Lithos Documentation

Release 0.18.4

Paul Colomiets

Nov 27, 2018

Contents

Contents:

CHAPTER 1

Configuration Overview

Lithos has 4 configs:

1. `/etc/lithos/master.yaml` – global configuration for whole lithos daemon. Empty config should work most of the time. *Master Config*
2. `/etc/lithos/sandboxes/<NAME>.yaml` – the allowed paths and other system limits for every sandbox. You may think of a sandbox as a single application. *Sandbox Config*
3. `/etc/lithos/processes/<NAME>.yaml` – you may think of it as a list of pairs (image_name, num_of_processes_to_run). It's only a tiny bit longer than that. *Process Config*
4. `<IMAGE>/config/<NAME>.yaml` – configuration of process to run. It's where all the needed to run process are. It's stored inside the image (so updated with new image), and limited by limits in sandbox config. *Container Configuration*

Four configs look superfluous, but they aren't hard. Let's see why are they needed...

1.1 Separation of Concerns

There are three roles which influence lithos containers.

- *Developer*
- *Platform Maintainer*
  - *User Namespaces*
  - *Filesystem*
  - *Network*
  - *Master Config*
- *Orchestration System*

1.1.1 Developer

Developer is the owner of the service. They needs to configure as much as possible for their container with the following limitations:

1. They can't break into host system including:
   - remote code execution (RCE) vulnerabilities in the code
   - even if malicious party controls source code, configuration and binary artifacts running inside the containers
2. Service must run on any host without changes. This might include different filesystem layouts of the host system (i.e. different names/numbers of disks)

Config for developer, which we call *container config* comes **within container itself**. And usually defines command-line, environment and resource limits:

```
executable: /usr/bin/python3.6
arguments:
- myapp.py
- --port=8080

work-dir: /app
environ:
  LANG: en_US.utf-8

memory-limit: 100Mi
fileno-limit: 1k
```

This is almost it. Sometimes container needs disk:

```
executable: /usr/bin/python3.6
arguments:
- myapp.py
- --port=8080

work-dir: /app
environ:
  LANG: en_US.utf-8
volumes:
  /var/lib/sqlite: !Persistent /db

memory-limit: 100Mi
fileno-limit: 1k
```

Note the following things:

1. It doesn't define where filesystem root is because config itself lies in the filesystem root.
2. Volumes don't specify path in the host filesystem, it's a virtual path (`/db` in this case). This is because otherwise the config would depend on exact filesystem layout on host system **and** in some cases it might be a vulnerability (or at least exposure of unnecessary data). Later we'll describe how it's mapped to the real filesystem.

Container operates inside a **sandbox** defined by platform maintainers.

1.1.2 Platform Maintainer

Platform maintainers define how containers are run. They define *sandbox config* and *master config*.

Former defines sandbox for a specific application. Let's see an example *(don't use it in production, see below)*:

```
image-dir: /opt/app1-images
allow-users: [100000-165535]
allow-groups: [100000-165535]
default-user: 100000
default-group: 100000
```

This says that images of this application are in `/opt/app1-images` and it's allowed to use user-ids in the range `100000-165535`.

User Namespaces

First thing to configure here is to make a user namespace per application:

```
image-dir: /opt/app1-images

uid-map:
- { inside: 0, outside: 10002, count: 2 }
gid-map:
- { inside: 0, outside: 10002, count: 2 }

allow-users: [1]
allow-groups: [1]
default-user: 1
default-group: 1
```

Note the following things:

1. We introduced uid/gid map. this means that two users starting with user id `10002` in the host system will be two users `0,1` in the container.
2. Allowed and default users are set relative to the container ids not host system ones
3. We allow only single user id and group id in the container. And this is number `1` (i.e. first non-root user)
4. This scheme works for 99% applications. But in case you need containers in containers or some other specific scenario you can enlarge uid-map and allowed groups as much as OS allows.

The id `10002` is arbitrary. You can use any one. For security and monitoring purposes you should keep separate user ids for each app. Whether they are same across the cluster or allocated on each node is irrelevant unless you have shared filesystem between machines. *Keeping them same uids across cluster is still recommended for easier monitoring and debugging*.

You can allow uid `0` too. When using uid name spaces it **should not** cause any elevated privileges. But this allows creating mountpoints, spawning other namespaces and do lots of things which creates larger vector of attack. This has caused vulnerabilities due to kernel bugs in the past.

Filesystem

As you have already seen, sandbox config defines a place with container base directories:

```
image-dir: /opt/app1-images
image-dir-levels: 1   # default value
```

In this config, directories named like this `/opt/app1-images/some-name1` serve as the root directory for containers(we'll show later how to find out which specific directory is used now). They are mounted **readonly**. With this config:

```
image-dir: /opt/app2-images
image-dir-levels: 2
```

Images are located in `/opt/app2-images/service1/version1`. I.e. two directory components below the image dir. Arbitrary *`image-dir-levels`* can be used. Only fixed number of components supported for each specific sandbox, though.

Extra directories can be specified as follows:

```
readonly-paths:
  /timezones: /usr/share/timezones

writable-paths:
  /db: /var/lib/app1-database
```

There are virtual paths on the left. These can be mounted by referencing them in *container config*:

```
executable: /usr/bin/python3.6
arguments:
- myapp.py
- --port=8080
volumes:
  /etc/timezones: !Readonly /timezones
  /var/lib/sqlite: !Persistent /db
```

This allows platform maintainers to move directories around in the host system and map different directories on different systems without ever interfering the container.

Network

Sandbox also contains network configuration. By default all containers have host network (i.e. they operate in the same network namespace, just like non-containerized processes).

There is also support for bridged network:

```
bridged-network:
  bridge: br0
  network: 10.64.0.0/16
  default-gateway: 10.64.255.254
  after-setup-command: [/usr/bin/arping, -U, -c1, '@{container_ip}']
```

This enables network isolation for containers. Every container in the sandbox have its own network config with separate IP address (see below which one) but all of them derive their configuration from the sandbox config.

Different sandboxes may have the same or different bridged network configs.

See *reference* for more info.

Master Config

Along with sandbox configs, master config is also a part of the "platform maintainer" zone of responsibility. It contains things that are common for all containers and is usually the same across cluster.

You may run with **empty** config. But most commonly it's expected to contain cgroup controllers for lithos to manage:

```
cgroup-controllers: [name,cpu,memory]
```

This is needed for lithos to correctly support memory limits and CPU quotes.

You might also want to nullify *config-log-dir* if you don't use `lithos_clean`:

```
cgroup-controllers: [name,cpu,memory]
config-log-dir: null
```

Usually, you don't need to set anything else. There are various directories to configure in case you have non-standard filesystem layout. See *reference* for full list of settings.

1.1.3 Orchestration System

The last part of configuration is thing that ties sandboxes, images and container configs together. We call it *process config*.

The general idea is that this config is created by an orchestration system. I.e. system that decides where, which version and how many processes to run. This can be some real system like verwalter or just an ansible/chef/salt/bash script that writes required configs.

Basically it looks like:

```
web-worker:
    kind: Daemon
    image: web-wrk/d7399260
    config: "/config/web-worker.yaml"
    instances: 2

background-worker:
    kind: Daemon
    image: task-queue/d7399260
    config: "/config/task-queue.yaml"
    instances: 3
```

Here we run two kinds of services "web worker" with 2 instances (equal processes/containers) and "background worker" with 3 instances.

The `image` is a directory path relative to *image-dir*. Path must contain the number of path components specified in *image-dir-levels*. It is also expected that the diretory is immutable, so each new version of container is run from a different directory and directory path contains some notion of the container version.

`config` is the path inside the container. There is no limit on how many configs might be in the same container. Not all of them might be running at any moment in time.

There are few other things that can be configured in this config. If you're using bridged networking, you need to specify IP address for each container:

```
web-worker:
    kind: Daemon
    image: web-wrk/d7399260
    config: "/config/web-worker.yaml"
    instances: 2
    ip_addresses:
    - 10.64.0.10
    - 10.64.0.11
```

And sometimes containers allow to customize their config with *variables*:

```
background-worker:
  kind: Daemon
  image: task-queue/d7399260
  config: "/config/task-queue.yaml"
  instances: 3
  variables:
    queue_name: "main-queue"
```

See *reference* for more info.

CHAPTER 2

Master Config

Master configuration file is the one that usually at `/etc/lithos/master.yaml` and defines small subset of global configuration parameters. Minimal configuration is an *empty file* but it **must exist** anyway. Here is the reference of the parameters along with the default values:

See *overview* for guidelines.

`sandboxes-dir`
: The directory for per-application configuration files which contain limits of what application might use. If path is relative it's relative to the directory where configuration file is. Default is `./sandboxes`.

`processes-dir`
: The directory for per-application configuration files which contain name of image directory, instance number, etc., to run. If path is relative it's relative to the directory where configuration file is. Default is `./processes`.

`runtime-dir`
: The directory where `pid` file of master process is stored and also the base directory for `state-dir` and `mount-dir`. Path must be absolute. It's expected to be stored on `tmpfs`. Default `/run/lithos`.

`state-dir`
: The directory where to keep container's state dirs. If path is relative it's relative to `runtime-dir`. Default `state` (i.e. `/run/lithos/state`). Path should be on `tmpfs`.

`mount-dir`
: An empty directory to use for mounting. If path is relative it's relative to `runtime-dir`. Default `mnt`.

`devfs-dir`
: The directory where `/dev` filesystem for container exists. If it's not `/dev` (which is not recommended), you should create the directory with `lithos_mkdev` script. Default `/var/lib/lithos/dev`.

`cgroup-name`
: The name of the root cgroup for all lithos processes. Specify `null` (or any other form of YAMLy null) to turn cgroups off completely.

`cgroup-controllers`
: List of cgroup controllers to initialize for each container. Note: the empty list is treated as default. Default is `[name, cpu, cpuacct, memory, blkio]`. If you have some controllers joined together like `cpu, cpuacct` it's ok.

Use `cgroup-name: null` to turn cgroup tracking off (not empty list here). And use `cgroup-controllers: [name]` to only use cgroups for naming processes but not for resource control.

Note: turning off cgroups means that resource limits does not work completely. lithos will not try to enforce them by polling or some other means

---

`default-log-dir`
(default `/var/log/lithos`) The directory where master and each of the application logs are created (unless are overrided by sandbox config).

`config-log-dir`
(default `/var/log/lithos/config`) The directory where configurations of the processes are stored. These are used by `lithos_clean` to find out when it's safe to clean directories. You may also reconstruct processes configuration at any point in time using this directory.

Changed in version 0.10.2: Parameter can be `null`:

```
config-log-dir: null
```

In this case no configuration logging is done. This is mainly useful if you track configurations and versions by some other means.

Note: This is enabled by default for backwards-compatibility reasons. We consider resetting this value to `null` by default in `lithos 1.0` as this parameter is not as useful as were expected.

`stdio-log-dir`
(default `/var/log/lithos/stderr`) The directory where stderr of the processes will be forwarded. One file per sandbox is created.

These files are created by lithos and file descriptor is passed to the application as both the stdout and stderr. Lithos does not parse, copy or otherwise proxy the data. The operating system does all the work. This also means lithos can't rotate or do any other magical things with the log.

This should be used only to tackle the critical errors. Application should send log to a syslog or write some rotating log files on it's own, because there is no good tools to groups lines of the stderr into solid log messages that include tracebacks and other fancy stuff.

Good utilities to manage the files:

- `logrotate` in `copytruncate` mode
- `rsyslog` with file input plugin

This can be overridden in process by *`stdout-stderr-file`*.

Note: The path is reopened on process restart. If *`restart-process-only`* is *true* then it's only reopened when configuration changes. This is good to know if you remove or rename the file by hand.

`log-file`
(default `master.log`) Master log file. Relative paths are treated from *`default-log-dir`*.

`log-level`
(default `warn`) Level of logging. Can be overriden on the command line.

`syslog-facility`
(no default) Enables logging to syslog (with specified facility) instead of file.

syslog-name
: (default `lithos`) Application name for master process in syslog. The child processes are prefixed by this value. For example `lithos-django` (where `django` is a sandbox name).

CHAPTER 3

Sandbox Config

This config resides in `/etc/lithos/sandboxes/NAME.yaml` (by default). Where `NAME` is the name of a sandbox.

The configuration file contains security and resource limits for the container. Including:

- A directory where image resides
- Set of directories that are mounted inside the container (i.e. all writable directories for the container, the `/tmp...`)
- User and group limits

See *overview* for guidelines.

3.1 Reference

`config-file`
: The path for the *processes config*. In most cases should be left unset. Default is `null` which is results into `/etc/lithos/processes/NAME.yaml` with all other settings being defaults.

`image-dir`
: Directory where application images are. Every subdir of the `image-dir` may be mounted as a root file system in the container. **Required**.

`image-dir-levels`
: (default `1`) A number of directory components required for image name in *`image-dir`*

`log-file`
: The file name where to put **supervisor** log of the container. Default is `/var/log/lithos/SANDBOX_NAME.yaml`.

`log-level`
: (default `warn`). The logging level of the supervisor.

readonly-paths
The mapping of `virtual_directory: host_system_directory` of folders which are visible for the container in read-only mode. (Note currently if you have submounts in the source directory, thay may be available as writeable). See *Volumes* for more details.

writable-paths
The mapping of `virtual_directory: host_system_directory` of folders which are visible for the container in writable mode. See *Volumes* for more details.

allow-users
List of ranges of user ids which can be used by container. For containers without user namespaces, it's just a limit of the `user-id` setting.

Example:

```
allow-users: [1, 99, 1000-2000]
```

For containers which have uid maps enabled **in sandbox** this is a list of users available *after* uid mapping applied. For example, the following maps uid 100000 as root in namespace (e.g. for file permissions), but doesn't allow to start process as root (even if it's 100000 ouside):

```
uid-map: [{outside: 100000, inside: 0, count: 65536}]
allow-users: [1-65535]
```

For containers which do have uid maps enabled **in container config**, it limits all the user ids available to the namespace (i.e. for the outside setting of the uid map).

default-user
(no default) A user id used in the container if no `user-id` is specified in container config. By default `user-id` is required.

Note: `default-user` value must be contained in the `allow-users` range

allow-groups
List of ranges of group ids for the container. Works similarly to *allow-users*.

default-group
(default `0`) A group id used in the container if no `group-id` is specified in container config.

Note: `default-group` value must be contained in the `allow-users` range

allow-tcp-ports
List of ranges of allowed TCP ports for container. This is currently not enforced in any way except:

1. Ports < 1024 are restricted by OS for non-root (but may be allowed here)
2. It restricts `bind-port` setting in container config

Note: if you have overlapping TCP port for different sandboxes, only single file descriptor will be used for each port. The config for opening port will be used arbitrary from single config amonst all users, which have obvious security implications.

Warning: *tcp-ports* bind at port in **host namespace**, i.e. it effectively discards *bridged-network* for that port this is both the feature and might be a pitfall. So most of the time you should avoid non-empty *allow-tcp-ports* if using *bridged-network*.

additional-hosts
Mapping of `hostname: ip` for names that will be added to `/etc/hosts` file. This is occasinally used for cheap but static service discovery.

uid-map, gid-map
The list of mapping for uids(gids) in the user namespace of the container. If they are not specified the user namespace is not used. This setting allows to run processes with `uid` zero without the risk of being the `root` on host system.

Here is a example of maps:

```
uid-map:
- {inside: 0, outside: 1000, count: 1}
- {inside: 1, outside: 1, count: 1}
gid-map:
- {inside: 0, outside: 100, count: 1}
```

Note: Currently you may have uid-map either in a sandbox or in a container config, not both.

used-images-list
(optional) A text file that is used by `lithos_clean` to keep images alive. It's not used by any other means except `lithos_clean` utility.

Each line of the file should contain image name relative to the `image_dir`.

It's expected that the list is kept up by some orchestration system or by deployment scripts or by any other tool meaningful for ops team.

This setting is only useful if `auto-clean` is `true` (default)

auto-clean
(default `true`) Clean images of this sandbox when running `lithos_clean`. This is a subject of the following caveats:

1. Lithos clean is not run by lithos automatically, you ought to run it using cron tab
2. If same `image-dir` is used for multiple sandboxes it will be cleaned if at least one of them has non-falsy `auto-clean`.

resolv-conf
(default `/etc/resolv.conf`) default place to copy `resolv.conf` from for containers.

Note: Container itself can override it's own resolv.conf file, but can't read original `/etc/resolv.conf` if this setting is changed.

hosts-file
(default `/etc/hosts`) default place to copy `hosts` from for containers.

Note: Container itself can override it's own `hosts` file, but can't read original `/etc/hosts` if this setting is changed.

bridged-network
(default is absent) a network bridge configuration for all the cotainers in the bridge

Example:

```
bridged-network:
  bridge: br0
  network: 10.0.0.0/24
```

(continues on next page)

(continued from previous page)

```
  default_gateway: 10.0.0.1
  after-setup-command: [/usr/bin/arping, -U, -c1, '@{container_ip}']
```

Note: when bridged network is active your *Process Config* should contain a list of ip addresses one for each container.

Note: this setting does not affect `tcp-ports`. So usually you should keep *`allow-tcp-ports`* setting empty when using bridged network.

Options:

`after-setup-command`
: Command to run after setting up container namespace but before running actual container. The example shown above sends unsolicited arp packet to notify router and other machines on the network that MAC address corresponding to container's IP is changed.

  Command must have absolute path, and has almost empty environment, so don't assume `PATH` is there if you're writing a script. Command runs in *container's network* namespace but with all other namespaces in host system (in particular in *host filesystem* and with permissions of root in host system)

  Replacement variables that work in command-line:

  - `@{container_ip}` – replaced with IP address of a container being set up

  Few examples:

  1. `[/usr/bin/arping, -U, -c1, '@{container_ip}']` – default in v0.17.x. This notifies other peers that MAC address for this IP changed.
  2. `[/usr/bin/arping, -c1, '10.0.0.1']` – other way to do that, that often does the same as in (1) a side-effect (where 10.0.0.1 is a default gateway)
  3. `[/usr/bin/ping, -c1, '10.0.0.1']` – doing same as (2) but using ICMP instead of ARP directly

  Most of the time containers should work with empty `after-setup-command`, but because container gets new MAC address each time it starts, there might be a small delay (~ 5 sec) after container's start where packets going to that IP are lost (so it appears that host is unavailable).

`secrets-private-key`
: (default is absent) Use the specified private key(s) to decode secrets in container's *`secret-environ`* setting.

  The key in this file is openssh-compatible ed25519 private key (RSA keys are *not* supported). File can contain multiple keys (concatenated), if secret matches any of them it will be decoded.

  To create a key use normal `ssh-keygen` and leave the password empty (password-protected keys aren't supported):

```
ssh-keygen -t ed25519 -t /etc/lithos/keys/secret.key
```

  Note: the key must be owned by root with permissions of 0600 (default for ssh-keygen).

`secrets-namespaces`
: (default is *[""]*) allow only secrets with listed namespaces. Useful only if `secrets-private-key` is set.

  For example:

```
secrets-namespaces:
- project1.web
- project1.celery
```

The idea is you might want to use single secret private key for a whole cluster. But diferent services having different "namespaces". This means you can use single public key for encyption and specify different namespace for each service. With this setup user can't just copy a key from one service to another if that another service isn't authorized to read the namespace using *secrets-namespaces*.

To encrypt secret for a specific namespace use:

```
lithos_crypt encrypt -k key.pub -d "secret" -n "project1.web"
```

By default both `lithos_crypt` and *secrets-namespaces* specify empty string as a namespace. This is good enough if you don't have multiple teams sharing the same cluster.

Currently namespaces are limited to a regexp `^[a-zA-Z0-9_.-]*$`

See *Encrypted Variables* for more info.

CHAPTER 4

Process Config

This config resides in `/etc/lithos/processes/NAME.yaml` (by default). Where `NAME` is the name of a sandbox.

It mainly contains three things:

- `image` the process is run from
- `config` file name inside the image that specifies command-line and other process execution parameters
- number `instances` of the process to run

For example:

```
django:
    image: django.v3.5.7
    config: /config/worker_process.yaml
    instances: 3

redis:
    image: redix.v1
    config: /config/redis.yaml
    instances: 1
```

This will start three python `django` worker processes and one redis.

Hint: Usually this config is generated by some tool like ansible or confd.

There is also a way to create **ad-hoc** commands. For example:

```
manage:
    kind: Command
    image: django.v3.5.7
    config: /config/manage_py.yaml
```

This will allow to start a `manage.py` command with:

```
$ lithos_cmd SANDBOX_NAME manage syncdb
```

This runs command in the same sandbox like the worker process itself but the command is actually attached to current shell. The commands may be freely mixed with `Daemon` items (which is default `kind`) in same config. The only limitation is that names must not be duplicated

The `Command` is occasionally useful, but should be used with care. To start a command you need root privileges on host system, so it's only useful for SysOp tasks or **may be** for cron tasks but not for normal operation of application.

4.1 Options

instances
: Number of instances to run

image
: Identifier of the image to run container from

config
: Configuration file name (absolute name in container) to run

ip-addresses
: A list of ip addresses if *bridged-network* is enforced in sandbox. Note the number of items in this list must match *instances* value.

variables
: A mapping of *variable: value* for variables that can be used in process config.

extra-secrets-namespaces
: Additional secrets namespaces allowed for this specific project. In addition to *secrets-namespaces*. See *Encrypted Variables* for more info.

4.2 Variables

You can also add variables for specific config:

For example:

```
django:
    image: django.v3.5.7
    config: /config/worker_process.yaml
    variables:
      tcp_port: 10001
    instances: 3
```

Only variables that are **declared** in *container config* can be substituted. Extra variables are ignored. If there is a declared variable but it's not present in process config, it doesn't pass configuration check.

CHAPTER 5

Container Configuration

Container configuration is a YAML file which is usually put into `/config/<service_name>.yaml` into container image itself.

Note: Curently container configuration may be put into any folder inside the image, but we may fix this folder later. The arbitrary path for container configuration may be a security vulnerability.

The somewhat minimal configuration is looks like following:

```
kind: Daemon
user-id: 1
volumes:
  /tmp: !Tmpfs { size: 100m }
executable: /bin/sleep
arguments: [60]
```

See *overview* for guidelines.

5.1 Variables

Container can declare some things, that can be changed in specific instantiation of the service, for example:

```
variables:
  tcp_port: !TcpPort
kind: Daemon
user-id: 1
volumes:
  /tmp: !Tmpfs { size: 100m }
executable: /bin/some_program
arguments:
- "--listen=localhost:@{tcp_port}"
```

The `variables` key declares variable names and types. Value for these variables can be provided in `variables` in *Process Config*.

There are the following types of variables:

TcpPort Allows a number between 1-65535 and ensures that the number matches port range allowed in sandbox (see *allow-tcp-ports*)

Changed in version 0.17.4: Added `activation` parameter as a shortcut to support systemd activation protocol. I.e. the following (showing two ports for more comprehensive example):

```
variables:
  port1: !TcpPort { activation: systemd }
  port2: !TcpPort { activation: systemd }
```

Means to add something like this:

```
  variables:
    port1: !TcpPort
    port2: !TcpPort
  tcp-ports:
    "@{port1}":
      fd: 3
    "@{port2}":
      fd: 4
  environ:
    LISTEN_FDS: 1
    LISTEN_FDNAMES: "port1:port2"
    LISTEN_PID: "@{lithos:pid}"

This works for any number of sockets. And it requires that
``LISTEN_FDS``, ``LISTEN_FDNAMES``, ``LISTEN_PID`` were absent in the
``environ`` as written in the file. Also it doesn't allow fine-grained
control over parameters of the socket and file descriptor numbers.
Use full form if you need specific options.
```

Choice Allows a value from a fixed set of choices (example: `!Choice ["high-priority", "low-priority"]`)

Name Allows a value that matches regex `^[0-9a-zA-Z_-]+$`. Useful for passing names of things into a script without having a chance to keep value unescaped when passing somewhere within a script or using it as a filename.

New in version 0.10.3.

DottedName Allows arbitrary DNS-like name. It's defined as dot-separated name with only alphanumeric and underscores, where no component could start or end with a dash and no consequent dots allowed.

New in version 0.17.4.

All entries of `@{variable_name}` are substituted in the following fields:

1. *arguments*
2. The values of *environ* (not in the keys yet)
3. The key in the *tcp-ports* (i.e. port number)

The expansion in any other place does not work yet, but may be implemented in the future. Only **declared** variables can be substituted. Trying to substitute undeclared variables or non-existing built-in variable results into configuration syntax error.

There are the number of builtin variables that start with `lithos:`:

lithos:name Name of the process, same as inserted in `LITHOS_NAME` environment variable

lithos:config_filename Full path of this configuration file as visible from within container

lithos:pid Pid of the process as visible inside of the container. Note: this variable can only be in environment and can only be full value of the variable. I.e. *PID: "@{lithos:pid}"* is fine, but *PID: "pid is @{lithos:pid}"* is **not allowed**. (In most cases this variable is exaclty `2`, this is expected but might not be always true in some cases).

More built-in variables may be added in the future. Built-in variables doesn't have to be declared.

5.2 Reference

`kind`
One of `Daemon` (default), `Command` or `CommandOrDaemon`.

The `Daemon` is long-running process that is monitored by supervisor.

The `Command` things are just one-off tasks, for example to initialize local file system data, or to check health of daemon process. The `Command` things are run by `lithos_cmd` utility

The `CommandOrDaemon` may be used in both ways, based on how it was declared in *Process Config*. In the command itself you can distinguish how it is run by `/cmd.` in `LITHOS_NAME` or cgroup name or better you can pass *variable* to a specific command and/or daemon.

New in version 0.10.3: `ContainerOrDaemon` mode

`user-id`
The numeric user indentifier for the process. It must be one of the allowed values in lithos configuration. Usually value of `0` is not allowed.

`group-id`
The numeric group indentifier for the process. It must be one of the allowed values in lithos configuration. Usually value of `0` is not allowed.

`memory-limit`
The memory limit for process and it's children. This is enforced by cgroups, so this needs *memory* cgroup to be enabled (otherwise its no-op). See *`cgroup-controllers`* for more info. Default: nolimit.

You can use `ki`, `Mi` and `Gi` units for memory accounting. See integer-units.

Changed in version 0.14.0: Previously it only set `memory.limit_in_bytes` but now it also sets `memory.memsw.limit_in_bytes` if the latter exists (otherwise skipping silently). This helps to kill processes earlier instead of swapping out to disk.

`cpu-shares`
The number of CPU shares for the process. Default is `1024` which means all processes get equal share. You may split them to different values like `768` for one process and `256` for another one.

This is enforced by cgroups, so this needs *cpu* cgroup to be enabled (otherwise its no-op). See *`cgroup-controllers`* for more info.

`fileno-limit`
The limit on file descriptors for process. Default `1024`.

`restart-timeout`
The minimum time to wait between subsequent restarts of failed processes in seconds. This is to ensure that it doesn't boggles down CPU. Default is `1` second. It's enough so that lithos itself do not hang. But it should be bigger for heavy-weight processes. Note: this is time between restarts, i.e. if process were running more than this number of seconds it will be restarted immediately.

kill-timeout
(default 5 seconds) The time to wait for application to die. If it is not dead by this number of seconds we kill it with KILL.

You should not rely on this timeout to be precise for multiple reasons:

1. Unidentified children are killed with a default timeout (5 sec). This includes children which are being killed when their configuration is removed.
2. When lithos is restarted (i.e. to reload a configuration) during the timeout, the timeout is reset. I.e. the process may hang more than this time.

executable
The path to executable to run. Only absolute paths are allowed.

arguments
The list of arguments for the command. Except argument zero.

environ
The mapping of values that are set for process. You must set all needed environment variables here. The only variable that is propagated by default is TERM. Also few special LITHOS_ variables may be set. This means you must set all the basic LANG, HOME and so on explicitly. This is to ensure that your environment is always the same regardless of where you run process.

secret-environ
Similarlty to environ but contains encrypted environment variables. For example:

```
secret-environ:
  DB_PASSWORD:␣
↪v2:ROit92I5:82HdsExJ:Gd3ocJsr:Hp3pngQZUos5b8ioKVUx40kegM1uDsYWwsWqC1cJ1/
↪1KmQPQQWJZe86xgl1EOIxbuLj6PUlBH8yz5qCnWp//Ofbc
```

Note: if environment variable is both in environ and secret-environ which one overrides is not specified for now.

You can encrypt variables using lithos_crypt:

```
lithos_crypt encrypt -k key.pub -d "secret" -n "some.namespace"
```

You only need public key for encryption. So the idea is that public key is published somewhere and anyone, even users having to access to server/private key can add a secret.

The -n / --namespace parameter must match one of the *secrets-namespaces* defined for project's sandbox.

Usually there is only one private key for every deployment (cluster), and a single namespace per project. But in some cases you might need single lithos config for multiple destinations or just want to rotate private key smoothly. So you can put secret(s) encoded for multiple keys and/or namespaces:

```
secret-environ:
  DB_PASSWORD:
  - v2:h+M9Ue9x:82HdsExJ:Gd3ocJsr:/+f4ezLfKIP/
↪mp0xdF7H6gfdM7onHWwbGFQX+M1aB+PoCNQidKyz/1yEGrwxD+i+qBGwLVBIXRqIc5FJ6/hw26CE
  -␣
↪v2:ROit92I5:cX9ciQzf:Gd3ocJsr:LMHBRtPFpMRRrljNnkaU6Y9JyVvEukRiDs4mitnTksNGSX5xU/
↪zADWDwEOCOtYoelbJeyDdPhM7Q1mEOSwjeyO317Q==
  - v2:h+M9Ue9x:82HdsExJ:Gd3ocJsr:/+f4ezLfKIP/
↪mp0xdF7H6gfdM7onHWwbGFQX+M1aB+PoCNQidKyz/1yEGrwxD+i+qBGwLVBIXRqIc5FJ6/hw26CE
```

Note: technically you can encrypt different secrets here, we can't enforce that, but it's very discouraged.

The underlying encyrption is curve25519xsalsa20poly1305 which is compatible with libnacl and libsodium.

See *Encrypted Variables* for more info.

This option conflicts with *secret-environ-file*.

secret-environ-file
Path to the file where to read secret environ from. Instead of including `secret-environ` in the container config itself you can use a separate file where data is contained. This is useful to keep single set of secrets shared between multiple containers.

The target file is also yaml, but it containers just mapping of names of the secrets to their values (or lists). For example:

```
PASSWD1: v2:ROit92I5:82HdsExJ:Gd3ocJsr:Hp3pngQZUos5b8ioKVUx40kegM1uDsYWwsWqC1cJ1/
↪1KmQPQQWJZe86xgl1EOIxbuLj6PUlBH8yz5qCnWp//Ofbc
PASSWD2:
- v2:h+M9Ue9x:82HdsExJ:Gd3ocJsr:/+f4ezLfKIP/
↪mp0xdF7H6gfdM7onHWwbGFQX+M1aB+PoCNQidKyz/1yEGrwxD+i+qBGwLVBIXRqIc5FJ6/hw26CE
- v2:ROit92I5:cX9ciQzf:Gd3ocJsr:LMHBRtPFpMRRrljNnkaU6Y9JyVvEukRiDs4mitnTksNGSX5xU/
↪zADWDwEOCOtYoelbJeyDdPhM7Q1mEOSwjeyO317Q==
```

Absolute paths here interpreted relative to the container root and relative paths are interpreted relative to the container config itself. Note: we currently support reading file from container's filesystem only, whether reading from a volume works or not is *unspecified* at the moment.

This option conflicts with *secret-environ*.

workdir
The working directory for target process. Default is `/`. Working directory must be absolute.

resolv-conf

Parameters of the `/etc/resolv.conf` file to generate. Default configuration is:

```
resolv-conf:
    mount: nil  # which basically means "auto"
    copy-from-host: true
```

Which means `resolv.conf` from host where lithos is running is copied to the "state" directory of the container. Then if `/etc/resolv.conf` in container is a file (and not a symlink) resolv conf is mounted over the `/etc/resolv.conf`.

More options are expected to be added later.

Changed in version 0.15.0: `mount` option added. Previously to make use of `resolv.conf` you should symlink `ln -s /state/resolv.conf /etc/resolv.conf` in the container's image.

Another change is that `copy-from-host` copies file that is specified in sandbox's `resolv.conf` which default to `/etc/resolv.conf` but may be different.

Parameters:

copy-from-host (default `true`) Copy `resolv.conf` file from host machine.

Note: even if `copy-from-host` is `true`, *additional-hosts* from sandbox config work, which may lead to duplicate or conflicting entries if some names are specified in both places.

Changed in version v0.11.0: The parameter used to be `false` by default, because we were thinking about better (perceived) isolation.

mount (default `nil`, which means "auto") Mount copied `resolv.conf` file over `/etc/resolv.conf`.

nil enables mounting if `/etc/resolv.conf` is present in the container and is a file (not a symlink) and also `copy-from-host` is true

New in version 0.15.0.

`hosts-file`

Parameters of the `/etc/hosts` file to generate. Default configuration is:

```
hosts-file:
    mount: nil  # which basically means "auto"
    localhost: true
    public-hostname: true
    copy-from-host: false
```

Changed in version 0.15.0: `mount` option added. Previously to make use of `resolv.conf` you should symlink `ln -s /state/resolv.conf /etc/resolv.conf` in the container's image.

Another change is that `copy-from-host` copies file that is specified in sandbox's `resolv.conf` which default to `/etc/resolv.conf` but may be different.

Parameters:

copy-from-host (default `true`) Copy hosts file from host machine.

Note: even if `copy-from-host` is `true`, *`additional-hosts`* from sandbox config work, which may lead to duplicate or conflicting entries if some names are specified in both places.

Changed in version v0.11.0: The parameter used to be `false` by default, because we were thinking about better (perceived) isolation. And also because hostname in Ubuntu doesn't resolve to real IP of the host. But we find those occassions where it matters to be quite rare in practice and using `hosts-file` as well as `resolv.conf` from the host system as the most expected and intuitive behavior.

mount (default `nil`, which means "auto") Mount produced `hosts` file over `/etc/hosts`.

nil enables mounting if `/etc/hosts` is present in the container and is a file (not a symlink).

Value of `true` fails if `/etc/hosts` is not a file. Value of `false` leaves `/etc/hosts` intact.

New in version 0.15.0.

localhost (default is true when `copy-from-host` is false) A boolean which defines whether to add `127.0.0.1 localhost` record to `hosts`

public-hostname (default is true when `copy-from-host` is false) Add to `hosts` file the result of `gethostname` system call along with the ip address that name resolves into.

`uid-map, gid-map`

The list of mapping for uids(gids) in the user namespace of the container. If they are not specified the user namespace is not used. This setting allows to run processes with `uid` zero without the risk of being the `root` on host system.

Here is a example of maps:

```
uid-map:
- {inside: 0, outside: 1000, count: 1}
- {inside: 1, outside: 1, count: 1}
gid-map:
- {inside: 0, outside: 100, count: 1}
```

Note: Currently you may have uid-map either in a sandbox or in a container config, not both.

stdout-stderr-file
This redirects both stdout and stderr to a file. The path is opened inside the container. So must reside on one of the mounted writeable *Volumes*. Probably you want `Persistent` volume. While it can be on `Tmpfs` or `Statedir` the applicability of such thing is very limited.

Usually log is put into the directory specified by `stdio-log-dir`.

interactive
(default `false`) Useful only for containers of kind `Command`. If `true` lithos_cmd doesn't clobber stdin and doesn't redirect stdout and stderr to a log file, effectively allowing command to be used for interactive commands or as a part of pipeline.

Note: for certain use cases, like pipelines it might be better to use fifo's (see `man mkfifo`) and a `Daemon` instead of this one because daemons may be restarted on death or for software upgrade, while `Command` is not supervised by lithos.

New in version 0.6.3.

Changed in version 0.5: Commands were always interactive

restart-process-only
(default `false`) If true when restarting process (i.e. in case process died or was killed), lithos restarts just the failed process. This means container will not be recreated, volumes will not be remounted, tmpfs will not be cleaned and some daemon processes may leave running.

By default `lithos_knot` which is pid 1 in the container exits when process dies. Which means all other processes will die on `KILL` signal, and container will be removed and created again. It's a little bit slower but safer default. This leaves no hanging daemons, orphan files in state dir and tmpfs garbage.

volumes
The mapping of mountpoint to volume definition. See *Volumes* for more info

tcp-ports
Binds address and provides file descriptor to the child process. All the children receive dup of the same file descriptor, so may all do `accept()` simultaneously. The configuration looks like:

```
tcp-ports:
  7777:
    fd: 3
    host: 0.0.0.0
    listen-backlog: 128
    reuse-addr: true
    reuse-port: false
```

All the fields except `fd` are optional.

Programs may require to pass listening file descriptor number by some means (usually environment). For example to run nginx with port bound (so you don't need to start it as root) you need:

```
tcp-ports:
  80:
    fd: 3
    set-non-block: true
```

(continues on next page)

(continued from previous page)

```
environ:
  NGINX: "3;"
```

To run gunicorn you may want:

```
tcp-ports:
  80:
    fd: 3
environ:
  GUNICORN_FD: "3"
```

More examples are in *Handing TCP Ports*

Parameters:

key TCP port number.

> **Warning:**
>
> - The paramters (except `fd`) do not change after socket is bound even if configuration change
> - You can't bind same port with different hostnames in a **single process** (previously there was a global limit for the single port for whole lithos master, currently this is limited just because `tcp-ports` is a mapping)

Port parameter should be unique amongst all containers. But sharing port works because it is useful if you are doing smooth software upgrade (i.e. you have few old processes running and few new processes running both sharing same port/file-descriptor). *Running them on single port is not the best practices for smooth software upgrade but that topic if out of scope of this documentation.*

fd *Required*. File descriptor number

host (default is `0.0.0.0` meaning all addresses) Host to bind to. It must be IP address, hostname is not supported.

listen-backlog (default `128`) the value to pass to the *listen()* system call. The value is capped by `net.core.somaxconn`

reuse-addr (default `true`) Sets `SO_REUSEADDR` socket option

reuse-port (default `false`) If set to `true` this changes behavior of the lithos with respect of the socket. In default case lithos binds socket as quick as possible and passes to each child on start. When this set to `true`, lithos creates a separate socket and calls bind for each process start. This has two consequences:

- Socket is not bound when no processes started (i.e. they are failing)
- Each process gets separate in-kernel queue of connections to accept

This should be set to `true` only on very high performant servers that experience assymetric workload in default case.

set-non-block (default `false`) Sets socket into non-blocking mode. This is usually done by an application itself but some of them (especially ones, that don't expect socket to be created by an external utility, e.g. nginx) don't do it themselves.

external (default `false`) If set to `true` listen on the port in the external network (host network of the system not bridged network). This is only effective if *`bridged-network`* is enabled for container.

Changed in version 0.17.0: Previously we only allowed external ports to be declared in lithos config. It was expected that container in bridged network can listen port itself. But it turned out file descriptors are still convenient for some use-cases even inside a bridge.

metadata
(optional) Allows to add arbitrary metadata to lithos configuration file. Lithos does not use and does not validate this data in any way (except that it must be a valid YAML). The metadata can be used by other tools that inspect lithos configs and extract data from it. In particular, we use metadata for our deployment tools (to keep configuration files more consolidated instead of keeping then in small fragments).

normal-exit-codes
(optional) A list of exit codes which are considered normal for process death. This currently only improves `failures` metric. See *Determining Failure*.

Note: by default even `0` exit code is considered an error for daemons, and for commands (`lithos_cmd`) `0` is considered successful.

This setting is intended for daemons which may voluntarily exit for some reason (soft memory limit, version upgrade, configuration reload).

It's not recommended to add *0* or *1* into the list, as some commands threat them pretty arbitrarily. For example *0* is exit code of most utilities running *–help* so this mistake will not be detected. And *1* is used for arbitrary crashes in scripting languages. So the good idea is to define some specific code in range of *8..120* to define successful exit.

CHAPTER 6

Metrics

Lithos submits metrics via a cantal-compatible protocol.

All metrics usually belong to lithos's cgroup, so for example in graphite you can find them under `cantal.<cluster-name>.<hostname>.lithos.groups.*`. Or you cand find them without this prefix in `http://hostname:22682/local/process_metrics` without a prefix.

In the following description we skip the common prefix and only show metric names.

Metrics of lithos master process:

- `master.restarts` (counter) amount of restarts of a master process. Usually restart equals to configuration reload via `lithos_switch` or any other way.
- `master.sandboxes` (gauge) number of sandboxes configured
- `master.containers` (gauge) number of containers (processes) conigured
- `master.queue` (gauge) length of the internal queue, the queue consists of processes to run and hanging processes to kill

Per-process metrics:

- `processes.<sandbox_name>.<process_name>.started` – (counter) number of times process have been started
- `processes.<sandbox_name>.<process_name>.deaths` – (counter) number of times process have exited for any reason
- `processes.<sandbox_name>.<process_name>.failures` – (counter) number of times process have exited for failure reason, for whatever reason lithos thinks it was failure. See *Determining Failure*
- `processes.<sandbox_name>.<process_name>.running` – (gauge) number of procesess that are currently running (was started but not yet found to be exited)

Global metrics for all sandboxes and containers:

- `containers.started` – (counter) same as for `processes.*` but for all containers
- `containers.deaths` – (counter) see above

- `containers.failures` – (counter) see above
- `containers.running` – (gauge) see above
- `containers.unknown` – (gauge) number of child processes of lithos that are found to be running but do not belong to any of the process groups known to lithos (they are being killed, and they are probably from deleted configs)

6.1 Determining Failure

Currently there are two kinds of process death that are considered non-failures:

1. Processes that had been sent `SIGTERM` signal to (with any exit status) or ones dead on `SIGTERM` signal are considered non-failed.
2. Processes exited with one of the exit codes specified in *`normal-exit-codes`*

CHAPTER 7

Volumes

Volumes in lithos are just some kind of mount-points. The mount points are not created by `lithos` itself. So they must exist either in original image. Or on respective volume (if mount point is inside a volume).

There are the following kinds of volumes:

`Readonly`
: Example: `!Readonly "/path/to/dir"`

  A **read-only** bind mount for some dir. The directory is mounted with `ro,nosuid,noexec,nodev`

`Persistent`
: Example: `!Persistent { path: /path/to/dir, mkdir: false, mode: 0o700, user: 0, group: 0 }`

  A **writeable** bind mount. The directory is mounted with `rw,nosuid,noexec,nodev`. If you need directory to be created set `mkdir` to `true`. You also probably need to customize either the user (to the one running command e.g. same as `user-id` of the container) or the mode (to something like `0o1777`, i.e. sticky writable by anyone).

`Statedir`
: Example: `!Statedir { path: /, mode: 0o700, user: 0, group: 0 }`

  Mount subdir of the container's own state directory. This directory is used to store generated `resolv.conf` and `hosts` files as well as for other kinds of small state which is dropped when container dies. If you mount something other than `/` you should custimize mode or an owner similarly to `!Persistent` volumes (except that you can't create statedir subdirectory by hand because statedir is created for each process at start)

`Tmpfs`
: Example: `!Tmpfs { size: 100Mi, mode: 0o766 }`

  The tmpfs mount point. Currently only `size` and `mode` options supported. Note that syntax of size and mode is generic syntax for numbers for our configuration library, not the syntax supported by kernel.

CHAPTER 8

Tips and Conventions

This documents describes how to prepare images to run by lithos. You don't have to obey all the rules. And you are free to create your own rules within the organization. But hopefully this will help you a lot when you're confused.

Contents:

8.1 Handing TCP Ports

There are couple of reasons you want `lithos` to open tcp port on behalf of your application:

1. Running multiple instances of the application, each sharing the same port
2. Smooth upgrade of you app, where some of processes are running old version of software and some run new one
3. Grow and shrink number of processes without any application code to support that
4. Using port < 1024 and not starting process as root
5. Each process is in separate cgroup, so monitoring tools can have fine-grained metrics over them

---

Note: While you could use `SO_REUSE_PORT` socket option for solving #1 it's not universally available option.

Forking inside the application doesn't work as well as running each process by lithos because in the former case your memory limits apply to all the processes rather than being fine-grained.

Following sections describe how to configure various software stacks and frameworks to use tcp-ports opened by lithos.

It's possible to run any software that supports systemd socket activation with *`tcp-ports`* of lithos. With the config similar to this:

```
environ:
  LISTEN_FDS: 1   # application receives single file descriptor
  # ... more env vars ...
tcp-ports:
  8080: # port number
    fd: 3   # SD_LISTEN_FDS_START, first fd number systemd passes
    host: 0.0.0.0
    listen-backlog: 128   # application may change this on its own
    reuse-addr: true
# ... other process settings ...
```

8.1.1 Python3 + Asyncio

For development purposes you probably have the code like this:

```
async def init(app):
    ...
    handler = app.make_handler()
    srv = await loop.create_server(handler, host, port)
```

To use tcp-ports you should check environment variable and pass socket if that exists:

```
import os
import socket

async def init(app):
    ...
    handler = app.make_handler()
    if os.environ.get("LISTEN_FDS") == "1":
        srv = await loop.create_server(handler,
            sock=socket.fromfd(3, socket.AF_INET, socket.SOCK_STREAM))
    else:
        srv = await loop.create_server(handler, host, port)
```

This assumes you are configured `environ` and `tcp-ports` as *described above*.

8.1.2 Python + Werkzeug (Flask)

Werkzeug supports the functionality out of the box, just put configure the environment:

```
environ:
  WERKZEUG_SERVER_FD: 3
  # ... more env vars ...
tcp-ports:
  8080: # port number
    fd: 3  # this corresponds to WERKZEUG_SERVER_FD
    host: 0.0.0.0
    listen-backlog: 128   # default in werkzeug
    reuse-addr: true
# ... other process settings ...
```

Or you can pass `fd=3` to `werkzeug.serving.BaseWSGIServer`.

Another hint: **do not use processes != 1**. Better use lithos's `instances` to control the number of processes.

8.1.3 Python + Twisted

Old code that looks like:

```
reactor.listenTCP(PORT, factory)
```

You need to change into something like this:

```
if os.environ.get("LISTEN_FD") == "1":
    import socket
    sock = socket.fromfd(3, socket.AF_INET, socket.SOCK_STREAM)
    sock.set_blocking(False)
    reactor.adoptStreamPort(sock.fileno(), AF_INET, factory)
    sock.close()
    os.close(3)
else:
    reactor.listenTCP(PORT, factory)
```

8.1.4 Golang + net/http

Previous code like this:

```
import "net/http"

srv := &http.Server{ .. }
if err := srv.ListenAndServe(); err != nil {
    log.Fatalf("Error listening")
}
```

You should wrap into something like this:

```
import "os"
import "net"
import "net/http"

srv := &http.Server{ .. }
if os.Getenv("LISTEN_FDS") == "1" {
    listener, err := net.FileListener(os.NewFile(3, "fd 3"))
    if err != nil {
        log.Fatalf("Can't open fd 3")
    }
    if err := srv.Serve(listener); err != nil {
        log.Fatalf("Error listening on fd 3")
    }
} else {
    if err := srv.ListenAndServe(); err != nil {
        log.Fatalf("Error listening")
    }
}
```

8.1.5 Node.js with Express Framework

Normal way to run express:

```
let port = 3000
app.listen(port, function() {
    console.log('server is listening on', this.address().port);
})
```

Turns into the following code:

```
let port = 3000;
if (process.env.LISTEN_FDS && parseInt(process.env.LISTEN_FDS, 10) === 1) {
    port = {fd:3};
}
app.listen(port, function() {
    console.log('server is listening on', this.address().port);
})
```

8.2 Deploying Vagga Containers

Vagga is a common way to develop applications for later deployment using lithos. Also vagga is a common way to prepare a container image for use with lithos.

Usually vagga does it's best to make containers as close to production as possible. Still vagga tries to make good trade-off to make it's easier to use for development, so there are few small quircks that you may or may not notice when deploying.

Here is a boring list, later sections describe some things in more detail:

1. Unsurprisingly `/work` directory is absent in production container. Usually this means three things:
   (a) Your sources must be copied/installed into container (e.g. using Copy)
   (b) There is no current working directory, unless you specify it explicitly current directory is root `/`
   (c) You can't **write** into working directory or `/work/somewhere`
2. All directories are read-only by default. Basic consequences are:
   (a) There is no writable `/tmp` unless you specify one. This also means there is no default for temporary dir, you have to chose whether this is an in-memory *`Tmpfs`* or on-disk *`Persistent`*.
   (b) There is no `/dev/shm` by default. This is just another `tmpfs` volume in every system nowadays, so just measure how much you need and mount a *`Tmpfs`*. Be aware that each container even on same machine get's it's own instance.
   (c) We can't even overwrite `/etc/resolv.conf` and `/etc/hosts`, see below.
3. There are few environment variables that vagga sets in container by default:
   (a) `TERM` – is propagated from external environment. For daemons it should never matter. For *`interactive`* commands it may matter.
   (b) `PATH` – in vagga is set to hard-coded value. There is no default value in lithos. If your program runs any binaries (and usually lots of them do, even if you don't expect), you want to set `PATH`.
   (c) Various `*_proxy` variables are propagated. They are almost never useful for daemons. But are written here for completeness.
4. In vagga we don't update `/etc/resolv.conf` and `/etc/hosts`, but in lithos we have such mechanism. The mechanism is following:

   (a) In container you make the symlinks `/etc/resolv.conf -> /state/resolv.conf`, `/etc/hosts -> /state/hosts`

   (b) The `/state` directory is mounted as *`Statedir`*

   (c) Lithos automatically puts `resolv.conf` and `hosts` into statedir when container is created (respecting *`resolv-conf`* and *`hosts-file`*)

   (d) Then files can be updated by updating files in `/var/run/lithos/state/<sandbox>/<process>/`

5. Because by default neither vagga nor lithos have network isolation, some things that are accessible in the dev system may not be accessible in the server system. This includes both, services on `localhost` as well as in **abstract unix socket namespace**. Known examples are:

   (a) Dbus: for example if `DBUS_SESSION_BUS_ADDRESS` starts with `unix:abstract=`

   (b) Xorg: X Window System, the thing you configure with `DISPLAY`

   (c) nscd: name service cache daemon (this thing may resolve DNS names even if TCP/IP network is absent for your container)

   (d) systemd-resolved: listens at `127.0.0.53:53` as well as on **dbus**

8.3 Storing Secrets

There are currently two ways to provide "secrets" for containers:

1. Encrypted values inserted into environment variable
2. Mount a directory from the host system

8.3.1 Encrypted Variables

Guide

Note: this guide covers both server setup and configuring specific containers. Usually setup (steps 1-3) is done once. And adding keys to a container (steps 4-5) is more regular job.

1. Create a key private key on the server:

   ```
   ssh-keygen -f /etc/lithos/keys/main.key -t ed25519 -P ""
   ```

   You can create a shared key or a per-project key. Depending on your convenience. Synchronize the key accross all the servers in the same cluster. This key should **never leave** that set of servers.

2. Add the reference to the key into your *Sandbox Config* (e.g. `/etc/lithos/sandboxes/myapp.yaml`):

```
secrets-private-key: /etc/lithos/keys/main.key
secrets-namespaces: [myapp]
```

You can omit `secrets-namespaces` if you're sole owner of this server/cluster (it allows only empty string as a namespace). You can also make per-process namespaces (*`extra-secrets-namespaces`*).

3. Publish your public key `/etc/lithos/keys/main.key.pub` for your users. *(Cryptography guarantees that even if this key is shared publically, i.e. commited into a git repo, or accessible over non-authorized web URL system is safe)*

4. Your users may now fetch the public key and encrypt their secrets with `lithos_crypt` (get static binary on releases page):

```
$ lithos_crypt encrypt -k main.key.pub -n myapp -d the_secret
v2:ROit92I5:KqWSX0BY:8MtOoWUX:nHcVCIWZG2hivi0rKa8MRnAIbt7TDTHB8YC8bBnac3IGMzk57R/
↪HsBhxeqCdC7Ljyf8pszBBjIGD33f6lwBM7Q==
```

The important thing here is to encrypt with the right key **and** the right namespace.

5. Then put a secret into your *Container Configuration*:

```
executable: /usr/bin/python3
environ:
  DATABASE_URL: postgresql://myappuser@db.example.com/myappdb
secret-environ:
  DATABASE_PASSWORD:␣
↪v2:ROit92I5:KqWSX0BY:8MtOoWUX:nHcVCIWZG2hivi0rKa8MRnAIbt7TDTHB8YC8bBnac3IGMzk57R/
↪HsBhxeqCdC7Ljyf8pszBBjIGD33f6lwBM7Q==
```

That's it. To add a new password to the same or another container repeat steps 4-5.

This scheme is specifically designed to be safe to store in a (public) git repository by using secure encryption.

Anatomy of the Encrypted Key

As you might see there is a pattern in an encrypted key. Here is how it looks like:

```
v2:ROit92I5:KqWSX0BY:8MtOoWUX:nHcVCIWZG2hivi0rKa8MRnAIbt7TDTHB8YC8bBnac3IGMwBM7Q==
                           ^-- encrypted "namespace:actual_secret"
                     ^^^^^^^^-- short hash of the password itself
            ^^^^^^^^-- short hash of the secrets namespace
   ^^^^^^^^-- short hash of the public key used for encryption
^^-- encryption version
```

Note the following things:

1. Only version `v2` is supported (`v1` was broken and dropped in 0.16.0)

2. The short hash is base64-encoded 6-bytes length blake2b hash of the value. You can check in using `b2sum` utility from recent version of `coreutils`:

```
$ echo -n "the_secret" | b2sum -l48 | xxd -r -p | base64
8MtOoWUX
```

(Note: we need `xxd` because `b2sum` outputs hexadecimal bytes, also note `-n` in `echo` command, as it's a common mistake, without the option `echo` outputs newline at the end).

3. The encrypted payload contains `<namespace>:` prefix. While we could check just the hash. Prefix allows providing better error messages.

   The underlying encyrption is curve25519xsalsa20poly1305 which is compatible with libnacl and libsodium.

Let's see how it might be helpful, here is the list of keys:

```
v2:h+M9Ue9x:82HdsExJ:Gd3ocJsr:/+f4ezLfKIP/mp0xdF7H6gfdM7onHWwbGFQX+M1aB+PoCNQidKyz/
↪1yEGrwxD+i+qBGwLVBIXRqIc5FJ6/hw26CE
v2:ROit92I5:cX9ciQzf:Gd3ocJsr:LMHBRtPFpMRRrljNnkaU6Y9JyVvEukRiDs4mitnTksNGSX5xU/
↪zADWDwEOCOtYoelbJeyDdPhM7Q1mEOSwjeyO317Q==
v2:ROit92I5:82HdsExJ:Gd3ocJsr:Hp3pngQZUos5b8ioKVUx40kegM1uDsYWwsWqC1cJ1/
↪1KmQPQQWJZe86xgl1EOIxbuLj6PUlBH8yz5qCnWp//Ofbc
```

You can see that:

1. All of them have same secret (3rd column)
2. Second and third ones have same encryption key (1st column)
3. First and third ones have the same namespace (2nd column)

This is useful for versioning and debugging problems. You can't deduce the actual password from this data anyway unless your password is very simple (dictioanry attack) or you already know it.

Note: even if all three {encryption key, namespace, secret} match, the last part of data (encrypted payload) will be different each time you encode that same value. All of the outputs are equally right.

Security Notes

1. Namespaces allow to divide security zones between many projects without nightmare of generating, syncing and managing secret keys per project.
2. Namespaces match exactly they aren't prefixes or any other kind of pattern
3. If you rely on `lithos_switch` to switch containers securely (with untrusted *Process Config*), you need to use different private key per project (as otherwise `extra-secrets-namespaces` can be used to steal keys)

CHAPTER 9

Frequently Asked Questions

9.1 How do I Start/Stop/Restart Processes Running By Lithos?

Short answer: You can't.

Long answer: Lithos keep running all the processes that it's configured to run. So:

- To stop process: remove it from the config
- To start process: add it to the config. If it's added, it will be restarted indefinitely. Sometimes may want to fix *restart-timeout*
- To restart process: well, kill it (with whatever signal you want).

The ergonomic of these operations is intentionally not very pleasing. This is because you are supposed to have higher-level tool to manage lithos. At least you want to use ansible, chef or puppet.

9.2 Why /run/lithos/mnt is empty?

This is a mount point. It's never mounted in host system namespace (well it's never visible in guest namespace too). The containerization works as follows:

1. The mount namespace is *unshared* (which means no future mounts are visible in the host system)
2. The root filesystem image is mounted to `/run/lithos/mnt`
3. Other things set up in root file system (`/dev`, `/etc/hosts`, whatever)
4. Pivot root is done, which means that `/run/lithos/mnt` is now visible as root dir, i.e. just plain `/` (you can think of it as good old `chroot`)

This all means that if you error like this:

```
[2015-11-17T10:29:40Z][ERROR] Fatal error: Can't mount pseudofs /run/lithos/mnt/dev/
↪pts (newinstance, options: devpts): No such file or directory (os error 2)
```

Or like this:

```
[2015-10-19T15:04:48Z][ERROR] Fatal error: Can't mount bind /whereever/external/
↪storage/is to /run/lithos/mnt/storage: No such file or directory (os error 2)
```

It means that lithos have failed on step #3. And that it failed to mount the directory in the guest container file system (`/dev/pts` and `/storage` respectively)

9.3 How to Organize Logging?

There is variety of ways. Here are some hints...

9.3.1 Syslog

You may accept logs by UDP. Since lithos has no network namespacing (yet). The UDP syslog just works.

To setup syslog using unix sockets you may configure syslog daemon on the host system to listen for the socket inside the container's `/dev`. For example, here is how to configure rsyslog for default lithos config:

```
module(load="imuxsock") # needs to be done just once
input(type="imuxsock" Socket="/var/lib/lithos/dev/log")
```

Alternatively, (but *not* recommended) you may configure *devfs-dir*:

```
devfs-dir: /dev
```

9.3.2 Stdout/Stderr

It's recommended to use syslog or any similar solutions for logs. But there are still reasons to write logs to a file:

1. You may want to log early start errors (when you have not yet initialized the logging subsystem of the application)
2. If you have single server and don't want additional daemons

Starting with version `v0.5.0` lithos has a per-sandbox log file which contains all the stdout/stderr output of the processes. By default it's in `/var/log/lithos/stderr/<sandbox_name>.log`. See *stdio-log-dir* for more info.

9.4 How to Update Configs?

The best way to update config of *processes* is to put it into a temporary file and run `lithos_switch` (see `lithos_switch --help` for more info). This is a main kind config you update multiple times a day.

In case you've already put config in place, or for *master* and *sandbox* config, you should first run `lithos_check` to check that all configs are valid. Then just send `QUIT` signal to the `lithos_tree` process. Usually the following command-line is enough for manual operation:

```
pkill -QUIT lithos_tree
```

But if you for authomation it's better to use `lithos_switch`.

Note: note

By sending `QUIT` signal we're effectivaly emulate crash of the supervisor daemon. It's designed in a way that allows it survive crash and keep all fresh child processes alive. After an **in-place restart** it checks configuration of child processes, kills outdated ones and executes new configs.

9.5 How to Run Commands in Container?

There are two common ways:

1. If you have container already running use `nsenter`
2. Prepare a special command for `lithos_cmd`

9.5.1 Running `nsenter`

This way only works if you have a running container. It's hard to get work if your process crashes too fast after start.

You must also have a working shell in container, we use `/bin/sh` in examples.

You can use `nsenter` to join most namespaces, except user namespace. For example, if you know pid, the following command would allow you to run shell in container and investigate files:

```
nsenter -m -p --target 12345 /bin/sh
```

If you don't know PID, you may easily discover it with `lithos_ps` or automate it with `pgrep`:

```
nsenter -m -p \
    --target=$(pgrep -f 'lithos_knot --name sandbox-name/process-name.0') \
    /bin/sh
```

Warning: This method is very insecure. It runs command in original user namespace with the host root user. While basic sandboxing (i.e. filesystem root) is enabled by *-m* and *-p*, the program that you're trying to run (i.e. the shell itself) can still escape that sandbox.

Because we do mount namespaces and user namespaces in different stages of container initialization there is currently no way to join both user namespace and mount namespace. (You can join just user namespace by running `nsenter -U --target=1235` where 123 is the pid of the process inside the container, not lithos_knot. But this is probably useless)

9.5.2 Running `lithos_cmd`

In some cases you may want to have a special container with a shell to run with `lithos_cmd`. This is just a normal lithos container configuration with `kind: Command` and `interactive: true` and shell being specified as a command. So you run your `shell.yaml` with:

```
lithos_cmd sandbox-name shell
```

There are three important points about this method:

1. If you're trying to investigate problem with the daemon config you copy daemon config into this interactive command. It's your job to keep both configs in sync. This config must also be exposed in *processes* config just like any other.
2. It will run another (although identical) container on each run. You will not see processes running as daemons and other shells in `ps` or similar commands.
3. You must have shell in container to get use of it. Sometimes you just don't have it. But you may use any interactive interpreter, like `python` or even non-interactive commands.

9.6 How to Find Files Mounted in Container?

Linux provides many great tools to introspect running container. Here is short overview:

1. `/proc/<pid>/root` is a directory where you can `cd` into and look at files
2. `/proc/<pid>/mountinfo` is a mapping between host system directories and ones container
3. And you can *join container's namespace*

9.6.1 Example 1

Let's try to explore some common tasks. First, let's find container's pid:

```
$ pgrep -f 'lithos_name --name sandbox-name/process-name.0'
12345
```

Now we can find out the OS release used to build container:

```
$ sudo cat /proc/12345/root/etc/alpine-release
3.4.6
```

Warning: There is a caveat. Symlinks that point to paths starting with root are resolved differently that in container. So ensure that you're not accessing a symlink (and that any intermediate components is not a symlink).

9.6.2 Example 2

Now, let's find out which volume is mounted as `/app/data` inside the container.

If you have quire recent `findmnt` it's easy:

```
$ findmnt -N 12345 /app/data
TARGET     SOURCE                                          FSTYPE OPTIONS
/app/data  /dev/mapper/Disk-main[/all-storages/myproject] ext4   rw,noatime,discard,
↪data=ordered
```

Here we can see that `/app/data` in container is a LVM partition `main` in group `Disk` with the path `all-storages/myproject` relative to the root of the partition. You can find out where this volume is mounted on host system by inspecting the output of `mount` or `findmnt` commands.

Manual way is to look at `/proc/<pid>/mountinfo` (stripped output):

```
$ cat /proc/12345/mountinfo
347 107 9:1 /all-images/sandbox-name/myproject.c17cb162 / ro,relatime - ext4 /dev/md1␣
↪rw,data=ordered
356 347 0:267 / /tmp rw,nosuid,nodev,relatime - tmpfs tmpfs rw,size=102400k
360 347 9:1 /all-storages/myproject /app/data rw,relatime - ext4 /dev/mapper/Disk-
↪main rw,data=ordered
```

Here you can observe same info. Important parts are:

- Fifth column is the mountpoint (but be careful in complex cases there might be multiple overlapping mount points);
- Fourth column is the path relative to the volume root;
- And, 9th column (next to the last) is the volume name.

Let's find out where it is on host system:

```
$ mount | grep Disk-main
/dev/mapper/Disk-main on /srv type ext4 (rw,noatime,discard,data=ordered)
```

That's it, now you can look at `/srv/all-storages/myproject` to find files seen by an application.

CHAPTER 10

Lithos Changes By Release

10.1 v0.19.0

- Feature: new process in bridged network gets CAP_NET_BIND_SERVICE capability in it's own network namespace (effectively allowing it to bind port 80, 443 or any other port < 1024)
- Bugfix: made `default-gateway` in `bridged-network` optional
- Bugfix: lithos now deletes veth interface if that exists, before starting a process (previously you needed to manually resolve this issue)

Note: we're making this release major to show that it requires more testing than regular update. This is because we changed internals quite a bit to allow network namespace owned by process.

10.2 v0.18.4

- Bugfix: only send SIGTERM to the process once when upgrading or stopping it (this prevents certain issues with the applications themselves)
- Bugfix: use don't reset kill timeout on SIGQUIT of lithos_tree
- Bugfix: correctly wait for kill timeout for retired children (not in the config any more)

10.3 v0.18.3

- Bugfix: it looks like that reading through `/proc/` is inherently racy, i.e. some process may be skipped. This commit fixes walk faster and traverse directory twice. More elaborate fix will be implemented in future.

10.4 v0.18.2

- Feature: add `secret-environ-file` which can be used to offload secrets to a separate (perhaps shared) file

10.5 v0.18.1

- Feature: add `set-non-block` option to tcp-ports

10.6 v0.18.0

- Breaking: we don't run `arping` after container setup by default, as it doesn't work in certain environments. Use *`after-setup-command`* instead.

10.7 v0.17.8

- Bugfix: fixes issue with bridged networking when host system is alpine (#15)

10.8 v0.17.7

- Bugfix: log name of the process when lithos_knot failed
- Bugfix: more robust parsing of process names by lithos_ps
- Feature: add `@{lithos:pid}` magic variable

10.9 v0.17.6

- Bugfix: systemd protocol support fixed: LISTEN_FDNAMES and LISTEN_PID

10.10 v0.17.5

- Feature: check variable substitution with `lithos_check` even in `--check-container` (out of system) mode

10.11 v0.17.4

- Feature: Add `DottedName` *variable type*
- Feature: Add `activation` parameter to `TcpPort` variable

10.12 v0.17.3

- Bugfix: fix EADDRINUSE error when all children requiring file descriptor where queued for restart (throttled), bug was due to duped socket lying in scheduled command (where main socket is closed to notify peers there are no listeners)

10.13 v0.17.2

- Bugfix: previously lithos_tree process after fork but before execing lithos_knot could be recognized as undefined child and killed. This race-condition sometimes led to closing sockets prematurely and being unable to listen them again

10.14 v0.17.1

- Bugfix: passing sockets as FDs in non-bridged network was broken in v0.17.0

10.15 v0.17.0

- Breaking: add `external` flag to *`tcp-ports`*, which by default is `false` (previous behavior was equal to `external: true`)
- Bugfix: `lithos_cmd` now returns exit code 0 if underlying command is exited successfully (was broken in 0.15.5)

10.16 v0.16.0

- Breaking: remove `v1` encryption for secrets (it was alive for a week)
- Feature: add *`secrets-namespaces`* and `extra-secrets-namespaces` option to allow namespacing secrets on top of a single key
- Feature: add `v2` key encryption scheme

10.17 v0.15.6

- Feature: add *`secret-environ`* and `secrets-private-key` settings which allow to pass to the application decrypted environment variables
- Bugfix: when bridged network is enabled we use `arping` to update ARP cache

10.18 v0.15.5

- Bugfix: add support for bridged-network and ip-addresses for lithos_cmd
- Bugfix: initialize looppack interface in container when `bridged-network` is configured

- Feature: allow `lithos_cmd` without `ip_addresses` (only loopback is initialized in this case)
- Bugfix: return error result from `lithos_cmd` if inner process failed

10.19 v0.15.4

- First release that stops support of ubuntu precise and adds repository for ubuntu bionic
- Bugfix: passing TCP port as fd < 3 didn't work before, now we allow `fd: 0` and fail gracefully on 1, 2.

10.20 v0.15.3

- feature: Add *`default-user`* and *`default-group`* to simplify container config
- bugfix: fix containers having symlinks at `/etc/{resolv.conf, hosts}` (broken in v0.15.0)

10.21 v0.15.2

- bugfix: containers without bridged network work again

10.22 v0.15.1

- nothing changed, fixed tests only

10.23 v0.15.0

- feature: Add *`normal-exit-codes`* setting
- feature: Add *`resolv-conf`* and *`hosts-file`* to sandbox config
- feature: Add *`bridged-network`* option to sandbox config
- breaking: By default `/etc/hosts` and `/etc/resolv.conf` will be mounted if they are proper mount points (can be opt out in container config)

10.24 v0.14.3

- Bugfix: when more than one variable is used lithos were restarting process every time (because of unstable serialization of hashmap)

10.25 v0.14.2

- Bugfix: if `auto-clean` is different in several sandboxes looking at the same image directory we skip cleaning the dir and print a warning
- Add a timestamp to `lithos_clean` output (in `--delete-unused` mode)

10.26 v0.14.1

- Bugfix: variable substitution was broken in v0.14.0

10.27 v0.14.0

- Sets `memory.memsw.limit_in_bytes` if that exists (usually requires `swapaccount=1` in kernel params).
- Adds a warning-level message on process startup
- Duplicates startup and death messages into stderr log, so you can corelate them with application messages

10.28 v0.13.2

- Upgrades many dependencies, no significant changes or bugfixes

10.29 v0.13.1

- Adds *`auto-clean`* setting

10.30 v0.13.0

- `/dev/pts/ptmx` is created with `ptmxmode=0666`, which makes it suitable for creating ptys by unprivileged users. We always used `newinstance` option, so it should be safe enough. And it also matches how `ptmx` is configured on most systems by default

10.31 v0.12.1

- Added `image-dir-levels` parameter which allows using images in form of `xx/yy/zz` (for value of `3`) instead of bare name

10.32 v0.12.0

- Fixed order of `sandbox-name.process-name` in metrics
- Dropped setting `cantal-appname` (never were useful, because cantal actually uses cgroup name, and lithos master process actually has one)

10.33 v0.11.0

- Option `cantal-appname` added to a config
- If no `CANTAL_PATH` present in environment we set it to some default, along with `CANTAL_APPNAME=lithos` unless `cantal-appname` is overriden.
- Added default container environment `LITHOS_CONFIG`. It may be used to log config name, read metadata and other purposes.

10.34 v0.10.7

- Cantal metrics added

CHAPTER 11

Indices and tables

- genindex

Index

S

T

U

V

W