdev

Supported instance types: VM

Creates and passes through a virtual GPU into the instance. A list of available mdev profiles can be found by running `lxc info --resources`.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the GPU device
productid	string	-	no	The product id of the GPU device
id	string	-	no	The card id of the GPU device
pci	string	-	no	The pci address of the GPU device
mdev	string	-	yes	The mdev profile to use (e.g. i915-GVTg_V5_4)

gpu: mig

Supported instance types: container

Creates and passes through a MIG compute instance. This currently requires NVIDIA MIG instances to be pre-created.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the GPU device
productid	string	-	no	The product id of the GPU device
id	string	-	no	The card id of the GPU device
pci	string	-	no	The pci address of the GPU device
mig.ci	int	-	no	Existing MIG compute instance ID
mig.gi	int	-	no	Existing MIG GPU instance ID
mig.uuid	string	-	no	Existing MIG device UUID (“MIG-” prefix can be omitted)

Note: Either “mig.uuid” (Nvidia drivers 470+) or both “mig.ci” and “mig.gi” (old Nvidia drivers) must be set.

gpu: sriov

Supported instance types: VM

Passes a virtual function of an SR-IOV enabled GPU into the instance.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the parent GPU device
productid	string	-	no	The product id of the parent GPU device
id	string	-	no	The card id of the parent GPU device
pci	string	-	no	The pci address of the parent GPU device

Type: proxy

Supported instance types: container (`nat` and non-`nat` modes), VM (`nat` mode only)

Proxy devices allow forwarding network connections between host and instance. This makes it possible to forward traffic hitting one of the host's addresses to an address inside the instance or to do the reverse and have an address in the instance connect through the host.

The supported connection types are:

- `tcp <-> tcp`
- `udp <-> udp`
- `unix <-> unix`
- `tcp <-> unix`
- `unix <-> tcp`
- `udp <-> tcp`
- `tcp <-> udp`
- `udp <-> unix`
- `unix <-> udp`

The proxy device also supports a `nat` mode where packets are forwarded using NAT rather than being proxied through a separate connection. This has benefit that the client address is maintained without the need for the target destination to support the `PROXY` protocol (which is the only way to pass the client address through when using the proxy device in non-nat mode).

When configuring a proxy device with `nat=true`, you will need to ensure that the target instance has a static IP configured in LXD on its NIC device. E.g.

```
lxc config device set <instance> <nic> ipv4.address=<ipv4.address> ipv6.address=<ipv6.
↪address>
```

In order to define a static IPv6 address, the parent managed network needs to have `ipv6.dhcp.stateful` enabled.

In NAT mode the supported connection types are:

- `tcp <-> tcp`
- `udp <-> udp`

When defining IPv6 addresses use square bracket notation, e.g.

```
connect=tcp:[2001:db8::1]:80
```

You can specify that the connect address should be the IP of the instance by setting the connect IP to the wildcard address (`0.0.0.0` for IPv4 and `[::]` for IPv6).

The listen address can also use wildcard addresses when using non-NAT mode. However when using `nat` mode you must specify an IP address on the LXD host.

Key	Type	Default	Required	Description
listen	string	-	yes	The address and port to bind and listen (<type>:<addr>:<port>[-<port>][,<port>])
connect	string	-	yes	The address and port to connect to (<type>:<addr>:<port>[-<port>][,<port>])
bind	string	host	no	Which side to bind on (host/instance)
uid	int	0	no	UID of the owner of the listening Unix socket
gid	int	0	no	GID of the owner of the listening Unix socket
mode	int	0644	no	Mode for the listening Unix socket
nat	bool	false	no	Whether to optimize proxying via NAT (requires instance NIC has static IP address)
proxy_protocol	bool	false	no	Whether to use the HAProxy PROXY protocol to transmit sender information
security.uid	int	0	no	What UID to drop privilege to
security.gid	int	0	no	What GID to drop privilege to

```
lxc config device add <instance> <device-name> proxy listen=<type>:<addr>:<port>[-<port>
↪][,<port>] connect=<type>:<addr>:<port> bind=<host/instance>
```

Type: unix-hotplug

Supported instance types: container

Unix hotplug device entries make the requested unix device appear in the instance's `/dev` and allow read/write operations to it if the device exists on the host system. Implementation depends on systemd-udev to be run on the host.

The following properties exist:

Key	Type	Default	Required	Description
vendorid	string	-	no	The vendor id of the unix device
productid	string	-	no	The product id of the unix device
uid	int	0	no	UID of the device owner in the instance
gid	int	0	no	GID of the device owner in the instance
mode	int	0660	no	Mode of the device in the instance
required	boolean	false	no	Whether or not this device is required to start the instance. (The default is false, and all devices are hot-pluggable)

Units for storage and network limits

Any value representing bytes or bits can make use of a number of useful suffixes to make it easier to understand what a particular limit is.

Both decimal and binary (kibi) units are supported with the latter mostly making sense for storage limits.

The full list of bit suffixes currently supported is:

- bit (1)
- kbit (1000)
- Mbit (1000^2)
- Gbit (1000^3)
- Tbit (1000^4)
- Pbit (1000^5)
- Ebit (1000^6)
- Kibit (1024)
- Mibit (1024^2)
- Gibit (1024^3)
- Tibit (1024^4)
- Pibit (1024^5)
- Eibit (1024^6)

The full list of byte suffixes currently supported is:

- B or bytes (1)
- kB (1000)
- MB (1000^2)
- GB (1000^3)
- TB (1000^4)
- PB (1000^5)
- EB (1000^6)
- KiB (1024)
- MiB (1024^2)
- GiB (1024^3)
- TiB (1024^4)
- PiB (1024^5)
- EiB (1024^6)

Instance types

LXD supports simple instance types. Those are represented as a string which can be passed at instance creation time.

There are three allowed syntaxes:

- `<instance type>`
- `<cloud>:<instance type>`
- `c<CPU>-m<RAM in GB>`

For example, those 3 are equivalent:

- t2.micro
- aws:t2.micro
- c1-m1

On the command line, this is passed like this:

```
lxc launch ubuntu:20.04 my-instance -t t2.micro
```

The list of supported clouds and instance types can be found here:

https://github.com/dustinkirkland/instance-type

Hugepage limits via `limits.hugepages.[size]`

LXD allows to limit the number of hugepages available to a container through the `limits.hugepage.[size]` key. Limiting hugepages is done through the hugetlb cgroup controller. This means the host system needs to expose the hugetlb controller in the legacy or unified cgroup hierarchy for these limits to apply. Note that architectures often expose multiple hugepage sizes. In addition, architectures may expose different hugepage sizes than other architectures.

Limiting hugepages is especially useful when LXD is configured to intercept the mount syscall for the `hugetlbfs` filesystem in unprivileged containers. When LXD intercepts a `hugetlbfs` mount syscall, it will mount the `hugetlbfs` filesystem for a container with correct `uid` and `gid` values as mount options. This makes it possible to use hugepages from unprivileged containers. However, it is recommended to limit the number of hugepages available to the container through `limits.hugepages.[size]` to stop the container from being able to exhaust the hugepages available to the host.

Resource limits via `limits.kernel.[limit name]`

LXD exposes a generic namespaced key `limits.kernel.*` which can be used to set resource limits for a given instance. It is generic in the sense that LXD will not perform any validation on the resource that is specified following the `limits.kernel.*` prefix. LXD cannot know about all the possible resources that a given kernel supports. Instead, LXD will simply pass down the corresponding resource key after the `limits.kernel.*` prefix and its value to the kernel. The kernel will do the appropriate validation. This allows users to specify any supported limit on their system. Some common limits are:

Key	Resource	Description
limits.kernel.as	RLIMIT_AS	Maximum size of the process's virtual memory
limits.kernel.core	RLIMIT_CORE	Maximum size of the process's coredump file
limits.kernel.cpu	RLIMIT_CPU	Limit in seconds on the amount of cpu time the process can consume
limits.kernel.data	RLIMIT_DATA	Maximum size of the process's data segment
limits.kernel.fsize	RLIMIT_FSIZE	Maximum size of files the process may create
limits.kernel.locks	RLIMIT_LOCKS	Limit on the number of file locks that this process may establish
limits.kernel.memlock	RLIMIT_MEMLOCK	Limit on the number of bytes of memory that the process may lock in RAM
limits.kernel.nice	RLIMIT_NICE	Maximum value to which the process's nice value can be raised
limits.kernel.nofile	RLIMIT_NOFILE	Maximum number of open files for the process
limits.kernel.nproc	RLIMIT_NPROC	Maximum number of processes that can be created for the user of the calling process
limits.kernel.rtprio	RLIMIT_RTPRIO	Maximum value on the real-time-priority that maybe set for this process
limits.kernel.sigpending	RLIMIT_SIGPENDING	Maximum number of signals that maybe queued for the user of the calling process

A full list of all available limits can be found in the manpages for the `getrlimit(2)/setrlimit(2)` system calls. To specify a limit within the `limits.kernel.*` namespace use the resource name in lowercase without the `RLIMIT_` prefix, e.g. `RLIMIT_NOFILE` should be specified as `nofile`. A limit is specified as two colon separated values which are either numeric or the word `unlimited` (e.g. `limits.kernel.nofile=1000:2000`). A single value can be used as a shortcut to set both soft and hard limit (e.g. `limits.kernel.nofile=3000`) to the same value. A resource with no explicitly configured limitation will be inherited from the process starting up the instance. Note that this inheritance is not enforced by LXD but by the kernel.

Snapshot scheduling and configuration

LXD supports scheduled snapshots which can be created at most once every minute. There are three configuration options:

- `snapshots.schedule` takes a shortened cron expression: `<minute> <hour> <day-of-month> <month> <day-of-week>`. If this is empty (default), no snapshots will be created.
- `snapshots.schedule.stopped` controls whether or not stopped instance are to be automatically snapshotted. It defaults to `false`.
- `snapshots.pattern` takes a pongo2 template string to format the snapshot name. To name snapshots with time stamps, the pongo2 context variable `creation_date` can be used. Be aware that you should format the date (e.g. use `{{ creation_date|date:"2006-01-02_15-04-05" }}`) in your template string to avoid forbidden characters in the snapshot name. Another way to avoid name collisions is to use the placeholder `%d`. If a snapshot with the same name (excluding the placeholder) already exists, all existing snapshot names will be taken into account to find the highest number at the placeholders position. This number will be incremented by one for the new name. The starting number if no snapshot exists will be `0`. The default behavior of `snapshots.pattern` is equivalent to a format string of `snap%d`.

Example of using pongo2 syntax to format snapshot names with timestamps:

```
lxc config set INSTANCE snapshots.pattern "{{ creation_date|date:'2006-01-02_15-04-05' }}
↪"
```

This results in snapshots named `{date/time of creation}` down to the precision of a second.

3.2.3 Network configuration

LXD supports the following network types:

- *bridge*: Creates an L2 bridge for connecting instances to (can provide local DHCP and DNS).

The configuration keys are namespaced with the following namespaces currently supported for all network types:

- `maas` (MAAS network identification)
- `user` (free form key/value for user metadata)

network: bridge

As one of the possible network configuration types under LXD, LXD supports creating and managing network bridges. LXD bridges can leverage underlying native Linux bridges and Open vSwitch.

Creation and management of LXD bridges is performed via the `lxc network` command. A bridge created by LXD is by default "managed" which means that LXD also will additionally set up a local `dnsmasq` DHCP server and if desired also perform NAT for the bridge (this is the default.)

When a bridge is managed by LXD, configuration values under the `bridge` namespace can be used to configure it.

Additionally, LXD can utilize a pre-existing Linux bridge. In this case, the bridge does not need to be created via `lxc network` and can simply be referenced in an instance or profile device configuration as follows:

```
devices:
  eth0:
     name: eth0
     nictype: bridged
     parent: br0
     type: nic
```

Network configuration properties:

A complete list of configuration settings for LXD networks can be found below.

The following configuration key namespaces are currently supported for bridge networks:

- `bridge` (L2 interface configuration)
- `fan` (configuration specific to the Ubuntu FAN overlay)
- `tunnel` (cross-host tunneling configuration)
- `ipv4` (L3 IPv4 configuration)
- `ipv6` (L3 IPv6 configuration)
- `dns` (DNS server and resolution configuration)
- `raw` (raw configuration file content)

It is expected that IP addresses and subnets are given using CIDR notation (`1.1.1.1/24` or `fd80:1234::1/64`).

The exception being tunnel local and remote addresses which are just plain addresses (`1.1.1.1` or `fd80:1234::1`).

Key	Type	Condition	Default	Description
bridge.driver	string	-	native	Bridge driver ("native" or "openvswitch")
bridge.external_interfaces	string	-	-	Comma separate list of unconfigured network int
bridge.hwaddr	string	-	-	MAC address for the bridge

Table 2 – continued from previous page

Key	Type	Condition	Default	Description
bridge.mode	string	-	standard	Bridge operation mode (“standard” or “fan”)
bridge.mtu	integer	-	1500	Bridge MTU (default varies if tunnel or fan setup
dns.domain	string	-	lxd	Domain to advertise to DHCP clients and use for
dns.mode	string	-	managed	DNS registration mode (“none” for no DNS reco
fan.overlay_subnet	string	fan mode	240.0.0.0/8	Subnet to use as the overlay for the FAN (CIDR r
fan.type	string	fan mode	vxlan	The tunneling type for the FAN (“vxlan” or “ipip
fan.underlay_subnet	string	fan mode	auto (on create only)	Subnet to use as the underlay for the FAN (CIDR
ipv4.address	string	standard mode	auto (on create only)	IPv4 address for the bridge (CIDR notation). Use
ipv4.dhcp	boolean	ipv4 address	true	Whether to allocate addresses using DHCP
ipv4.dhcp.expiry	string	ipv4 dhcp	1h	When to expire DHCP leases
ipv4.dhcp.gateway	string	ipv4 dhcp	ipv4.address	Address of the gateway for the subnet
ipv4.dhcp.ranges	string	ipv4 dhcp	all addresses	Comma separated list of IP ranges to use for DHC
ipv4.firewall	boolean	ipv4 address	true	Whether to generate filtering firewall rules for thi
ipv4.nat	boolean	ipv4 address	false	Whether to NAT (defaults to true for regular brid
ipv4.nat.order	string	ipv4 address	before	Whether to add the required NAT rules before or
ipv4.nat.address	string	ipv4 address	-	The source address used for outbound traffic from
ipv4.routes	string	ipv4 address	-	Comma separated list of additional IPv4 CIDR su
ipv4.routing	boolean	ipv4 address	true	Whether to route traffic in and out of the bridge
ipv6.address	string	standard mode	auto (on create only)	IPv6 address for the bridge (CIDR notation). Use
ipv6.dhcp	boolean	ipv6 address	true	Whether to provide additional network configurat
ipv6.dhcp.expiry	string	ipv6 dhcp	1h	When to expire DHCP leases
ipv6.dhcp.ranges	string	ipv6 stateful dhcp	all addresses	Comma separated list of IPv6 ranges to use for D
ipv6.dhcp.stateful	boolean	ipv6 dhcp	false	Whether to allocate addresses using DHCP
ipv6.firewall	boolean	ipv6 address	true	Whether to generate filtering firewall rules for thi
ipv6.nat	boolean	ipv6 address	false	Whether to NAT (will default to true if unset and
ipv6.nat.order	string	ipv6 address	before	Whether to add the required NAT rules before or
ipv6.nat.address	string	ipv6 address	-	The source address used for outbound traffic from
ipv6.routes	string	ipv6 address	-	Comma separated list of additional IPv6 CIDR su
ipv6.routing	boolean	ipv6 address	true	Whether to route traffic in and out of the bridge
maas.subnet.ipv4	string	ipv4 address	-	MAAS IPv4 subnet to register instances in (when
maas.subnet.ipv6	string	ipv6 address	-	MAAS IPv6 subnet to register instances in (when
raw.dnsmasq	string	-	-	Additional dnsmasq configuration to append to th
tunnel.NAME.group	string	vxlan	239.0.0.1	Multicast address for vxlan (used if local and rem
tunnel.NAME.id	integer	vxlan	0	Specific tunnel ID to use for the vxlan tunnel
tunnel.NAME.interface	string	vxlan	-	Specific host interface to use for the tunnel
tunnel.NAME.local	string	gre or vxlan	-	Local address for the tunnel (not necessary for m
tunnel.NAME.port	integer	vxlan	0	Specific port to use for the vxlan tunnel
tunnel.NAME.protocol	string	standard mode	-	Tunneling protocol (“vxlan” or “gre”)
tunnel.NAME.remote	string	gre or vxlan	-	Remote address for the tunnel (not necessary for
tunnel.NAME.ttl	integer	vxlan	1	Specific TTL to use for multicast routing topolog

Those keys can be set using the lxc tool with:

```
lxc network set <network> <key> <value>
```

Integration with systemd-resolved

If the system running LXD uses systemd-resolved to perform DNS lookups, it's possible to notify resolved of the domain(s) that LXD is able to resolve. This requires telling resolved the specific bridge(s), nameserver address(es), and dns domain(s).

For example, if LXD is using the `lxdbr0` interface, get the ipv4 address with `lxc network get lxdbr0 ipv4.address` command (the ipv6 can be used instead or in addition), and the domain with `lxc network get lxdbr0 dns.domain` (if unset, the domain is `lxd` as shown in the table above). Then notify resolved:

```
systemd-resolve --interface lxdbr0 --set-domain '~lxd' --set-dns n.n.n.n
```

Replace `lxdbr0` with the actual bridge name, and `n.n.n.n` with the actual address of the nameserver (without the subnet netmask).

Also replace `lxd` with the domain name. Note the ~ before the domain name is important; it tells resolved to use this nameserver to look up only this domain; no matter what your actual domain name is, you should prefix it with ~. Also, since the shell may expand the ~ character, you may need to include it in quotes.

In newer releases of systemd, the `systemd-resolve` command has been deprecated, however it is still provided for backwards compatibility (as of this writing). The newer method to notify resolved is using the `resolvectl` command, which would be done in two steps:

```
resolvectl dns lxdbr0 n.n.n.n
resolvectl domain lxdbr0 '~lxd'
```

This resolved configuration will persist as long as the bridge exists, so you must repeat this command each reboot and after LXD is restarted (see below on how to automate this).

Also note this only works if the bridge `dns.mode` is not `none`.

Note that depending on the `dns.domain` used, you may need to disable DNSSEC in resolved to allow for DNS resolution. This can be done through the `DNSSEC` option in `resolved.conf`.

To automate the `systemd-resolved` DNS configuration when LXD creates the `lxdbr0` interface so that it is applied on system start you need to create a systemd unit file `/etc/systemd/system/lxd-dns-lxdbr0.service` containing:

```
[Unit]
Description=LXD per-link DNS configuration for lxdbr0
BindsTo=sys-subsystem-net-devices-lxdbr0.device
After=sys-subsystem-net-devices-lxdbr0.device

[Service]
Type=oneshot
ExecStart=/usr/bin/resolvectl dns lxdbr0 n.n.n.n
ExecStart=/usr/bin/resolvectl domain lxdbr0 '~lxd'

[Install]
WantedBy=sys-subsystem-net-devices-lxdbr0.device
```

Be sure to replace `n.n.n.n` in that file with the IP of the `lxdbr0` bridge.

Then enable and start it using:

```
sudo systemctl daemon-reload
sudo systemctl enable --now lxd-dns-lxdbr0
```

If the `lxdbr0` interface already exists (i.e LXD is running), then you can check that the new service has started:

```
sudo systemctl status lxd-dns-lxdbr0.service
 lxd-dns-lxdbr0.service - LXD per-link DNS configuration for lxdbr0
     Loaded: loaded (/etc/systemd/system/lxd-dns-lxdbr0.service; enabled; vendor preset:␣
↪enabled)
     Active: inactive (dead) since Mon 2021-06-14 17:03:12 BST; 1min 2s ago
    Process: 9433 ExecStart=/usr/bin/resolvectl dns lxdbr0 n.n.n.n (code=exited,␣
↪status=0/SUCCESS)
    Process: 9434 ExecStart=/usr/bin/resolvectl domain lxdbr0 ~lxd (code=exited,␣
↪status=0/SUCCESS)
   Main PID: 9434 (code=exited, status=0/SUCCESS)
```

You can then check it has applied the settings using:

```
sudo resolvectl status lxdbr0
Link 6 (lxdbr0)
      Current Scopes: DNS
DefaultRoute setting: no
       LLMNR setting: yes
MulticastDNS setting: no
  DNSOverTLS setting: no
      DNSSEC setting: no
    DNSSEC supported: no
  Current DNS Server: n.n.n.n
         DNS Servers: n.n.n.n
          DNS Domain: ~lxd
```

IPv6 prefix size

For optimal operation, a prefix size of 64 is preferred. Larger subnets (prefix smaller than 64) should work properly too but aren't typically that useful for SLAAC.

Smaller subnets while in theory possible when using stateful DHCPv6 for IPv6 allocation aren't properly supported by dnsmasq and may be the source of issue. If you must use one of those, static allocation or another standalone RA daemon be used.

Allow DHCP, DNS with Firewalld

In order to allow instances to access the DHCP and DNS server that LXD runs on the host when using firewalld you need to add the host's bridge interface to the `trusted` zone in firewalld.

To do this permanently (so that it persists after a reboot) run the following command:

```
firewall-cmd --zone=trusted --change-interface=<LXD network name> --permanent
```

E.g. for a bridged network called `lxdbr0` run the command:

```
firewall-cmd --zone=trusted --change-interface=lxdbr0 --permanent
```

This will then allow LXD's own firewall rules to take effect.

How to let Firewalld control the LXD's iptables rules

When using firewalld and LXD together, iptables rules can overlaps. For example, firewalld could erase LXD iptables rules if it is started after LXD daemon, then LXD container will not be able to do any oubound internet access. One way to fix it is to delegate to firewalld the LXD's iptables rules and to disable the LXD ones.

First step is to *allow DNS and DHCP*.

Then to tell to LXD totally stop to set iptables rules (because firewalld will do it):

```
lxc network set lxdbr0 ipv4.nat false
lxc network set lxdbr0 ipv6.nat false
lxc network set lxdbr0 ipv6.firewall false
lxc network set lxdbr0 ipv4.firewall false
```

Finally, to enable iptables firewalld's rules for LXD usecase (in this example, we suppose the bridge interface is `lxdbr0` and the associated IP range is `10.0.0.0/24`:

```
firewall-cmd --permanent --direct --add-rule ipv4 filter INPUT 0 -i lxdbr0 -s 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 filter OUTPUT 0 -o lxdbr0 -d 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 filter FORWARD 0 -i lxdbr0 -s 10.0.0.0/
↪24 -m comment --comment "generated by firewalld for LXD" -j ACCEPT
firewall-cmd --permanent --direct --add-rule ipv4 nat POSTROUTING 0 -s 10.0.0.0/24 ! -d␣
↪10.0.0.0/24 -m comment --comment "generated by firewalld for LXD" -j MASQUERADE
firewall-cmd --reload
```

To check the rules are taken into account by firewalld:

```
firewall-cmd --direct --get-all-rules
```

Warning: what is exposed above is not a fool-proof approach and may end up inadvertently introducing a security risk.

3.2.4 Non-interactive configuration via preseed YAML

The `lxd init` command supports a `--preseed` command line flag that makes it possible to fully configure LXD daemon settings, storage pools, network devices and profiles, in a non-interactive way.

For example, starting from a brand new LXD installation, the command line:

```
    cat <<EOF | lxd init --preseed
config:
  core.https_address: 192.168.1.1:9999
  images.auto_update_interval: 15
networks:
- name: lxdbr0
  type: bridge
  config:
    ipv4.address: auto
    ipv6.address: none
EOF
```

will configure the LXD daemon to listen for HTTPS connections on port 9999 of the 192.168.1.1 address, to automatically update images every 15 hours, and to create a network bridge device named `lxdbr0`, which will get assigned an

IPv4 address automatically.

Configure a brand new LXD

If you are configuring a brand new LXD instance, then the preseed command will always succeed and apply the desired configuration (as long as the given YAML contains valid keys and values), since there is no existing state that might conflict with the desired one.

Re-configuring an existing LXD

If you are re-configuring an existing LXD instance using the preseed command, then the provided YAML configuration is meant to completely overwrite existing entities (if the provided entities do not exist, they will just be created, as in the brand new LXD case).

In case you are overwriting an existing entity you must provide the full configuration of the new desired state for the entity (i.e. the semantics is the same as a `PUT` request in the *RESTful API*).

Rollback

If some parts of the new desired configuration conflict with the existing state (for example they try to change the driver of a storage pool from `dir` to `zfs`), then the preseed command will fail and will automatically try its best to rollback any change that was applied so far.

For example it will delete entities that were created by the new configuration and revert overwritten entities back to their original state.

Failure modes when overwriting entities are the same as `PUT` requests in the *RESTful API*.

Note however, that the rollback itself might potentially fail as well, although rarely (typically due to backend bugs or limitations). Thus care must be taken when trying to reconfigure a LXD daemon via preseed.

Default profile

Differently from the interactive init mode, the `lxd init --preseed` command line will not modify the default profile in any particular way, unless you explicitly express that in the provided YAML payload.

For instance, you will typically want to attach a root disk device and a network interface to your default profile. See below for an example.

Configuration format

The supported keys and values of the various entities are the same as the ones documented in the *RESTful API*, but converted to YAML for easier reading (however you can use JSON too, since YAML is a superset of JSON).

Here follows an example of a preseed payload containing most of the possible configuration knobs. You can use it as a template for your own one, and add, change or remove what you need:

```
# Daemon settings
config:
  core.https_address: 192.168.1.1:9999
  core.trust_password: sekret
  images.auto_update_interval: 6
```

(continues on next page)

(continued from previous page)

```
# Storage pools
storage_pools:
- name: data
  driver: zfs
  config:
    source: my-zfs-pool/my-zfs-dataset

# Network devices
networks:
- name: lxd-my-bridge
  type: bridge
  config:
    ipv4.address: auto
    ipv6.address: none

# Profiles
profiles:
- name: default
  devices:
    root:
      path: /
      pool: data
      type: disk
- name: test-profile
  description: "Test profile"
  config:
    limits.memory: 2GB
  devices:
    test0:
      name: test0
      nictype: bridged
      parent: lxd-my-bridge
      type: nic
```

3.2.5 Profiles

Introduction

Profiles can store any configuration that an instance can (key/value or devices) and any number of profiles can be applied to an instance.

Profiles are applied in the order they are specified so the last profile to specify a specific key wins.

In any case, instance-specific configuration always overrides that coming from the profiles.

Default profile

If not present, LXD will create a `default` profile. The `default` profile cannot be renamed or removed. The `default` profile is set for any new instance created which doesn't specify a different profiles list.

Configuration

As profiles aren't specific to containers or virtual machines, they may contain configuration and devices that are valid for either type.

This differs from the behavior when applying those config/devices directly to an instance where its type is then taken into consideration and keys that aren't allowed result in an error.

See *instance configuration* for valid configuration options.

3.2.6 Project configuration

LXD supports projects as a way to split your LXD server. Each project holds its own set of instances and may also have its own images and profiles.

What a project contains is defined through the `features` configuration keys. When a feature is disabled, the project inherits from the `default` project.

By default all new projects get the entire feature set, on upgrade, existing projects do not get new features enabled.

The key/value configuration is namespaced with the following namespaces currently supported:

- `features` (What part of the project featureset is in use)
- `limits` (Resource limits applied on containers and VMs belonging to the project)
- `user` (free form key/value for user metadata)

Key	Type	Condition	Default	Description
features.images	boolean	-	true	Separate set of images and image aliases for the project
features.profiles	boolean	-	true	Separate set of profiles for the project
features.storage.volumes	boolean	-	true	Separate set of storage volumes for the project
limits.containers	integer	-	-	Maximum number of containers that can be created in the project
limits.cpu	integer	-	-	Maximum value for the sum of individual "limits.cpu" configs set on the instances of the project
limits.disk	string	-	-	Maximum value of aggregate disk space used by all instances volumes, custom volumes and images of the project
limits.memory	string	-	-	Maximum value for the sum of individual "limits.memory" configs set on the instances of the project
limits.processes	integer	-	-	Maximum value for the sum of individual "limits.processes" configs set on the instances of the project
limits.virtual-machines	integer	-	-	Maximum number of VMs that can be created in the project
restricted	boolean	-	false	Block access to security-sensitive features (this must be enabled to allow the `restricted.*` keys to take effect, this is so it can be tempoarily disabled if needed without having to clear the related keys)
restricted.containers.lowlevel	string	-	block	Prevents use of low-level container options like raw.lxc, raw.idmap, volatile, etc.
restricted.containers.nesting	string	-	block	Prevents setting security.nesting=true.
restricted.containers.privilege	string	-	unprivileged	If "unpriviliged", prevents setting security.privileged=true. If "isolated", prevents setting security.privileged=true and also security.idmap.isolated=true. If "allow", no restriction apply.
restricted.devices.disk	string	-	managed	If "block" prevent use of disk devices except the root one. If "managed" allow use of disk devices only if "pool=" is set. If "allow", no restrictions apply.
restricted.devices.gpu	string	-	block	Prevents use of devices of type "gpu"
restricted.devices.infiniband	string	-	block	Prevents use of devices of type "infiniband"
restricted.devices.nic	string	-	managed	If "block" prevent use of all network devices. If "managed" allow use of network devices only if "network=" is set. If "allow", no restrictions apply.
restricted.devices.pci	string	-	block	Prevents use of devices of type "pci"
restricted.devices.proxy	string	-	block	Prevents use of devices of type "proxy"
restricted.devices.unix-block	string	-	block	Prevents use of devices of type "unix-block"
restricted.devices.unix-char	string	-	block	Prevents use of devices of type "unix-char"
restricted.devices.unix-hotplug	string	-	block	Prevents use of devices of type "unix-hotplug"
restricted.devices.usb	string	-	block	Prevents use of devices of type "usb"
restricted.virtual-machines.lowlevel	string	-	block	Prevents use of low-level virtual-machine options like raw.qemu, volatile, etc.

Those keys can be set using the lxc tool with:

```
lxc project set <project> <key> <value>
```

Project limits

Note that to be able to set one of the `limits.*` config keys, **all** instances in the project **must** have that same config key defined, either directly or via a profile.

In addition to that:

- The `limits.cpu` config key also requires that CPU pinning is **not** used.
- The `limits.memory` config key must be set to an absolute value, **not** a percentage.

The `limits.*` config keys defined on a project act as a hard upper bound for the **aggregate** value of the individual `limits.*` config keys defined on the project's instances, either directly or via profiles.

For example, setting the project's `limits.memory` config key to `50GB` means that the sum of the individual values of all `limits.memory` config keys defined on the project's instances will be kept under `50GB`. Trying to create or modify an instance assigning it a `limits.memory` value that would make the total sum exceed `50GB`, will result in an error.

Similarly, setting the project's `limits.cpu` config key to `100`, means that the **sum** of individual `limits.cpu` values will be kept below `100`.

Project restrictions

If the `restricted` config key is set to `true`, then the instances of the project won't be able to access security-sensitive features, such as container nesting, raw LXC configuration, etc.

The exact set of features that the `restricted` config key blocks may grow across LXD releases, as more features are added that are considered security-sensitive.

Using the various `restricted.*` sub-keys, it's possible to pick individual features which would be normally blocked by `restricted` and allow them, so they can be used by instances of the project.

For example:

```
lxc project set <project> restricted=true
lxc project set <project> restricted.containers.nesting=allow
```

will block all security-sensitive features **except** container nesting.

Each security-sensitive feature has an associated `restricted.*` project config sub-key whose default value needs to be explicitly changed if you want for that feature to be allowed it in the project.

Note that changing the value of a specific `restricted.*` config key has an effect only if the top-level `restricted` key itself is currently set to `true`. If `restricted` is set to `false`, changing a `restricted.*` sub-key is effectively a no-op.

Most `'restricted.*` config keys are binary switches that can be set to either `block` (the default) or `allow`. However some of them support other values for more fine-grained control.

Setting all `restricted.*` keys to `allow` is effectively equivalent to setting `restricted` itself to `false`.

3.2.7 Server configuration

The server configuration is a simple set of key and values.

The key/value configuration is namespaced with the following namespaces currently supported:

- `backups` (backups configuration)
- `candid` (External user authentication through Candid)
- `cluster` (cluster configuration)
- `core` (core daemon configuration)
- `images` (image configuration)
- `maas` (MAAS integration)
- `rbac` (Role Based Access Control through external Candid + Canonical RBAC)

Key	Type	Scope	Default	Description
backups.compression_algorithm	string	global	gzip	Compression algorithm to use for new images (bzip2, gzip, lzma
candid.api.key	string	global	-	Public key of the candid server (required for HTTP-only servers)
candid.api.url	string	global	-	URL of the the external authentication endpoint using Candid
candid.domains	string	global	-	Comma-separated list of allowed Candid domains (empty string
candid.expiry	integer	global	3600	Candid macaroon expiry in seconds
cluster.https_address	string	local	-	Address to use for clustering traffic
cluster.images_minimal_replica	integer	global	3	Minimal numbers of cluster members with a copy of a particular
cluster.max_standby	integer	global	2	Maximum number of cluster members that will be assigned the c
cluster.max_voters	integer	global	3	Maximum number of cluster members that will be assigned the c
cluster.offline_threshold	integer	global	20	Number of seconds after which an unresponsive node is consider
core.debug_address	string	local	-	Address to bind the pprof debug server to (HTTP)
core.https_address	string	local	-	Address to bind for the remote API (HTTPS)
core.https_allowed_credentials	boolean	global	-	Whether to set Access-Control-Allow-Credentials http header va
core.https_allowed_headers	string	global	-	Access-Control-Allow-Headers http header value
core.https_allowed_methods	string	global	-	Access-Control-Allow-Methods http header value
core.https_allowed_origin	string	global	-	Access-Control-Allow-Origin http header value
core.https_trusted_proxy	string	global	-	Comma-separated list of IP addresses of trusted servers to provic
core.proxy_https	string	global	-	https proxy to use, if any (falls back to HTTPS_PROXY environ
core.proxy_http	string	global	-	http proxy to use, if any (falls back to HTTP_PROXY environme
core.proxy_ignore_hosts	string	global	-	hosts which don't need the proxy for use (similar format to NO_
core.shutdown_timeout	integer	global	5	Number of minutes to wait for running operations to complete be
core.trust_ca_certificates	boolean	global	-	Whether to automatically trust clients signed by the CA
core.trust_password	string	global	-	Password to be provided by clients to setup a trust
images.auto_update_cached	boolean	global	true	Whether to automatically update any image that LXD caches
images.auto_update_interval	integer	global	6	Interval in hours at which to look for update to cached images (0
images.compression_algorithm	string	global	gzip	Compression algorithm to use for new images (bzip2, gzip, lzma
images.remote_cache_expiry	integer	global	10	Number of days after which an unused cached remote image will
maas.api.key	string	global	-	API key to manage MAAS
maas.api.url	string	global	-	URL of the MAAS server
maas.machine	string	local	hostname	Name of this LXD host in MAAS
rbac.agent.private_key	string	global	-	The Candid agent private key as provided during RBAC registrat
rbac.agent.public_key	string	global	-	The Candid agent public key as provided during RBAC registrati
rbac.agent.url	string	global	-	The Candid agent url as provided during RBAC registration
rbac.agent.username	string	global	-	The Candid agent username as provided during RBAC registratio

Table 3 – continued from previous page

Key	Type	Scope	Default	Description
rbac.api.expiry	integer	global	-	RBAC macaroon expiry in seconds
rbac.api.key	string	global	-	Public key of the RBAC server (required for HTTP-only servers)
rbac.api.url	string	global	-	URL of the external RBAC server
storage.backups_volume	string	local	-	Volume to use to store the backup tarballs (syntax is POOL/VOL
storage.images_volume	string	local	-	Volume to use to store the image tarballs (syntax is POOL/VOLU

Those keys can be set using the lxc tool with:

```
lxc config set <key> <value>
```

When operating as part of a cluster, the keys marked with a `global` scope will immediately be applied to all the cluster members. Those keys with a `local` scope must be set on a per member basis using the `--target` option of the command line tool.

Exposing LXD to the network

By default, LXD can only be used by local users through a UNIX socket.

To expose LXD to the network, you'll need to set `core.https_address`. All remote clients can then connect to LXD and access any image which was marked for public use.

Trusted clients can be manually added to the trust store on the server with `lxc config trust add` or the `core.trust_password` key can be set allowing for clients to self-enroll into the trust store at connection time by providing the configured password.

More details about authentication can be found *here*.

External authentication

LXD when accessed over the network can be configured to use external authentication through Candid.

Setting the `candid.*` configuration keys above to the values matching your Candid deployment will allow users to authenticate through their web browsers and then get trusted by LXD.

For those that have a Canonical RBAC server in front of their Candid server, they can instead set the `rbac.*` configuration keys which are a superset of the `candid.*` ones and allow for LXD to integrate with the RBAC service.

When integrated with RBAC, individual users and groups can be granted various level of access on a per-project basis. All of this is driven externally through the RBAC service.

More details about authentication can be found *here*.

3.2.8 Storage configuration

LXD supports creating and managing storage pools and storage volumes. General keys are top-level. Driver specific keys are namespaced by driver name. Volume keys apply to any volume created in the pool unless the value is overridden on a per-volume basis. The following types are supported:

- *dir*
- *ceph*
- *cephfs*
- *btrfs*

- *lvm*
- *zfs*

Storage pool configuration keys can be set using the lxc tool with:

```
lxc storage set [<remote>:]<pool> <key> <value>
```

Storage volume configuration keys can be set using the lxc tool with:

```
lxc storage volume set [<remote>:]<pool> <volume> <key> <value>
```

To set default volume configurations for a storage pool, set a storage pool configuration with a volume prefix i.e. `volume.<VOLUME_CONFIGURATION>=<VALUE>`. For an example, to set the default volume size of a pool with the lxc tool, use:

```
lxc storage set [<remote>:]<pool> volume.size <value>
```

Storage volume content types

Storage volumes can be either `filesystem` or `block` type.

Containers and container images are always going to be using `filesystem`. Virtual machines and virtual machine images are always going to be using `block`.

Custom storage volumes can be either types with the default being `filesystem`. Those custom storage volumes of type `block` can only be attached to virtual machines.

Block custom storage volumes can be created with:

```
lxc storage volume create [<remote>]:<pool> <name> --type=block
```

Where to store LXD data

Depending on the storage backends used, LXD can either share the filesystem with its host or keep its data separate.

Sharing with the host

This is usually the most space efficient way to run LXD and possibly the easiest to manage. It can be done with:

- `dir` backend on any backing filesystem
- `btrfs` backend if the host is btrfs and you point LXD to a dedicated subvolume
- `zfs` backend if the host is zfs and you point LXD to a dedicated dataset on your zpool

Dedicated disk/partition

In this mode, LXD's storage will be completely independent from the host. This can be done by having LXD use an empty partition on your main disk or by having it use a full dedicated disk.

This is supported by all storage drivers except `dir`, `ceph` and `cephfs`.

Loop disk

If neither of the options above are possible for you, LXD can create a loop file on your main drive and then have the selected storage driver use that.

This is functionally similar to using a disk/partition but uses a large file on your main drive instead. This comes at a performance penalty as every writes need to go through the storage driver and then your main drive's filesystem. The loop files also usually cannot be shrunk. They will grow up to the limit you select but deleting instances or images will not cause the file to shrink.

Storage Backends and supported functions

Feature comparison

LXD supports using ZFS, btrfs, LVM or just plain directories for storage of images, instances and custom volumes. Where possible, LXD tries to use the advanced features of each system to optimize operations.

Feature	Directory	Btrfs	LVM	ZFS	CEPH
Optimized image storage	no	yes	yes	yes	yes
Optimized instance creation	no	yes	yes	yes	yes
Optimized snapshot creation	no	yes	yes	yes	yes
Optimized image transfer	no	yes	no	yes	yes
Optimized instance transfer	no	yes	no	yes	yes
Copy on write	no	yes	yes	yes	yes
Block based	no	no	yes	no	yes
Instant cloning	no	yes	yes	yes	yes
Storage driver usable inside a container	yes	yes	no	no	no
Restore from older snapshots (not latest)	yes	yes	yes	no	yes
Storage quotas	yes(*)	yes	yes	yes	yes

Recommended setup

The two best options for use with LXD are ZFS and btrfs. They have about similar functionalities but ZFS is more reliable if available on your particular platform.

Whenever possible, you should dedicate a full disk or partition to your LXD storage pool. While LXD will let you create loop based storage, this isn't recommended for production use.

Similarly, the directory backend is to be considered as a last resort option. It does support all main LXD features, but is terribly slow and inefficient as it can't perform instant copies or snapshots and so needs to copy the entirety of the instance's storage every time.

Security Considerations

Currently, the Linux Kernel may not apply mount options and silently ignore them when a block-based filesystem (e.g. `ext4`) is already mounted with different options. This means when dedicated disk devices are shared between different storage pools with different mount options set, the second mount may not have the expected mount options. This becomes security relevant, when e.g. one storage pool is supposed to provide `acl` support and the second one is supposed to not provide `acl` support. For this reason it is currently recommended to either have dedicated disk devices per storage pool or ensure that all storage pools that share the same dedicated disk device use the same mount options.

Optimized image storage

All backends but the directory backend have some kind of optimized image storage format. This is used by LXD to make instance creation near instantaneous by simply cloning a pre-made image volume rather than unpack the image tarball from scratch.

As it would be wasteful to prepare such a volume on a storage pool that may never be used with that image, the volume is generated on demand, causing the first instance to take longer to create than subsequent ones.

Optimized instance transfer

ZFS, btrfs and CEPH RBD have an internal send/receive mechanisms which allow for optimized volume transfer. LXD uses those features to transfer instances and snapshots between servers.

When such capabilities aren't available, either because the storage driver doesn't support it or because the storage backend of the source and target servers differ, LXD will fallback to using rsync to transfer the individual files instead.

When rsync has to be used LXD allows to specify an upper limit on the amount of socket I/O by setting the `rsync.bwlimit` storage pool property to a non-zero value.

Default storage pool

There is no concept of a default storage pool in LXD. Instead, the pool to use for the instance's root is treated as just another "disk" device in LXD.

The device entry looks like:

```
root:
  type: disk
  path: /
  pool: default
```

And it can be directly set on an instance ("-s" option to "lxc launch" and "lxc init") or it can be set through LXD profiles.

That latter option is what the default LXD setup (through "lxd init") will do for you. The same can be done manually against any profile using (for the "default" profile):

```
lxc profile device add default root disk path=/ pool=default
```

I/O limits

I/O limits in IOp/s or MB/s can be set on storage devices when attached to an instance (see *Instances*).

Those are applied through the Linux `blkio` cgroup controller which makes it possible to restrict I/O at the disk level (but nothing finer grained than that).

Because those apply to a whole physical disk rather than a partition or path, the following restrictions apply:

- Limits will not apply to filesystems that are backed by virtual devices (e.g. device mapper).
- If a filesystem is backed by multiple block devices, each device will get the same limit.
- If the instance is passed two disk devices that are each backed by the same disk, the limits of the two devices will be averaged.

It's also worth noting that all I/O limits only apply to actual block device access, so you will need to consider the filesystem's own overhead when setting limits. This also means that access to cached data will not be affected by the limit.

Notes and examples

dir

- While this backend is fully functional, it's also much slower than all the others due to it having to unpack images or do instant copies of instances, snapshots and images.
- Quotas are supported with the directory backend when running on either ext4 or XFS with project quotas enabled at the filesystem level.

Storage pool configuration

Key	Type	Default	Description
rsync.bwlimit	string	0 (no limit)	Specifies the upper limit to be placed on the socket I/O whenever rsync has to be used to transfer storage entities
rsync.compression	bool	true	Whether to use compression while migrating storage pools
source	string	-	Path to block device or loop file or filesystem entry

Storage volume configuration

Key	Type	Condition	Default	Description
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

The following commands can be used to create directory storage pools

- Create a new directory pool called “pool1”.

```
lxc storage create pool1 dir
```

- Use an existing directory for “pool2”.

```
lxc storage create pool2 dir source=/data/lxd
```

CEPH

- Uses RBD images for images, then snapshots and clones to create instances and snapshots.
- Due to the way copy-on-write works in RBD, parent filesystems can’t be removed until all children are gone. As a result, LXD will automatically prefix any removed but still referenced object with “zombie_” and keep it until such time the references are gone and it can safely be removed.
- Note that LXD will assume it has full control over the osd storage pool. It is recommended to not maintain any non-LXD owned filesystem entities in a LXD OSD storage pool since LXD might delete them.
- Note that sharing the same osd storage pool between multiple LXD instances is not supported. LXD only allows sharing of an OSD storage pool between multiple LXD instances only for backup purposes of existing instances

via `lxd import`. In line with this, LXD requires the "ceph.osd.force_reuse" property to be set to true. If not set, LXD will refuse to reuse an osd storage pool it detected as being in use by another LXD instance.

- When setting up a ceph cluster that LXD is going to use we recommend using `xfs` as the underlying filesystem for the storage entities that are used to hold OSD storage pools. Using `ext4` as the underlying filesystem for the storage entities is not recommended by Ceph upstream. You may see unexpected and erratic failures which are unrelated to LXD itself.
- To use ceph osd pool of type "erasure" you **must** have the osd pool created beforehand, as well as a separate osd pool of type "replicated" that will be used for storing metadata. This is required as RBD & CephFS do not support omap. To specify which pool is "earasure coded" you need to use the `ceph.osd.data_pool_name=<erasure-coded-pool-name>` and `source=<replicated-pool-name>` for the replicated pool.

Storage pool configuration

Key	Type	Default	Description
ceph.cluster_name	string	ceph	Name of the ceph cluster in which to create new storage pools
ceph.osd.data_pool_name	string	-	Name of the osd data pool
ceph.osd.force_reuse	bool	false	Force using an osd storage pool that is already in use by another LXD instance
ceph.osd.pg_num	string	32	Number of placement groups for the osd storage pool
ceph.osd.pool_name	string	name of the pool	Name of the osd storage pool
ceph.rbd.clone_copy	bool	true	Whether to use RBD lightweight clones rather than full dataset copies
ceph.rbd.du	bool	true	Whether to use rbd du to obtain disk usage data for stopped instances.
ceph.rbd.features	string	layering	Comma separate list of RBD features to enable on the volumes
ceph.user.name	string	admin	The ceph user to use when creating storage pools and volumes
volatile.pool.pristine	string	true	Whether the pool has been empty on creation time

Storage volume configuration

Key	Type	Condition	Default	Description
block.filesystem	string	block based driver	same as volume.block.filesystem	Filesystem of the storage volume
block.mount_options	string	block based driver	same as volume.block.mount_options	Mount options for block devices
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

The following commands can be used to create Ceph storage pools

- Create a osd storage pool named “pool1” in the CEPH cluster “ceph”.

```
lxc storage create pool1 ceph
```

- Create a osd storage pool named “pool1” in the CEPH cluster “my-cluster”.

```
lxc storage create pool1 ceph ceph.cluster_name=my-cluster
```

- Create a osd storage pool named “pool1” with the on-disk name “my-osd”.

```
lxc storage create pool1 ceph ceph.osd.pool_name=my-osd
```

- Use the existing osd storage pool “my-already-existing-osd”.

```
lxc storage create pool1 ceph source=my-already-existing-osd
```

- Use the existing osd erasure coded pool "ecpool" and osd replicated pool "rpl-pool".

```
lxc storage create pool1 ceph source=rpl-pool ceph.osd.data_pool_name=ecpool
```

CEPHFS

- Can only be used for custom storage volumes
- Supports snapshots if enabled on the server side

Storage pool configuration

Key	Type	Default	Description
ceph.cluster_name	string	ceph	Name of the ceph cluster in which to create new storage pools
ceph.user.name	string	admin	The ceph user to use when creating storage pools and volumes
cephfs.cluster_name	string	ceph	Name of the ceph cluster in which to create new storage pools
cephfs.path	string	/	The base path for the CEPHFS mount
cephfs.user.name	string	admin	The ceph user to use when creating storage pools and volumes
volatile.pool.pristine	string	true	Whether the pool has been empty on creation time

Storage volume configuration

Key	Type	Condition	Default	Description
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

Btrfs

- Uses a subvolume per instance, image and snapshot, creating btrfs snapshots when creating a new object.
- btrfs can be used as a storage backend inside a container (nesting), so long as the parent container is itself on btrfs. (But see notes about btrfs quota via qgroups.)
- btrfs supports storage quotas via qgroups. While btrfs qgroups are hierarchical, new subvolumes will not automatically be added to the qgroups of their parent subvolumes. This means that users can trivially escape any quotas that are set. If adherence to strict quotas is a necessity users should be mindful of this and maybe consider using a zfs storage pool with refquotas.
- When using quotas it is critical to take into account that btrfs extents are immutable so when blocks are written they end up in new extents and the old ones remain until all of its data is dereferenced or rewritten. This means that a quota can be reached even if the total amount of space used by the current files in the subvolume is smaller than the quota. This is seen most often when using VMs on BTRFS due to the random I/O nature of using raw disk image files on top of a btrfs subvolume. Our recommendation is to not use VMs with btrfs storage pools, but if you insist then please ensure that the instance root disk's `size.state` property is set to 2x the size of the root disk's size to allow all blocks in the disk image file to be rewritten without reaching the qgroup quota. You may also find that using the `btrfs.mount_options=compress-force` storage pool option avoids this scenario as a side effect of enabling compression is to reduce the maximum extent size such that block rewrites don't cause as much storage to be double tracked. However as this is a storage pool option it will affect all volumes on the pool.

Storage pool configuration

Key	Type	Condition	Default	Description
btrfs.mount_options	string	btrfs driver	user_subvol_rm_allowed	Mount options for block devices

Storage volume configuration

Key	Type	Condition	Default	Description
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

The following commands can be used to create BTRFS storage pools

- Create loop-backed pool named “pool1”.

```
lxc storage create pool1 btrfs
```

- Create a new pool called “pool1” using an existing btrfs filesystem at `/some/path`.

```
lxc storage create pool1 btrfs source=/some/path
```

- Create a new pool called “pool1” on `/dev/sdX`.

```
lxc storage create pool1 btrfs source=/dev/sdX
```

Growing a loop backed btrfs pool

LXD doesn't let you directly grow a loop backed btrfs pool, but you can do so with:

```
sudo truncate -s +5G /var/lib/lxd/disks/<POOL>.img
sudo losetup -c <LOOPDEV>
sudo btrfs filesystem resize max /var/lib/lxd/storage-pools/<POOL>/
```

(NOTE: For users of the snap, use `/var/snap/lxd/common/mntns/var/snap/lxd/common/lxd/` instead of `/var/lib/lxd/`)

- LOOPDEV refers to the mounted loop device (e.g. `/dev/loop8`) associated with the storage pool image.
- The mounted loop devices can be found using the following command:

```
losetup -l
```

LVM

- Uses LVs for images, then LV snapshots for instances and instance snapshots.
- The filesystem used for the LVs is ext4 (can be configured to use xfs instead).
- By default, all LVM storage pools use an LVM thinpool in which logical volumes for all LXD storage entities (images, instances, etc.) are created. This behavior can be changed by setting "lvm.use_thinpool" to "false". In this case, LXD will use normal logical volumes for all non-instance snapshot storage entities (images, instances, etc.). This means most storage operations will need to fallback to rsyncing since non-thinpool logical volumes do not support snapshots of snapshots. Note that this entails serious performance impacts for the LVM driver causing it to be close to the fallback DIR driver both in speed and storage usage. This option should only be chosen if the use-case renders it necessary.
- For environments with high instance turn over (e.g continuous integration) it may be important to tweak the archival `retain_min` and `retain_days` settings in `/etc/lvm/lvm.conf` to avoid slowdowns when interacting with LXD.

Storage pool configuration

Key	Type	Default	Description
lvm.thinpool_name	string	LXDThin-Pool	Thin pool where volumes are created
lvm.use_thinpool	bool	true	Whether the storage pool uses a thinpool for logical volumes
lvm.vg.force_reuse	bool	false	Force using an existing non-empty volume group
lvm.vg_name	string	name of the pool	Name of the volume group to create
rsync.bwlimit	string	0 (no limit)	Specifies the upper limit to be placed on the socket I/O whenever rsync has to be used to transfer storage entities
rsync.compression	bool	true	Whether to use compression while migrating storage pools
source	string	-	Path to block device or loop file or filesystem entry

Storage volume configuration

Key	Type	Condition	Default	Description
block.filesystem	string	block based driver	same as volume.block.filesystem	Filesystem of the storage volume
block.mount_options	string	block based driver	same as volume.block.mount_options	Mount options for block devices
lvm.stripes	string	lvm driver	-	Number of stripes to use for new volumes (or thin pool volume)
lvm.stripes.size	string	lvm driver	-	Size of stripes to use (at least 4096 bytes and multiple of 512bytes)
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`

The following commands can be used to create LVM storage pools

- Create a loop-backed pool named “pool1”. The LVM Volume Group will also be called “pool1”.

```
lxc storage create pool1 lvm
```

- Use the existing LVM Volume Group called “my-pool”

```
lxc storage create pool1 lvm source=my-pool
```

- Use the existing LVM Thinpool called “my-pool” in Volume Group “my-vg”.

```
lxc storage create pool1 lvm source=my-vg lvm.thinpool_name=my-pool
```

- Create a new pool named "pool1" on `/dev/sdX`. The LVM Volume Group will also be called "pool1".

```
lxc storage create pool1 lvm source=/dev/sdX
```

- Create a new pool called "pool1" using `/dev/sdX` with the LVM Volume Group called "my-pool".

```
lxc storage create pool1 lvm source=/dev/sdX lvm.vg_name=my-pool
```

ZFS

- When LXD creates a ZFS pool, compression is enabled by default.
- Uses ZFS filesystems for images, then snapshots and clones to create instances and snapshots.
- Due to the way copy-on-write works in ZFS, parent filesystems can't be removed until all children are gone. As a result, LXD will automatically rename any removed but still referenced object to a random deleted/ path and keep it until such time the references are gone and it can safely be removed.
- ZFS as it is today doesn't support delegating part of a pool to a container user. Upstream is actively working on this.
- ZFS doesn't support restoring from snapshots other than the latest one. You can however create new instances from older snapshots which makes it possible to confirm the snapshots is indeed what you want to restore before you remove the newer snapshots.

  LXD can be configured to automatically discard the newer snapshots during restore. This can be configured through the `volume.zfs.remove_snapshots` pool option.

  However note that instance copies use ZFS snapshots too, so you also cannot restore an instance to a snapshot taken before the last copy without having to also delete all its descendants.

  Copying the wanted snapshot into a new instance and then deleting the old instance does however work, at the cost of losing any other snapshot the instance may have had.
- Note that LXD will assume it has full control over the ZFS pool or dataset. It is recommended to not maintain any non-LXD owned filesystem entities in a LXD zfs pool or dataset since LXD might delete them.
- When quotas are used on a ZFS dataset LXD will set the ZFS "quota" property. In order to have LXD set the ZFS "refquota" property, either set "zfs.use_refquota" to "true" for the given dataset or set "volume.zfs.use_refquota" to true on the storage pool. The former option will make LXD use refquota only for the given storage volume the latter will make LXD use refquota for all storage volumes in the storage pool.
- I/O quotas (IOps/MBs) are unlikely to affect ZFS filesystems very much. That's because of ZFS being a port of a Solaris module (using SPL) and not a native Linux filesystem using the Linux VFS API which is where I/O limits are applied.

Storage pool configuration

Key	Type	Default	Description
size	string	0	Size of the storage pool in bytes (suffixes supported). (Currently valid for loop based pools and zfs.)
source	string	-	Path to block device or loop file or filesystem entry
zfs.clone_copy	bool	true	Whether to use ZFS lightweight clones rather than full dataset copies
zfs.pool_name	string	name of the pool	Name of the zpool

Storage volume configuration

Key	Type	Condition	Default	Description
security.shifted	bool	custom volume	false	Enable id shifting overlay (allows attach by multiple isolated instances)
security.unmapped	bool	custom volume	false	Disable id mapping for the volume
size	string	appropriate driver	same as volume.size	Size of the storage volume
snapshots.expiry	string	custom volume	-	Controls when snapshots are to be deleted (expects expression like `1M 2H 3d 4w 5m 6y`)
snapshots.pattern	string	custom volume	snap%d	Pongo2 template string which represents the snapshot name (used for scheduled snapshots and unnamed snapshots)
snapshots.schedule	string	custom volume	-	Cron expression (`<minute> <hour> <dom> <month> <dow>`), or a comma separated list of schedule aliases `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>`
zfs.remove_snapshots	string	zfs driver	same as volume.zfs.remove_snapshots	Remove snapshots as needed
zfs.use_refquota	string	zfs driver	same as volume.zfs.zfs_refquota	Use refquota instead of quota for space

The following commands can be used to create ZFS storage pools

- Create a loop-backed pool named "pool1". The ZFS Zpool will also be called "pool1".

```
lxc storage create pool1 zfs
```

- Create a loop-backed pool named "pool1" with the ZFS Zpool called "my-tank".

```
lxc storage create pool1 zfs zfs.pool_name=my-tank
```

- Use the existing ZFS Zpool "my-tank".

```
lxc storage create pool1 zfs source=my-tank
```

- Use the existing ZFS dataset "my-tank/slice".

```
lxc storage create pool1 zfs source=my-tank/slice
```

- Create a new pool called "pool1" on `/dev/sdX`. The ZFS Zpool will also be called "pool1".

```
lxc storage create pool1 zfs source=/dev/sdX
```

- Create a new pool on `/dev/sdX` with the ZFS Zpool called "my-tank".

```
lxc storage create pool1 zfs source=/dev/sdX zfs.pool_name=my-tank
```

Growing a loop backed ZFS pool

LXD doesn't let you directly grow a loop backed ZFS pool, but you can do so with:

```
sudo truncate -s +5G /var/lib/lxd/disks/<POOL>.img
sudo zpool set autoexpand=on lxd
sudo zpool online -e lxd /var/lib/lxd/disks/<POOL>.img
sudo zpool set autoexpand=off lxd
```

(NOTE: For users of the snap, use `/var/snap/lxd/common/lxd/` instead of `/var/lib/lxd/`)

Enabling TRIM on existing pools

LXD will automatically enable trimming support on all newly created pools on ZFS 0.8 or later.

This helps with the lifetime of SSDs by allowing better block re-use by the controller. This also will allow freeing space on the root filesystem when using a loop backed ZFS pool.

For systems which were upgraded from pre-0.8 to 0.8, this can be enabled with a one time action of:

- zpool upgrade ZPOOL-NAME
- zpool set autotrim=on ZPOOL-NAME
- zpool trim ZPOOL-NAME

This will make sure that TRIM is automatically issued in the future as well as cause TRIM on all currently unused space.

3.2.9 Virtual Machines

Introduction

Virtual machines are a new instance type supported by LXD alongside containers.

They are implemented through the use of `qemu`.

Please note, currently not all features that are available with containers have been implemented for VMs, however we continue to strive for feature parity with containers.

Configuration

See *instance configuration* for valid configuration options.

3.3 Images

3.3.1 Architectures

Introduction

LXD just like LXC can run on just about any architecture that's supported by the Linux kernel and by Go.

Some objects in LXD are tied to an architecture, like the container, container snapshots and images.

This document lists all the supported architectures, their unique identifier (used in the database), how they should be named and some notes.

Please note that what LXD cares about is the kernel architecture, not the particular userspace flavor as determined by the toolchain.

That means that LXD considers armv7 hard-float to be the same as armv7 soft-float and refers to both as "armv7". If useful to the user, the exact userspace ABI may be set as an image and container property, allowing easy query.

Architectures

ID	Name	Notes	Personalities
1	i686	32bit Intel x86	
2	x86_64	64bit Intel x86	x86
3	armv7l	32bit ARMv7 little-endian	
4	aarch64	64bit ARMv8 little-endian	armv7 (optional)
5	ppc	32bit PowerPC big-endian	
6	ppc64	64bit PowerPC big-endian	powerpc
7	ppc64le	64bit PowerPC little-endian	
8	s390x	64bit ESA/390 big-endian	
9	mips	32bit MIPS	
10	mips64	64bit MIPS	mips
11	riscv32	32bit RISC-V little-endian	
12	riscv64	64bit RISC-V little-endian	

The architecture names above are typically aligned with the Linux kernel architecture names.

3.3.2 Custom network configuration with cloud-init

cloud-init may be used for custom network configuration of instances.

Before trying to use it, however, first determine which image source you are about to use as not all images have cloud-init package installed.

The images from the `ubuntu` and `ubuntu-daily` remotes are all cloud-init enabled. Images from the `images` remote have cloud-init enabled variants using the `/cloud` suffix.

cloud-init uses the network-config data to render the relevant network configuration on the system using either ifupdown or netplan depending on the Ubuntu release.

The default behavior is to use a DHCP client on an instance's eth0 interface.

In order to change this you need to define your own network configuration using user.network-config key in the config dictionary which will override the default configuration (this is due to how the template is structured).

For example, to configure a specific network interface with a static IPv4 address and also use a custom nameserver use

```
config:
  user.network-config: |
    version: 1
    config:
      - type: physical
        name: eth1
        subnets:
          - type: static
```

(continues on next page)

(continued from previous page)

```
            ipv4: true
            address: 10.10.101.20
            netmask: 255.255.255.0
            gateway: 10.10.101.1
            control: auto
        - type: nameserver
          address: 10.10.10.254
```

An instance's rootfs will contain the following files as a result:

- `/var/lib/cloud/seed/nocloud-net/network-config`
- `/etc/network/interfaces.d/50-cloud-init.cfg` (if using ifupdown)
- `/etc/netplan/50-cloud-init.yaml` (if using netplan)

Implementation Details

cloud-init allows you to seed instance configuration using the following files located at `/var/lib/cloud/seed/nocloud-net`:

- `user-data` (required)
- `meta-data` (required)
- `vendor-data` (optional)
- `network-config` (optional)

The network-config file is written to by LXD using data provided in templates that come with an image. This is governed by metadata.yaml but naming of the configuration keys and template content is not hard-coded as far as LXD is concerned - this is purely image data that can be modified if needed.

- NoCloud data source documentation
- The source code for NoCloud data source
- A good reference on which values you can use are unit tests for cloud-init
- cloud-init directory layout

A default `cloud-init-network.tpl` provided with images from the "ubuntu:" image source looks like this:

```
{% if config\_get("user.network-config", "") == "" %}version: 1
config:
    - type: physical
      name: eth0
      subnets:
          - type: {% if config_get("user.network_mode", "") == "link-local" %}manual{%␣
↪else %}dhcp{% endif %}
            control: auto{% else %}{{ config_get("user.network-config", "") }}{% endif %}
```

The template syntax is the one used in the pongo2 template engine. A custom `config_get` function is defined to retrieve values from an instance configuration.

Options available with such a template structure:

- Use DHCP by default on your eth0 interface;
- Set `user.network_mode` to `link-local` and configure networking by hand;

- Seed cloud-init by defining `user.network-config`.

3.3.3 Image handling

Introduction

LXD uses an image based workflow. It comes with a built-in image store where the user or external tools can import images.

Containers are then started from those images.

It's possible to spawn remote instances using local images or local instances using remote images. In such cases, the image may be cached on the target LXD.

Sources

LXD supports importing images from three different sources:

- Remote image server (LXD or simplestreams)
- Direct pushing of the image files
- File on a remote web server

Remote image server (LXD or simplestreams)

This is the most common source of images and the only one of the three options which is supported directly at instance creation time.

With this option, an image server is provided to the target LXD server along with any needed certificate to validate it (only HTTPS is supported).

The image itself is then selected either by its fingerprint (SHA256) or one of its aliases.

From a CLI point of view, this is what's done behind those common actions:

- lxc launch ubuntu:20.04 u1
- lxc launch images:centos/8 c1
- lxc launch my-server:SHA256 a1
- lxc image copy images:gentoo local: –copy-aliases –auto-update

In the cases of `ubuntu` and `images` above, those remotes use simplestreams as a read-only image server protocol and select images by one of their aliases.

The `my-server` remote there is another LXD server and in that example selects an image based on its fingerprint.

Direct pushing of the image files

This is mostly useful for air-gapped environments where images cannot be directly retrieved from an external server.

In such a scenario, image files can be downloaded on another system using:

- lxc image export ubuntu:20.04

Then transferred to the target system and manually imported into the local image store with:

- lxc image import META ROOTFS –alias ubuntu-20.04

`lxc image import` supports both unified images (single file) and split images (two files) with the example above using the latter.

File on a remote web server

As an alternative to running a full image server only to distribute a single image to users, LXD also supports importing images by URL.

There are a few limitations to that method though:

- Only unified (single file) images are supported
- Additional http headers must be returned by the remote server

LXD will set the following headers when querying the server:

- `LXD-Server-Architectures` to a comma separate list of architectures the client supports
- `LXD-Server-Version` to the version of LXD in use

And expects `LXD-Image-Hash` and `LXD-Image-URL` to be set by the remote server. The former being the SHA256 of the image being downloaded and the latter the URL to download the image from.

This allows for reasonably complex image servers to be implemented using only a basic web server with support for custom headers.

On the client side, this is used with:

```
lxc image import URL --alias some-name
```

Publishing an instance or snapshot as a new image

An instance or one of its snapshots can be turned into a new image. This is done on the CLI with `lxc publish`.

When doing this, you will most likely first want to cleanup metadata and templates on the instance you're publishing using the `lxc config metadata` and `lxc config template` commands. You will also want to remove any instance-specific state like host SSH keys, dbus/systemd machine-id, ...

The publishing process can take quite a while as a tarball must be generated from the instance and then be compressed. As this can be particularly I/O and CPU intensive, publish operations are serialized by LXD.

Caching

When spawning an instance from a remote image, the remote image is downloaded into the local image store with the cached bit set. The image will be kept locally as a private image until either it's been unused (no new instance spawned) for the number of days set in `images.remote_cache_expiry` or until the image's expiry is reached whichever comes first.

LXD keeps track of image usage by updating the `last_used_at` image property every time a new instance is spawned from the image.

Auto-update

LXD can keep images up to date. By default, any image which comes from a remote server and was requested through an alias will be automatically updated by LXD. This can be changed with `images.auto_update_cached`.

On startup and then every 6 hours (unless `images.auto_update_interval` is set), the LXD daemon will go look for more recent version of all the images in the store which are marked as auto-update and have a recorded source server.

When a new image is found, it is downloaded into the image store, the aliases pointing to the old image are moved to the new one and the old image is removed from the store.

The user can also request a particular image be kept up to date when manually copying an image from a remote server.

If a new upstream image update is published and the local LXD has the previous image in its cache when the user requests a new instance to be created from it, LXD will use the previous version of the image rather than delay the instance creation.

This behavior only happens if the current image is scheduled to be auto-updated and can be disabled by setting `images.auto_update_interval` to 0.

Profiles

A list of profiles can be associated with an image using the `lxc image edit` command. After associating profiles with an image, an instance launched using the image will have the profiles applied in order. If `nil` is passed as the list of profiles, only the `default` profile will be associated with the image. If an empty list is passed, then no profile will be associated with the image, not even the `default` profile. An image's associated profiles can be overridden when launching an instance by using the `--profile` and the `--no-profiles` flags to `lxc launch`.

Special image properties

Image properties beginning with the prefix ***requirements*** (e.g. requirements.XYZ) are used by LXD to determine the compatibility of the host system and the instance to be created by said image. In the event that these are incompatible, LXD will not start the instance.

At the moment, the following requirements are supported:

Key	Type	Default	Description
requirements.secureboot	string	-	If set to "false", indicates the image will not boot under secureboot
requirements.cgroup	string	-	If set to "v1", indicates the image requires the host to run CGroupV1

Image format

LXD currently supports two LXD-specific image formats.

The first is a unified tarball, where a single tarball contains both the instance root and the needed metadata.

The second is a split model, using two files instead, one containing the root, the other containing the metadata.

The former is what's produced by LXD itself and what people should be using for LXD-specific images.

The latter is designed to allow for easy image building from existing non-LXD rootfs tarballs already available today.

Unified tarball

Tarball, can be compressed and contains:

- `rootfs/`
- `metadata.yaml`
- `templates/` (optional)

In this mode, the image identifier is the SHA-256 of the tarball.

Split tarballs

Two (possibly compressed) tarballs. One for metadata, one for the rootfs.

`metadata.tar` contains:

- `metadata.yaml`
- `templates/` (optional)

`rootfs.tar` contains a Linux root filesystem at its root.

In this mode the image identifier is the SHA-256 of the concatenation of the metadata and rootfs tarball (in that order).

Supported compression

LXD supports a wide variety of compression algorithms for tarballs though for compatibility purposes, gzip or xz should be preferred.

For split images, the rootfs file can also be squashfs formatted in the container case. For virtual machines, the `rootfs.img` file is always qcow2 and can optionally be compressed using qcow2's native compression.

Content

For containers, the rootfs directory (or tarball) contains a full file system tree of what will become the `/`. For VMs, this is instead a `rootfs.img` file which becomes the main disk device.

The templates directory contains pongo2-formatted templates of files inside the instance.

`metadata.yaml` contains information relevant to running the image under LXD, at the moment, this contains:

```
architecture: x86_64
creation_date: 1424284563
properties:
  description: Ubuntu 20.04 LTS Intel 64bit
  os: Ubuntu
  release: focal 20.04
templates:
  /etc/hosts:
    when:
      - create
      - rename
    template: hosts.tpl
    properties:
      foo: bar
  /etc/hostname:
    when:
      - start
    template: hostname.tpl
  /etc/network/interfaces:
    when:
      - create
    template: interfaces.tpl
    create_only: true
```

The `architecture` and `creation_date` fields are mandatory, the properties are just a set of default properties for the image. The os, release, name and description fields while not mandatory in any way, should be pretty common.

For templates, the `when` key can be one or more of:

- `create` (run at the time a new instance is created from the image)
- `copy` (run when an instance is created from an existing one)
- `start` (run every time the instance is started)

The templates will always receive the following context:

- `trigger`: name of the event which triggered the template (string)
- `path`: path of the file being templated (string)
- `container`: key/value map of instance properties (name, architecture, privileged and ephemeral) (map[string]string) (deprecated in favor of `instance`)
- `instance`: key/value map of instance properties (name, architecture, privileged and ephemeral) (map[string]string)
- `config`: key/value map of the instance's configuration (map[string]string)
- `devices`: key/value map of the devices assigned to this instance (map[string]map[string]string)
- `properties`: key/value map of the template properties specified in metadata.yaml (map[string]string)

The `create_only` key can be set to have LXD only only create missing files but not overwrite an existing file.

As a general rule, you should never template a file which is owned by a package or is otherwise expected to be overwritten by normal operation of the instance.

For convenience the following functions are exported to pongo templates:

- `config_get("user.foo", "bar")` => Returns the value of `user.foo` or `"bar"` if unset.

3.4 Operation

3.4.1 Backing up a LXD server

What to backup

When planning to backup a LXD server, consider all the different objects that are stored/managed by LXD:

- Instances (database records and filesystems)
- Images (database records, image files and filesystems)
- Networks (database records and state files)
- Profiles (database records)
- Storage volumes (database records and filesystems)

Only backing up the database or only backing up the instances will not get you a fully functional backup.

In some disaster recovery scenarios, that may be reasonable but if your goal is to get back online quickly, consider all the different pieces of LXD you're using.

Full backup

A full backup would include the entirety of `/var/lib/lxd` or `/var/snap/lxd/common/lxd` for snap users.

You will also need to appropriately backup any external storage that you made LXD use, this can be LVM volume groups, ZFS zpools or any other resource which isn't directly self-contained to LXD.

Restoring involves stopping LXD on the target server, wiping the lxd directory, restoring the backup and any external dependency it requires.

If not using the snap package and your source system has a /etc/subuid and /etc/subgid file, restoring those or at least the entries inside them for both the `lxd` and `root` user is also a good idea (avoids needless shifting of container filesystems).

Then start LXD again and check that everything works fine.

Secondary backup LXD server

LXD supports copying and moving instances and storage volumes between two hosts.

So with a spare server, you can copy your instances and storage volumes to that secondary server every so often, allowing it to act as either an offline spare or just as a storage server that you can copy your instances back from if needed.

Instance backups

The `lxc export` command can be used to export instances to a backup tarball. Those tarballs will include all snapshots by default and an "optimized" tarball can be obtained if you know that you'll be restoring on a LXD server using the same storage pool backend.

You can use any compressor installed on the server using the `--compression` flag. There is no validation on the LXD side, any command that is available to LXD and supports `-c` for stdout should work.

Those tarballs can be saved any way you want on any filesystem you want and can be imported back into LXD using the `lxc import` command.

Disaster recovery

LXD provides the `lxd recover` command (note the the `lxd` command rather than the normal `lxc` command). This is an interactive CLI tool that will attempt to scan all storage pools that exist in the database looking for missing volumes that can be recovered. It also provides the ability for the user to specify the details of any unknown storage pools (those that exist on disk but do not exist in the database) and it will attempt to scan those too.

Because LXD maintains a `backup.yaml` file in each instance's storage volume which contains all necessary information to recover a given instance (including instance configuration, attached devices, storage volume and pool configuration) it can be used to rebuild the instance, storage volume and storage pool database records.

The `lxd recover` tool will attempt to mount the storage pool (if not already mounted) and scan it for unknown volumes that look like they are associated with LXD. For each instance volume LXD will attempt to mount it and access the `backup.yaml` file. From there it will perform some consistency checks to compare what is in the `backup.yaml` file with what is actually on disk (such as matching snapshots) and if all checks out then the database records are recreated.

If the storage pool database record also needs to be created then it will prefer to use an instance `backup.yaml` file as the basis of its config, rather than what the user provided during the discovery phase, however if not available then it will fallback to restoring the pool's database record with what was provided by the user.

3.4.2 Clustering

LXD can be run in clustering mode, where any number of LXD servers share the same distributed database and can be managed uniformly using the lxc client or the REST API.

Note that this feature was introduced as part of the API extension "clustering".

Forming a cluster

First you need to choose a bootstrap LXD node. It can be an existing LXD server or a brand new one. Then you need to initialize the bootstrap node and join further nodes to the cluster. This can be done interactively or with a preseed file.

Note that all further nodes joining the cluster must have identical configuration to the bootstrap node, in terms of storage pools and networks. The only configuration that can be node-specific are the `source` and `size` keys for storage pools and the `bridge.external_interfaces` key for networks.

It is strongly recommended that the number of nodes in the cluster be at least three, so the cluster can survive the loss of at least one node and still be able to establish quorum for its distributed state (which is kept in a SQLite database replicated using the Raft algorithm). If the number of nodes is less than three, then only one node in the cluster will store the SQLite database. When the third node joins the cluster, both the second and third nodes will receive a replica of the database.

Interactively

Run `lxd init` and answer `yes` to the very first question ("Would you like to use LXD clustering?"). Then choose a name for identifying the node, and an IP or DNS address that other nodes can use to connect to it, and answer `no` to the question about whether you're joining an existing cluster. Finally, optionally create a storage pool and a network bridge. At this point your first cluster node should be up and available on your network.

You can now join further nodes to the cluster. Note however that these nodes should be brand new LXD servers, or alternatively you should clear their contents before joining, since any existing data on them will be lost.

There are two ways to add a member to an existing cluster; using the trust password or using a join token. A join token for a new member is generated in advance on the existing cluster using the command:

```
lxc cluster add <new member name>
```

This will return a single-use join token which can then be used in the join token question stage of `lxd init`. The join token contains the addresses of the existing online members, as well as a single-use secret and the fingerprint of the cluster certificate. This reduces the amount of questions you have to answer during `lxd init` as the join token can be used to answer these questions automatically.

Alternatively you can use the trust password instead of using a join token.

To add an additional node, run `lxd init` and answer `yes` to the question about whether to use clustering. Choose a node name that is different from the one chosen for the bootstrap node or any other nodes you have joined so far. Then pick an IP or DNS address for the node and answer `yes` to the question about whether you're joining an existing cluster.

If you have a join token then answer `yes` to the question that asks if you have a join token and then copy it in when it asks for it.

If you do not have a join token, but have a trust password instead then, then answer `no` to the question that asks if you have a join token. Then pick an address of an existing node in the cluster and check the fingerprint that gets printed matches the cluster certificate of the existing members.

Per-server configuration

As mentioned previously, LXD cluster members are generally assumed to be identical systems.

However to accommodate things like slightly different disk ordering or network interface naming, LXD records some settings as being server-specific. When such settings are present in a cluster, any new server being added will have to provide a value for it.

This is most often done through the interactive `lxd init` which will ask the user for the value for a number of configuration keys related to storage or networks.

Those typically cover:

- Source device for a storage pool (leaving empty would create a loop)
- Name for a ZFS zpool (defaults to the name of the LXD pool)
- External interfaces for a bridged network (empty would add none)
- Name of the parent network device for managed physical or macvlan networks (must be set)

It's possible to lookup the questions ahead of time (useful for scripting) by querying the `/1.0/cluster` API endpoint. This can be done through `lxc query /1.0/cluster` or through other API clients.

Preseed

Create a preseed file for the bootstrap node with the configuration you want, for example:

```
config:
  core.trust_password: sekret
  core.https_address: 10.55.60.171:8443
  images.auto_update_interval: 15
storage_pools:
- name: default
  driver: dir
networks:
- name: lxdbr0
```

(continues on next page)

(continued from previous page)

```
  type: bridge
  config:
    ipv4.address: 192.168.100.14/24
    ipv6.address: none
profiles:
- name: default
  devices:
    root:
      path: /
      pool: default
      type: disk
    eth0:
      name: eth0
      nictype: bridged
      parent: lxdbr0
      type: nic
cluster:
  server_name: node1
  enabled: true
```

Then run `cat <preseed-file> | lxd init --preseed` and your first node should be bootstrapped.

Now create a bootstrap file for another node. You only need to fill in the `cluster` section with data and config values that are specific to the joining node.

Be sure to include the address and certificate of the target bootstrap node. To create a YAML-compatible entry for the `cluster_certificate` key you can use a command like `sed ':a;N;$!ba;s/\n/\n\n/g' /var/lib/lxd/cluster.crt` (or `sed ':a;N;$!ba;s/\n/\n\n/g' /var/snap/lxd/common/lxd/cluster.crt` for snap users), which you have to run on the bootstrap node. `cluster_certificate_path` key (which should contain valid path to cluster certificate) can be used instead of `cluster_certificate` key.

For example:

```
cluster:
  enabled: true
  server_name: node2
  server_address: 10.55.60.155:8443
  cluster_address: 10.55.60.171:8443
  cluster_certificate: "-----BEGIN CERTIFICATE-----

opyQ1VRpAg2sV2C4W8irbNqeUsTeZZxhLqp4vNOXXBBrSqUCdPu1JXADV0kavg1l

2sXYoMobyV3K+RaJgsr1OiHjacGiGCQT3YyNGGY/n5zgT/8xI0Dquvja0bNkaf6f

...

-----END CERTIFICATE-----
"
  cluster_password: sekret
  member_config:
  - entity: storage-pool
    name: default
    key: source
    value: ""
```

When joining a cluster using a cluster join token, the following fields can be omitted:

- server_name
- cluster_address
- cluster_certificate
- cluster_password

And instead the full token be passed through the `cluster_token` field.

Managing a cluster

Once your cluster is formed you can see a list of its nodes and their status by running `lxc cluster list`. More detailed information about an individual node is available with `lxc cluster show <node name>`.

Voting and stand-by members

The cluster uses a distributed *database* to store its state. All nodes in the cluster need to access such distributed database in order to serve user requests.

If the cluster has many nodes, only some of them will be picked to replicate database data. Each node that is picked can replicate data either as "voter" or as "stand-by". The database (and hence the cluster) will remain available as long as a majority of voters is online. A stand-by node will automatically be promoted to voter when another voter is shutdown gracefully or when its detected to be offline.

The default number of voting nodes is 3 and the default number of stand-by nodes is 2. This means that your cluster will remain operation as long as you switch off at most one voting node at a time.

You can change the desired number of voting and stand-by nodes with:

```
lxc config set cluster.max_voters <n>
```

and

```
lxc config set cluster.max_standby <n>
```

with the constraint that the maximum number of voters must be odd and must be least 3, while the maximum number of stand-by nodes must be between 0 and 5.

Deleting nodes

To cleanly delete a node from the cluster use `lxc cluster remove <node name>`.

Offline nodes and fault tolerance

At each time there will be an elected cluster leader that will monitor the health of the other nodes. If a node is down for more than 20 seconds, its status will be marked as OFFLINE and no operation will be possible on it, as well as operations that require a state change across all nodes.

If the node that goes offline is the leader itself, the other nodes will elect a new leader.

As soon as the offline node comes back online, operations will be available again.

If you can't or don't want to bring the node back online, you can delete it from the cluster using `lxc cluster remove --force <node name>`.

You can tweak the amount of seconds after which a non-responding node will be considered offline by running:

```
lxc config set cluster.offline_threshold <n seconds>
```

The minimum value is 10 seconds.

Upgrading nodes

To upgrade a cluster you need to upgrade all of its nodes, making sure that they all upgrade to the same version of LXD.

To upgrade a single node, simply upgrade the lxd/lxc binaries on the host (via snap or other packaging systems) and restart the lxd daemon.

If the new version of the daemon has database schema or API changes, the restarted node might transition into a Blocked state. That happens if there are still nodes in the cluster that have not been upgraded and that are running an older version. When a node is in the Blocked state it will not serve any LXD API requests (in particular, lxc commands on that node will not work, although any running instance will continue to run).

You can see if some nodes are blocked by running `lxc cluster list` on a node which is not blocked.

As you proceed upgrading the rest of the nodes, they will all transition to the Blocked state, until you upgrade the very last one. At that point the blocked nodes will notice that there is no out-of-date node left and will become operational again.

Recover from quorum loss

Every LXD cluster has up to 3 members that serve as database nodes. If you permanently lose a majority of the cluster members that are serving as database nodes (for example you have a 3-member cluster and you lose 2 members), the cluster will become unavailable. However, if at least one database node has survived, you will be able to recover the cluster.

In order to check which cluster members are configured as database nodes, log on any survived member of your cluster and run the command:

```
lxd cluster list-database
```

This will work even if the LXD daemon is not running.

Among the listed members, pick the one that has survived and log into it (if it differs from the one you have run the command on).

Now make sure the LXD daemon is not running and then issue the command:

```
lxd cluster recover-from-quorum-loss
```

At this point you can restart the LXD daemon and the database should be back online.

Note that no information has been deleted from the database, in particular all information about the cluster members that you have lost is still there, including the metadata about their instances. This can help you with further recovery steps in case you need to re-create the lost instances.

In order to permanently delete the cluster members that you have lost, you can run the command:

```
lxc cluster remove <name> --force
```

Note that this time you have to use the regular `lxc` command line tool, not `lxd`.

Recover cluster members with changed addresses

If some members of your cluster are no longer reachable, or if the cluster itself is unreachable due to a change in IP address or listening port number, the cluster can be reconfigured.

On each member of the cluster, with LXD not running, run the following command:

```
lxd cluster edit
```

Note that all commands in this section will use `lxd` instead of `lxc`.

This will present a YAML representation of this node's last recorded information about the rest of the cluster:

```
# Latest dqlite segment ID: 1234

members:
  - id: 1             # Internal ID of the node (Read-only)
    name: node1       # Name of the cluster member (Read-only)
    address: 10.0.0.10:8443 # Last known address of the node (Writeable)
    role: voter             # Last known role of the node (Writeable)
  - id: 2
    name: node2
    address: 10.0.0.11:8443
    role: stand-by
  - id: 3
    name: node3
    address: 10.0.0.12:8443
    role: spare
```

Members may not be removed from this configuration, and a spare node cannot become a voter, as it may lack a global database. Importantly, keep in mind that at least 2 nodes must remain voters (except in the case of a 2-member cluster, where 1 voter suffices), or there will be no quorum.

Once the necessary changes have been made, repeat the process on each member of the cluster. Upon reloading LXD on each member, the cluster in its entirety should be back online with all nodes reporting in.

Note that no information has been deleted from the database, all information about the cluster members and their instances is still there.

Instances

You can launch an instance on any node in the cluster from any node in the cluster. For example, from node1:

```
lxc launch --target node2 ubuntu:20.04 c1
```

will launch an Ubuntu 20.04 container on node2.

When you launch an instance without defining a target, the instance will be launched on the server which has the lowest number of instances. If all the servers have the same amount of instances, it will choose one at random.

You can list all instances in the cluster with:

```
lxc list
```

The NODE column will indicate on which node they are running.

After an instance is launched, you can operate it from any node. For example, from node1:

```
lxc exec c1 ls /
lxc stop c1
lxc delete c1
lxc pull file c1/etc/hosts .
```

Manually altering Raft membership

There might be situations in which you need to manually alter the Raft membership configuration of the cluster because some unexpected behavior occurred.

For example if you have a cluster member that was removed uncleanly it might not show up in `lxc cluster list` but still be part of the Raft configuration (you can see that with `lxd sql local "SELECT * FROM raft_nodes").

In that case you can run:

```
lxd cluster remove-raft-node <address>
```

to remove the leftover node.

Images

By default, LXD will replicate images on as many cluster members as you have database members. This typically means up to 3 copies within the cluster.

That number can be increased to improve fault tolerance and likelihood of the image being locally available.

The special value of "-1" may be used to have the image copied on all nodes.

You can disable the image replication in the cluster by setting the count down to 1:

```
lxc config set cluster.images_minimal_replica 1
```

Storage pools

As mentioned above, all nodes must have identical storage pools. The only difference between pools on different nodes might be their `source`, `size` or `zfs.pool\_name` configuration keys.

To create a new storage pool, you first have to define it across all nodes, for example:

```
lxc storage create --target node1 data zfs source=/dev/vdb1
lxc storage create --target node2 data zfs source=/dev/vdc1
```

Note that when defining a new storage pool on a node the only valid configuration keys you can pass are the node-specific ones mentioned above.

At this point the pool hasn't been actually created yet, but just defined (it's state is marked as Pending if you run `lxc storage list`).

Now run:

```
lxc storage create data zfs
```

and the storage will be instantiated on all nodes. If you didn't define it on a particular node, or a node is down, an error will be returned.

You can pass to this final `storage create` command any configuration key which is not node-specific (see above).

Storage volumes

Each volume lives on a specific node. The `lxc storage volume list` includes a `NODE` column to indicate on which node a certain volume resides.

Different volumes can have the same name as long as they live on different nodes (for example image volumes). You can manage storage volumes in the same way you do in non-clustered deployments, except that you'll have to pass a `--target <node name>` parameter to volume commands if more than one node has a volume with the given name.

For example:

```
# Create a volume on the node this client is pointing at
lxc storage volume create default web

# Create a volume with the same node on another node
lxc storage volume create default web --target node2

# Show the two volumes defined
lxc storage volume show default web --target node1
lxc storage volume show default web --target node2
```

Networks

As mentioned above, all nodes must have identical networks defined. The only difference between networks on different nodes might be their `bridge.external_interfaces` optional configuration key (see also documentation about *network configuration*).

To create a new network, you first have to define it across all nodes, for example:

```
lxc network create --target node1 my-network
lxc network create --target node2 my-network
```

Note that when defining a new network on a node the only valid configuration key you can pass is `bridge.external_interfaces`, as mentioned above.

At this point the network hasn't been actually created yet, but just defined (it's state is marked as Pending if you run `lxc network list`).

Now run:

```
lxc network create my-network
```

and the network will be instantiated on all nodes. If you didn't define it on a particular node, or a node is down, an error will be returned.

You can pass to this final `network create` command any configuration key which is not node-specific (see above).

Separate REST API and clustering networks

You can configure different networks for the REST API endpoint of your clients and for internal traffic between the nodes of your cluster (for example in order to use a virtual address for your REST API, with DNS round robin).

To do that, you need to bootstrap the first node of the cluster using the `cluster.https_address` config key. For example, when using preseed:

```
config:
  core.trust_password: sekret
  core.https_address: my.lxd.cluster:8443
  cluster.https_address: 10.55.60.171:8443
...
```

(the rest of the preseed YAML is the same as above).

To join a new node, first set its REST API address, for instance using the `lxc` client:

```
lxc config set core.https_address my.lxd.cluster:8443
```

and then use the `PUT /1.0/cluster` API endpoint as usual, specifying the address of the joining node with the `server_address` field. If you use preseed, the YAML payload would be exactly like the one above.

Updating the cluster certificate

In a LXD cluster, all servers respond with the same shared certificate. This is usually a standard self-signed certificate with an expiry set to 10 years.

If you wish to replace it with something else, for example a valid certificate obtained through Let's Encrypt, `lxc cluster update-certificate` can be used to replace the certificate on all servers in your cluster.

3.4.3 Instance command execution

LXD makes it easy to run a command inside a given instance. For containers, this always works and is handled directly by LXD. For virtual machines, this relies on the `lxd-agent` process running inside of the virtual machine.

At the CLI level, this is achieved through the `lxc exec` command which supports specifying not only the command to executed but also the execution mode, user, group and working directory.

At the API level, this is done through `/1.0/instances/NAME/exec`.

Execution mode

LXD can execute commands either interactively or non-interactively.

In interactive mode, a pseudo-terminal device (PTS) will be used to handle input (stdin) and output (stdout, stderr). This is automatically selected by the CLI if connected to a terminal emulator (not run from a script).

In non-interactive mode, pipes are allocated instead, one for each of stdin, stdout and stderr. This allows running a command and properly getting separate stdin, stdout and stderr as required by many scripts.

User, groups and working directory

LXD has a policy not to read data from within the instances or trusting anything that can be found in it. This means that LXD will not be parsing things like `/etc/passwd`, `/etc/group` or `/etc/nsswitch.conf` to handle user and group resolution.

As a result, LXD also doesn't know where the home directory for the user may be or what supplementary groups the user may be in.

By default, LXD will run the command as root (uid 0) with the default group (gid 0) and the working directory set to /root.

The user, group and working directory can all be overridden but absolute values (uid, gid, path) have to be provided as LXD will not do any resolution for you.

Environment

The environment variables set during an exec session come from a few sources:

- `environment.KEY=VALUE` directly set on the instance
- Environment variables directly passed during the exec session
- Default variables set by LXD

For that last category, LXD will set the `PATH` to `/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin` and extend it with `/snap` and `/etc/NIXOS` if applicable. Additionally `LANG` will be set to `C.UTF-8`.

When running as root (uid 0), the following variables will also be set:

- `HOME` to `/root`
- `USER` to `root`

When running as another user, it is the responsibility of the user to specify the correct values.

Those defaults only get set if they're not in the instance configuration or directly overridden for the exec session.

3.4.4 Production setup

Introduction

So you've made it past trying out LXD live online, or on a server scavenged from random parts. You like what you see, and now you want to try doing some serious work with LXD.

The vast majority of Linux distributions do not come with optimized kernel settings suitable for the operation of a large number of containers. The instructions in this document cover the most common limits that you're likely to hit when running containers and suggested updated values.

Common errors that may be encountered

```
Failed to allocate directory watch:  Too many open files
<Error> <Error>:  Too many open files
failed to open stream:  Too many open files in...
neighbour:  ndisc_cache:  neighbor table overflow!
```

Server Changes

/etc/security/limits.conf

Domain	Type	Item	Value	Default	Description
*	soft	nofile	1048576	unset	maximum number of open files
*	hard	nofile	1048576	unset	maximum number of open files
root	soft	nofile	1048576	unset	maximum number of open files
root	hard	nofile	1048576	unset	maximum number of open files
*	soft	memlock	unlimited	unset	maximum locked-in-memory address space (KB)
*	hard	memlock	unlimited	unset	maximum locked-in-memory address space (KB)

NOTE: For users of the snap, those ulimits are automatically raised by the snap/LXD.

/etc/sysctl.conf

Parameter	Value	Default	Description
fs.aio-max-nr	524288	65536	This is the maximum number of concurrent async I/O operations. You might need to increase it further if you have a lot of workloads that use the AIO subsystem (e.g. MySQL)
fs.inotify.max_queued_events	1048576	16384	This specifies an upper limit on the number of events that can be queued to the corresponding inotify instance. [1]
fs.inotify.max_user_instances	1048576	128	This specifies an upper limit on the number of inotify instances that can be created per real user ID. [1]
fs.inotify.max_user_watches	1048576	8192	This specifies an upper limit on the number of watches that can be created per real user ID. [1]
kernel.dmesg_restrict	1	0	This denies container access to the messages in the kernel ring buffer. Please note that this also will deny access to non-root users on the host system.
kernel.keys.maxbytes	2000000	20000	This is the maximum size of the keyring non-root users can use
kernel.keys.maxkeys	2000	200	This is the maximum number of keys a non-root user can use, should be higher than the number of containers
net.ipv4.neigh.default.gc_thresh3	8192	1024	This is the maximum number of entries in ARP table (IPv4). You should increase this if you create over 1024 containers. Otherwise, you will get the error `neighbour: ndisc_cache: neighbor table overflow!` when the ARP table gets full and those containers will not be able to get a network configuration. [2]
net.ipv6.neigh.default.gc_thresh3	8192	1024	This is the maximum number of entries in ARP table (IPv6). You should increase this if you plan to create over 1024 containers. Otherwise, you will get the error `neighbour: ndisc_cache: neighbor table overflow!` when the ARP table gets full and those containers will not be able to get a network configuration. [2]
vm.max_map_count	262144	65530	This file contains the maximum number of memory map areas a process may have. Memory map areas are used as a side-effect of calling malloc, directly by mmap and mprotect, and also when loading shared libraries.

Then, reboot the server.

Prevent container name leakage

Both /sys/kernel/slab and /proc/sched_debug make it easy to list all cgroups on the system and by extension, all containers.

If this is something you'd like to see blocked, make sure you have the following done before any container is started:

- chmod 400 /proc/sched_debug
- chmod 700 /sys/kernel/slab/

Network Bandwidth Tweaking

If you have at least 1GbE NIC on your lxd host with a lot of local activity (container - container connections, or host - container connections), or you have 1GbE or better internet connection on your lxd host it worth play with txqueuelen. These settings work even better with 10GbE NIC.

Server Changes

txqueuelen

You need to change `txqueuelen` of your real NIC to 10000 (not sure about the best possible value for you), and change and change lxdbr0 interface `txqueuelen` to 10000.

In Debian-based distros you can change `txqueuelen` permanently in `/etc/network/interfaces`
You can add for ex.: `up ip link set eth0 txqueuelen 10000` to your interface configuration to set txqueuelen value on boot.
You could set it txqueuelen temporary (for test purpose) with `ifconfig <interface> txqueuelen 10000`

/etc/sysctl.conf

You also need to increase `net.core.netdev_max_backlog` value.
You can add `net.core.netdev_max_backlog = 182757` to `/etc/sysctl.conf` to set it permanently (after reboot) You set `netdev_max_backlog` temporary (for test purpose) with `echo 182757 > /proc/sys/net/core/netdev_max_backlog` Note: You can find this value too high, most people prefer set `netdev_max_backlog` = `net.ipv4.tcp_mem` min. value. For example I use this values `net.ipv4.tcp_mem = 182757 243679 365514`

Containers changes

You also need to change txqueuelen value for all you ethernet interfaces in containers.
In Debian-based distros you can change txqueuelen permanently in `/etc/network/interfaces`
You can add for ex.: `up ip link set eth0 txqueuelen 10000` to your interface configuration to set txqueuelen value on boot.

Notes regarding this change

10000 txqueuelen value commonly used with 10GbE NICs. Basically small txqueuelen values used with slow devices with a high latency, and higher with devices with low latency. I personally have like 3-5% improvement with these settings for local (host with container, container vs container) and internet connections. Good thing about txqueuelen value tweak, the more containers you use, the more you can be can benefit from this tweak. And you can always temporary set this values and check this tweak in your environment without lxd host reboot.

3.4.5 Remote API authentication

Remote communications with the LXD daemon happen using JSON over HTTPS.

To be able to access the remote API, clients must authenticate with the LXD server. The following authentication methods are supported:

- *TLS client certificates*
- *Candid-based authentication*
- *Role Based Access Control (RBAC)*

TLS client certificates

When using TLS client certificates for authentication, both the client and the server will generate a key pair the first time they're launched. The server will use that key pair for all HTTPS connections to the LXD socket. The client will use its certificate as a client certificate for any client-server communication.

To cause certificates to be regenerated, simply remove the old ones. On the next connection, a new certificate is generated.

Communication protocol

The supported protocol must be TLS 1.2 or better. All communications must use perfect forward secrecy, and ciphers must be limited to strong elliptic curve ones (such as ECDHE-RSA or ECDHE-ECDSA).

Any generated key should be at least 4096 bit RSA, preferably EC384. When using signatures, only SHA-2 signatures should be trusted.

Since we control both client and server, there is no reason to support any backward compatibility to broken protocol or ciphers.

Trusted TLS clients

You can obtain the list of TLS certificates trusted by a LXD server with `lxc config trust list`.

Trusted clients can be added in either of the following ways:

- *Adding trusted certificates to the server*
- *Adding client certificates using a trust password*

The workflow to authenticate with the server is similar to that of SSH, where an initial connection to an unknown server triggers a prompt:

1. When the user adds a server with `lxc remote add`, the server is contacted over HTTPS, its certificate is downloaded and the fingerprint is shown to the user.
2. The user is asked to confirm that this is indeed the server's fingerprint, which they can manually check by connecting to the server or by asking someone with access to the server to run the info command and compare the fingerprints.
3. The server attempts to authenticate the client:
    - If the client certificate is in the server's trust store, the connection is granted.

- If the client certificate is not in the server's trust store and a trust password is set, the server prompts the user for the trust password. If the provided trust password matches, the client certificate is added to the server's trust store and the connection is granted. Otherwise, the connection is rejected.
- If the client certificate is not in the server's trust store and no trust password is set, the connection is rejected.

To revoke trust to a client, remove its certificate from the server with `lxc config trust remove FINGERPRINT`.

It's possible to restrict a TLS client to one or multiple projects. In this case, the client will also be prevented from performing global configuration changes or altering the configuration (limits, restrictions) of the projects it's allowed access to.

To restrict access, use `lxc config trust edit FINGERPRINT`. Set the `restricted` key to `true` and specify a list of projects to restrict the client to. If the list of projects is empty, the client will not be allowed access to any of them.

Adding trusted certificates to the server

The preferred way to add trusted clients is to directly add their certificates to the trust store on the server. To do so, copy the client certificate to the server and register it using `lxc config trust add <file>`.

Adding client certificates using a trust password

To allow establishing a new trust relationship from the client side, you must set a trust password (`core.trust_password`, see *Server configuration*) for the server. Clients can then add their own certificate to the server's trust store by providing the trust password when prompted.

In a production setup, unset `core.trust_password` after all clients have been added. This prevents brute-force attacks trying to guess the password.

Using a PKI system

In a PKI (Public key infrastructure) setup, a system administrator manages a central PKI that issues client certificates for all the lxc clients and server certificates for all the LXD daemons.

To enable PKI mode, complete the following steps:

1. Add the CA (Certificate authority) certificate to all machines:
   - Place the `client.ca` file in the clients' configuration directories (`~/.config/lxc`).
   - Place the `server.ca` file in the server's configuration directory (`/var/lib/lxd` or `/var/snap/lxd/common/lxd` for snap users).
2. Place the certificates issued by the CA on the clients and the server, replacing the automatically generated ones.
3. Restart the server.

In that mode, any connection to a LXD daemon will be done using the preseeded CA certificate.

If the server certificate isn't signed by the CA, the connection will simply go through the normal authentication mechanism. If the server certificate is valid and signed by the CA, then the connection continues without prompting the user for the certificate.

Note that the generated certificates are not automatically trusted. You must still add them to the server in one of the ways described in *Trusted TLS clients*.

Candid-based authentication

When LXD is configured to use Candid authentication, clients that try to authenticate with the server must get a Discharge token from the authentication server specified by the `candid.api.url` setting (see *Server configuration*).

The authentication server certificate must be trusted by the LXD server.

To add a remote pointing to a LXD server configured with Candid/Macaroon authentication, run `lxc remote add REMOTE ENDPOINT --auth-type=candid`. To verify the user, the client will prompt for the credentials required by the authentication server. If the authentication is successful, the client will connect to the LXD server and present the token received from the authentication server. The LXD server verifies the token, thus authenticating the request. The token is stored as cookie and is presented by the client at each request to LXD.

For instructions on how to set up Candid-based authentication, see the Candid authentication for LXD tutorial.

Role Based Access Control (RBAC)

LXD supports integrating with the Canonical RBAC service. Combined with Candid-based authentication, RBAC (Role Based Access Control) can be used to limit what an API client is allowed to do on LXD.

In such a setup, authentication happens through Candid, while the RBAC service maintains roles to user/group relationships. Roles can be assigned to individual projects, to all projects or to the entire LXD instance.

The meaning of the roles when applied to a project is as follows:

- auditor: Read-only access to the project
- user: Ability to do normal life cycle actions (start, stop, ...), execute commands in the instances, attach to console, manage snapshots, ...
- operator: All of the above + the ability to create, re-configure and delete instances and images
- admin: All of the above + the ability to reconfigure the project itself

---

Important: In an unrestricted project, only the `auditor` and the `user` roles are suitable for users that you wouldn't trust with root access to the host.

In a *restricted project*, the `operator` role is safe to use as well if configured appropriately.

Failure scenarios

In the following scenarios, authentication is expected to fail.

Server certificate changed

The server certificate might change in the following cases:

- The server was fully reinstalled and therefore got a new certificate.
- The connection is being intercepted (MITM (Man in the middle)).

In such cases, the client will refuse to connect to the server because the certificate fingerprint does not match the fingerprint in the configuration for this remote.

It is then up to the user to contact the server administrator to check if the certificate did in fact change. If it did, the certificate can be replaced by the new one, or the remote can be removed altogether and re-added.

Server trust relationship revoked

The server trust relationship is revoked for a client if another trusted client or the local server administrator removes the trust entry for the client on the server.

In this case, the server still uses the same certificate, but all API calls return a 403 code with an error indicating that the client isn't trusted.

3.5 REST API

3.5.1 REST API

Introduction

All the communications between LXD and its clients happen using a RESTful API over http which is then encapsulated over either SSL for remote operations or a unix socket for local operations.

API versioning

The list of supported major API versions can be retrieved using `GET /`.

The reason for a major API bump is if the API breaks backward compatibility.

Feature additions done without breaking backward compatibility only result in addition to `api_extensions` which can be used by the client to check if a given feature is supported by the server.

Return values

There are three standard return types:

- Standard return value
- Background operation
- Error

Standard return value

For a standard synchronous operation, the following dict is returned:

```
{
    "type": "sync",
    "status": "Success",
    "status_code": 200,
    "metadata": {}                          // Extra resource/action specific metadata
}
```

HTTP code must be 200.

Background operation

When a request results in a background operation, the HTTP code is set to 202 (Accepted) and the Location HTTP header is set to the operation URL.

The body is a dict with the following structure:

```
{
    "type": "async",
    "status": "OK",
    "status_code": 100,
    "operation": "/1.0/instances/<id>",                    // URL to the background␣
↪operation
    "metadata": {}                                         // Operation metadata (see␣
↪below)
}
```

The operation metadata structure looks like:

```
{
    "id": "a40f5541-5e98-454f-b3b6-8a51ef5dbd3c",          // UUID of the operation
    "class": "websocket",                                  // Class of the operation␣
↪(task, websocket or token)
    "created_at": "2015-11-17T22:32:02.226176091-05:00",   // When the operation was␣
↪created
    "updated_at": "2015-11-17T22:32:02.226176091-05:00",   // Last time the operation␣
↪was updated
    "status": "Running",                                   // String version of the␣
↪operation's status
    "status_code": 103,                                    // Integer version of the␣
↪operation's status (use this rather than status)
    "resources": {                                         // Dictionary of resource␣
↪types (container, snapshots, images) and affected resources
      "containers": [
        "/1.0/instances/test"
      ]
    },
    "metadata": {                                          // Metadata specific to the␣
↪operation in question (in this case, exec)
      "fds": {
        "0": "2a4a97af81529f6608dca31f03a7b7e47acc0b8dc6514496eb25e325f9e4fa6a",
        "control": "5b64c661ef313b423b5317ba9cb6410e40b705806c28255f601c0ef603f079a7"
      }
    },
    "may_cancel": false,                                   // Whether the operation can␣
↪be canceled (DELETE over REST)
    "err": ""                                              // The error string should␣
↪the operation have failed
}
```

The body is mostly provided as a user friendly way of seeing what's going on without having to pull the target operation, all information in the body can also be retrieved from the background operation URL.

Error

There are various situations in which something may immediately go wrong, in those cases, the following return value is used:

```
{
    "type": "error",
    "error": "Failure",
    "error_code": 400,
    "metadata": {}                      // More details about the error
}
```

HTTP code must be one of of 400, 401, 403, 404, 409, 412 or 500.

Status codes

The LXD REST API often has to return status information, be that the reason for an error, the current state of an operation or the state of the various resources it exports.

To make it simple to debug, all of those are always doubled. There is a numeric representation of the state which is guaranteed never to change and can be relied on by API clients. Then there is a text version meant to make it easier for people manually using the API to figure out what's happening.

In most cases, those will be called status and `status_code`, the former being the user-friendly string representation and the latter the fixed numeric value.

The codes are always 3 digits, with the following ranges:

- 100 to 199: resource state (started, stopped, ready, ...)
- 200 to 399: positive action result
- 400 to 599: negative action result
- 600 to 999: future use

List of current status codes

Code	Meaning
100	Operation created
101	Started
102	Stopped
103	Running
104	Cancelling
105	Pending
106	Starting
107	Stopping
108	Aborting
109	Freezing
110	Frozen
111	Thawed
112	Error
200	Success
400	Failure
401	Cancelled

Recursion

To optimize queries of large lists, recursion is implemented for collections. A `recursion` argument can be passed to a GET query against a collection.

The default value is 0 which means that collection member URLs are returned. Setting it to 1 will have those URLs be replaced by the object they point to (typically a dict).

Recursion is implemented by simply replacing any pointer to an job (URL) by the object itself.

Filtering

To filter your results on certain values, filter is implemented for collections. A `filter` argument can be passed to a GET query against a collection.

Filtering is available for the instance and image endpoints.

There is no default value for filter which means that all results found will be returned. The following is the language used for the filter argument:

?filter=field_name eq desired_field_assignment

The language follows the OData conventions for structuring REST API filtering logic. Logical operators are also supported for filtering: not(not), equals(eq), not equals(ne), and(and), or(or). Filters are evaluated with left associativity. Values with spaces can be surrounded with quotes. Nesting filtering is also supported. For instance, to filter on a field in a config you would pass:

?filter=config.field_name eq desired_field_assignment

For filtering on device attributes you would pass:

?filter=devices.device_name.field_name eq desired_field_assignment

Here are a few GET query examples of the different filtering methods mentioned above:

containers?filter=name eq “my container” and status eq Running

containers?filter=config.image.os eq ubuntu or devices.eth0.nictype eq bridged

images?filter=Properties.os eq Centos and not UpdateSource.Protocol eq simplestreams

Async operations

Any operation which may take more than a second to be done must be done in the background, returning a background operation ID to the client.

The client will then be able to either poll for a status update or wait for a notification using the long-poll API.

Notifications

A websocket based API is available for notifications, different notification types exist to limit the traffic going to the client.

It’s recommended that the client always subscribes to the operations notification type before triggering remote operations so that it doesn’t have to then poll for their status.

PUT vs PATCH

The LXD API supports both PUT and PATCH to modify existing objects.

PUT replaces the entire object with a new definition, it's typically called after the current object state was retrieved through GET.

To avoid race conditions, the Etag header should be read from the GET response and sent as If-Match for the PUT request. This will cause LXD to fail the request if the object was modified between GET and PUT.

PATCH can be used to modify a single field inside an object by only specifying the property that you want to change. To unset a key, setting it to empty will usually do the trick, but there are cases where PATCH won't work and PUT needs to be used instead.

Instances, containers and virtual-machines

This documentation will always show paths such as `/1.0/instances/...`. Those are fairly new, introduced with LXD 3.19 when virtual-machine support.

Older releases that only supported containers will instead use the exact same API at `/1.0/containers/...`.

For backward compatibility reasons, LXD does still expose and support that `/1.0/containers` API, though for the sake of brevity, we decided not to double-document everything below.

An additional endpoint at `/1.0/virtual-machines` is also present and much like `/1.0/containers` will only show you instances of that type.

API structure

LXD has an auto-generated Swagger specification describing its API endpoints. The YAML version of this API specification can be found in `rest-api.yaml`. See *Main API specification* for a convenient web rendering of it.

3.5.2 Main API specification

3.5.3 API extensions

The changes below were introduced to the LXD API after the 1.0 API was finalized.

They are all backward compatible and can be detected by client tools by looking at the `api_extensions` field in `GET /1.0/`.

storage_zfs_remove_snapshots

A `storage.zfs_remove_snapshots` daemon configuration key was introduced.

It's a boolean that defaults to false and that when set to true instructs LXD to remove any needed snapshot when attempting to restore another.

This is needed as ZFS will only let you restore the latest snapshot.

container_host_shutdown_timeout

A `boot.host_shutdown_timeout` container configuration key was introduced.

It's an integer which indicates how long LXD should wait for the container to stop before killing it.

Its value is only used on clean LXD daemon shutdown. It defaults to 30s.

container_stop_priority

A `boot.stop.priority` container configuration key was introduced.

It's an integer which indicates the priority of a container during shutdown.

Containers will shutdown starting with the highest priority level.

Containers with the same priority will shutdown in parallel. It defaults to 0.

container_syscall_filtering

A number of new syscalls related container configuration keys were introduced.

- `security.syscalls.blacklist_default`
- `security.syscalls.blacklist_compat`
- `security.syscalls.blacklist`
- `security.syscalls.whitelist`

See *Instance configuration* for how to use them.

auth_pki

This indicates support for PKI authentication mode.

In this mode, the client and server both must use certificates issued by the same PKI.

See *security.md* for details.

container_last_used_at

A `last_used_at` field was added to the `GET /1.0/containers/<name>` endpoint.

It is a timestamp of the last time the container was started.

If a container has been created but not started yet, `last_used_at` field will be `1970-01-01T00:00:00Z`

etag

Add support for the ETag header on all relevant endpoints.

This adds the following HTTP header on answers to GET:

- ETag (SHA-256 of user modifiable content)

And adds support for the following HTTP header on PUT requests:

- If-Match (ETag value retrieved through previous GET)

This makes it possible to GET a LXD object, modify it and PUT it without risking to hit a race condition where LXD or another client modified the object in the meantime.

patch

Add support for the HTTP PATCH method.

PATCH allows for partial update of an object in place of PUT.

usb_devices

Add support for USB hotplug.

https_allowed_credentials

To use LXD API with all Web Browsers (via SPAs) you must send credentials (certificate) with each XHR (in order for this to happen, you should set "withCredentials=true" flag to each XHR Request).

Some browsers like Firefox and Safari can't accept server response without `Access-Control-Allow-Credentials: true` header. To ensure that the server will return a response with that header, set `core.https_allowed_credentials=true`.

image_compression_algorithm

This adds support for a `compression_algorithm` property when creating an image (`POST /1.0/images`).

Setting this property overrides the server default value (`images.compression_algorithm`).

directory_manipulation

This allows for creating and listing directories via the LXD API, and exports the file type via the X-LXD-type header, which can be either "file" or "directory" right now.

container_cpu_time

This adds support for retrieving cpu time for a running container.

storage_zfs_use_refquota

Introduces a new server property `storage.zfs_use_refquota` which instructs LXD to set the "refquota" property instead of "quota" when setting a size limit on a container. LXD will also then use "usedbydataset" in place of "used" when being queried about disk utilization.

This effectively controls whether disk usage by snapshots should be considered as part of the container's disk space usage.

storage_lvm_mount_options

Adds a new `storage.lvm_mount_options` daemon configuration option which defaults to "discard" and allows the user to set addition mount options for the filesystem used by the LVM LV.

network

Network management API for LXD.

This includes:

- Addition of the "managed" property on `/1.0/networks` entries
- All the network configuration options (see *Network configuration* for details)
- `POST /1.0/networks` (see *RESTful API* for details)
- `PUT /1.0/networks/<entry>` (see *RESTful API* for details)
- `PATCH /1.0/networks/<entry>` (see *RESTful API* for details)
- `DELETE /1.0/networks/<entry>` (see *RESTful API* for details)
- `ipv4.address` property on "nic" type devices (when nictype is "bridged")
- `ipv6.address` property on "nic" type devices (when nictype is "bridged")
- `security.mac_filtering` property on "nic" type devices (when nictype is "bridged")

profile_usedby

Adds a new used_by field to profile entries listing the containers that are using it.

container_push

When a container is created in push mode, the client serves as a proxy between the source and target server. This is useful in cases where the target server is behind a NAT or firewall and cannot directly communicate with the source server and operate in pull mode.

container_exec_recording

Introduces a new boolean "record-output", parameter to `/1.0/containers/<name>/exec` which when set to "true" and combined with with "wait-for-websocket" set to false, will record stdout and stderr to disk and make them available through the logs interface.

The URL to the recorded output is included in the operation metadata once the command is done running.

That output will expire similarly to other log files, typically after 48 hours.

certificate_update

Adds the following to the REST API:

- ETag header on GET of a certificate
- PUT of certificate entries
- PATCH of certificate entries

container_exec_signal_handling

Adds support `/1.0/containers/<name>/exec` for forwarding signals sent to the client to the processes executing in the container. Currently SIGTERM and SIGHUP are forwarded. Further signals that can be forwarded might be added later.

gpu_devices

Enables adding GPUs to a container.

container_image_properties

Introduces a new `image` config key space. Read-only, includes the properties of the parent image.

migration_progress

Transfer progress is now exported as part of the operation, on both sending and receiving ends. This shows up as a "fs_progress" attribute in the operation metadata.

id_map

Enables setting the `security.idmap.isolated` and `security.idmap.isolated`, `security.idmap.size`, and `raw.id_map` fields.

network_firewall_filtering

Add two new keys, `ipv4.firewall` and `ipv6.firewall` which if set to false will turn off the generation of iptables FORWARDING rules. NAT rules will still be added so long as the matching `ipv4.nat` or `ipv6.nat` key is set to true.

Rules necessary for dnsmasq to work (DHCP/DNS) will always be applied if dnsmasq is enabled on the bridge.

network_routes

Introduces `ipv4.routes` and `ipv6.routes` which allow routing additional subnets to a LXD bridge.

storage

Storage management API for LXD.

This includes:

- `GET /1.0/storage-pools`
- `POST /1.0/storage-pools` (see *RESTful API* for details)
- `GET /1.0/storage-pools/<name>` (see *RESTful API* for details)
- `POST /1.0/storage-pools/<name>` (see *RESTful API* for details)
- `PUT /1.0/storage-pools/<name>` (see *RESTful API* for details)
- `PATCH /1.0/storage-pools/<name>` (see *RESTful API* for details)
- `DELETE /1.0/storage-pools/<name>` (see *RESTful API* for details)
- `GET /1.0/storage-pools/<name>/volumes` (see *RESTful API* for details)
- `GET /1.0/storage-pools/<name>/volumes/<volume_type>` (see *RESTful API* for details)
- `POST /1.0/storage-pools/<name>/volumes/<volume_type>` (see *RESTful API* for details)
- `GET /1.0/storage-pools/<pool>/volumes/<volume_type>/<name>` (see *RESTful API* for details)
- `POST /1.0/storage-pools/<pool>/volumes/<volume_type>/<name>` (see *RESTful API* for details)
- `PUT /1.0/storage-pools/<pool>/volumes/<volume_type>/<name>` (see *RESTful API* for details)
- `PATCH /1.0/storage-pools/<pool>/volumes/<volume_type>/<name>` (see *RESTful API* for details)
- `DELETE /1.0/storage-pools/<pool>/volumes/<volume_type>/<name>` (see *RESTful API* for details)
- All storage configuration options (see *Storage configuration* for details)

file_delete

Implements `DELETE` in `/1.0/containers/<name>/files`

file_append

Implements the `X-LXD-write` header which can be one of `overwrite` or `append`.

network_dhcp_expiry

Introduces `ipv4.dhcp.expiry` and `ipv6.dhcp.expiry` allowing to set the DHCP lease expiry time.

storage_lvm_vg_rename

Introduces the ability to rename a volume group by setting `storage.lvm.vg_name`.

storage_lvm_thinpool_rename

Introduces the ability to rename a thinpool name by setting `storage.thinpool_name`.

network_vlan

This adds a new `vlan` property to `macvlan` network devices.

When set, this will instruct LXD to attach to the specified VLAN. LXD will look for an existing interface for that VLAN on the host. If one can't be found it will create one itself and then use that as the macvlan parent.

image_create_aliases

Adds a new `aliases` field to `POST /1.0/images` allowing for aliases to be set at image creation/import time.

container_stateless_copy

This introduces a new `live` attribute in `POST /1.0/containers/<name>`. Setting it to false tells LXD not to attempt running state transfer.

container_only_migration

Introduces a new boolean `container_only` attribute. When set to true only the container will be copied or moved.

storage_zfs_clone_copy

Introduces a new boolean `storage_zfs_clone_copy` property for ZFS storage pools. When set to false copying a container will be done through zfs send and receive. This will make the target container independent of its source container thus avoiding the need to keep dependent snapshots in the ZFS pool around. However, this also entails less efficient storage usage for the affected pool. The default value for this property is true, i.e. space-efficient snapshots will be used unless explicitly set to "false".

unix_device_rename

Introduces the ability to rename the unix-block/unix-char device inside container by setting `path`, and the `source` attribute is added to specify the device on host. If `source` is set without a `path`, we should assume that `path` will be the same as `source`. If `path` is set without `source` and `major`/`minor` isn't set, we should assume that `source` will be the same as `path`. So at least one of them must be set.

storage_rsync_bwlimit

When rsync has to be invoked to transfer storage entities setting `rsync.bwlimit` places an upper limit on the amount of socket I/O allowed.

network_vxlan_interface

This introduces a new `tunnel.NAME.interface` option for networks.

This key control what host network interface is used for a VXLAN tunnel.

storage_btrfs_mount_options

This introduces the `btrfs.mount_options` property for btrfs storage pools.

This key controls what mount options will be used for the btrfs storage pool.

entity_description

This adds descriptions to entities like containers, snapshots, networks, storage pools and volumes.

image_force_refresh

This allows forcing a refresh for an existing image.

storage_lvm_lv_resizing

This introduces the ability to resize logical volumes by setting the `size` property in the containers root disk device.

id_map_base

This introduces a new `security.idmap.base` allowing the user to skip the map auto-selection process for isolated containers and specify what host uid/gid to use as the base.

file_symlinks

This adds support for transferring symlinks through the file API. X-LXD-type can now be "symlink" with the request content being the target path.

container_push_target

This adds the `target` field to `POST /1.0/containers/<name>` which can be used to have the source LXD host connect to the target during migration.

network_vlan_physical

Allows use of `vlan` property with `physical` network devices.

When set, this will instruct LXD to attach to the specified VLAN on the `parent` interface. LXD will look for an existing interface for that `parent` and VLAN on the host. If one can't be found it will create one itself. Then, LXD will directly attach this interface to the container.

storage_images_delete

This enabled the storage API to delete storage volumes for images from a specific storage pool.

container_edit_metadata

This adds support for editing a container metadata.yaml and related templates via API, by accessing urls under `/1.0/containers/<name>/metadata`. It can be used to edit a container before publishing an image from it.

container_snapshot_stateful_migration

This enables migrating stateful container snapshots to new containers.

storage_driver_ceph

This adds a ceph storage driver.

storage_ceph_user_name

This adds the ability to specify the ceph user.

instance_types

This adds the `instance_type` field to the container creation request. Its value is expanded to LXD resource limits.

storage_volatile_initial_source

This records the actual source passed to LXD during storage pool creation.

storage_ceph_force_osd_reuse

This introduces the `ceph.osd.force_reuse` property for the ceph storage driver. When set to `true` LXD will reuse a osd storage pool that is already in use by another LXD instance.

storage_block_filesystem_btrfs

This adds support for btrfs as a storage volume filesystem, in addition to ext4 and xfs.

resources

This adds support for querying a LXD daemon for the system resources it has available.

kernel_limits

This adds support for setting process limits such as maximum number of open files for the container via `nofile`. The format is `limits.kernel.[limit name]`.

storage_api_volume_rename

This adds support for renaming custom storage volumes.

external_authentication

This adds support for external authentication via Macaroons.

network_sriov

This adds support for SR-IOV enabled network devices.

console

This adds support to interact with the container console device and console log.

restrict_devlxd

A new security.devlxd container configuration key was introduced. The key controls whether the /dev/lxd interface is made available to the container. If set to false, this effectively prevents the container from interacting with the LXD daemon.

migration_pre_copy

This adds support for optimized memory transfer during live migration.

infiniband

This adds support to use infiniband network devices.

maas_network

This adds support for MAAS network integration.

When configured at the daemon level, it's then possible to attach a "nic" device to a particular MAAS subnet.

devlxd_events

This adds a websocket API to the devlxd socket.

When connecting to /1.0/events over the devlxd socket, you will now be getting a stream of events over websocket.

proxy

This adds a new `proxy` device type to containers, allowing forwarding of connections between the host and container.

network_dhcp_gateway

Introduces a new ipv4.dhcp.gateway network config key to set an alternate gateway.

file_get_symlink

This makes it possible to retrieve symlinks using the file API.

network_leases

Adds a new /1.0/networks/NAME/leases API endpoint to query the lease database on bridges which run a LXD-managed DHCP server.

unix_device_hotplug

This adds support for the "required" property for unix devices.

storage_api_local_volume_handling

This add the ability to copy and move custom storage volumes locally in the same and between storage pools.

operation_description

Adds a "description" field to all operations.

clustering

Clustering API for LXD.

This includes the following new endpoints (see *RESTful API* for details):

- `GET /1.0/cluster`
- `UPDATE /1.0/cluster`
- `GET /1.0/cluster/members`
- `GET /1.0/cluster/members/<name>`
- `POST /1.0/cluster/members/<name>`
- `DELETE /1.0/cluster/members/<name>`

The following existing endpoints have been modified:

- `POST /1.0/containers` accepts a new target query parameter
- `POST /1.0/storage-pools` accepts a new target query parameter
- `GET /1.0/storage-pool/<name>` accepts a new target query parameter
- `POST /1.0/storage-pool/<pool>/volumes/<type>` accepts a new target query parameter
- `GET /1.0/storage-pool/<pool>/volumes/<type>/<name>` accepts a new target query parameter
- `POST /1.0/storage-pool/<pool>/volumes/<type>/<name>` accepts a new target query parameter
- `PUT /1.0/storage-pool/<pool>/volumes/<type>/<name>` accepts a new target query parameter
- `PATCH /1.0/storage-pool/<pool>/volumes/<type>/<name>` accepts a new target query parameter
- `DELETE /1.0/storage-pool/<pool>/volumes/<type>/<name>` accepts a new target query parameter
- `POST /1.0/networks` accepts a new target query parameter
- `GET /1.0/networks/<name>` accepts a new target query parameter

event_lifecycle

This adds a new `lifecycle` message type to the events API.

storage_api_remote_volume_handling

This adds the ability to copy and move custom storage volumes between remote.

nvidia_runtime

Adds a `nvidia_runtime` config option for containers, setting this to true will have the NVIDIA runtime and CUDA libraries passed to the container.

container_mount_propagation

This adds a new "propagation" option to the disk device type, allowing the configuration of kernel mount propagation.

container_backup

Add container backup support.

This includes the following new endpoints (see *RESTful API* for details):

- `GET /1.0/containers/<name>/backups`
- `POST /1.0/containers/<name>/backups`
- `GET /1.0/containers/<name>/backups/<name>`
- `POST /1.0/containers/<name>/backups/<name>`
- `DELETE /1.0/containers/<name>/backups/<name>`
- `GET /1.0/containers/<name>/backups/<name>/export`

The following existing endpoint has been modified:

- `POST /1.0/containers` accepts the new source type `backup`

devlxd_images

Adds a `security.devlxd.images` config option for containers which controls the availability of a `/1.0/images/FINGERPRINT/export` API over devlxd. This can be used by a container running nested LXD to access raw images from the host.

container_local_cross_pool_handling

This enables copying or moving containers between storage pools on the same LXD instance.

proxy_unix

Add support for both unix sockets and abstract unix sockets in proxy devices. They can be used by specifying the address as `unix:/path/to/unix.sock` (normal socket) or `unix:@/tmp/unix.sock` (abstract socket).

Supported connections are now:

- `TCP <-> TCP`
- `UNIX <-> UNIX`
- `TCP <-> UNIX`
- `UNIX <-> TCP`

proxy_udp

Add support for udp in proxy devices.

Supported connections are now:

- `TCP <-> TCP`
- `UNIX <-> UNIX`
- `TCP <-> UNIX`
- `UNIX <-> TCP`
- `UDP <-> UDP`
- `TCP <-> UDP`
- `UNIX <-> UDP`

clustering_join

This makes GET /1.0/cluster return information about which storage pools and networks are required to be created by joining nodes and which node-specific configuration keys they are required to use when creating them. Likewise the PUT /1.0/cluster endpoint now accepts the same format to pass information about storage pools and networks to be automatically created before attempting to join a cluster.

proxy_tcp_udp_multi_port_handling

Adds support for forwarding traffic for multiple ports. Forwarding is allowed between a range of ports if the port range is equal for source and target (for example `1.2.3.4 0-1000 -> 5.6.7.8 1000-2000`) and between a range of source ports and a single target port (for example `1.2.3.4 0-1000 -> 5.6.7.8 1000`).

network_state

Adds support for retrieving a network's state.

This adds the following new endpoint (see *RESTful API* for details):

- `GET /1.0/networks/<name>/state`

proxy_unix_dac_properties

This adds support for gid, uid, and mode properties for non-abstract unix sockets.

container_protection_delete

Enables setting the `security.protection.delete` field which prevents containers from being deleted if set to true. Snapshots are not affected by this setting.

proxy_priv_drop

Adds security.uid and security.gid for the proxy devices, allowing privilege dropping and effectively changing the uid/gid used for connections to Unix sockets too.

pprof_http

This adds a new core.debug_address config option to start a debugging HTTP server.

That server currently includes a pprof API and replaces the old cpu-profile, memory-profile and print-goroutines debug options.

proxy_haproxy_protocol

Adds a proxy_protocol key to the proxy device which controls the use of the HAProxy PROXY protocol header.

network_hwaddr

Adds a bridge.hwaddr key to control the MAC address of the bridge.

proxy_nat

This adds optimized UDP/TCP proxying. If the configuration allows, proxying will be done via iptables instead of proxy devices.

network_nat_order

This introduces the `ipv4.nat.order` and `ipv6.nat.order` configuration keys for LXD bridges. Those keys control whether to put the LXD rules before or after any pre-existing rules in the chain.

container_full

This introduces a new recursion=2 mode for `GET /1.0/containers` which allows for the retrieval of all container structs, including the state, snapshots and backup structs.

This effectively allows for "lxc list" to get all it needs in one query.

candid_authentication

This introduces the new candid.api.url config option and removes core.macaroon.endpoint.

backup_compression

This introduces a new `backups.compression_algorithm` config key which allows configuration of backup compression.

candid_config

This introduces the config keys `candid.domains` and `candid.expiry`. The former allows specifying allowed/valid Candid domains, the latter makes the macaroon's expiry configurable. The `lxc remote add` command now has a `--domain` flag which allows specifying a Candid domain.

nvidia_runtime_config

This introduces a few extra config keys when using nvidia.runtime and the libnvidia-container library. Those keys translate pretty much directly to the matching nvidia-container environment variables:

- nvidia.driver.capabilities => NVIDIA_DRIVER_CAPABILITIES
- nvidia.require.cuda => NVIDIA_REQUIRE_CUDA
- nvidia.require.driver => NVIDIA_REQUIRE_DRIVER

storage_api_volume_snapshots

Add support for storage volume snapshots. They work like container snapshots, only for volumes.

This adds the following new endpoint (see *RESTful API* for details):

- `GET /1.0/storage-pools/<pool>/volumes/<type>/<name>/snapshots`
- `POST /1.0/storage-pools/<pool>/volumes/<type>/<name>/snapshots`
- `GET /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `PUT /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `POST /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`
- `DELETE /1.0/storage-pools/<pool>/volumes/<type>/<volume>/snapshots/<name>`

storage_unmapped

Introduces a new `security.unmapped` boolean on storage volumes.

Setting it to true will flush the current map on the volume and prevent any further idmap tracking and remapping on the volume.

This can be used to share data between isolated containers after attaching it to the container which requires write access.

projects

Add a new project API, supporting creation, update and deletion of projects.

Projects can hold containers, profiles or images at this point and let you get a separate view of your LXD resources by switching to it.

candid_config_key

This introduces a new `candid.api.key` option which allows for setting the expected public key for the endpoint, allowing for safe use of a HTTP-only candid server.

network_vxlan_ttl

This adds a new `tunnel.NAME.ttl` network configuration option which makes it possible to raise the ttl on VXLAN tunnels.

container_incremental_copy

This adds support for incremental container copy. When copying a container using the `--refresh` flag, only the missing or outdated files will be copied over. Should the target container not exist yet, a normal copy operation is performed.

usb_optional_vendorid

As the name implies, the `vendorid` field on USB devices attached to containers has now been made optional, allowing for all USB devices to be passed to a container (similar to what's done for GPUs).

snapshot_scheduling

This adds support for snapshot scheduling. It introduces three new configuration keys: `snapshots.schedule`, `snapshots.schedule.stopped`, and `snapshots.pattern`. Snapshots can be created automatically up to every minute.

snapshots_schedule_aliases

Snapshot schedule can be configured by a comma separated list of schedule aliases. Available aliases are `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly> <@startup>` for instances, and `<@hourly> <@daily> <@midnight> <@weekly> <@monthly> <@annually> <@yearly>` for storage volumes.

container_copy_project

Introduces a `project` field to the container source dict, allowing for copy/move of containers between projects.

clustering_server_address

This adds support for configuring a server network address which differs from the REST API client network address. When bootstrapping a new cluster, clients can set the new `cluster.https_address` config key to specify the address of the initial server. When joining a new server, clients can set the `core.https_address` config key of the joining server to the REST API address the joining server should listen at, and set the `server_address` key in the `PUT /1.0/cluster` API to the address the joining server should use for clustering traffic (the value of `server_address` will be automatically copied to the `cluster.https_address` config key of the joining server).

clustering_image_replication

Enable image replication across the nodes in the cluster. A new cluster.images_minimal_replica configuration key was introduced can be used to specify to the minimal numbers of nodes for image replication.

container_protection_shift

Enables setting the `security.protection.shift` option which prevents containers from having their filesystem shifted.

snapshot_expiry

This adds support for snapshot expiration. The task is run minutely. The config option `snapshots.expiry` takes an expression in the form of `1M 2H 3d 4w 5m 6y` (1 minute, 2 hours, 3 days, 4 weeks, 5 months, 6 years), however not all parts have to be used.

Snapshots which are then created will be given an expiry date based on the expression. This expiry date, defined by `expires_at`, can be manually edited using the API or `lxc config edit`. Snapshots with a valid expiry date will be removed when the task in run. Expiry can be disabled by setting `expires_at` to an empty string or `0001-01-01T00:00:00Z` (zero time). This is the default if `snapshots.expiry` is not set.

This adds the following new endpoint (see *RESTful API* for details):

- `PUT /1.0/containers/<name>/snapshots/<name>`

snapshot_expiry_creation

Adds `expires\_at` to container creation, allowing for override of a snapshot's expiry at creation time.

network_leases_location

Introductes a "Location" field in the leases list. This is used when querying a cluster to show what node a particular lease was found on.

resources_cpu_socket

Add Socket field to CPU resources in case we get out of order socket information.

resources_gpu

Add a new GPU struct to the server resources, listing all usable GPUs on the system.

resources_numa

Shows the NUMA node for all CPUs and GPUs.

kernel_features

Exposes the state of optional kernel features through the server environment.

id_map_current

This introduces a new internal `volatile.idmap.current` key which is used to track the current mapping for the container.

This effectively gives us:

- `volatile.last_state.idmap` => On-disk idmap
- `volatile.idmap.current` => Current kernel map
- `volatile.idmap.next` => Next on-disk idmap

This is required to implement environments where the on-disk map isn't changed but the kernel map is (e.g. shiftfs).

event_location

Expose the location of the generation of API events.

storage_api_remote_volume_snapshots

This allows migrating storage volumes including their snapshots.

network_nat_address

This introduces the `ipv4.nat.address` and `ipv6.nat.address` configuration keys for LXD bridges. Those keys control the source address used for outbound traffic from the bridge.

container_nic_routes

This introduces the `ipv4.routes` and `ipv6.routes` properties on "nic" type devices. This allows adding static routes on host to container's nic.

rbac

Adds support for RBAC (role based access control). This introduces new config keys:

- rbac.api.url
- rbac.api.key
- rbac.api.expiry
- rbac.agent.url
- rbac.agent.username
- rbac.agent.private_key
- rbac.agent.public_key

cluster_internal_copy

This makes it possible to do a normal "POST /1.0/containers" to copy a container between cluster nodes with LXD internally detecting whether a migration is required.

seccomp_notify

If the kernel supports seccomp-based syscall interception LXD can be notified by a container that a registered syscall has been performed. LXD can then decide to trigger various actions.

lxc_features

This introduces the `lxc_features` section output from the `lxc info` command via the `GET /1.0/` route. It outputs the result of checks for key features being present in the underlying LXC library.

container_nic_ipvlan

This introduces the `ipvlan` "nic" device type.

network_vlan_sriov

This introduces VLAN (`vlan`) and MAC filtering (`security.mac_filtering`) support for SR-IOV devices.

storage_cephfs

Add support for CEPHFS as a storage pool driver. This can only be used for custom volumes, images and containers should be on CEPH (RBD) instead.

container_nic_ipfilter

This introduces container IP filtering (`security.ipv4_filtering` and `security.ipv6_filtering`) support for `bridged` nic devices.

resources_v2

Rework the resources API at /1.0/resources, especially:

- CPU
  - Fix reporting to track sockets, cores and threads
  - Track NUMA node per core
  - Track base and turbo frequency per socket
  - Track current frequency per core
  - Add CPU cache information
  - Export the CPU architecture
  - Show online/offline status of threads
- Memory
  - Add hugepages tracking
  - Track memory consumption per NUMA node too
- GPU
  - Split DRM information to separate struct
  - Export device names and nodes in DRM struct
  - Export device name and node in NVIDIA struct
  - Add SR-IOV VF tracking

container_exec_user_group_cwd

Adds support for specifying User, Group and Cwd during `POST /1.0/containers/NAME/exec`.

container_syscall_intercept

Adds the `security.syscalls.intercept.\*` configuration keys to control what system calls will be interecepted by LXD and processed with elevated permissions.

container_disk_shift

Adds the `shift` property on `disk` devices which controls the use of the shiftfs overlay.

storage_shifted

Introduces a new `security.shifted` boolean on storage volumes.

Setting it to true will allow multiple isolated containers to attach the same storage volume while keeping the filesystem writable from all of them.

This makes use of shiftfs as an overlay filesystem.

resources_infiniband

Export infiniband character device information (issm, umad, uverb) as part of the resources API.

daemon_storage

This introduces two new configuration keys `storage.images_volume` and `storage.backups_volume` to allow for a storage volume on an existing pool be used for storing the daemon-wide images and backups artifacts.

instances

This introduces the concept of instances, of which currently the only type is "container".

image_types

This introduces support for a new Type field on images, indicating what type of images they are.

resources_disk_sata

Extends the disk resource API struct to include:

- Proper detection of sata devices (type)
- Device path
- Drive RPM
- Block size
- Firmware version

- Serial number

clustering_roles

This adds a new `roles` attribute to cluster entries, exposing a list of roles that the member serves in the cluster.

images_expiry

This allows for editing of the expiry date on images.

resources_network_firmware

Adds a FirmwareVersion field to network card entries.

backup_compression_algorithm

This adds support for a `compression_algorithm` property when creating a backup (`POST /1.0/containers/<name>/backups`).

Setting this property overrides the server default value (`backups.compression_algorithm`).

ceph_data_pool_name

This adds support for an optional argument (`ceph.osd.data_pool_name`) when creating storage pools using Ceph RBD, when this argument is used the pool will store it's actual data in the pool specified with `data_pool_name` while keeping the metadata in the pool specified by `pool_name`.

container_syscall_intercept_mount

Adds the `security.syscalls.intercept.mount`, `security.syscalls.intercept.mount.allowed`, and `security.syscalls.intercept.mount.shift` configuration keys to control whether and how the mount system call will be interecepted by LXD and processed with elevated permissions.

compression_squashfs

Adds support for importing/exporting of images/backups using SquashFS file system format.

container_raw_mount

This adds support for passing in raw mount options for disk devices.

container_nic_routed

This introduces the `routed` "nic" device type.

container_syscall_intercept_mount_fuse

Adds the `security.syscalls.intercept.mount.fuse` key. It can be used to redirect filesystem mounts to their fuse implementation. To this end, set e.g. `security.syscalls.intercept.mount.fuse=ext4=fuse2fs`.

container_disk_ceph

This allows for existing a CEPH RDB or FS to be directly connected to a LXD container.

virtual_machines

Add virtual machine support.

image_profiles

Allows a list of profiles to be applied to an image when launching a new container.

clustering_architecture

This adds a new `architecture` attribute to cluster members which indicates a cluster member's architecture.

resources_disk_id

Add a new device_id field in the disk entries on the resources API.

storage_lvm_stripes

This adds the ability to use LVM stripes on normal volumes and thin pool volumes.

vm_boot_priority

Adds a `boot.priority` property on nic and disk devices to control the boot order.

unix_hotplug_devices

Adds support for unix char and block device hotplugging.

api_filtering

Adds support for filtering the result of a GET request for instances and images.

instance_nic_network

Adds support for the `network` property on a NIC device to allow a NIC to be linked to a managed network. This allows it to inherit some of the network's settings and allows better validation of IP settings.

clustering_sizing

Support specifying a custom values for database voters and standbys. The new `cluster.max_voters` and `cluster.max_standby` configuration keys were introduced to specify to the ideal number of database voter and standbys.

firewall_driver

Adds the `Firewall` property to the ServerEnvironment struct indicating the firewall driver being used.

storage_lvm_vg_force_reuse

Introduces the ability to create a storage pool from an existing non-empty volume group. This option should be used with care, as LXD can then not guarantee that volume name conflicts won't occur with non-LXD created volumes in the same volume group. This could also potentially lead to LXD deleting a non-LXD volume should name conflicts occur.

container_syscall_intercept_hugetlbfs

When mount syscall interception is enabled and hugetlbfs is specified as an allowed filesystem type LXD will mount a separate hugetlbfs instance for the container with the uid and gid mount options set to the container's root uid and gid. This ensures that processes in the container can use hugepages.

limits_hugepages

This allows to limit the number of hugepages a container can use through the hugetlb cgroup. This means the hugetlb cgroup needs to be available. Note, that limiting hugepages is recommended when intercepting the mount syscall for the hugetlbfs filesystem to avoid allowing the container to exhaust the host's hugepages resources.

container_nic_routed_gateway

This introduces the `ipv4.gateway` and `ipv6.gateway` NIC config keys that can take a value of either "auto" or "none". The default value for the key if unspecified is "auto". This will cause the current behaviour of a default gateway being added inside the container and the same gateway address being added to the host-side interface. If the value is set to "none" then no default gateway nor will the address be added to the host-side interface. This allows multiple routed NIC devices to be added to a container.

projects_restrictions

This introduces support for the `restricted` configuration key on project, which can prevent the use of security-sensitive features in a project.

custom_volume_snapshot_expiry

This allows custom volume snapshots to expiry. Expiry dates can be set individually, or by setting the `snapshots.expiry` config key on the parent custom volume which then automatically applies to all created snapshots.

volume_snapshot_scheduling

This adds support for custom volume snapshot scheduling. It introduces two new configuration keys: `snapshots.schedule` and `snapshots.pattern`. Snapshots can be created automatically up to every minute.

trust_ca_certificates

This allows for checking client certificates trusted by the provided CA (`server.ca`). It can be enabled by setting `core.trust_ca_certificates` to true. If enabled, it will perform the check, and bypass the trusted password if true. An exception will be made if the connecting client certificate is in the provided CRL (`ca.crl`). In this case, it will ask for the password.

snapshot_disk_usage

This adds a new `size` field to the output of `/1.0/instances/<name>/snapshots/<snapshot>` which represents the disk usage of the snapshot.

clustering_edit_roles

This adds a writable endpoint for cluster members, allowing the editing of their roles.

container_nic_routed_host_address

This introduces the `ipv4.host_address` and `ipv6.host_address` NIC config keys that can be used to control the host-side veth interface's IP addresses. This can be useful when using multiple routed NICs at the same time and needing a predictable next-hop address to use.

This also alters the behaviour of `ipv4.gateway` and `ipv6.gateway` NIC config keys. When they are set to "auto" the container will have its default gateway set to the value of `ipv4.host_address` or `ipv6.host_address` respectively.

The default values are:

`ipv4.host_address`: 169.254.0.1 `ipv6.host_address`: fe80::1

This is backward compatible with the previous default behaviour.

container_nic_ipvlan_gateway

This introduces the `ipv4.gateway` and `ipv6.gateway` NIC config keys that can take a value of either "auto" or "none". The default value for the key if unspecified is "auto". This will cause the current behaviour of a default gateway being added inside the container and the same gateway address being added to the host-side interface. If the value is set to "none" then no default gateway nor will the address be added to the host-side interface. This allows multiple ipvlan NIC devices to be added to a container.

resources_usb_pci

This adds USB and PCI devices to the output of `/1.0/resources`.

resources_cpu_threads_numa

This indicates that the numa_node field is now recorded per-thread rather than per core as some hardware apparently puts threads in different NUMA domains.

resources_cpu_core_die

Exposes the `die_id` information on each core.

api_os

This introduces two new fields in `/1.0`, `os` and `os_version`.

Those are taken from the os-release data on the system.

resources_system

This adds system information to the output of `/1.0/resources`.

resources_cpu_isolated

Add an `Isolated` property on CPU threads to indicate if the thread is physically `Online` but is configured not to accept tasks.

usedby_consistency

This extension indicates that UsedBy should now be consistent with suitable ?project= and ?target= when appropriate.

The 5 entities that have UsedBy are:

- Profiles
- Projects
- Networks
- Storage pools
- Storage volumes

container_syscall_filtering_allow_deny_syntax

A number of new syscalls related container configuration keys were updated.

- `security.syscalls.deny_default`
- `security.syscalls.deny_compat`
- `security.syscalls.deny`
- `security.syscalls.allow`

resources_gpu_mdev

Expose available mediated device profiles and devices in /1.0/resources.

console_vga_type

This extends the `/1.0/console` endpoint to take a `?type=` argument, which can be set to `console` (default) or `vga` (the new type added by this extension).

When POST'ing to `/1.0/<instance name>/console?type=vga` the data websocket returned by the operation in the metadata field will be a bidirectional proxy attached to a SPICE unix socket of the target virtual machine.

projects_limits_disk

Add `limits.disk` to the available project configuration keys. If set, it limits the total amount of disk space that instances volumes, custom volumes and images volumes can use in the project.

storage_rsync_compression

Adds `rsync.compression` config key to storage pools. This key can be used to disable compression in rsync while migrating storage pools.

gpu_mdev

This adds support for virtual GPUs. It introduces the `mdev` config key for GPU devices which takes a supported mdev type, e.g. i915-GVTg_V5_4.

resources_pci_iommu

This adds the IOMMUGroup field for PCI entries in the resources API.

resources_network_usb

Adds the usb_address field to the network card entries in the resources API.

resources_disk_address

Adds the usb_address and pci_address fields to the disk entries in the resources API.

network_state_vlan

This adds a "vlan" section to the /1.0/networks/NAME/state API.

Those contain additional state information relevant to VLAN interfaces:

- lower_device
- vid

gpu_sriov

This adds support for SR-IOV enabled GPUs. It introduces the `sriov` gpu type property.

migration_stateful

Add a new `migration.stateful` config key.

disk_state_quota

This introduces the `size.state` device config key on `disk` devices.

storage_ceph_features

Adds a new `ceph.rbd.features` config key on storage pools to control the RBD features used for new volumes.

gpu_mig

This adds support for NVIDIA MIG. It introduces the `mig` gputype and associated config keys.

clustering_join_token

Adds `POST /1.0/cluster/members` API endpoint for requesting a join token used when adding new cluster members without using the trust password.

clustering_join_token

Adds `POST /1.0/cluster/members` API endpoint for requesting a join token used when adding new cluster members without using the trust password.

clustering_description

Adds an editable description to the cluster members.

server_trusted_proxy

This introduces support for `core.https_trusted_proxy` which has LXD parse a HAProxy style connection header on such connections and if present, will rewrite the request's source address to that provided by the proxy server.

clustering_update_cert

Adds `PUT /1.0/cluster/certificate` endpoint for updating the cluster certificate across the whole cluster

storage_api_project

This adds support for copy/move custom storage volumes between projects.

server_instance_driver_operational

This modifies the `driver` output for the `/1.0` endpoint to only include drivers which are actually supported and operational on the server (as opposed to being included in LXD but not operational on the server).

server_supported_storage_drivers

This adds supported storage driver info to server environment info.

event_lifecycle_requestor_address

Adds a new address field to lifecycle requestor.

resources_gpu_usb

Add a new USBAddress (usb_address) field to ResourcesGPUCard (GPU entries) in the resources API.

network_counters_errors_dropped

This adds the received and sent errors as well as inbound and outbound dropped packets to the network counters.

image_source_project

Adds a new `project` field to `POST /1.0/images` allowing for the source project to be set at image copy time.

database_leader

Adds new "database-leader" role which is assigned to cluster leader.

instance_all_projects

This adds support for displaying instances from all projects.

ceph_rbd_du

Adds a new `ceph.rbd.du` boolean on Ceph storage pools which allows disabling the use of the potentially slow `rbd du` calls.

qemu_metrics

This adds a new `security.agent.metrics` boolean which defaults to `true`. When set to `false`, it doesn't connect to the lxd-agent for metrics and other state information, but relies on stats from QEMU.

gpu_mig_uuid

Adds support for the new MIG UUID format used by Nvidia `470+` drivers (eg. `MIG-74c6a31a-fde5-5c61-973b-70e12346c202`), the `MIG-` prefix can be omitted

This extension supersedes old `mig.gi` and `mig.ci` parameters which are kept for compatibility with old drivers and cannot be set together.

event_project

Expose the project an API event belongs to.

instance_allow_inconsistent_copy

Adds `allow_inconsistent` field to instance source on `POST /1.0/instances`. If true, rsync will ignore the `Partial transfer due to vanished source files` (code 24) error when creating an instance from a copy.

image_restrictions

This extension adds on to the image properties to include image restrictions/host requirements. These requirements help determine the compatbility between an instance and the host system.

3.5.4 Communication between instance and host

Introduction

Communication between the hosted workload (instance) and its host while not strictly needed is a pretty useful feature.

In LXD, this feature is implemented through a `/dev/lxd/sock` node which is created and setup for all LXD instances.

This file is a Unix socket which processes inside the instance can connect to. It's multi-threaded so multiple clients can be connected at the same time.

Implementation details

LXD on the host binds `/var/lib/lxd/devlxd/sock` and starts listening for new connections on it.

This socket is then exposed into every single instance started by LXD at `/dev/lxd/sock`.

The single socket is required so we can exceed 4096 instances, otherwise, LXD would have to bind a different socket for every instance, quickly reaching the FD limit.

Authentication

Queries on `/dev/lxd/sock` will only return information related to the requesting instance. To figure out where a request comes from, LXD will extract the initial socket ucred and compare that to the list of instances it manages.

Protocol

The protocol on `/dev/lxd/sock` is plain-text HTTP with JSON messaging, so very similar to the local version of the LXD protocol.

Unlike the main LXD API, there is no background operation and no authentication support in the `/dev/lxd/sock` API.

REST-API

API structure

- /
  - /1.0
    - /1.0/config
      - /1.0/config/{key}
    - /1.0/events
    - /1.0/images/{fingerprint}/export
    - /1.0/meta-data

API details

/

GET

- Description: List of supported APIs
- Return: list of supported API endpoint URLs (by default `['/1.0']`)

Return value:

```
[
    "/1.0"
]
```

/1.0

GET

- Description: Information about the 1.0 API
- Return: dict

Return value:

```
{
    "api_version": "1.0"
}
```

/1.0/config

GET

- Description: List of configuration keys
- Return: list of configuration keys URL

Note that the configuration key names match those in the instance config, however not all configuration namespaces will be exported to `/dev/lxd/sock`. Currently only the `user.*` keys are accessible to the instance.

At this time, there also aren't any instance-writable namespace.

Return value:

```
[
    "/1.0/config/user.a"
]
```

/1.0/config/<KEY>

GET

- Description: Value of that key
- Return: Plain-text value

Return value:

```
blah
```

/1.0/events

GET

- Description: websocket upgrade
- Return: none (never ending flow of events)

Supported arguments are:

- type: comma separated list of notifications to subscribe to (defaults to all)

The notification types are:

- config (changes to any of the user.* config keys)
- device (any device addition, change or removal)

This never returns. Each notification is sent as a separate JSON dict:

```
{
    "timestamp": "2017-12-21T18:28:26.846603815-05:00",
    "type": "device",
    "metadata": {
        "name": "kvm",
        "action": "added",
        "config": {
            "type": "unix-char",
            "path": "/dev/kvm"
        }
    }
}
```

```
{
    "timestamp": "2017-12-21T18:28:26.846603815-05:00",
    "type": "config",
    "metadata": {
        "key": "user.foo",
        "old_value": "",
        "value": "bar"
    }
}
```

/1.0/images/<FINGERPRINT>/export

GET

- Description: Download a public/cached image from the host
- Return: raw image or error
- Access: Requires security.devlxd.images set to true

Return value:

```
See /1.0/images/<FINGERPRINT>/export in the daemon API.
```

/1.0/meta-data

GET

- Description: Container meta-data compatible with cloud-init
- Return: cloud-init meta-data

Return value:

```
#cloud-config
instance-id: abc
local-hostname: abc
```

3.5.5 Events

Introduction

Events are messages about actions that have occurred over LXD. Using the API endpoint `/1.0/events` directly or via `lxc monitor` will connect to a WebSocket through which logs and lifecycle messages will be streamed.

Event types

LXD Currently supports three event types.

- **Logging**: Shows all logging messages regardless of the server logging level.
- **Operation**: Shows all ongoing operations from creation to completion (including updates to their state and progress metadata).
- **Lifecycle**: Shows an audit trail for specific actions occurring over LXD.

Event structure

Example:

```
location: cluster_name
metadata:
  action: network-updated
  requestor:
    protocol: unix
    username: root
  source: /1.0/networks/lxdbr0
timestamp: "2021-03-14T00:00:00Z"
type: lifecycle
```

- `location`: The cluster member name (if clustered).
- `timestamp`: Time that the event occurred in RFC3339 format.
- `type`: The type of event this is (one of `logging`, `operation`, or `lifecycle`).
- `metadata`: Information about the specific event type.

Logging event structure

- `message`: The log message.
- `level`: The log-level of the log.
- `context`: Additional information included in the event.

Operation event structure

- `id`: The UUID of the operation.
- `class`: The type of operation (task, token, or websocket).
- `description`: A description of the operation.
- `created_at`: The operation's creation date.
- `updated_at`: The operation's date of last change.
- `status`: The current state of the operation.
- `status_code`: The operation status code.
- `resources`: Resources affected by this operation.
- `metadata`: Operation specific metadata.
- `may_cancel`: Whether the operation may be cancelled.
- `err`: Error message of the operation.
- `location`: The cluster member name (if clustered).

Lifecycle event structure

- `action`: The lifecycle action that occurred.
- `requestor`: Information about who is making the request (if applicable).
- `source`: Path to what is being acted upon.
- `context`: Additional information included in the event.

Supported lifecycle events

Name	Description	Additiona
`certificate-created`	A new certificate has been added to the server trust store.	
`certificate-deleted`	The certificate has been deleted from the trust store.	
`certificate-updated`	The certificate's configuration has been updated.	
`cluster-certificate-updated`	The certificate for the whole cluster has changed.	
`cluster-disabled`	Clustering has been disabled for this machine.	
`cluster-enabled`	Clustering has been enabled for this machine.	
`cluster-member-added`	A new machine has joined the cluster.	
`cluster-member-removed`	The cluster member has been removed from the cluster.	
`cluster-member-renamed`	The cluster member has been renamed.	`old_name`
`cluster-member-updated`	The cluster member's configuration been edited.	
`cluster-token-created`	A join token for adding a cluster member has been created.	
`config-updated`	The server configuration has changed.	
`image-alias-created`	An alias has been created for an existing image.	`target`: t
`image-alias-deleted`	An alias has been deleted for an existing image.	`target`: t
`image-alias-renamed`	The alias for an existing image has been renamed.	`old_name`
`image-alias-updated`	The configuration for an image alias has changed.	`target`: t
`image-created`	A new image has been added to the image store.	`type`: con
`image-deleted`	The image has been deleted from the image store.	
`image-refreshed`	The local image copy has updated to the current source image version.	
`image-retrieved`	The raw image file has been downloaded from the server.	`target`: c
`image-secret-created`	A one-time key to fetch this image has been created.	
`image-updated`	The image's configuration has changed.	
`instance-backup-created`	A backup of the instance has been created.	
`instance-backup-deleted`	The instance backup has been deleted.	
`instance-backup-renamed`	The instance backup has been renamed.	`old_name`
`instance-backup-retrieved`	The raw instance backup file has been downloaded.	
`instance-console`	Connected to the console of the instance.	`type`: con
`instance-console-reset`	The console buffer has been reset.	
`instance-console-retrieved`	The console log has been downloaded.	
`instance-created`	A new instance has been created.	
`instance-deleted`	The instance has been deleted.	
`instance-exec`	A command has been executed on the instance.	`command`:
`instance-file-deleted`	A file on the instance has been deleted.	`file`: pat
`instance-file-pushed`	The file has been pushed to the instance.	`file-sou`
`instance-file-retrieved`	The file has been downloaded from the instance.	`file-sou`
`instance-log-deleted`	The instance's specified log file has been deleted.	
`instance-log-retrieved`	The instance's specified log file has been downloaded.	
`instance-metadata-retrieved`	The instance's image metadata has been downloaded.	

Table 4 – continued from previous page

Name	Description	Additiona
instance-metadata-updated	The instance's image metadata has changed.	
instance-metadata-template-created	A new image template file for the instance has been created.	path: rela
instance-metadata-template-deleted	The image template file for the instance has been deleted.	path: rela
instance-metadata-template-retrieved	The image template file for the instance has been downloaded.	path: rela
instance-paused	The instance has been put in a paused state.	
instance-renamed	The instance has been renamed.	old_name
instance-restarted	The instance has restarted.	
instance-restored	The instance has been restored from a snapshot.	snapshot
instance-resumed	The instance has resumed after being paused.	
instance-shutdown	The instance has shut down.	
instance-started	The instance has started.	
instance-stopped	The instance has stopped.	
instance-updated	The instance's configuration has changed.	
instance-snapshot-created	A snapshot of the instance has been created.	
instance-snapshot-deleted	The instance snapshot has been deleted.	
instance-snapshot-renamed	The instance snapshot has been renamed.	old_name
instance-snapshot-updated	The instance snapshot's configuration has changed.	
network-created	A network device has been created.	
network-deleted	The network device has been deleted.	
network-renamed	The network device has been renamed.	old_name
network-updated	The network device's configuration has changed.	
operation-cancelled	The operation has been cancelled.	
profile-created	A new profile has been created.	
profile-deleted	The profile has been deleted.	
profile-renamed	The profile has been renamed .	old_name
profile-updated	The profile's configuration has changed.	
project-created	A new project has been created.	
project-deleted	The project has been deleted.	
project-renamed	The project has been renamed.	old_name
project-updated	The project's configuration has changed.	
storage-pool-created	A new storage pool has been created.	target: c
storage-pool-deleted	The storage pool has been deleted.	
storage-pool-updated	The storage pool's configuration has changed.	target: c
storage-volume-created	A new storage volume has been created.	type: con
storage-volume-deleted	The storage volume has been deleted.	
storage-volume-renamed	The storage volume has been renamed.	old_name
storage-volume-restored	The storage volume has been restored from a snapshot.	snapshot
storage-volume-updated	The storage volume's configuration has changed.	
storage-volume-snapshot-created	A new storage volume snapshot has been created.	type: con
storage-volume-snapshot-deleted	The storage volume's snapshot has been deleted.	
storage-volume-snapshot-renamed	The storage volume's snapshot has been renamed.	old_name
storage-volume-snapshot-updated	The configuration for the storage volume's snapshot has changed.	

3.6 Internals & debugging

3.6.1 Container runtime environment

LXD attempts to present a consistent environment to the container it runs.

The exact environment will differ slightly based on kernel features and user configuration but will otherwise be identical for all containers.

PID1

LXD spawns whatever is located at `/sbin/init` as the initial process of the container (PID 1). This binary should act as a proper init system, including handling re-parented processes.

LXD's communication with PID1 in the container is limited to two signals:

- `SIGINT` to trigger a reboot of the container
- `SIGPWR` (or alternatively `SIGRTMIN+3`) to trigger a clean shutdown of the container

The initial environment of PID1 is blank except for `container=lxc` which can be used by the init system to detect the runtime.

All file descriptors above the default 3 are closed prior to PID1 being spawned.

Filesystem

LXD assumes that any image it uses to create a new container from will come with at least:

- `/dev` (empty)
- `/proc` (empty)
- `/sbin/init` (executable)
- `/sys` (empty)

Devices

LXD containers have a minimal and ephemeral `/dev` based on a tmpfs filesystem. Since this is a tmpfs and not a devtmpfs, device nodes will only appear if manually created.

The standard set of device nodes will be setup:

- `/dev/console`
- `/dev/fd`
- `/dev/full`
- `/dev/log`
- `/dev/null`
- `/dev/ptmx`
- `/dev/random`
- `/dev/stdin`
- `/dev/stderr`

- /dev/stdout
- /dev/tty
- /dev/urandom
- /dev/zero

On top of the standard set of devices, the following are also setup for convenience:

- /dev/fuse
- /dev/net/tun
- /dev/mqueue

Mounts

The following mounts are setup by default under LXD:

- /proc (proc)
- /sys (sysfs)
- /sys/fs/cgroup/* (cgroupfs) (only on kernels lacking cgroup namespace support)

The following paths will also be automatically mounted if present on the host:

- /proc/sys/fs/binfmt_misc
- /sys/firmware/efi/efivars
- /sys/fs/fuse/connections
- /sys/fs/pstore
- /sys/kernel/debug
- /sys/kernel/security

The reason for passing all of those is legacy init systems which require those to be mounted or be mountabled inside the container.

The majority of those will not be writable (or even readable) from inside an unprivileged container and will be blocked by our AppArmor policy inside privileged containers.

Network

LXD containers may have any number of network devices attached to them. The naming for those unless overridden by the user is ethX where X is an incrementing number.

Container to host communication

LXD sets up a socket at /dev/lxd/sock which root in the container can use to communicate with LXD on the host.

The API is *documented here*.

LXCFS

If LXCFS is present on the host, it will automatically be setup for the container.

This normally results in a number of `/proc` files being overridden through bind-mounts. On older kernels a virtual version of `/sys/fs/cgroup` may also be setup by LXCFS.

3.6.2 Live Migration in LXD

Overview

Migration has two pieces, a "source", that is, the host that already has the instance, and a "sink", the host that's getting the instance. Currently, in the `pull` mode, the source sets up an operation, and the sink connects to the source and pulls the instance.

There are three websockets (channels) used in migration:

1. the control stream
2. the criu images stream
3. the filesystem stream

When a migration is initiated, information about the instance, its configuration, etc. are sent over the control channel (a full description of this process is below), the criu images and instance filesystem are synced over their respective channels, and the result of the restore operation is sent from the sink to the source over the control channel.

In particular, the protocol that is spoken over the criu channel and filesystem channel can vary, depending on what is negotiated over the control socket. For example, both the source and the sink's LXD directory is on btrfs, the filesystem socket can speak btrfs-send/receive. Additionally, although we do a "stop the world" type migration right now, support for criu's p.haul protocol will happen over the criu socket at some later time.

Control Socket

Once all three websockets are connected between the two endpoints, the source sends a MigrationHeader (protobuf description found in `/lxd/migration/migrate.proto`). This header contains the instance configuration which will be added to the new instance.

There are also two fields indicating the filesystem and criu protocol to speak. For example, if a server is hosted on a btrfs filesystem, it can indicate that it wants to do a `btrfs send` instead of a simple rsync (similarly, it could indicate that it wants to speak the p.haul protocol, instead of just rsyncing the images over slowly).

The sink then examines this message and responds with whatever it supports. Continuing our example, if the sink is not on a btrfs filesystem, it responds with the lowest common denominator (rsync, in this case), and the source is to send the root filesystem using rsync. Similarly with the criu connection; if the sink doesn't have support for the p.haul protocol (or whatever), we fall back to rsync.

3.6.3 Daemon behavior

Introduction

This specification covers some of the daemon's behavior, such as reaction to given signals, crashes, ...

Startup

On every start, LXD checks that its directory structure exists. If it doesn't, it'll create the required directories, generate a keypair and initialize the database.

Once the daemon is ready for work, LXD will scan the instances table for any instance for which the stored power state differs from the current one. If an instance's power state was recorded as running and the instance isn't running, LXD will start it.

Signal handling

SIGINT, SIGQUIT, SIGTERM

For those signals, LXD assumes that it's being temporarily stopped and will be restarted at a later time to continue handling the instances.

The instances will keep running and LXD will close all connections and exit cleanly.

SIGPWR

Indicates to LXD that the host is going down.

LXD will attempt a clean shutdown of all the instances. After 30s, it will kill any remaining instance.

The instance `power_state` in the instances table is kept as it was so that LXD after the host is done rebooting can restore the instances as they were.

SIGUSR1

Write a memory profile dump to the file specified with `--memprofile`.

3.6.4 Database

Introduction

So first of all, why a database?

Rather than keeping the configuration and state within each instance's directory as is traditionally done by LXC, LXD has an internal database which stores all of that information. This allows very quick queries against all instances configuration.

An example is the rather obvious question "what instances are using br0?". To answer that question without a database, LXD would have to iterate through every single instance, load and parse its configuration and then look at what network devices are defined in there.

While that may be quick with a few instance, imagine how many filesystem access would be required for 2000 instances. Instead with a database, it's only a matter of accessing the already cached database with a pretty simple query.

Database engine

Since LXD supports clustering, and all members of the cluster must share the same database state, the database engine is based on a distributed version of SQLite, which provides replication, fault-tolerance and automatic failover without the need of external database processes. We refer to this database as the "global" LXD database.

Even when using LXD as single non-clustered node, the global database will still be used, although in that case it effectively behaves like a regular SQLite database.

The files of the global database are stored under the `./database/global` sub-directory of your LXD data dir (e.g. `/var/lib/lxd/database/global` or `/var/snap/lxd/common/lxd/database/global` for snap users).

Since each member of the cluster also needs to keep some data which is specific to that member, LXD also uses a plain SQLite database (the "local" database), which you can find in `./database/local.db`.

Backups of the global database directory and of the local database file are made before upgrades, and are tagged with the `.bak` suffix. You can use those if you need to revert the state as it was before the upgrade.

Dumping the database content or schema

If you want to get a SQL text dump of the content or the schema of the databases, use the `lxd sql <local|global> [.dump|.schema]` command, which produces the equivalent output of the `.dump` or `.schema` directives of the sqlite3 command line tool.

Running custom queries from the console

If you need to perform SQL queries (e.g. `SELECT`, `INSERT`, `UPDATE`) against the local or global database, you can use the `lxd sql` command (run `lxd sql --help` for details).

You should only need to do that in order to recover from broken updates or bugs. Please consult the LXD team first (creating a GitHub issue or forum post).

Running custom queries at LXD daemon startup

In case the LXD daemon fails to start after an upgrade because of SQL data migration bugs or similar problems, it's possible to recover the situation by creating `.sql` files containing queries that repair the broken update.

To perform repairs against the local database, write a `./database/patch.local.sql` file containing the relevant queries, and similarly a `./database/patch.global.sql` for global database repairs.

Those files will be loaded very early in the daemon startup sequence and deleted if the queries were successful (if they fail, no state will change as they are run in a SQL transaction).

As above, please consult the LXD team first.

Syncing the cluster database to disk

If you want to flush the content of the cluster database to disk, use the `lxd sql global .sync` command, that will write a plain SQLite database file into `./database/global/db.bin`, which you can then inspect with the `sqlite3` command line tool.

3.6.5 Debugging

For information on debugging instance issues, see *Frequently Asked Questions*

Debugging `lxc` and `lxd`

Here are different ways to help troubleshooting `lxc` and `lxd` code.

lxc –debug

Adding `--debug` flag to any client command will give extra information about internals. If there is no useful info, it can be added with the logging call:

```
logger.Debugf("Hello: %s", "Debug")
```

lxc monitor

This command will monitor messages as they appear on remote server.

lxd –debug

Shutting down `lxd` server and running it in foreground with `--debug` flag will bring a lot of (hopefully) useful info:

```
systemctl stop lxd lxd.socket
lxd --debug --group lxd
```

`--group lxd` is needed to grant access to unprivileged users in this group.

REST API through local socket

On server side the most easy way is to communicate with LXD through local socket. This command accesses `GET /1.0` and formats JSON into human readable form using jq utility:

```
curl --unix-socket /var/lib/lxd/unix.socket lxd/1.0 | jq .
```

or for snap users:

```
curl --unix-socket /var/snap/lxd/common/lxd/unix.socket lxd/1.0 | jq .
```

See the *RESTful API* for available API.

REST API through HTTPS

HTTPS connection to LXD requires valid client certificate, generated in `~/.config/lxc/client.crt` on first `lxc remote add`. This certificate should be passed to connection tools for authentication and encryption.

Examining certificate. In case you are curious:

```
openssl x509 -in client.crt -purpose
```

Among the lines you should see:

```
Certificate purposes:
SSL client : Yes
```

with command line tools

```
wget --no-check-certificate https://127.0.0.1:8443/1.0 --certificate=$HOME/.config/lxc/
↪client.crt --private-key=$HOME/.config/lxc/client.key -O - -q
```

with browser

Some browser plugins provide convenient interface to create, modify and replay web requests. To authenticate againsg LXD server, convert `lxc` client certificate into importable format and import it into browser.

For example this produces `client.pfx` in Windows-compatible format:

```
openssl pkcs12 -clcerts -inkey client.key -in client.crt -export -out client.pfx
```

After that, opening https://127.0.0.1:8443/1.0 should work as expected.

3.6.6 Environment variables

Introduction

The LXD client and daemon respect some environment variables to adapt to the user's environment and to turn some advanced features on and off.

Common

Name	Description
`LXD_DIR`	The LXD data directory
`LXD_INSECURE_TLS`	If set to true, allows all default Go ciphers both for client <-> server communication and server <-> image servers (server <-> server and clustering are not affected)
`PATH`	List of paths to look into when resolving binaries
`http_proxy`	Proxy server URL for HTTP
`https_proxy`	Proxy server URL for HTTPS
`no_proxy`	List of domains, IP addresses or CIDR ranges that don't require the use of a proxy

Client environment variable

Name	Description
EDITOR	What text editor to use
VISUAL	What text editor to use (if EDITOR isn't set)
LXD_CONF	Path to the LXC configuration directory
LXD_GLOBAL_CONF	Path to the global LXC configuration directory
LXC_REMOTE	Name of the remote to use (overrides configured default remote)

Server environment variable

Name	Description
LXD_EXEC_PATH	Full path to the LXD binary (used when forking subcommands)
LXD_LXC_TEMPLATE_CONFIG	Path to the LXC template configuration directory
LXD_SECURITY_APPARMOR	If set to false, forces AppArmor off
LXD_UNPRIVILEGED_ONLY	If set to true, enforces that only unprivileged containers can be created. Note that any privileged containers that have been created before setting LXD_UNPRIVILEGED_ONLY will continue to be privileged. To use this option effectively it should be set when the LXD daemon is first setup.
LXD_OVMF_PATH	Path to an OVMF build including OVMF_CODE.fd and OVMF_VARS.ms.fd
LXD_SHIFTFS_DISABLE	Disable shiftfs support (useful when testing traditional UID shifting)
LXD_DEVMONITOR_DIR	Path to be monitored by the device monitor. This is primarily for testing.

3.6.7 System call interception

LXD supports intercepting some specific system calls from unprivileged containers and if they're considered to be safe, will executed with elevated privileges on the host.

Doing so comes with a performance impact for the syscall in question and will cause some work for LXD to evaluate the request and if allowed, process it with elevated privileges.

Available system calls

mknod / mknodat

The `mknod` and `mknodat` system calls can be used to create a variety of special files.

Most commonly inside containers, they may be called to create block or character devices. Creating such devices isn't allowed in unprivileged containers as this is a very easy way to escalate privileges by allowing direct write access to resources like disks or memory.

But there are files which are safe to create. For those, intercepting this syscall may unblock some specific workloads and allow them to run inside an unprivileged containers.

The devices which are currently allowed are:

- overlayfs whiteout (char 0:0)
- /dev/console (char 5:1)
- /dev/full (char 1:7)
- /dev/null (char 1:3)

- /dev/random (char 1:8)
- /dev/tty (char 5:0)
- /dev/urandom (char 1:9)
- /dev/zero (char 1:5)

All file types other than character devices are currently sent to the kernel as usual, so enabling this feature doesn't change their behavior at all.

This can be enabled by setting `security.syscalls.intercept.mknod` to `true`.

setxattr

The `setxattr` system call is used to set extended attributes on files.

The attributes which are handled by this currently are:

- trusted.overlay.opaque (overlayfs directory whiteout)

Note that because the mediation must happen on a number of character strings, there is no easy way at present to only intercept the few attributes we care about. As we only allow the attributes above, this may result in breakage for other attributes that would have been previously allowed by the kernel.

This can be enabled by setting `security.syscalls.intercept.setxattr` to `true`.

3.6.8 Idmaps for user namespace

Introduction

LXD runs safe containers. This is achieved mostly through the use of user namespaces which make it possible to run containers unprivileged, greatly limiting the attack surface.

User namespaces work by mapping a set of uids and gids on the host to a set of uids and gids in the container.

For example, we can define that the host uids and gids from 100000 to 165535 may be used by LXD and should be mapped to uid/gid 0 through 65535 in the container.

As a result a process running as uid 0 in the container will actually be running as uid 100000.

Allocations should always be of at least 65536 uids and gids to cover the POSIX range including root (0) and nobody (65534).

Kernel support

User namespaces require a kernel >= 3.12, LXD will start even on older kernels but will refuse to start containers.

Allowed ranges

On most hosts, LXD will check `/etc/subuid` and `/etc/subgid` for allocations for the "lxd" user and on first start, set the default profile to use the first 65536 uids and gids from that range.

If the range is shorter than 65536 (which includes no range at all), then LXD will fail to create or start any container until this is corrected.

If some but not all of `/etc/subuid`, `/etc/subgid`, `newuidmap` (path lookup) and `newgidmap` (path lookup) can be found on the system, LXD will fail the startup of any container until this is corrected as this shows a broken shadow setup.

If none of those files can be found, then LXD will assume a 1000000000 uid/gid range starting at a base uid/gid of 1000000.

This is the most common case and is usually the recommended setup when not running on a system which also hosts fully unprivileged containers (where the container runtime itself runs as a user).

Varying ranges between hosts

The source map is sent when moving containers between hosts so that they can be remapped on the receiving host.

Different idmaps per container

LXD supports using different idmaps per container, to further isolate containers from each other. This is controlled with two per-container configuration keys, `security.idmap.isolated` and `security.idmap.size`.

Containers with `security.idmap.isolated` will have a unique id range computed for them among the other containers with `security.idmap.isolated` set (if none is available, setting this key will simply fail).

Containers with `security.idmap.size` set will have their id range set to this size. Isolated containers without this property set default to a id range of size 65536; this allows for POSIX compliance and a "nobody" user inside the container.

To select a specific map, the `security.idmap.base` key will let you override the auto-detection mechanism and tell LXD what host uid/gid you want to use as the base for the container.

These properties require a container reboot to take effect.

Custom idmaps

LXD also supports customizing bits of the idmap, e.g. to allow users to bind mount parts of the host's filesystem into a container without the need for any uid-shifting filesystem. The per-container configuration key for this is `raw.idmap`, and looks like:

```
both 1000 1000
uid 50-60 500-510
gid 100000-110000 10000-20000
```

The first line configures both the uid and gid 1000 on the host to map to uid 1000 inside the container (this can be used for example to bind mount a user's home directory into a container).

The second and third lines map only the uid or gid ranges into the container, respectively. The second entry per line is the source id, i.e. the id on the host, and the third entry is the range inside the container. These ranges must be the same size.

This property requires a container reboot to take effect.

3.7 External resources