Red Hat Enterprise Linux 7 Resource Management and Linux Containers Guide

Managing system resources and administering Linux Containers on Red Hat Enterprise Linux 7

Peter Ondrejka
Douglas Silas
Martin Prpič
Rüdiger Landmann

Red Hat Enterprise Linux 7 Resource Management and Linux Containers Guide

Managing system resources and administering Linux Containers on Red Hat Enterprise Linux 7

Peter Ondrejka
Red Hat Engineering Content Services
pondrejk@redhat.com

Douglas Silas
Red Hat Engineering Content Services
dhensley@redhat.com

Martin Prpič
Red Hat Security Response Team
mprpic@redhat.com

Rüdiger Landmann
Red Hat Engineering Content Services
r.landmann@redhat.com

Abstract

Managing system resources, administering Linux Containers, and using Docker on Red Hat Enterprise Linux 7.

Table of Contents

Part I. Resource Management With Control Groups

This part covers the concepts of resource management with use of kernel control groups, describing common tasks such as creating, modifying and monitoring control groups, as well as assigning system resources to these groups.

Chapter 1. Introduction to Control Groups (Cgroups)

1.1. What are Control Groups

The *control groups*, abbreviated as *cgroups* in this guide, are a Linux kernel feature that allows you to allocate resources — such as CPU time, system memory, network bandwidth, or combinations of these resources — among hierarchically ordered groups of processes running on a system. By using cgroups, system administrators gain fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. Hardware resources can be smartly divided up among applications and users, increasing overall efficiency.

Control Groups provide a way to hierarchically group and label processes, and to apply resource limits to them. Traditionally, all processes received similar amount of system resources that administrator could modulate with the process *niceness* value. With this approach, applications that involved a large number of processes got more resources than applications with few processes, regardless of the relative importance of these applications.

Red Hat Enterprise Linux 7 moves the resource management settings from the process level to the application level by binding the system of cgroup hierarchies with the systemd unit tree. Therefore, you can manage system resources with **`systemctl`** commands, see Chapter 2, *Using Control Groups* for more information on how to use it.

In previous versions of Red Hat Enterprise Linux, system administrators built custom cgroup hierarchies with use of the **`cgconfig`** command from the *libcgroup* package. This package is now deprecated and it is not recommended to use it since it can easily create conflicts with the default cgroup hierarchy. However, *libcgroup* is still available to cover for certain specific cases, where **systemd** is not yet applicable, most notably for using the *net-prio* subsystem. See Chapter 3, *Using libcgroup Tools*.

The aforementioned tools provide a high-level interface to interact with cgroup controllers (also known as subsystems) in Linux kernel. The main cgroup controllers for resource management are *cpu*, *memory* and *blkio*, see Available Controllers in Red Hat Enterprise Linux 7 for the list of controllers enabled by default. For detailed description of resource controllers and their configurable parameters, refer to Controller-Specific Kernel Documentation.

1.2. Default Cgroup Hierarchies

By default, **systemd** automatically creates a hierarchy of *slices*, *scopes* and *services* to provide a unified structure for the cgroup tree. With the **`systemctl`** command, you can further modify this structure by creating custom slices, as shown in Section 2.1, “Creating Control Groups”. Also, **systemd** automatically mounts hierarchies for important kernel resource controllers (see Available Controllers in Red Hat Enterprise Linux 7) in the **`/sys/fs/cgroups/`** directory.

Warning

The deprecated **`cgconfig`** tool from the **`libcgroup`** package is available to mount and handle hierarchies for controllers not yet supported by **systemd** (most notably the **`net-prio`** controller). Never use **`libcgropup`** tools to modify the default hierarchies mounted by **systemd** since it would lead to unexpected behavior. The **`libcgroup`** library will be removed in the future versions of Red Hat Enterprise Linux. For more information on how to use **`cgconfig`**, see Chapter 3, *Using libcgroup Tools*.

Systemd Unit Types

All processes running on your system are child processes of the **systemd** init process. Systemd provides three unit types that are used for the purpose of resource control (for a complete list of **systemd**'s unit types, see the chapter called *Managing Services with systemd* in Red Hat Enterprise Linux 7 System Administrators Guide):

- **Service** — A group of processes, which **systemd** started based on unit configuration file. Services encapsulate the specified processes so that they can be started and stopped as a one set. Services are named in the following way:

  ```
  name.service
  ```

  Where *name* stands for the name of service.

- **Scope** — A group of externally created processes. Scopes encapsulate processes that are started and stopped by arbitrary processes via the **fork()** function and then registered at runtime with PID1. For instance, user sessions, containers, and virtual machines are exposed as scopes. Scopes are named in the following form:

  ```
  name.scope
  ```

  Here, *name* stands for the name of scope.

- **Slice** — A group of hierarchically organized units that manage system processes. Slices do not contain processes, they organize a hierarchy in which scopes and services are placed. The actual processes are contained in scopes or in services. In this hierarchical tree, every name of a slice unit corresponds with the path to the location in the hierarchy. The dash ("**-**") character acts as a separator the path components. For example, if the name of a slice looks as follows:

  ```
  parent-name.slice
  ```

  it means that a slice called *parent-name*.**slice** is a subslice of the *parent*.**slice**. This slice can have its own subslice named *parent-name-name2*.**slice**, and so on.

  There is one root slice of all slices denoted as:

  ```
  -.slice
  ```

Service, scope and slice units directly map to objects in the cgroup tree. When these units are activated, they each map directly to cgroup paths built from the unit names. For example, a service *ex.service* contained in a *test-waldo.slice* is found in the cgroup **test.slice/test-waldo.slice/ex.service/**.

Services, scopes and slices may be created freely by the administrator and also dynamically by programs. By default, the operating system defines a number of built-in services that are necessary to start-up the system. Also, there are four slices defined by default:

- **-.slice** — the root slice
- **system.slice** — the default place for all system services
- **user.slice** — the default place for all user sessions
- **machine.slice** — the default place for all virtual machines and containers

The above is the default configuration, the administrator may define new slices and assign services and scopes to them. Also note that all login sessions are automatically placed in an individual scope unit, same as virtual machines and container processes. Furthermore, all users logging in are assigned with an implicit slice.

The following is a simplified example of a cgroup tree. This output was generated with the **systemd-cgls** command (see Section 2.4, “Obtaining Information About Control Groups”):

```
├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
├─user.slice
│ └─user-1000.slice
│   └─session-1.scope
│     ├─11459 gdm-session-worker [pam/gdm-password]
│     ├─11471 gnome-session --session gnome-classic
│     ├─11479 dbus-launch --sh-syntax --exit-with-session
│     ├─11480 /bin/dbus-daemon --fork --print-pid 4 --print-address 6 --session
│     ...
│
└─system.slice
  ├─systemd-journald.service
  │ └─422 /usr/lib/systemd/systemd-journald
  ├─bluetooth.service
  │ └─11691 /usr/sbin/bluetoothd -n
  ├─systemd-localed.service
  │ └─5328 /usr/lib/systemd/systemd-localed
  ├─colord.service
  │ └─5001 /usr/libexec/colord
  ├─sshd.service
  │ └─1191 /usr/sbin/sshd -D
  │
  ...
```

As you can see, services and scopes contain process and are placed in slices that do not contain processes of their own. The only exception is PID 1 that is located in the special systemd.slice. Also note that **-.slice** is not shown as it is implicitly identified with the root of the entire tree.

Service and slice units may be configured via unit files on disk (see Section 2.3.2, “Modifying Unit Files”), or alternatively be created dynamically at runtime via API calls to PID 1 (see Section 1.4, “Online Documentation” for API reference). Scope units may only be created at runtime via API calls to PID 1, but not from unit files on disk. Units that are created dynamically at runtime via API calls are called *transient units*. Transient units exist only during runtime and are released automatically as soon as they finished or they got deactivated or the system is rebooted.

1.3. Resource Controllers in Linux Kernel

A resource controller, also called cgroup subsystem, represents a single resource, such as CPU time or memory. The Linux kernel provides a range of resource controllers, that are mounted automatically by **systemd**. Find the list of currently mounted resource controllers in **/proc/cgroups**, or use the **lssubsys** monitoring tool. In Red Hat Enterprise Linux 7, **systemd** mounts the following controllers by default:

Available Controllers in Red Hat Enterprise Linux 7

- **blkio** — sets limits on input/output access to and from block devices such as physical drives (disk, solid state, USB, etc.).

- **cpu** — uses the scheduler to provide cgroup tasks access to the CPU. It is mounted together with **cpuacct** on the same mount.
- **cpuacct** — automatic reports on CPU resources used by tasks in a cgroup. It is mounted together with **cpu** on the same mount.
- **cpuset** — assigns individual CPUs (on a multicore system) and memory nodes to tasks in a cgroup.
- **devices** — allows or denies access to devices by tasks in a cgroup.
- **freezer** — suspends or resumes tasks in a cgroup.
- **memory** — sets limits on memory use by tasks in a cgroup, and generates automatic reports on memory resources used by those tasks.
- **net_cls** — tags network packets with a class identifier (classid) that allows the Linux traffic controller (**tc**) to identify packets originating from a particular cgroup task.
- **perf_event** — allows to monitor cgroups with the **perf** tool.
- **hugetlb** — allows to use virtual memory pages of large sizes, and to enforce resource limits on these pages.

The Linux Kernel exposes a wide range of tunable parameters for resource controllers that can be configured with **systemd**. See the kernel documentation (list of references in Controller-Specific Kernel Documentation) for detailed description of these parameters.

1.4. Additional Resources

To find more information about resource control under systemd, the unit hierarchy, as well as the kernel resource controllers, refer to the materials listed below:

Installed Documentation

Cgroup-Related Systemd Documentation

The following man pages contain general information unified cgroup hierarchy under **systemd**:

- **systemd.resource-control**(5) — describes the configuration options for resource control shared by system units.
- **systemd.unit**(5) — describes common options of all unit configuration files.
- **systemd.slice**(5) — provides general information about *.slice* units.
- **systemd.scope**(5) — provides general information about *.scope* units.
- **systemd.service**(5) — provides general information about *.service* units.

Controller-Specific Kernel Documentation

The *kernel-doc* package provides a detailed documentation of all resource controllers. This package is included in the optional *subscription* channel, to install it, type as **root**:

```
yum install kernel-doc
```

After the installation, the following files will appear under the **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/cgroups/** directory:

- **blkio** subsystem — **blkio-controller.txt**
- **cpuacct** subsystem — **cpuacct.txt**
- **cpuset** subsystem — **cpusets.txt**
- **devices** subsystem — **devices.txt**
- **freezer** subsystem — **freezer-subsystem.txt**
- **memory** subsystem — **memory.txt**
- **net_cls** subsystem — **net_cls.txt**

Additionally, refer to the following files on further information about the **cpu** subsystem:

- Real-Time scheduling — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-rt-group.txt**
- CFS scheduling — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-bwc.txt**

Online Documentation

- Red Hat Enterprise Linux 7 System Administrators Guide — The *System Administrator's Guide* documents relevant information regarding the deployment, configuration and administration of Red Hat Enterprise Linux 7. It is oriented towards system administrators with a basic understanding of the system.
- The D-Bus API of systemd — The reference for D-Bus API commands for accessing **systemd**.

Chapter 2. Using Control Groups

The following sections provide an overview of tasks related to creation and management of control groups. This guide focuses on utilities provided by **systemd** that are preferred as a way of cgroup management and will be supported in the future. Previous versions of Red Hat Enterprise Linux used the *libcgroup* package for the same purpose. This package is still available to assure backward compatibility (see Warning), but it will not be supported in the future versions of Red Hat Enterprise Linux.

2.1. Creating Control Groups

From the **systemd**'s perspective, a cgroup is bound to a system unit configurable with a unit file and manageable with systemd's command-line utilities. Depending on the type of application, your resource management settings can be *transient* or *persistent*.

You can create a **transient cgroup** for a service by starting this service with the **systemd-run** command. This way, you can set limits on resources consumed by the service during its runtime. Applications can create transient cgroups dynamically by using API calls to **systemd**. See Section 2.5, "Online Documentation" for API reference. Transient unit is removed automatically as soon as the service is stopped.

To assign a **persistent cgroup** to a service, edit its unit configuration file. This configuration is preserved after the system reboot, so it can be used to manage services that are started automatically. Note that scope units can not be created this way.

2.1.1. Creating Transient Cgroups with systemd-run

The **systemd-run** command is used to create and start a transient *service* or *scope* unit and run a custom command in this unit. Commands executed in service units are started asynchronously in the background, where they are invoked from the **systemd** process. Commands run in scope units are started directly from the **systemd-run** process and thus inherit the execution environment of the caller. Execution in this case is synchronous.

To run a command in a specified cgroup, type as **root**:

```
systemd-run --unit=name --scope --slice=slice_name command
```

- The *name* stands for the name you want this unit to be known under. If **--unit** is not specified, a unit name will be generated automatically. It is recommended to choose a descriptive name, since it will represent the unit in the **systemctl** output. This name must be unique during runtime of the unit.
- Use the optional **--scope** parameter to create a transient *scope* unit instead of *service* unit that is created by default.
- With the **--slice** option, you can make your newly created *service* or *scope* unit a member of a specified slice. Replace *slice_name* with the name of an existing slice (as shown in the output of **systemctl -t slice**), or create a new slice by passing a unique name. By default, services and scopes are created as members of the **system.slice**.
- Replace *command* with the command you wish to execute in the service unit. Place this command at the very end of the **systemd-run** syntax, so that the parameters of this command are not confused for parameters of **systemd-run**.

Besides the above options, there are several other parameters you can specify for **systemd-run**. With **--description**, you can add a description to the unit, **--remain-after-exit** lets you to collect runtime

information after terminating the service's process. The **--machine** option executes the command in a confined container. See the **systemd-run** man page to learn more.

Example 2.1. Starting a New Service with systemd-run

Use the following command to run the **top** utility in a service unit in a new slice called **test**. Type as **root**:

```
~]# systemd-run --unit=toptest --slice=test top -b
```

The following message is displayed to confirm that you started the service successfully:

```
Running as unit toptest.service
```

Now, you can use the name *toptest.service* to monitor or to modify the cgroup with **systemctl** commands.

2.1.2. Creating Persistent Cgroups

You can configure a unit to be started automatically on system boot by executing the **systemctl enable** command (see the chapter called *Managing Services with systemd* in Red Hat Enterprise Linux 7 System Administrators Guide). Running this command automatically creates a unit file in the **/usr/lib/systemd/system/** directory. To make persistent changes to the cgroup, add or modify configuration parameters in its unit file. For more information, see Section 2.3.2, “Modifying Unit Files”.

2.2. Removing Control Groups

Transient cgroups are released automatically as soon as the processes they contain finish. By passing the **--remain-after-exit** option to **systemd-run** you can keep the unit running after its processes finished to collect runtime information. To stop the unit gracefully, type:

```
systemctl stop name.service
```

Replace *name* with the name of the service you wish to stop. To terminate one or more of the unit's processes, type as **root**:

```
systemctl kill name.service --kill-who=PID,... --signal=signal
```

Replace *name* with a name of the unit, for example *httpd.service*. Use **--kill-who** to select which processes from the cgroup you wish to terminate. To kill more processes at the same time, pass a comma-separated list of PIDs. Replace *signal* with the type of POSIX signal you wish to send to specified processes. Default is *SIGTERM*. For more information, see the **systemd.kill** manual page.

Persistent cgroups are released when the unit is disabled and its configuration file is deleted by running:

```
systemctl disable name.service
```

where *name* stands for the name of the service to be disabled.

2.3. Modifying Control Groups

Each unit supervised by **systemd** has a unit configuration file in the **/usr/lib/systemd/system/** directory. To change parameters of a service unit, modify this configuration file. You can do that manually or from the command-line interface by using the **systemctl set-property** command.

2.3.1. Setting Parameters from the Command-Line Interface

The **systemctl set-property** command allows you to change resource control settings during the application runtime. To do so, use the following syntax as **root**:

```
systemctl set-property name parameter=value
```

Replace *name* with the name of the systemd unit you wish to modify, *parameter* with a name of the parameter to be changed, and *value* with a new value you want to assign to this parameter.

Not all unit parameters may be changed at runtime, but most of those related to resource control may, see Section 2.3.2, “Modifying Unit Files” for a list. Note that **systemctl set-property** allows you to change multiple properties at once, which is preferable over setting them individually.

The changes are applied instantly, and written into unit file so that they are preserved after reboot. You can change this behavior by passing the **--runtime** option that makes your settings transient:

```
systemctl set-property --runtime name property=value
```

Example 2.2. Using systemctl set-property

To limit the CPU and memory usage of *httpd.service* from the command line, type:

```
~]# systemctl set-property httpd.service CPUShares=600 MemoryLimit=500M
```

To make this a temporary change, add the **--runtime** option:

```
~]# systemctl set-property --runtime httpd.service CPUShares=600
MemoryLimit=500M
```

2.3.2. Modifying Unit Files

Systemd service unit files, by default stored in the **/usr/lib/systemd/system/** directory, provide a number of high-level configuration parameters useful for resource management. These parameters communicate with Linux cgroup controllers, that must be enabled in the kernel. With these parameters, you can manage CPU, memory consumption, block IO, as well as some more fine-grained unit properties.

Managing CPU

The *cpu* controller is enabled by default in kernel, and consequently every system service receives the same amount of CPU, regardless of how many processes it contains. This default behavior can be changed with the **DefaultControllers** parameter in the **/etc/systemd/system.conf** configuration file. To manage the CPU allocation, use the following directive in the **[Service]** section of the unit configuration file:

CPUShares=*value*

Replace *value* with a number of CPU shares. The default value is 1024, by increasing this number you assign more CPU to the unit. This parameter implies that **CPUAccounting** is turned on in the unit file.

The **CPUShares** parameter controls the *cpu.shares* control group parameter. See the description of the **cpu** controller in Controller-Specific Kernel Documentation to see other CPU-related control parameters.

Example 2.3. Limiting CPU Consumption of a Unit

Imagine you wish to assign the Apache service 1500 CPU shares instead of the default 1024. To do so, modify the **CPUShares** setting in the **/usr/lib/systemd/system/httpd.service** unit file:

```
[Service]
CPUShares=1500
```

To apply your changes, reload systemd's configuration and restart Apache so that the modified service file is taken into account:

```
~]# systemctl daemon-reload
```

```
~]# systemctl restart httpd.service
```

Managing Memory

To enforce limits on memory the unit's memory consumption, use the following directives in the **[Service]** section of the unit configuration file:

MemoryLimit=*value*

Replace *value* with a limit on maximum memory usage of the processes executed in the cgroup. Use *K*, *M*, *G*, *T* suffixes to identify Kilobyte, Megabyte, Gigabyte, or Terabyte as a unit of measurement. Also, the **MemoryAccounting** parameter must be enabled for the same unit.

The **MemoryLimit** parameter controls the *memory.limit_in_bytes* control group parameter. For more information, see the description of the **memory** controller in Controller-Specific Kernel Documentation.

Example 2.4. Limiting Memory Consumption of a Unit

Imagine you wish to assign a 1GB memory limit to the Apache service. To do so, modify the **MemoryLimit** setting in the **/usr/lib/systemd/system/httpd.service** unit file:

```
[Service]
MemoryLimit=1G
```

To apply your changes, you must reload systemd's configuration and restart Apache so that the modified service file is taken into account:

```
~]# systemctl daemon-reload
```

```
~]# systemctl restart httpd.service
```

Managing Block IO

To manage the Block IO, use the following directives in the **[Service]** section of the unit configuration file. Directives listed below assume that the **BlockIOAccounting** parameter is enabled:

BlockIOWeight=*value*

Replace *value* with a new overall block IO weight for the executed processes. You can choose a single value between 10 and 1000, the default setting is 1000.

BlockIODeviceWeight=*device_name value*

Replace *value* with a block IO weight for a device specified with *device_name*. Replace *device_name* either with a name or with a path to a device. As with **BlockIOWeight**, you can set a single weight value between 10 and 1000.

BlockIOReadBandwidth=*device_name value*

This directive allows you to limit a specific bandwidth for a unit. Replace *device_name* with the name of a device or with a path to a block device node, *value* stands for a bandwidth rate. Use *K*, *M*, *G*, *T* suffixes to specify units of measurement, value with no suffix is interpreted as bytes per second.

BlockIOWriteBandwidth=*device_name value*

Limits the write bandwidth for a specified device. Accepts the same arguments as **BlockIOReadBandwidth**.

Each of the aforementioned directives control a corresponding cgroup parameter. See the description of the **blkio** controller in Controller-Specific Kernel Documentation.

Note

Currently, the **blkio** resource controller does not support buffered write operations. It is primarily targeted at direct I/O, so the services that use buffered write will ignore the limits set with **BlockIOWriteBandwidth**. On the other hand, buffered read operations are supported, and **BlockIOReadBandwidth** limits will be applied correctly both on direct and buffered read.

Example 2.5. Limiting Block IO of a Unit

To lower the block IO weight for the Apache service accessing the **/home/jdoe/** directory add the following text into the **/usr/lib/systemd/system/httpd.service** unit file:

```
[Service]
BlockIODeviceWeight=/home/jdoe 750
```

To set the maximum bandwidth for Apache reading from the **/var/log/** directory to 5MB per second, use the following syntax:

```
[Service]
BlockIOReadBandwith=/var/log 5M
```

To apply your changes, you must reload systemd's configuration and restart Apache so that the modified service file is taken into account:

```
~]# systemctl daemon-reload
```

```
~]# systemctl restart httpd.service
```

Managing Other System Resources

There are several other directives you can use in the unit file to facilitate resource management:

DeviceAllow=*device_name options*

With this option, you can control access to specific device nodes. Here, *device_name* stands for a path to a device node or a device group name as specified in **/proc/devices**. Replace **options** with a combination of **r**, **w**, and **m** to allow the unit to read, write, or create device nodes.

DevicePolicy=*value*

Here, *value* is one of: *strict* (only allows the types of access explicitly specified with **DeviceAllow**), *closed* (allows access to standard pseudo devices including /dev/null, /dev/zero, /dev/full, /dev/random, and /dev/urandom) or *auto* (allows access to all devices if no explicit **DeviceAllow** is present, which is default behavior)

Slice=*slice_name*

Replace *slice_name* with the name of the slice to place the unit in. The default is *system.slice*. Scope units can not be arranged this way, since they are tied to their parent slices.

ControlGroupAttribute=*attribute value*

With this option, you can set various control group parameters exposed by Linux cgroup controllers. Replace *attribute* with a low-level cgroup parameter you wish to modify and *value* with a new value for this parameter. Refer to Controller-Specific Kernel Documentation for more information on cgroup controllers.

Example 2.6. Changing Low-level Cgroup Attributes

Imagine that you wish change the *memory.swappiness* setting that sets the tendency of the kernel to swap out process memory used by tasks in the cgroup. For more information on this setting, see the description of the memory controller in Controller-Specific Kernel Documentation. To set *memory.swappiness* to 70 for the Apache service, add the following text to **/usr/lib/systemd/system/httpd.service**:

```
[Service]
ControlGroupAttribute=memory.swappiness 70
```

To apply your changes, you must reload systemd's configuration and restart Apache so that the modified service file is taken into account:

```
~]# systemctl daemon-reload
```

```
~]# systemctl restart httpd.service
```

2.4. Obtaining Information About Control Groups

You can use the **systemctl** command to list system units and to view their status. **Systemd** also provides **systemd-cgls** to view the hierarchy of control groups and **systemd-cgtop** to monitor their resource consumption in real time.

2.4.1. Listing Units

Use the following command to list all active units on the system:

```
systemctl list-units
```

This command lists all active units on the system. The **list-units** option is executed by default, which means that you will receive the same output when you omit this option and execute just:

```
systemctl
UNIT                     LOAD   ACTIVE SUB     DESCRIPTION
abrt-ccpp.service        loaded active exited  Install ABRT coredump hook
abrt-oops.service        loaded active running ABRT kernel log watcher
abrt-vmcore.service      loaded active exited  Harvest vmcores for ABRT
abrt-xorg.service        loaded active running ABRT Xorg log watcher
...
```

The output contains four rows:

- *UNIT* — the name of unit that also reflects the unit's position in the cgroup tree. As mentioned in Section 1.2, “Systemd Unit Types”, three unit types are relevant for resource control: *slice*, *scope* and *service*. For a complete list of **systemd**'s unit types, see the chapter called *Managing Services with systemd* in Red Hat Enterprise Linux 7 System Administrators Guide.
- *LOAD* — reflects whether the unit configuration file was properly loaded. If the unit file failed to load, the field will contain *error* instead of *loaded*. Other unit load states are: *stub*, *merged*, and *masked*.
- *ACTIVE* — the high-level unit activation state, which is a generalization of SUB.

- *SUB* — the low-level unit activation state. The range of possible values depends on the unit type.
- *DESCRIPTION* — the description of the unit's content and functionality.

By default, **systemctl** lists only active units (in terms of high-level activations state — the ACTIVE field). Use the **--all** option to see inactive units too. To limit the amount of information in the output list, use the **--type** (**-t**) parameter that requires a comma-separated list of unit types such as *service* and *slice*, or unit load states such as *loaded* and *masked*.

Example 2.7. Using systemctl list-units

To view a list of all slices used on the system, type:

```
~]$ systemctl -t slice
```

To list all active masked services, type:

```
~]$ systemctl -t service,masked
```

To list all unit files installed on your system and their status, type:

```
systemctl list-unit-files
```

2.4.2. Viewing the Control Group Hierarchy

The aforementioned listing commands do not let you go beyond the unit level to see the actual processes running in cgroups. Also, the output of **systemctl** does not show the hierarchy of units. You can achieve both by using the **systemd-cgls** command that groups the running process according to cgroups. To display the whole cgroup hierarchy on your system, type:

```
systemd-cgls
```

When **systemd-cgls** is issued without parameters, it returns the entire cgroup hierarchy. The highest level of the cgroup tree is formed by slices and can look as follows:

```
├─system
│ ├─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 20
│ ...
│
├─user
│ ├─user-1000
│ │ └─ ...
│ ├─user-2000
│ │ └─ ...
│ ...
│
└─machine
  ├─machine-1000
  │ └─ ...
  ...
```

Note that machine slice is present only if you are running a virtual machine or container. For more info on the cgroup tree see Section 1.2, “Systemd Unit Types”.

To reduce the output of **systemd-cgls**, and to view a specified part of the hierarchy, type:

```
systemd-cgls name
```

Replace *name* with a name of the resource controller you wish to inspect.

As an alternative, use the **systemctl status** to display detailed information about a system unit. A cgroup subtree is a part of the output of this command.

```
systemctl status name
```

To learn more about **systemctl status**, see the chapter called *Managing Services with systemd* in Red Hat Enterprise Linux 7 System Administrators Guide.

Example 2.8. Viewing the Control Group Hierarchy

To see a cgroup tree of the **memory** resource controller, execute:

```
~]$ systemd-cgls memory
memory:
├─    1 /usr/lib/systemd/systemd --switched-root --system --deserialize 23
├─  475 /usr/lib/systemd/systemd-journald
...
```

The output of the above command lists the services that interact with the selected controller. A different approach is to view a part of the cgroup tree for a certain service, slice, or scope unit:

```
~]# systemctl status httpd.service
httpd.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled)
   Active: active (running) since Sun 2014-03-23 08:01:14 MDT; 33min ago
  Process: 3385 ExecReload=/usr/sbin/httpd $OPTIONS -k graceful (code=exited,
status=0/SUCCESS)
 Main PID: 1205 (httpd)
   Status: "Total requests: 0; Current requests/sec: 0; Current traffic:   0 B/sec"
   CGroup: /system.slice/httpd.service
           ├─1205 /usr/sbin/httpd -DFOREGROUND
           ├─3387 /usr/sbin/httpd -DFOREGROUND
           ├─3388 /usr/sbin/httpd -DFOREGROUND
           ├─3389 /usr/sbin/httpd -DFOREGROUND
           ├─3390 /usr/sbin/httpd -DFOREGROUND
           └─3391 /usr/sbin/httpd -DFOREGROUND

...
```

Beides the aforementioned tools, there is also the **machinectl** command dedicated to Linux containers. See Section 7.8, “Monitoring a Container” for more information on monitoring Linux containers.

2.4.3. Viewing Resource Controllers

The aforementioned **systemctl** commands let you monitor the higher-level unit hierarchy, but do not show which resource controllers in Linux kernel are actually used by which processes. This information is stored in dedicated proces file, to view it, type as **root**:

```
cat proc/PID/cgroup
```

Where *PID* stands for the ID of the process you wish to examine. By default, the list is the same for all units started by **systemd**, since it automatically mounts all default controllers. See the following example:

```
~]# cat proc/27/cgroup
10:hugetlb:/
9:perf_event:/
8:blkio:/
7:net_cls:/
6:freezer:/
5:devices:/
4:memory:/
3:cpuacct,cpu:/
2:cpuset:/
1:name=systemd:/
```

By checking this file, you can determine if the process has been placed in the desired cgroups as defined by the systemd unit file specifications.

2.4.4. Monitoring Resource Consumption

The **systemd-cgls** command provides a static snapshot of the cgroup hierarchy. To see a dynamic account of currently running cgroups ordered by their resource usage (CPU, Memory, and IO), use:

```
systemd-cgtop
```

The behavior, provided statistics, and control options of **systemd-cgtop** are akin of those of the **top** utility. See **systemd-cgtop** man page for more information.

2.5. Additional Resources

For more information on how to use systemd and related tools to manage system resources on Red Hat Enterprise Linux, refer to the sources listed below:

Installed Documentation

Man Pages of Cgroup-Related Systemd Tools

- **systemd-run**(1) — The manual page lists all command-line options of the **systemd-run** utiltiy.
- **systemctl**(1) — The manual page of the **systemctl** utility that lists available options and commands.
- **systemd-cgls**(1) — This manual page lists all command-line options of the **systemd-cgls** utiltiy.
- **systemd-cgtop**(1) — The manual page contains the list of all command-line options of the **systemd-cgtop** utiltiy.
- **machinectl**(1) — This manual page lists all command-line options of the **machinectl** utility.
- **systemd.kill**(5) — This manual page provides an overview of kill configuration options for system units.

Controller-Specific Kernel Documentation

The *kernel-doc* package provides a detailed documentation of all resource controllers. This package is included in the optional *subscription* channel, to install it, type as **root**:

```
yum install kernel-doc
```

After the installation, the following files will appear under the **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/cgroups/** directory:

- **blkio** subsystem — **blkio-controller.txt**
- **cpuacct** subsystem — **cpuacct.txt**
- **cpuset** subsystem — **cpusets.txt**
- **devices** subsystem — **devices.txt**
- **freezer** subsystem — **freezer-subsystem.txt**
- **memory** subsystem — **memory.txt**
- **net_cls** subsystem — **net_cls.txt**

Additionally, refer to the following files on further information about the **cpu** subsystem:

- Real-Time scheduling — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-rt-group.txt**
- CFS scheduling — **/usr/share/doc/kernel-doc-<*kernel_version*>/Documentation/scheduler/sched-bwc.txt**

Online Documentation

- Red Hat Enterprise Linux 7 System Administrators Guide — The *System Administrator's Guide* documents relevant information regarding the deployment, configuration and administration of Red Hat Enterprise Linux 7. It is oriented towards system administrators with a basic understanding of the system.
- The D-Bus API of systemd — The reference for D-Bus API commands for accessing systemd.

Chapter 3. Using libcgroup Tools

As mentioned above, the *libcgroup* package, which was the main tool for cgroup management in previous versions of Red Hat Enterprise Linux, is now deprecated. To avoid conflicts, do not use *libcgroup* tools for default resource controllers (listed in Available Controllers in Red Hat Enterprise Linux 7) that are now an exclusive domain of **systemd**. This leaves a limited space for applying *libcgroup* tools, use it only when you need to manage controllers not currently supported by **systemd**, such as *net_prio*.

The following sections describe how to use *libcgroup* tools in relevant scenarios without conflicting with the default system of hierarchi.

Note

In order to use *libcgroup* tools, first ensure the *libcgroup* and *libcgroup-tools* packages are installed on your system. To install them, run as **root**:

```
~]# yum install libcgroup
~]# yum install libcgroup-tools
```

Note

The **net_prio** controller is not compiled in the kernel like the rest of the controllers, rather it is a module that has to be loaded before attempting to mount it. To load this module, type as **root**:

```
modprobe netprio_cgroup
```

3.1. Mounting a Hierarchy

To use a kernel resource controller that is not mounted automatically, you have to create a hierarchy that will contain this controller. You can add or detach a hierarchy by editing the **mount** section of the **/etc/cgconfig.conf**. This method makes the controller attachment persistent, which means your settings will be preserved after system reboot. As an alternative, you can use the **mount** command to create a transient mount established only for the current session.

Using the cgconfig Service

The **cgconfig** service installed with the *libcgroup-tools* package provides a way to mount hierarchies for additional resource controllers. By default, this service is not started automatically. When you start **cgconfig**, it applies the settings from the **/etc/cgconfig.conf** configuration file. The configuration is therefore recreated from session to session and becomes persistent. Note that if you stop **cgconfig**, it unmounts all the hierarchies that it mounted.

The default **/etc/cgconfig.conf** file installed with the *libcgroup* package does not contain any configuration settings, just an information that **systemd** mounts the main resource controllers automatically.

You can create three types of entry in **/etc/cgconfig.conf** — *mount*, *group*, and *template*. Mount entries are used to create and mount hierarchies as virtual file systems, and attach controllers to those hierarchies. In Red Hat Enterprise Linux 7, default hierarchies are mounted automatically to the **/sys/fs/cgroup/** directory, **cgconfig** is therefore used solely to attach non-default controllers. Mount entries are defined using the following syntax:

```
mount {
    controller_name = /sys/fs/cgroup/controller_name;
    …
}
```

Replace *controller_name* with a name of the kernel resource controller you wish to mount to the hierarchy. See Example 3.1, “Creating a mount entry” for an example.

Example 3.1. Creating a mount entry

To attach the **net_prio** controller to the default cgroup tree, add the following text to the **/etc/cgconfig.conf** configuration file:

```
mount {
    net_prio = /sys/fs/cgroup/net_prio;
}
```

Then restart the **cgconfig** service to apply the setting:

```
systemctl restart cgconfig.service
```

Group entries in **/etc/cgconfig.conf** can be used to set the parameters of resource controllers. See Section 3.5, “Setting Cgroup Parameters” for more information about group entries.

Template entries in **/etc/cgconfig.conf** can be used to create a group definition applied to all processes.

Using the mount Command

You can also use the **mount** command to temporarily mount a hierarchy. To do so, first create a *mount point* in the **/sys/fs/cgroup/** directory where **systemd** mounts the main resource controllers. Type as **root**:

```
mkdir /sys/fs/cgroup/name
```

Replace *name* with a name of the new mount destination, usually the name of the controller is used. Next, use the **mount** command to mount the hierarchy and simultaneously attach one or more subsystems. Type as **root**:

```
mount -t cgroup -o controller_name none /sys/fs/cgroup/controller_name
```

Where *controller_name* with a name of the controller to specify both the device to be mounted as well as the destination folder. The **-t cgroup** parameter specifies the type of mount.

Example 3.2. Using the mount command to attach controllers

To mount a hierarchy for the **net_prio** controller with use of the **mount** command, first create the mount point:

```
~]# mkdir /sys/fs/cgroup/net_prio
```

Then mount **net_prio** to the destination you created in the previous step:

```
~]# mount -t cgroup -o net_prio none /sys/fs/cgroup/net_prio
```

You can verify if you attached the hierarchy correctly by listing all available hierarchies along with their current mount points with the **lssubsys** command (see Section 3.8, “Listing Controllers”):

```
~]# lssubsys -am
cpuset /sys/fs/cgroup/cpuset
cpu,cpuacct /sys/fs/cgroup/cpu,cpuacct
memory /sys/fs/cgroup/memory
devices /sys/fs/cgroup/devices
freezer /sys/fs/cgroup/freezer
net_cls /sys/fs/cgroup/net_cls
blkio /sys/fs/cgroup/blkio
perf_event /sys/fs/cgroup/perf_event
hugetlb /sys/fs/cgroup/hugetlb
net_prio /sys/fs/cgroup/net_prio
```

3.2. Unmounting a Hierarchy

If you mounted a hierarchy by editing the **/etc/cgconfig.conf** configuration file, you can unmount it simply by removing the configuration directive from the *mount* section of this configuration file. Then restart the service to apply the new configuration.

Similarly, you can unmount a hierarchy by executing the following command as **root**:

```
~]# umount /sys/fs/cgroups/controller_name
```

Replace *controller_name* with the name of the hierarchy that contains the resource controller you wish to detach.

Warning

Make sure that you use **umount** to remove only hierarchies that you mounted yourself. Detaching a hierarchy that contains a default controller (listed in Available Controllers in Red Hat Enterprise Linux 7) will most probably lead to complications requiring system reboot.

3.3. Creating Control Groups

Use the **cgcreate** command to create transient cgroups in hierarchies you created yourself. The syntax for **cgcreate** is:

```
cgcreate -t uid:gid -a uid:gid -g controllers:path
```

where:

- **-t** (optional) — specifies a user (by user ID, uid) and a group (by group ID, gid) to own the **tasks** pseudo-file for this cgroup. This user can add tasks to the cgroup.

  

  Removing processes

  Note that the only way to remove a process from a cgroup is to move it to a different cgroup. To move a process, the user must have write access to the *destination* cgroup; write access to the source cgroup is not important.

- **-a** (optional) — specifies a user (by user ID, uid) and a group (by group ID, gid) to own all pseudo-files other than **tasks** for this cgroup. This user can modify the access to system resources for tasks in this cgroup.

- **-g** — specifies the hierarchy in which the cgroup should be created, as a comma-separated list of the *controllers* associated with those hierarchies. The list of controllers is followed by a colon and the *path* to the child group relative to the hierarchy. Do not include the hierarchy mount point in the path.

Because all cgroups in the same hierarchy have the same controllers, the child group has the same controllers as its parent.

As an alternative, you can create a child of the cgroup directly, use the **mkdir** command:

```
~]# mkdir /sys/fs/cgroup/controller/name/child_name
```

For example:

```
~]# mkdir /sys/fs/cgroup/net_prio/lab1/group1
```

3.4. Removing Control Groups

Remove cgroups with the **cgdelete** command that has a syntax similar to that of **cgcreate**. Run the following command as **root**:

```
cgdelete controllers:path
```

where:

- *controllers* is a comma-separated list of controllers.
- *path* is the path to the cgroup relative to the root of the hierarchy.

For example:

```
~]# cgdelete net_prio:/test-subgroup
```

cgdelete can also recursively remove all subgroups when the **-r** option is specified.

Note that when you delete a cgroup, all its processes move to its parent group.

3.5. Setting Cgroup Parameters

Modify the parameters of the control groups by editing the **/etc/cgconfig.conf**, or by using the **cgset** command. Changes made to **/etc/cgconfig.conf** are preserved after reboot, while **cgset** changes the cgroup parameters only for the current session.

Modifying /etc/cgconfig.conf

You can set the controller parameters in the *Groups* section of **/etc/cgconfig.conf**. Group entries are defined using the following syntax:

```
group name {
[permissions]
    controller {
        param_name = param_value;
        …
    }
    …
}
```

Replace *name* with the name of your cgroup, *controller* stands for the name of the controller you wish to modify. This should be a controller you mounted yourself, not any of the default controllers mounted automatically by **systemd**. Replace *param_name* and *param_value* with the controller parameter you wish to change and its new value. Note that the **permissions** section is optional. To define permissions for a group entry, use the following syntax:

```
perm {
    task {
        uid = task_user;
        gid = task_group;
    }
    admin {
       uid = admin_name;
       gid = admin_group;
    }
}
```

Restart the cgconfig service for the changes to take effect

You must restart the **cgconfig** service for the changes in the **/etc/cgconfig.conf** to take effect. Restarting this service rebuilds hierarchies specified in the configuration file but does not affect all mounted hierarchies. You can restart a service by executing the **systemctl restart** command, however, it is recommended to first stop the **cgconfig** service:

```
~]# systemctl stop cgconfig
```

then edit the configuration file. After saving your changes, you can start **cgconfig** again with the following command:

```
~]# systemctl start cgconfig
```

Using the cgset Command

Set controller parameters by running the **cgset** command from a user account with permission to modify the relevant cgroup. Use this only for controllers you mounted manually.

The syntax for **cgset** is:

```
cgset -r parameter=value path_to_cgroup
```

where:

- *parameter* is the parameter to be set, which corresponds to the file in the directory of the given cgroup
- *value* is the value for the parameter
- *path_to_cgroup* is the path to the cgroup *relative to the root of the hierarchy*.

The values that you can set with **cgset** might depend on values set higher in a particular hierarchy. For example, if **group1** is limited to use only CPU 0 on a system, you cannot set **group1/subgroup1** to use CPUs 0 and 1, or to use only CPU 1.

You can also use **cgset** to copy the parameters of one cgroup into another, existing cgroup. The syntax to copy parameters with **cgset** is:

```
cgset --copy-from path_to_source_cgroup path_to_target_cgroup
```

where:

- *path_to_source_cgroup* is the path to the cgroup whose parameters are to be copied, relative to the root group of the hierarchy
- *path_to_target_cgroup* is the path to the destination cgroup, relative to the root group of the hierarchy

3.6. Moving a Process to a Control Group

Move a process into a cgroup by running the **cgclassify** command:

```
cgclassify -g controllers:path_to_cgroup pidlist
```

where:

- *controllers* is a comma-separated list of resource controllers, or ***** to launch the process in the hierarchies associated with all available subsystems. Note that if there are multiple cgroups of the same name, the **-g** option moves the processes in each of those groups.
- *path_to_cgroup* is the path to the cgroup within the hierarchy
- *pidlist* is a space-separated list of *process identifier* (PIDs)

You can also add the **--sticky** option before the *pid* to keep any child processes in the same cgroup. If you do not set this option and the **cgred** service is running, child processes will be allocated to cgroups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the cgroup in which you started it.

You can also use the **cgred** service (which starts the **cgrulesengd** daemon) that moves tasks into cgroups according to parameters set in the **/etc/cgrules.conf** file. Use **cgred** only to manage manually attached controllers. Entries in the **/etc/cgrules.conf** file can take one of the two forms:

- *user subsystems control_group*
- *user*:*command subsystems control_group*

For example:

```
maria   net_prio  /usergroup/staff
```

This entry specifies that any processes that belong to the user named **maria** access the **devices** subsystem according to the parameters specified in the **/usergroup/staff** cgroup. To associate particular commands with particular cgroups, add the *command* parameter, as follows:

```
maria:ftp  devices  /usergroup/staff/ftp
```

The entry now specifies that when the user named **maria** uses the **ftp** command, the process is automatically moved to the **/usergroup/staff/ftp** cgroup in the hierarchy that contains the **devices** subsystem. Note, however, that the daemon moves the process to the cgroup only after the appropriate condition is fulfilled. Therefore, the **ftp** process might run for a short time in the wrong group. Furthermore, if the process quickly spawns children while in the wrong group, these children might not be moved.

Entries in the **/etc/cgrules.conf** file can include the following extra notation:

- **@** — when prefixed to *user*, indicates a group instead of an individual user. For example, **@admins** are all users in the **admins** group.
- ***** — represents "all". For example, ***** in the **subsystem** field represents all subsystems.
- **%** — represents an item the same as the item in the line above. For example:

```
@adminstaff net_prio   /admingroup
@labstaff %   %
```

3.7. Starting a Process in a Control Group

Launch processes in a manually created cgroup by running the **cgexec** command. The syntax for **cgexec** is:

```
cgexec -g controllers:path_to_cgroup command arguments
```

where:

- *controllers* is a comma-separated list of controllers, or ***** to launch the process in the hierarchies associated with all available subsystems. Note that, as with the **cgset** command described in Section 3.5, “Setting Cgroup Parameters”, if cgroups of the same name exist, the **-g** option creates processes in each of those groups.
- *path_to_cgroup* is the path to the cgroup relative to the hierarchy.
- *command* is the command to run in the cgroup.
- *arguments* are any arguments for the command.

You can also add the **--sticky** option before the *command* to keep any child processes in the same cgroup. If you do not set this option and the **cgred** daemon is running, child processes will be allocated to cgroups based on the settings found in **/etc/cgrules.conf**. The process itself, however, will remain in the cgroup in which you started it.

3.8. Obtaining Information About Control Groups

The *libcgroup-tools* package contains several utilities for obtaining information about controllers, control groups, and their parameters.

Listing Controllers

To find the controllers that are available in your kernel and how are they mounted together to hierarchies, run:

```
cat /proc/cgroups
```

Or, to find the mount points of particular subsystems, run:

```
lssubsys -m controllers
```

where *controllers* is a list of the subsystems in which you are interested. Note that the **lssubsys -m** command returns only the top-level mount point per each hierarchy.

Finding Control Groups

To list the cgroups on a system, run as **root**:

```
lscgroup
```

You can restrict the output to a specific hierarchy by specifying a controller and path in the format ***controller*:*path***. For example:

```
~]$ lscgroup cpuset:adminusers
```

lists only subgroups of the **adminusers** cgroup in the hierarchy to which the **cpuset** subsystem is attached.

Displaying Parameters of Control Groups

To display the parameters of specific cgroups, run:

```
~]$ cgget -r parameter list_of_cgroups
```

where *parameter* is a pseudo-file that contains values for a subsystem, and *list_of_cgroups* is a list of cgroups separated with spaces.

If you do not know the names of the parameters themselves, use a command like:

```
~]$ cgget -g cpuset /
```

3.9. Additional Resources

The definitive documentation for cgroup commands are the manual pages provided with the *libcgroup* package. The section numbers are specified in the list of man pages below.

Installed Documentation

The libcgroup-related Man Pages

- **cgclassify**(1) — the **cgclassify** command is used to move running tasks to one or more cgroups.
- **cgclear**(1) — the **cgclear** command is used to delete all cgroups in a hierarchy.

  cgconfig.conf(5) — cgroups are defined in the **cgconfig.conf** file.
- **cgconfigparser**(8) — the **cgconfigparser** command parses the **cgconfig.conf** file and mounts hierarchies.
- **cgcreate**(1) — the **cgcreate** command creates new cgroups in hierarchies.
- **cgdelete**(1) — the **cgdelete** command removes specified cgroups.
- **cgexec**(1) — the **cgexec** command runs tasks in specified cgroups.
- **cgget**(1) — the **cgget** command displays cgroup parameters.
- **cgsnapshot**(1) — the **cgsnapshot** command generates a configuration file from existing subsystems.
- **cgred.conf**(5) — **cgred.conf** is the configuration file for the **cgred** service.
- **cgrules.conf**(5) — **cgrules.conf** contains the rules used for determining when tasks belong to certain cgroups.
- **cgrulesengd**(8) — the **cgrulesengd** service distributes tasks to cgroups.
- **cgset**(1) — the **cgset** command sets parameters for a cgroup.
- **lscgroup**(1) — the **lscgroup** command lists the cgroups in a hierarchy.
- **lssubsys**(1) — the **lssubsys** command lists the hierarchies containing the specified subsystems.

Chapter 4. Control Group Application Examples

This chapter provides application examples that take advantage of the cgroup functionality.

4.1. Prioritizing Database I/O

Running each instance of a database server inside its own dedicated virtual guest allows you to allocate resources per database based on their priority. Consider the following example: a system is running two database servers inside two KVM guests. One of the databases is a high priority database and the other one a low priority database. When both database servers are run simultaneously, the I/O throughput is decreased to accommodate requests from both databases equally; Figure 4.1, “I/O throughput without resource allocation” indicates this scenario — once the low priority database is started (around time 45), I/O throughput is the same for both database servers.

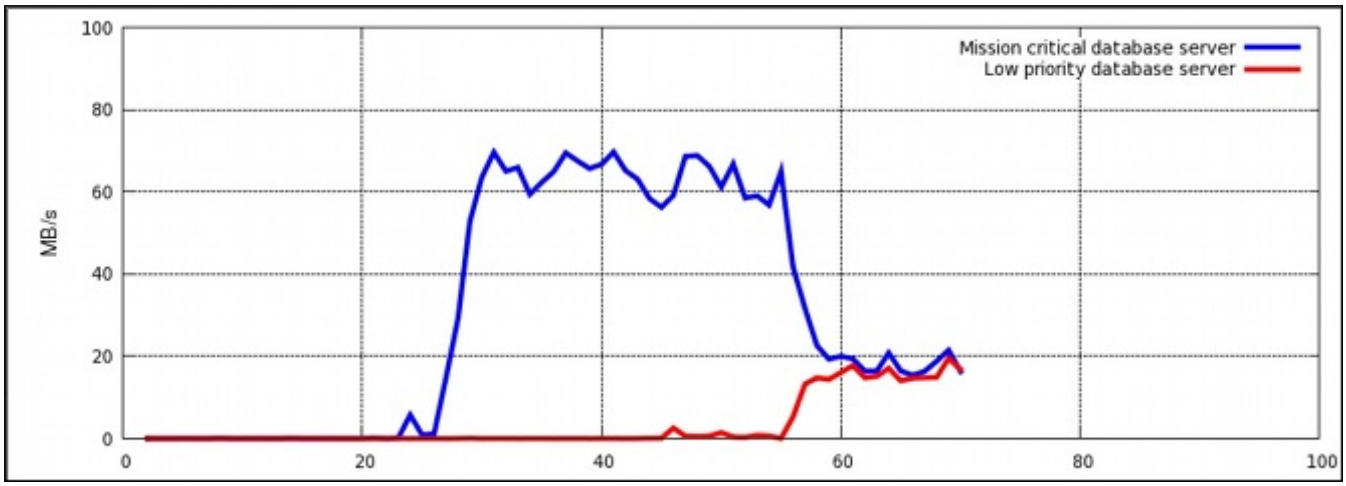

Figure 4.1. I/O throughput without resource allocation

To prioritize the high priority database server, it can be assigned to a cgroup with a high number of reserved I/O operations, whereas the low priority database server can be assigned to a cgroup with a low number of reserved I/O operations. To achieve this, follow the steps in Procedure 4.1, “I/O Throughput Prioritization”, all of which are performed on the host system.

Procedure 4.1. I/O Throughput Prioritization

1. Make sure resource accounting is on for both services:

```
~]# systemctl set-property db1.service BlockIOAccounting=true
~]# systemctl set-property db2.service BlockIOAccounting=true
```

2. Set a ratio of 10:1 for the high and low priority services. Processes running in those service units will use only the resources made available to them

```
~]# systemctl set-property db1.service BlockIOWeight=1000
~]# systemctl set-property db2.service BlockIOWeight=100
```

Figure 4.2, “I/O throughput with resource allocation” illustrates the outcome of limiting the low priority database and prioritizing the high priority database. As soon as the database servers are moved to their appropriate cgroups (around time 75), I/O throughput is divided among both servers with the ratio of 10:1.

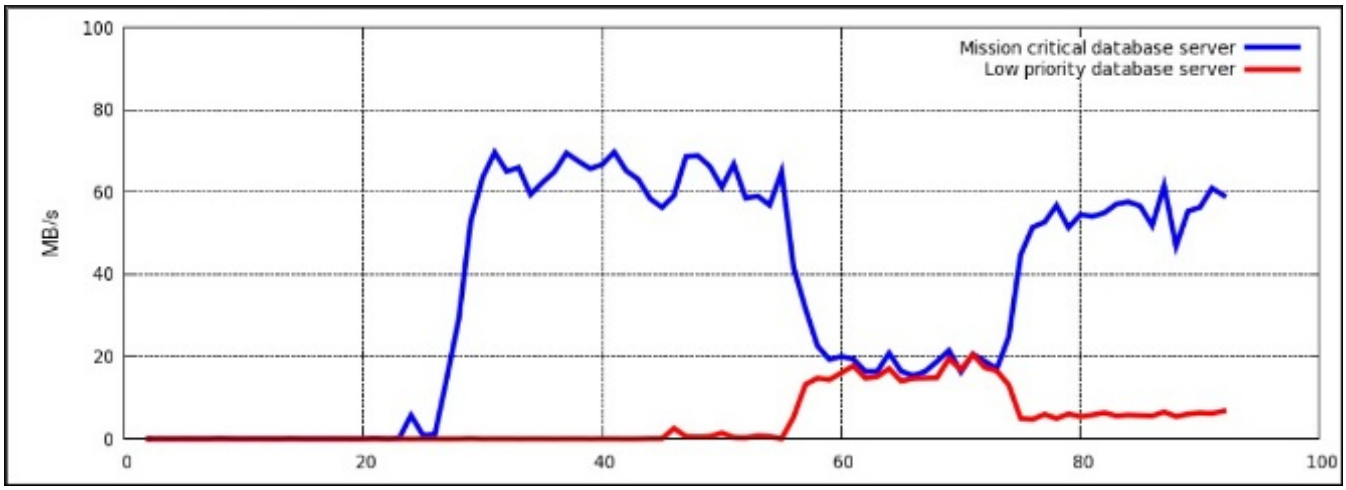

Figure 4.2. I/O throughput with resource allocation

Alternatively, block device I/O throttling can be used for the low priority database to limit its number of read and write operation. For more information refer to the description of the **blkio** controller in Controller-Specific Kernel Documentation.

4.2. Prioritizing Network Traffic

When running multiple network-related services on a single server system, it is important to define network priorities between these services. Defining these priorities ensures that packets originating from certain services have a higher priority than packets originating from other services. For example, such priorities are useful when a server system simultaneously functions as an NFS and Samba server. The NFS traffic must be of high priority as users expect high throughput. The Samba traffic can be deprioritized to allow better performance of the NFS server.

The **net_prio** controller can be used to set network priorities for processes in cgroups. These priorities are then translated into Type Of Service (TOS) bits and embedded into every packet. Follow the steps in Procedure 4.2, “Setting Network Priorities for File Sharing Services” to configure prioritization of two file sharing services (NFS and Samba).

Procedure 4.2. Setting Network Priorities for File Sharing Services

1. The **net_prio** controller is not compiled in the kernel, it is a module that must be loaded manually. To do so, type:

```
~]# modprobe netprio_cgroup
```

2. Attach the **net_prio** subsystem to the **/cgroup/net_prio** cgroup:

```
~]# mkdir sys/fs/cgroup/net_prio
~]# mount -t cgroup -o net_prio none sys/fs/cgroup/net_prio
```

3. Create two cgroups, one for each service:

```
~]# mkdir sys/fs/cgroup/net_prio/nfs_high
~]# mkdir sys/fs/cgroup/net_prio/samba_low
```

4. To automatically move the **nfs** services to the **nfs_high** cgroup, add the following line to the **/etc/sysconfig/nfs** file:

```
CGROUP_DAEMON="net_prio:nfs_high"
```

This configuration ensures that **nfs** service processes are moved to the **nfs_high** cgroup when the **nfs** service is started or restarted.

5. The **smbd** daemon does not have a configuration file in the **/etc/sysconfig** directory. To automatically move the **smbd** daemon to the **samba_low** cgroup, add the following line to the **/etc/cgrules.conf** file:

```
*:smbd                  net_prio                samba_low
```

Note that this rule moves every **smbd** daemon, not only **/usr/sbin/smbd**, into the **samba_low** cgroup.

You can define rules for the **nmbd** and **winbindd** daemons to be moved to the **samba_low** cgroup in a similar way.

6. Start the **cgred** service to load the configuration from the previous step:

```
~]# systemctl start cgred
Starting CGroup Rules Engine Daemon:                       [  OK  ]
```

7. For the purposes of this example, let us assume both services use the **eth1** network interface. Define network priorities for each cgroup, where **1** denotes low priority and **10** denotes high priority:

```
~]# echo "eth1 1" > /sys/fs/cgroup/net_prio/samba_low/net_prio.ifpriomap
~]# echo "eth1 10" > /sys/fs/cgroup/net_prio/nfs_high/net_prio.ifpriomap
```

8. Start the **nfs** and **smb** services and check whether their processes have been moved into the correct cgroups:

```
~]# systemctl start smb
Starting SMB services:                                     [  OK  ]
~]# cat /sys/fs/cgroup/net_prio/samba_low/tasks
16122
16124
~]# systemctl start nfs
Starting NFS services:                                     [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Starting NFS mountd:                                       [  OK  ]
Stopping RPC idmapd:                                       [  OK  ]
Starting RPC idmapd:                                       [  OK  ]
Starting NFS daemon:                                       [  OK  ]
~]# cat sys/fs/cgroup/net_prio/nfs_high/tasks
16321
16325
16376
```

Network traffic originating from NFS now has higher priority than traffic originating from Samba.

Similar to Procedure 4.2, “Setting Network Priorities for File Sharing Services”, the **net_prio** subsystem can be used to set network priorities for client applications, for example, Firefox.

Part II. Linux Containers

This part provides an overview of general Linux Containers concepts and their current capabilities implemented in Red Hat Enterprise Linux 7.

Chapter 5. Introduction to Linux Containers

Linux Containers have emerged as a key open source application packaging and delivery technology, combining lightweight application isolation with the flexibility of image-based deployment methods.

Red Hat Enterprise Linux 7 implements Linux Containers using core technologies such as Control Groups (Cgroups) for Resource Management, Namespaces for Process Isolation, SELinux for Security, enabling secure multi-tenancy and reducing the potential for security exploits

5.1. Linux Containers Architecture

Several components are needed for Linux Containers to function correctly, most of is them provided by the Linux kernel. Kernel *namespaces* ensure process isolation and *cgroups* are employed to control the system resources. *SELinux* is used to assure separation between the host and the container and also between the individual containers. *Management interface* forms a higher layer that interacts with the aforementioned kernel components and provides tools for construction and management of containers.

The following scheme illustrates the architecture of Linux Containers in Red Hat Enterprise Linux 7:

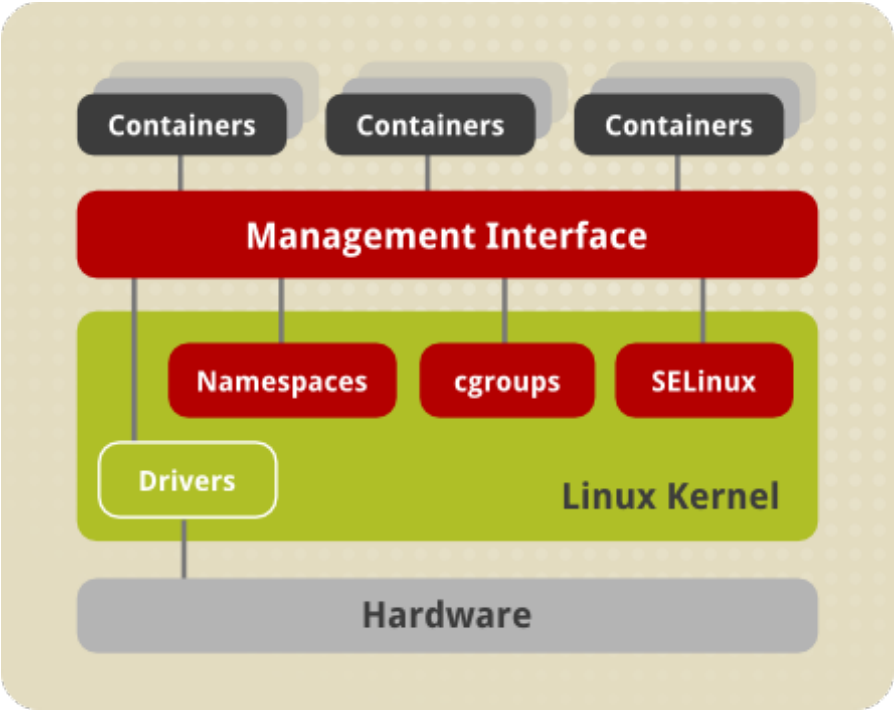

Figure 5.1. Linux Containers Architecture

Namespaces

The kernel provides process isolation by creating separate **namespaces** for containers. Namespaces allow you to create an abstraction of a particular global system resource and make it appear as a separated instance to processes within a namespace. Consequently, several containers can use the same resource simultaneously without creating a conflict. There are several types of namespaces:

- **Mount namespaces** isolate the set of file system mount points seen by a group of processes so that processes in different mount namespaces can have different views of the file system hierarchy. With mount namespaces, the **`mount()`** and **`umount()`** system calls cease to operate on a global set of mount points (visible to all processes) and instead perform operations that affect just the mount

namespace associated with the container process. For example, each container can have its own **/tmp** or **/var** directory or even have an entirely different userspace.

- **UTS namespaces** isolate two system identifiers – nodename and domainname, returned by the **uname()** system call. This allows each container to have its own hostname and NIS domain name, which is useful for initialization and configuration scripts based on these names. You can test this isolation by executing the **hostname** command on the host system and a container – the results will differ.
- **IPC namespaces** isolate certain interprocess communication (IPC) resources, such as System V IPC objects and POSIX message queues. This means that two containers can create shared memory segments and semaphores with the same name, but are not able to interact with other containers memory segments or shared memory.
- **PID namespaces** allow processes in different containers to have the same PID, so each container can have its own init (PID1) process that manages various system initialization tasks as well as containers life cycle. Also, each container has its unique **/proc** directory. Note that from within the container you can monitor only processes running inside this container. In other words, the container is only aware of its native processes and can not "see" the processes running in different parts of the system. On the other hand, the host operating system is aware of processes running inside of the container, but assigns them different PID numbers. For example, run the **ps -eZ | grep systemd$** command on host to see all instances of systemd including those running inside of containers.
- **Network namespaces** provide isolation of network controllers, system resources associated with networking, firewall and routing tables. This allows container to use separate virtual network stack, loopback device and process space. You can add virtual or real devices to the container, assign them their own IP Addresses and even full iptables rules. You can view the different network settings by executing the **ip addr** command on the host and inside the container.

Note

There is another type of namespace called **user namespace**. User namespaces are similar to PID namespaces, they allow you to specify a range of host UIDs dedicated to the container. Consequently, a process can have full root privileges for operations inside the container, and at the same time be unprivileged for operations outside the container. For compatibility reasons, user namespaces are turned off in the current version of Red Hat Enterprise Linux 7, but will be enabled in the near future.

Control Groups (cgroups)

The kernel uses **cgroups** to group processes for the purpose of system resource management. Cgroups let you allocate CPU time, system memory, network bandwidth, or combinations of these among user-defined groups of tasks. In Red Hat Enterprise Linux 7, cgroups are managed with systemd slice, scope, and service units. For more information on cgroups, see Part I, “Resource Management With Control Groups”.

SELinux

SELinux provides secure separation of containers by applying SELinux policy and labels. It integrates with virtual devices by using the **sVirt** technology. For more information see Section 5.2, “Secure Containers with SELinux”

Management Interface

Red Hat Enterprise Linux 7 provides the **Docker** application as a main management tool for Linux Containers. **Docker** builds on the aforementioned kernel capabilities, adding several enhancement features, such as portability or version control. To learn more, see Chapter 6, *Using Docker*.

As an alternative, you can use the **virsh** utility from the **Libvirt** toolkit, which provides a basic interface for launching and management of Linux Containers. See Chapter 7, *Using virsh* for more information on using **virsh**.

5.2. Secure Containers with SELinux

From the security point of view, there is a need to isolate the host system from a container and to isolate containers from each other. The kernel features used by containers, namely cgoups and namespaces, by itself provide a certain level of security. Cgroups ensure that a single container cannot exhaust a large amount of system resources, thus preventing some denial-of-service attacks. By virtue of namespaces, the **/dev** directory created within a container is private to each container, and therefore unaffected by the host changes. However, this can not prevent a hostile process from breaking out of the container since the entire system is not namespaced or containerized. Another level of separation, provided by SELinux, is therefore needed.

Security-Enhanced Linux (**SELinux**) is an implementation of a mandatory access control (MAC) mechanism, multi-level security (MLS), and multi-category security (MCS) in the Linux kernel. The **sVirt** project builds upon SELinux and integrates with **Libvirt** to provide a MAC framework for virtual machines and containers. This architecture provides a secure separation for containers as it prevents root processes within the container from interfering with other processes running outside this container. The containers created with **Docker** or **virsh** are automatically assigned with an SELinux context specified in the SELinux policy.

By default, containers created with **libvirt** tools are assigned with the **virtd_lxc_t** label (execute **ps -eZ | grep virtd_lxc_t**). You can apply **sVirt** by setting static or dynamic labeling for processes inside the container.

Note

You might notice that SELinux appears to be disabled inside the container even though it is running in enforcing mode on host system – you can verify this by executing the **getenforce** command on host and in the container. This is to prevent utilities that have SELinux awareness, such as **setenforce**, to perform any SELinux activity inside the container.

Note that if SELinux is disabled or running in permissive mode on the host machine, containers are not separated securely enough. For more information about SELinux, refer to Red Hat Enterprise Linux 7 SELinux Users and Administrators Guide, sVirt is described in Red Hat Enterprise Linux 7 Virtualization Security Guide.

5.3. Container Use Cases

There are two general scenarios for using Linux containers in Red Hat Enterprise Linux 7. You can work with *host containers* as a tool for application sandboxing, or you can utilize the extended features of *image-based containers*.

5.3.1. Host Containers

The Red Hat Enterprise Linux 7 host operating system with Linux container feature allows you to carve out containers as lightweight application sandboxes. All host containers launched are identical – each runs the same user space as the host system, so all applications running in in host containers are based on Red Hat Enterprise Linux 7 user space and run time. The advantage of this approach is that security erratas and other updates can be applied to these containers easily with the **`yum update`** command. You can create and manage host based containers with the **virsh** application, see Chapter 7, *Using virsh*.

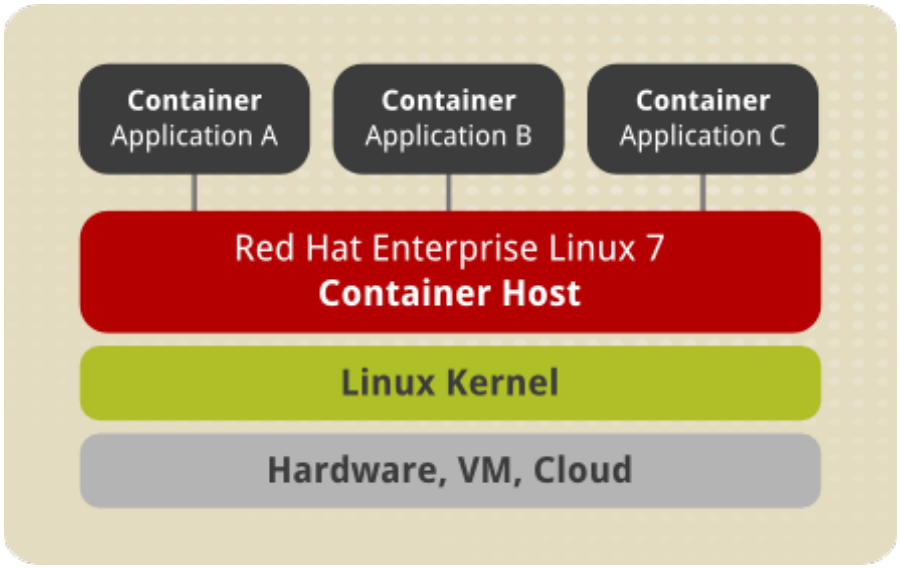

Figure 5.2. Host Containers

5.3.2. Image-based Containers

With image-based containers, an application is packaged with individual runtime stack, which makes it independent from the host operating system. This way, you can run several instances of an application, each developed for a different platform. This is possible because the container run time and the application run time are deployed in the form of an image. For example, *Runtime A* in Figure 5.3, “Image-based Containers” can stand for Red Hat Enterprise Linux 6.5, *Runtime B* could refer to version 6.6 and so on.

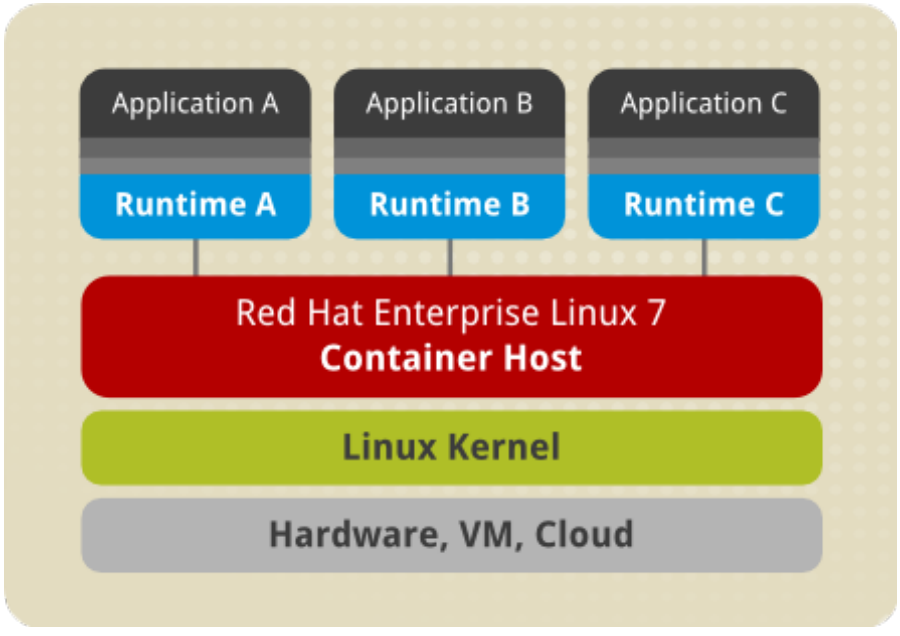

Figure 5.3. Image-based Containers

Image-based containers allow you to host multiple instances and versions of an application, with minimal overhead and increased flexibility. Such containers are not tied to the host-specific configuration, which makes them portable. These features are enabled by the *Docker* format for application packaging described in Section 5.4, “Application Packaging with Docker”

Docker format relies on the *device mapper thin provisioning* technology that is an advanced variation of LVM snapshots to implement copy-on-write in Red Hat Enterprise Linux 7.

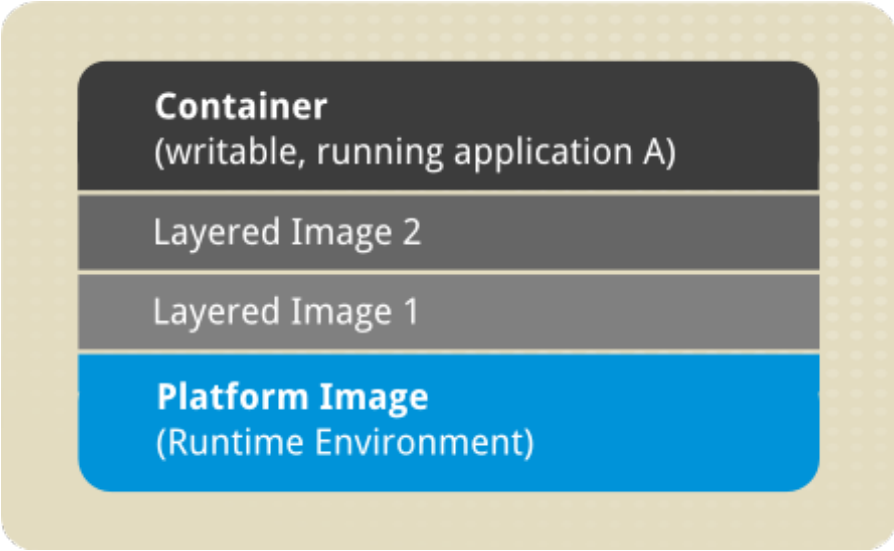

Figure 5.4. Image Layering Using Docker Format

The above image shows the fundamental components of any image-based container:

- **Container** – (in the narrow sense of the word) an active component in which you can run an application. Each container is based on an *image* that holds necessary configuration data. When you launch a container from an image, a writable layer is added on top of this image. Every time you commit a container (using the **`docker commit`** command), a new image layer is added to store your changes. To learn more about containers, see Section 6.2, “Managing Containers”.
- **Image** – a static snapshot of the containers' configuration. Image is a read-only layer that is never modified, all changes are made in top-most writable layer, and can be saved only by creating a new image. Each image depends on one more parent images. See Section 6.1, “Working with Docker Images”, for description of image-related command-line tools.
- **Platform Image** – an image that has no parent. Platform images define the runtime environment, packages and utilities necessary for containerized application to run. Your work with Docker usually starts by pulling the platform image. The platform image is read-only, so any changes are reflected in the copied images stacked on top of it. .

The images piled on top of the platform image create an *application layer* that contains software dependencies for the containerized application. For example, the layered images in Figure 5.4, “Image Layering Using Docker Format” could have added software dependencies required by the containerized application.

The whole container can be very large or it could be made really small depending on how many packages are included in the application layer. Further layering of the images is possible, such as software from 3rd party ISVs. From a user point of view there is still one container, but from an operational point of view there can be many layers behind it.

5.4. Application Packaging with Docker

Docker is a technology behind image-based containers. It is a tool and a format designed for shipping applications as self-contained units. Docker builds on the core capabilities of Linux containers, such as cgroups, namespaces and SeLinux (see Section 5.1, “Linux Containers Architecture”). It also depends to certain extent on the underlying operating system, namely on device mapper thin provisioning and on systemd for resource management.

Docker brings in an API for container management, an image format and a possibility to use a remote registry for sharing containers. The following advantages come from this scheme:

- **Portability across machines** – you can bundle an application and all its dependencies into a single container that is independent from the host version of Linux kernel, platform distribution, or deployment model. This container can be transfered to another machine that runs Docker, and executed there without compatibility issues.
- **Version control and component reuse** – you can track successive versions of a container, inspect differences, or roll-back to previous versions. As mentioned above, containers reuse components from the precessing layers, which makes them noticeably lightweight (see Figure 5.4, “Image Layering Using Docker Format”).
- **Sharing** – you can use a remote repository to share your container with others. Red Hat provides a registry ( ) for this purpose, it is also possible to configure your own private repository .

Docker also provides a configuration template called **Dockerfile** that contains a list of instructions for building images. Dockerfiles let you automate and share procedures you would otherwise have to repeat manually. To learn more about Dockerfile instructions, see Section 6.4, “Using Dockerfiles”.

The aforementioned features of Docker address the problem of multiple software stacks in a variety of hardware environments, which highly improves application packaging and delivery. A rich tool ecosystem already embraced Docker, see

5.5. Linux Containers Compared to KVM Virtualization

The main difference between the KVM virtualization and Linux Containers is that virtual machines require a separate kernel instance to run on, while containers can be deployed from the host operating system. This significantly reduces the complexity of container creation and maintenance. Also, the reduced overhead lets you create a large number of containers with faster startup and shutdown speeds.

Both Linux Containers and KVM virtualization have certain advantages and drawbacks that influence the use cases in which these technologies are typically applied:

KVM virtualization:

- KVM virtualization lets you boot full operating systems of different kinds, even non-Linux systems. However, a complex setup is sometimes needed. Virtual machines are resource-intensive so you can run only a limited number of them on your host machine.
- Running separate kernel instances generally means better separation and security. If one of the kernels terminates unexpectedly, it does not disable the whole system. On the other hand, this isolation makes it harder for virtual machines to communicate with the rest of the system, and therefore several interpretation mechanisms must be used.

- Guest virtual machine is isolated from host changes, which lets you run different versions of the same application on the host and virtual machine. KVM also provides many useful features such as live migration. For more information on these capabilities, see Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide.

Linux Containers:

- Linux Containers are designed to support isolation of one or more applications. You can create or destroy containers very easily and they are convenient to maintain.
- System-wide changes are visible in each container. For example, if you upgrade an application on the host machine, this change will apply to all sandboxes that run instances of this application.
- Since containers are lightweight, a large number of them can run simultaneously on a host machine. The theoretical maximum is 6000 containers and 12,000 bind mounts of root file system directories. Also, containers are faster to create and have low startup times.

5.6. Additional Resources

To learn more about general principles and architecture of Linux Containers, refer to the following resources.

Installed Documentation

- **`docker`**(1) — The manual page of the **`docker`** command.
- **`virsh`**(1) — The manual page of the **`virsh`** command.

Online Documentation

- Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide — This guide instructs how to configure a Red Hat Enterprise Linux 7 host physical machine and how to install and configure guest virtual machines with different distributions, using the KVM hypervisor. Also included PCI device configuration, SR-IOV, networking, storage, device and guest virtual machine management, as well as troubleshooting, compatibility and restrictions.
- Red Hat Enterprise Linux 7 SELinux Users and Administrators Guide — The *SELinux Users and Administrators Guide* for Red Hat Enterprise Linux 7 documents the basics and principles upon which SELinux functions, as well as practical tasks to set up and configure various services.
- Get Started with RHEL 7 Containers in Docker — This quick start guide describes the essential Docker-related tasks along with a number of examples.
- Documentation on the Docker Project Site — The official documentation of the Docker project.

Chapter 6. Using Docker

The following sections provide a description of **Docker** command-line and configuration options that allow you to package and share your applications.

First ensure that the *docker* package is installed on your system, if that is not the case, execute the following command as **root**:

```
~]# yum install docker
```

After you successfully installed the application, use the usual **systemctl** commands to start **docker** and to make it run automatically at boot time:

```
~]# systemctl start docker.service
~]# systemctl enable docker.service
```

For detailed instructions on how to set up **Docker** on Red Hat Enterprise Linux 7, see the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal.

To list all available command-line options provided by **Docker**, type:

```
docker
```

which is an equivalent of executing the **docker help** command without arguments. To find the version number of the *docker* package that is currently installed on your system, type:

```
docker version
```

6.1. Working with Docker Images

As mentioned in Section 5.3.2, “Image-based Containers”, a Docker *image* is a read-only snapshot of a container. Images are static layers that store information on processes started in the container. By committing a container, you add a new layer on top of the parent image. An image with no parent is called a *platform image* that also constitutes the environment for containers running on top of this image.

There are two approaches to image building, either you can create them interactively from the command-line interface or you can define a template called *Dockerfile* to hold image parameters necessary for build. Interactive approach is useful for troubleshooting and prototyping, while Dockerfiles are designed for sharing stable image configurations. The following sections focus on Docker CLI tools, to learn more about Dockerfiles, see Section 6.4, “Using Dockerfiles”.

6.1.1. Searching for Images

To search for existing images, type as **root**:

```
docker search name
```

Replace *name* with the name of the image you are looking for. You can type just a part of the whole name. This command searches the *Docker.io Index*, which is currently the main public registry for sharing repositories of images. Some repositories in docker index are marked as trusted, which means they are officially checked. By using the **--trusted** (or **-t**) option, you can limit your search to trusted builds only.

Users of docker index can reward repositories with stars, to search for images that achieves certain amount of stars, use the **--stars** (or **-s**) option.

Image name is preceded by the name of its containing repository to provide a unique identification of the image in docker index:

```
repository_name/image_name
```

Example 6.1. Searching for Public Images

Use the following command to search for fedora-related trusted images with at least one star ranking:

```
~]#  docker search --stars=1 --trusted fedora
NAME                DESCRIPTION                                     STARS
OFFICIAL  TRUSTED
goldmann/wildfly    A WildFly application server running on a ...   3
[OK]
tutum/fedora-20     Fedora 20 image with SSH access. For the r...   1
[OK]
...
```

Apart from the public repository in *Docker.io Index*, there is also a storage space enabled on the Red Hat Customer portal to allow you to view images in a Red Hat registry. See the *Working with Docker registries* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal for more information.

6.1.2. Pulling Images

To download a selected image from the remote registry to your local machine, type as **root**:

```
docker pull repository_name/image_name
```

Where *repository_name/image_name* identifies the image you wish to pull from the docker index. The *Getting images from outside Docker registries* section in the Get Started with RHEL 7 Containers in Docker article shows how to pull Red Hat Enterprise Linux platform image.

6.1.3. Listing Images

To list all locally installed images, execute the following command as **root**:

```
docker images
```

Use the following command-line options to modify the output of **docker images**:

- **-a** — returns verbose output including all intermediate images. By default, this option is set to **false**, so the intermediate images are filtered out from the output list.
- **-t** — creates output in a form of a hierarchical tree depicting relationships between images.
- **-v** — returns a graph of the commit tree in the *Graphviz* format. You can use the **dot** utility to convert the graph to PNG or another image format.

6.1.4. Modifying Images

Modify the image configuration by committing the changes you made in the container running on top of the image, as shown in Section 6.2.3, “Committing a Container”. This adds a new layered image that can serve as a base for other containers. Run **docker images** to confirm that you created the image successfully.

6.1.5. Sharing Images

To copy the image or the repository to a remote location, execute the following command as **root**:

```
docker push name
```

Here, *name* stands for the name or the id of the image or the repository. Default Docker commands push to the default *index.docker.io* registry. Note that it is not possible to push Red Hat Enterprise Linux images to the public registry. A specialized registry for that purpose is provided by Red Hat, refer to the *Creating a private Docker registry* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal.

For more information on sharing images and repositories, refer to Section 6.7, “Publishing Images”

6.1.6. Removing Images

To remove one or more images from your system, use to following syntax as **root**:

```
docker rmi image_name
```

Replace *image_name* with the name or ID of the image you want to remove. This command removes image from the host node, but it will not affect images in the remote registry until you push the changes with **docker push**. By default, you can not remove an image base of a running container, use the **-f** option to suppress this. Use the **docker images** command to make sure you removed the image successfully.

6.2. Managing Containers

Containers provide an environment for running applications that is securely isolated from the rest of the system. For more information on the principles of Linux Containers, see Part II, “Linux Containers”.

6.2.1. Starting a Container

To create a new container, run the following command as **root**:

```
docker run --name=container_name image_name command
```

Replace *command* with a command you want to execute in the container, replace *container_name* with a name for the new container, and *image_name* to specify an image on top of which will the container run.

There are several other command-line options you can use with **docker run** to set the parameters of the new container (see Table 6.1, “Command-line options for **docker run**”) or to modify the default parameters configured in the Dockerfile (refer to Section 6.2.1, “Overriding Image Defaults”).

Table 6.1. Command-line options for docker run

Usage	Option	Description

Usage	Option	Description
run mode	**-d**, **--detach**	Run the container in the background. Disabled by default. In this mode, the container is no longer listening to input from command line, so all communication is limited to network connections (see Section 6.5, “Networking”) and shared volumes (see Section 6.6.2, “Using Data Volume Containers”). Use the **docker attach** command to change the container to forward mode.
	-a, **--attach**	Attach the container to **stdin**, **stdout** or **stderr**.
	-i, **--interactive**	Keep **stdin** open even if not the container is not attached. Disabled by default.
	-t, **--tty**	Allocates a pseudo terminal to the container. Disabled by default.
container identification and privileges	**--name**	Assign the specified name or UUID to the container. If no name is set, **Docker** generates a random one.
	--privileged	Grant extended privileges to the container, including access to devices. Disabled by default.
networking	**-n**, **--networking**	Enable networking for the container (see Section 6.5, “Networking”). This option is enabled by default.
	--dns	Set custom DNS servers for the container.
resource constraints	**-c**, **--cpu-shares**	Set the cpu priority for the container.
	-m, **--memory**	Set the maximum amount of memory the container can use, specify units with **b**, **k**, **m** or **g**.
persistence	**-rm**	Automatically remove the container's file system when it exits. Disabled by default.

Overriding Image Defaults

Dockerfiles let you specify default configuration for images. This is convenient, since it helps you to share the configuration and automate image creation. In certain cases, you may need to override these default settings when starting a container. See Table 6.2, “Dockerfile Instructions” for a complete list of directives.

By executing a command with **docker run** you override the COMMAND directive set in Dockerfile. Not all parameters can be altered at runtime, **docker run** accepts the following options to modify preset Dockerfile instructions:

- **--entrypoint** overrides the ENTRYPOINT directive
- **--expose** overrides the EXPOSE directive
- **-e**, **--env** overrides to the ENV directive
- **-v**, **--volume** overrides the VOLUME directive. See Section 6.6.2, “Using Data Volume Containers” for more information on using volumes
- **-u**, **--user** overrides the USER directive
- **-w**, **--workdir** overrides the WORKDIR directive

Refer to Section 6.8, “Installed Documentation” for more information on **docker run**. The *Running Docker containers* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal provides examples of **docker run** in action.

6.2.2. Connecting to a Running Container

To execute commands inside of a running container, you need to connect to it through a command-line interface. The **docker attach** command is not suitable for this, since it only lets you observe the standard output of the application currently running in the container. Instead, use the **nsenter** command to enter the container namespace. This command requires the ID of the container as it appears on the host system, not the ID that appears in the output of **docker ps**. You can find this ID for example by executing:

```
docker inspect -f {{.State.Pid}} container_id
```

Here, *container_id* stands for the container name or its ID from the **docker ps** output. See Section 6.3.2, "Viewing Container Parameters" for more information on the **docker inspect** command. With the correct host ID, execute the following command as **root**:

```
nsenter -m -u -n -i -p -t container_host_id /bin/sh
```

Once you are connected to the container, you can execute commands affecting the container's isolated environment. To break out from the dedicated shell, type **exit**. See the *Investigating within a running Docker container* procedure in the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal. You can learn more about the **nsenter** command from its man page.

6.2.3. Committing a Container

To create a new image from changes made in the running container, type as **root**:

```
docker commit container_name [repository_name:tag]
```

Where *container_name* represents the name or ID of the container. Use **docker ps** to find the container ID. Optionally, set *repository_name* and *tag* to identify the image. The following command-line options are provided to help you to organize your commits:

- **-a**, **--author** – the name and contact information of the creator
- **-m**, **--message** – a commit message

Refer to the *Creating an image from a container* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal for example of **docker commit** usage.

6.2.4. Stopping a Container

To stop the running container gracefully, execute the following command as **root**:

```
docker stop container_name
```

Where *container_name* stands for the name or ID of the container you wish to stop. By executing the above command, you send the **SIGTERM** signal to the container following by **SIGKILL** after certain time period. This period can be set with the **-t** command-line option. To send **SIGKILL** immediately, for example to stop a container that is not responding, type:

```
docker kill container_name
```

You can also use the **--signal** option with the above command to define a custom signal to be sent to

the container instead of the default **SIGKILL**.

6.2.5. Restarting a Container

To start a previously stopped container, execute as **root**:

```
docker start container_name
```

Where *container_name* stands for the name or ID of the container you wish to start. To restart a running container, use:

```
docker restart container_name
```

6.2.6. Removing a Container

To remove a container, type the following command as **root**:

```
docker rm container_name
```

Where *container_name* stands for the name or ID of the container you wish to remove.

6.2.7. Automatically Starting a Container

To start a container automatically at boot time, first configure it as a systemd service by creating the unit configuration file in the **/etc/systemd/system/** directory. For example, the contents of the **/etc/systemd/system/redis-container.service** can look as follows:

```
[Unit]
Description=Redis container
Author=Me
After=docker.service

[Service]
Restart=always
ExecStart=/usr/bin/docker start -a redis_server
ExecStop=/usr/bin/docker stop -t 2 redis_server

[Install]
WantedBy=local.target
```

After creating the unit file, use the **systemctl enable** command to start the container automatically.

To learn more about configuring services with **systemd**, refer to chapter called *Managing Services with systemd* in Red Hat Enterprise Linux 7 System Administrators Guide.

6.3. Monitoring Images and Containers

Docker offers several ways to learn about the parameters of existing containers and images. Also, you can track changes made to the image and monitor the resource consumption of currently running containers.

6.3.1. Listing Containers

To view a list of currently running containers, type as **root**:

```
docker ps
```

Use the following command-line options to modify the output of **docker ps**:

- **-a** — returns a list of all containers. By default, this option is set to **false**, so only the currently running containers are included in the output
- **-l** — displays only the latest created container, including containers that are not running
- **-n** — displays only **n** latest created containers, including containers that are not running
- **-s** — includes container sizes into the output table
- **--since** — limits the output on containers that were created after the specified one. You can use name or ID to identify the oldest included container

Example 6.2. Example_Output of docker ps

Display the two latest containers by executing:

```
~]# docker ps -n 2
CONTAINER ID  IMAGE          COMMAND  CREATED          STATUS          PORTS
NAMES
4c01db0b339c  rhel7:12.04    bash     17 seconds ago  Up 16 seconds  6379/tcp  app
d7886598dbe2  fedora:latest top      3 days ago      Ghost                     top
```

Note: the *Ghost* entry in the STATUS row that marks an unresponsive container.

6.3.2. Viewing Container Parameters

To view an overall information on how **Docker** is configured on your system, execute as **root**:

```
docker info
```

Information displayed includes the number of containers and images, pool name, paths to data and metadata files, total data and metadata space used, and so on (see Example 6.3, “Example Output of **docker info**”).

Example 6.3. Example Output of `docker info`

The output of **`docker info`** can look as follows:

```
~]# docker info
Containers: 5
Images: 2
Storage Driver: devicemapper
 Pool Name: docker-253:0-35960696-pool
 Data file: /var/lib/docker/devicemapper/devicemapper/data
 Metadata file: /var/lib/docker/devicemapper/devicemapper/metadata
 Data Space Used: 994.4 Mb
 Data Space Total: 102400.0 Mb
 Metadata Space Used: 1.3 Mb
 Metadata Space Total: 2048.0 Mb
Execution Driver: native-0.2
Kernel Version: 3.10.0-122.el7.x86_64
```

To display a detailed information about an image or a container, type as **`root`**:

```
docker inspect container_name
```

By default, the above command returns the output in the *JSON* format. With the **`-f`** option, you are able to limit the output to a specific item of interest:

```
docker inspect -f {{.section.subsection}} container_name
```

The *section* and *subsection* values let you extract a specific value from the JSON file as shown in Example 6.4, “Example Output of **`docker inspect`**”.

Example 6.4. Example Output of `docker inspect`

Display an in-depth description of a container called **`test_container`**, type:

```
~]# docker inspect test_container
[2013/07/30 01:52:26 GET /v1.3/containers/efef/json
{
  "ID": "efefdc74a1d5900d7d7a74740e5261c09f5f42b6dae58ded6a1fde1cde7f4ac5",
  "Created": "2013-07-30T00:54:12.417119736Z",
  "Path": "ping",
  "Args": [
      "www.redhat.com"
  ],
```

The output of **`docker inspect`** starts with general identification data shown above, then continues with the list of configuration parameters specified for the container:

```
    "Config": {
      "Hostname": "efefdc74a1d5",
      "User": "",
      "Memory": 0,
      "MemorySwap": 0,
      "CpuShares": 0,
      "AttachStdin": false,
```

```
      "AttachStdout": true,
      "AttachStderr": true,
      "PortSpecs": null,
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": null,
      "Cmd": [
          "ping",
          "www.google.com"
      ],
      "Dns": null,
      "Image": "learn/ping",
      "Volumes": null,
      "VolumesFrom": "",
      "Entrypoint": null
  },
```

The current status of the container follows:

```
    "State": {
      "Running": true,
      "Pid": 22249,
      "ExitCode": 0,
      "StartedAt": "2013-07-30T00:54:12.424817715Z",
      "Ghost": false
  },
```

The platform image is identified afterwards:

```
    "Image": "a1dbb48ce764c6651f5af98b46ed052a5f751233d731b645a6c57f91a4cb7158",
```

Network settings follow:

```
    "NetworkSettings": {
      "IPAddress": "172.16.42.6",
      "IPPrefixLen": 24,
      "Gateway": "172.16.42.1",
      "Bridge": "docker0",
      "PortMapping": {
          "Tcp": {},
          "Udp": {}
      }
  },
```

The output concludes with paths to host directories and summary of data volumes.

```
    "SysInitPath": "/usr/bin/docker",
    "ResolvConfPath": "/etc/resolv.conf",
    "Volumes": {},
    "VolumesRW": {}
```

Use the **-f** option to extract data from the output of **docker inspect**:

```
~]# docker inspect -f {{.State.Pid}} test_container
22249
~]# docker inspect -f {{.NetworkSettings.Gateway}} test_container
172.16.42.1
```

```
~]# docker inspect -f {{.Config.Hostname}} test_container
efefdc74a1d5
```

6.3.3. Viewing Container Logs and History

To view changes introduced by the container, type as **root**:

```
docker diff container_name
```

Where *container_name* stands for the name or ID of the inspected container. The output of **docker diff** can look as follows:

```
~]# docker diff 7bb0e258a
C /dev
A /dev/kmsg
C /etc
D /etc/mtab
A /go
...
```

As you can see in the above example, there are three types of change that can be listed in the output of **docker diff**:

- *A* – the file was added.
- *D* – the file was deleted.
- *C* – the file was changed.

To display logs from a running container, type as **root**:

```
docker logs container_name
```

Replace *container_name* with a name of the inspected container.

Example 6.5. Sending Logs from Container to Host

To view log messages created in containers on your host system, first mount the **/dev/log/** directory on host to all containers you wish to monitor:

```
~]# docker run -v /dev/log:/dev/log rhel7 /bin/sh
```

When **/dev/log/** is mounted, the container will automatically direct log messages into this directory. For more information on mounting external directories to containers, refer to Section 6.6.3, “Mounting a Host Directory to a Container”.

Use the following command to view the history of an image:

```
docker history image_name
```

Replace *image_name* with the name of the image you wish to check. The output of **docker history** contains a list of precessing images together with information on image size, date of creation, and the command of application that created the image. For example:

```
~]# docker history rhel7
IMAGE          CREATED         CREATED BY                                      SIZE
105182bb5e8b   5 days ago      /bin/sh -c #(nop) ADD file:71356d2   372.7 MB
eaa0d1ee1547   2 months ago    /bin/sh -c #(nop) MAINTAINER Peter   0 B
...
```

6.3.4. Montitoring Resource Consumption of Containers

For a dynamic view of processes currently running inside of a certain container, execute the following command as **root**:

```
docker top container_name
```

Replace *container_name* with the name or ID of a container you wish to inspect. The range of provided statistics is akin to the **top** utility, includes CPU and memory consumption of individual processes. You can limit the output with the same set of command-line options as with the **docker ps** command (see Section 6.3.1, “Listing Containers”).

6.4. Using Dockerfiles

Dockerfiles provide a configuration syntax that lets you automate the image creation. Dockerfile is a simple list of instructions that copy the steps you would otherwise perform manually. This way you can translate whole procedures to sequences of instructions written into a reusable file. Dockerfiles are simple text files, no file name extension is required to build them correctly.

When you build an image from a Dockerfile, the configured steps are executed one-by-one. Each instruction is run independently and has no influence on following commands in the file. Each instruction is committed to new container that is then used as a base for the next instruction. Whenever possible, Docker will reuse the intermediate images, accelerating the build significantly.

Note

You can effectively utilize the Docker cache by keeping consistent structure of your Dockerfiles. The instructions that are common for all your Dockerfiles, for example ***MAINTAINER***, should be kept in front. If you append new instruction at the bottom of the file, **Docker** will simply use cache instead of recreating the intermediate images.

6.4.1. Dockerfile Syntax

The configuration directives used in Dockerfiles have the following syntax:

```
INSTRUCTION arguments
```

INSTRUCTION stands for a container parameter – see Table 6.2, “Dockerfile Instructions” for a complete list of configurable directives. Replace *arguments* with values you want to assign to the parameter.

Table 6.2. Dockerfile Instructions

Instruction	Description
FROM	This instruction specifies a platform image for subsequent instructions.
MAINTAINER	Contact information of the author.

Instruction	Description
RUN	Executes a command on the current image and commits the results, which is an equivalent of running **`docker run`** and **`docker commit`** commands. The resulting committed image is used by following instructions specified later in the Dockerfile.
CMD	Lets you specify default parameters for an executing container. CMD can be used only once in a Dockerfile.
EXPOSE	This instruction specifies the ports to be exposed when running a container. Specify only private port, public ports will be assigned automatically so that the Dockerfile stays reusable.
ENV	Allows you to set environment variables for the container. ENV instructions are persistent, and are passed to all RUN instructions specified later in the Dockerfile.
ADD	This instruction is used to copy files from a source directory on host system to the file system of the container.
ENTRYPOINT	Triggers a specified command as soon as the container starts.
VOLUME	Creates a mount point the specified name and marks it as holding externally mounted volumes from native host or other containers.
USER	Sets the user name and UID to be used when running a container.
WORKDIR	Sets the working directory for the RUN, CMD and ENTRYPOINT instructions that follow in the Dockerfile.
ONBUILD	Lets you specify an action that will be triggered when the image is used as the base for another build.

Example 6.6. Dockerfile for Apache Contianer

The following example shows a Dockerfile used for building a **httpd** container:

```
FROM registry.access.redhat.com/redhat/rhel7beta
MAINTAINER "Scott Collier" <scollier@redhat.com>

RUN yum -y update; yum clean all
RUN yum -y install httpd; yum clean all
RUN echo "Apache" >> /var/www/html/index.html

EXPOSE 80

# Simple startup script to avoid some issues observed with container restart
ADD run-apache.sh /run-apache.sh
RUN chmod -v +x /run-apache.sh

CMD ["/run-apache.sh"]
```

The opening lines are common to most Dockerfiles, as they identify the platform image and the author. The **yum** commands on following lines are necessary to ensure that application you are running is up to date. Keep these instructions in front to boost the build performance as mentioned in Note on Docker cache.

The EXPOSE directive exposes a private port to the container. **Docker** will automatically assign a public port, so do not specify it here (for instance 80:8080), since you will only be able to run one instance of the containerized application.

Dockerfiles let you import and execute custom scripts with ADD and CMD directives. In this case **run-apache.sh** is called in to remove any incompletely-shutdown **httpd** contexts:

```
#!/bin/bash

rm -rf /run/httpd/*
exec /usr/sbin/apachectl -D FOREGROUND
```

6.4.2. Building an Image from Dockerfile

Use the following command to build an image from valid Dockerfile, type as **root**:

```
docker build --tag=tag path
```

Where *path* stands for the path to the Dockerfile that can be stored locally or accessed through the network. Replace *tag* with a short description of an image. The **--tag** option is not required, but it is a good practice to use tags whenever possible, since they help you to keep track of your builds. See the Dockerfile in action in the *Building an image from a Dockerfile* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal.

6.5. Networking

By default, each container has a networking functionality enabled. Docker uses Linux bridge capabilities to provide network connectivity to containers. The **docker0** bridge interface is managed by Docker for this purpose. When the Docker daemon starts, it:

- creates the **docker0** bridge if not present,
- searches for an IP address range which does not overlap with an existing route,
- picks an IP in the selected range,
- assigns this IP to the **docker0** bridge,

To assign a public-facing port to a private port through NAT, type as **root**:

```
docker port container_name private_port
```

Replace *container_name* with the name or ID of the container, and *private_port* with a port you wish to assign to the container.

To receive information about events on the server in real time, type as **root**:

```
docker events
```

Use the **--since** option to limit the output of the above command to certain time period.

Example 6.7. Displaying Server Events

Use the following syntax to track server events that appeared after selected date:

```
~]# docker events --since '2014-04-12'
[2014-04-12 18:14:13 -0400 EDT] 786d69800457: (from whenry/testimage:latest)
create
[2014-04-12 18:14:13 -0400 EDT] 786d69800457: (from whenry/testimage:latest)
start
[2014-04-12 18:22:44 -0400 EDT] 786d69800457: (from whenry/testimage:latest) die
[2014-04-12 18:22:44 -0400 EDT] 786d69800457: (from whenry/testimage:latest) stop
...
```

Warning

Changing the default network binding from Linux bridge to a TCP port or Unix docker user group will increase your security risks by allowing non-root users to gain root access on the host. If you are binding to a TCP port, anyone with access to that port has full Docker access; so make sure you will not do it on an open network.

With the **-H** option, it is possible to make Docker listen on a specific IP and port. By default, it listens on **unix:///var/run/docker.sock** to allow only local connections by the root user. It is possible to change this default to give access to everybody, but you are strongly recommended not to do it, because then it is trivial for remote attacker to gain root access to the host where Docker is running.

6.6. Sharing Data Across Containers

Containers are by definition isolated from the host system and also form each other. However, many applications require sharing persistent data across containers. Apart from networking (see Section 6.5, “Networking”), there are several options for doing this locally. You can copy data form a container to the

host system, share data though volume containers, mount host directories to containers, or export data from containers to archive files.

6.6.1. Copying Data from Container to Host

The most convenient way to move files and directories from the container file system to the host file system is by using the following command as **root**:

```
docker cp container_name:container_path host_path
```

Specify the container name or ID with *container_name*, replace *container_path* with a path to the file of directory in the container file system. Finally, *host_path* stands for the destination path on the host file system.

> **Example 6.8. Copying Files from Container to Host**
>
> To copy the **/etc/hosts** file from the container called *test_container* to the same location on the host file system, use:
>
> ```
> ~]# docker cp test_container:/etc/hosts .
> ```
>
> The period (".") character means that the destination path is the same as the source. The above command will therefore create a **/etc/hosts** file on the host file system.

6.6.2. Using Data Volume Containers

A *data volume* is a directory available to a container, but located outside of its root file system. This allows volume to bypass image layering, which makes it ideal for sharing persistent data between containers. Changes to a data volume are made directly and are not included in the commit tree. By default, containers have dedicated directories in the **/var/lib/docker/** directory, while volumes are stored separately in **/var/lib/docker/volumes/**. Volumes persist until no containers use them.

To create a data volume to your container, execute the following command as **root**:

```
docker run --name=container_name --volume=volume_path image_name command
```

The above syntax creates a container based on the image specified with *image_name* that runs a selected *command*. Replace *volume_path* with a directory on your host system you want to bind mount to the container. The **--volume** (or **-v**) option can be used one or more times to add more mounts to the container.

When creating a container, you can attach data volumes that were already mounted to another container by executing:

```
docker run --name=container2 --volumes-from=container1 image_name command
```

This way, the newly created *container2* mounts all data volumes used by *container1*. Note that the volumes can be shared even if the original container is not running.

The above commands allow you to create a *Data Volume Container* that acts as a collection of data volumes that other containers can access with the **--volumes-from** option (see Example 6.9, “Sharing Data Volumes Between Containers”).

Example 6.9. Sharing Data Volumes Between Containers

To create a data volume container that other containers can mount from, type:

```
~]# docker run --name=data -v /var/volume1 -v /var/volume2 rhel7-data true
```

With this command, you created a data volume container called **data** that has two shareable volumes. Now, you can refer to this container when creating other containers:

```
~]# docker run --volumes-from=data --name=container1 rhel7 bash
```

You can also take volumes from the **container1** instead of the original data volume container. This makes **container1** an intermediary container, and hides the true source of data.

6.6.3. Mounting a Host Directory to a Container

You can also use data volumes to mount directories from your host machine to the container's file system. In that case, the **--volume** syntax looks as follows:

```
--volume=host_dir:container_dir:access_rights
```

Replace:

- *host_dir* with a path to the directory on your host system that will be mounted.
- *container_dir* with a path to the destination directory in container file system.
- *access_rights* either with **ro** for read-only mount, or with **rw** to allow container to write into this directory.

The mounted directory becomes a data volume you can share with other containers by using the **--volumes-from** option.

Note

Host volume settings are not portable, since they are host-dependent and might not work on any other machine. For this reason, there is no Dockerfile equivalent for mounting host directories to the container. Also, be aware that the host system has no knowledge of container SELinux policy. Therefore, if SELinux policy is enforced, the mounted host directory is not writable to the container, regardless of the **rw** setting. Currently, you can work around this by assigning the proper SELinux policy type to the host directory:

```
~]# chcon -Rt svirt_sandbox_file_t host_dir
```

Where *host_dir* is a path to the directory on host system that is mounted to the container.

6.6.4. Using Archive Files

Archive files are useful for backing up or restoring containers and images. Note that you can not backup data volumes this way since they are external to containers.

To export the contents of a container file system as an archive in tar compress format, type as **root**:

```
docker export container_name
```

The output is exported to STDOUT and can be redirected to an archive file. Here, *container_name* stands for a name or ID of the container. Conversely, you can create a file system by importing content from an URL or a tar archive as follows:

```
docker import source
```

Replace *source* with a URL or a path to the archive file. Similarly, you can store a Docker image by executing as **root**:

```
docker save --output=file_name image_name
```

By default, **docker save** prints the output to STDOUT. Pass *file_name* to the **--output** option to export the image to an archive. Conversely, to load an image from the archive, type:

```
docker load --input=archive
```

Where *archive* stands for the path to the archive file.

6.7. Publishing Images

With use of **docker push** command, you can synchronize your image with a remote repository. By default, **Docker** uses the public registry on **index.docker.io**. However you cannot use this registry for Red Hat Enterprise Linux platform images. Therefore, Red Hat provides a specialized registry, see the *Creating a private Docker registry* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal. Also, you can create your own internal registry as shown in Section 6.7.1, “Creating a Private Registry”.

Use the following command to log in or register to a Docker registry server:

```
docker login --email="email" --username="username" --password="password" server
```

Replace *server* with the URI of the registry server you wish to connect to. If no server is specified, the default **https://index.docker.io/v1/** is selected. Replace *email*, *username*, and *password* with your login credentials.

Tags provide a way to mark certain images in the repository, which helps you to organize your data and saves your time. To place a tag on a certain image, type as **root**:

```
docker tag image_ID name:tag
```

Here, *image_ID* stands for the image ID, *registry/username/name* for the name of the image and *tag* represents the assigned tag.

6.7.1. Creating a Private Registry

A private repository is enabled for you on the Red Hat Customer portal to allow you to view images in a Red Hat registry. See the *Creating a private Docker registry* section of the Get Started with RHEL 7 Containers in Docker article on Red Hat Customer Portal.

However, you can host your own private registry. To push to or pull from a repository on your own registry, first prefix the tag with the host address of your registry. For example, if you created a registry on **localhost.localdomain:5000**, you can initialize a repository called **new_repo** by executing:

```
~]# docker tag 0u812deadbeef localhost.localdomain:5000/new_repo
```

This command binds the image **0u812deadbeef** with your custom repository. Then you can push the new repository to its home location on localhost:

```
~]# docker push localhost.localdomain:5000/new_repo
```

Once a repository has your registry's host name as part of the tag, you can use it like any other repository to push and pull images.

6.8. Additional Resources

To learn more about using Docker in Red Hat Enterprise Linux 7, refer to the following resources.

Installed Documentation

- **docker**(1) — The manual page of the **docker** command.
- **nsenter**(1) — The manual page of the **nsenter** command.

Online Documentation

- Get Started with RHEL 7 Containers in Docker — This article on Red Hat Customer Portal describes the essential Docker-related tasks and provides a number of examples.
- Documentation on the Docker Project Site — The official documentation of the Docker project.
- Red Hat Enterprise Linux 7 System Administrators Guide — The *System Administrator's Guide* documents relevant information regarding the deployment, configuration and administration of Red Hat Enterprise Linux 7. It is oriented towards system administrators with a basic understanding of the system.

Chapter 7. Using virsh

The following sections provide an overview of tasks related to installation, configuration, and management of Linux containers. The material below focuses on tools provided by the **libvirt** library, which are useful for basic container-related operations. You can also use the **Docker** application that offers a wider range of capabilities; for more information on **Docker**, see Chapter 6, *Using Docker*.

The **libvirt** library provides a necessary infrastructure for general-purpose containers together with the **virsh** utility as a default command-line interface for managing guest domains, such as virtual machines and Linux containers.

There are two kinds of Linux Containers you can create, either they are *persistent* or *volatile*. Persistent containers are preserved after reboot, define them using an XML configuration file. Temporary containers are deleted as soon as the contained application finishes, you can create them with the **virsh create** command.

7.1. Connecting to the LXC Driver

To execute container-related commands correctly, libvirt must be connected to LXC driver. This is not done by default as each host can only have one default libvirt URI, and the KVM driver typically takes precedence over LXC. To temporarily change the driver to LXC, use the **-c** (connect) argument before a command as follows (execute as **root**):

```
virsh -c lxc:/// command
```

With **-c lxc:///** specified in front of the command you change the connected driver to LXC. Since this change is temporary, the default URI is reset right after execution. All examples of container usage in this guide assume that LXC is not the default driver and therefore, the above syntax is used when necessary. However, you can avoid typing **-c lxc:///** before every command if you explicitly override the default URI for the libvirt session using the LIBVIRT_DEFAULT_URI environment variable as shown in Procedure 7.1, “Changing the Default libvirt Driver”.

Procedure 7.1. Changing the Default libvirt Driver

1. To identify your default libvirt URI, type:

   ```
   ~]# virsh uri
   qemu:///system
   ```

   In this case, the **qemu:///system** URI is set as default, which means KVM driver is connected.

2. Change the default setting for the libvirt session by typing:

   ```
   ~]# export LIBVIRT_DEFAULT_URI=lxc:///
   ```

   Note that this change is not preserved after system reboot.

3. To verify your new cofiguration, type:

   ```
   ~]# virsh uri
   lxc:///
   ```

7.2. The virsh Utility

The **virsh** utility is a general-purpose command-line interface for administration of virtualization domains. As such, it can be used to manage the capabilities of LXC domains. The **virsh** utility can be used, for example, to create, modify, and destroy containers, display information about existing containers, or manage resources, storage, and network connectivity of a container.

The following table describes virsh commands that are most often used in connection with Linux containers. For a complete list of virsh commands see the **virsh** manual page.

Table 7.1. virsh Commands

Virsh Command	Description
define	Creates a new container based on parameters in supplied libvirt configuration file in XML format.
undefine	Deletes a container. If the container is running, it is converted to a transient container which is removed with an application shutdown.
start	Starts a previously-defined container. With the **--console** option, you can directly connect to the newly created container. If the **--autodestroy** option is specified, the container will be automatically destroyed on **virsh** exit.
autostart	Sets the container to start automatically on system boot.
create	Defines and starts a non-persistent container in one step. The temporary container is based on libvirt configuration file. By executing the **shutdown** command you automatically destroy this container. The **--console** and **--autodestroy** options can be used as with the **start** command.
console	Connects to the virtual console of the container.
shutdown	Coordinates with the domain operating system to perform a graceful shutdown. The exact behavior can be specified with the parameter in the container's XML definition.
destroy	Immediately terminates the container. This can be used to shut down the container forcefully if it is not responding after executing **shutdown**.
edit	Opens the container's configuration file for editing and validates the changes before applying them.

7.3. Creating a Container

To create a Linux Container using the **virsh** utility, follow these steps:

1. Create a Libvirt configuration file in the XML format with the following required parameters:

```
<domain type='lxc'>
    <name>container_name</name>
    <memory>mem_limit</memory>
    <os>
        <type>exe</type>
        <init>/bin/sh</init>
    </os>
```

```
    <devices>
        <console type='pty'/>
    </devices>
</domain>
```

Here, replace *container_name* with a name for your container, and *mem_limit* with an initial memory limit for the container. In **libvirt**, the virtualization type for containers is defined as **exe**. The ***<init>*** parameter defines a path to the binary to spawn as the container's *init* (the process with PID 1). The last required parameter is the text console device, specified with the ***<console>*** parameter.

Apart from the aforementioned required parameters, there are several other settings you can apply, see Example 7.3, "Modifying a Container" for a list of these parameters. For more information on the syntax and formatting of a Libvirt XML configuration file, refer to Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide.

Example 7.1. Creating Libvirt Configuration File

The following is an example of Libvirt configuration file **test-container.xml**:

```
<domain type='lxc'>
    <name>test-container</name>
    <memory>102400</memory>
    <os>
        <type>exe</type>
        <init>/bin/sh</init>
    </os>
    <devices>
        <console type='pty'/>
    </devices>
</domain>
```

2. To import a new container to **Libvirt**, use the following syntax:

```
~]# virsh -c lxc:/// define config_file
```

Here, *config_file* stands for the XML configuration file you created in the previous step.

Example 7.2. Defining test-container.xml

To import the **test_container.xml** file to to **Libvirt**, type:

```
~]# virsh -c lxc:/// define test_container.xml
```

The following message will be returned:

```
Domain test-container defined from test-container.xml
```

7.4. Starting, Connecting to, and Stopping a Container

To start a previously-defined container, use the following command as **root**:

```
virsh -c lxc:/// start container_name
```

Replace *container_name* with a name of the container. Once a container is started, you can connect to it using the following command:

```
virsh -c lxc:/// console container_name
```

Note

Note that if a container uses the **/bin/sh** process as the init process with a **PID** of 1, exiting the shell will also shut down the container.

To stop a running container, execute the following command as **root**:

```
virsh -c lxc:/// shutdown container_name
```

If a container is not responding, it can be shut down forcefully by executing:

```
virsh -c lxc:/// destroy container_name
```

7.5. Modifying a Container

To modify any of the configuration settings of an existing container, run the following command as **root**:

```
virsh -c lxc:/// edit container_name
```

With *container_name*, specify the container whose settings you wish to modify. The above command opens the XML configuration file of the specified container in a text editor and lets you change any of the settings. The default editor option is *vi*, you can change it by changing the ***EDITOR*** environment variable to your editor of choice.

Example 7.3. Modifying a Container

The following example shows how the configuration file of the **test-container** from Example 7.1, "Creating Libvirt Configuration File" looks when opened by **virsh edit**:

```
<domain type='lxc'>
  <name>test-container</name>
  <uuid>a99736bb-8a7e-4fc5-99dc-bd96f6116b1c</uuid>
  <memory unit='KiB'>102400</memory>
  <currentMemory unit='KiB'>102400</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='x86_64'>exe</type>
    <init>/bin/sh</init>
  </os>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/libexec/libvirt_lxc</emulator>
    <interface type='network'>
      <source network='default'/>
    </interface>
    <console type='pty'>
      <target type='lxc' port='0'/>
    </console>
  </devices>
</domain>
```

You may notice that the configuration file opened by the **virsh edit** differs from the original configuration file that was used to create the container. This change is to show all possible settings that can be configured, not only the required ones displayed in Example 7.1, "Creating Libvirt Configuration File". For instance, you can modify the container's behavior on reboot or on crash.

Once the file has been edited, save the file and exit the editor. After doing so, **virsh edit** automatically validates your modified configuration file and in case of syntax errors, it prompts you to open the file again. The modified configuration takes effect next time the container boots. To apply the changes immediately, reboot the container (as **root**):

```
virsh -c lxc:/// reboot container_name
```

7.6. Automatically Starting a Container on Boot

To start the container automatically on boot, use the following command as **root**:

```
virsh -c lxc:/// autostart container_name
```

Replace *container_name* with a name of the container you want to start automatically on system boot. To disable this automatic start, type as **root**:

```
virsh -c lxc:/// autostart --disable container_name
```

Example 7.4. Using virsh autostart

To start the *test-container* domain automatically at boot time, type:

```
~]# virsh -c lxc:/// autostart test-container
```

When the command is executed successfully, the following message appears:

```
Domain test-container marked as autostarted
```

You can use the **virsh dominfo** command (see Example 7.6, “The Example Output of virsh dominfo”), to verify your new setting:

```
~]# virsh -c lxc:/// dominfo test-container | grep Autostart
Autostart:      enable
```

7.7. Removing a Container

To remove an existing container, run the following command as **root**:

```
virsh -c lxc:/// undefine container_name
```

Replace *container_name* with the name of the container you wish to remove. Undefining a container simply removes its configuration file. Thus, the container can no longer be started. If the container is running and it is undefined, it enters a transient state in which it has no configuration file on the disk. Once a transient container is shut down, it can not be started again.

Warning

The container is removed immediately after you execute the **virsh undefine** command. **virsh** will not prompt you for confirmation before deleting the container. Think twice before executing the command, as the remove operation is not reversible.

7.8. Monitoring a Container

To view a simple list of all existing containers, both running and inactive, type the following command as **root**:

```
virsh -c lxc:/// list --all
```

Example 7.5. The Example Output of virsh list

The output of the **virsh list --all** command can look as follows:

```
 Id    Name                           State
----------------------------------------------------
 4369  httpd-container-001            running
 -     test-container                 shut off
```

Once you know the name of a container, or its process ID if it is running, you can view the meta data of this container with the following command:

```
virsh -c lxc:/// dominfo container_name
```

Replace *container_name* with a name or PID of the container you wish to examine.

Example 7.6. The Example Output of virsh dominfo

The following example shows meta data of the **httpd-container-001** domain:

```
~]# virsh -c lxc:/// dominfo httpd-container-001
Id:             4369
Name:           httpd-container-001
UUID:           4e96844c-2bc6-43ab-aef9-8fb93de53095
OS Type:        exe
State:          running
CPU(s):         1
CPU time:       0.3s
Max memory:     524288 KiB
Used memory:    8880 KiB
Persistent:     yes
Autostart:      enable
Managed save:   unknown
Security model: selinux
Security DOI:   0
Security label: system_u:system_r:svirt_lxc_net_t:s0 (enforcing)
```

For a live view of currently running Linux Containers, you can use the **virt-top** utility that provides a variety of statistics of virtualization systems. To use **virt-top**, first install it as **root**:

```
yum install virt-top
```

To launch the utility, type:

```
virt-top -c lxc:///
```

The range of provided statistics and operations is similar to the **top** utility. For more information, see the **virt-top** manual page.

The above commands allow you to monitor the overall status and resource consumption of your containers. To go deeper beyond the container level to track individual applications running inside of a container, first connect to this container with the **virsh console** command. Then you can execute the usual monitoring commands such as **top** inside the container.

However, when you are running a large number of containers simultaneously, you may want to gain an overview of containerized processes without connecting to individual containers. In this case, you can use the **systemd-cgls** command that groups all processes within a container into a cgroup, with the name of this container. As an alternative, you can use the **machinectl** command to get information about containers from the host system. First, list all running containers as shown in the following example:

```
~]# machinectl
MACHINE                          CONTAINER SERVICE
lxc-httpd-container-001          container libvirt-lxc
lxc-test-container               container libvirt-lxc

2 machines listed.
```

Then you can view the status of one or more containers by executing:

```
~]# machinectl status -l container_name
```

Replace *container_name* with a name of the container you wish to inspect. This command requires the *lxc-* prefix before the name as shown the output of the **machinectl** command in the above example. The **-l** option ensures that the output is not abbreviated.

Example 7.7. The Example Output of machinectl status

Use the following command to see the status of the *test-container*:

```
~]# machinectl status -l lxc-test-container
lxc-test-container(73369262eca04dbcac288b6030b46b4c)
           Since: Wed 2014-02-05 06:46:50 MST; 1h 3min ago
          Leader: 2300
         Service: libvirt-lxc; class container
            Unit: machine-lxc\x2dtest\x2dcontainer.scope
                  ├─12760 /usr/libexec/libvirt_lxc --name test-container --
console 21 --security=selinux --h
                  └─12763 /bin/sh
```

Once you have found the PID of the containerized process, you can use standard tools on the host system to monitor what the process is doing. See the **systemd-cgls** and **machinectl** man pages for more information.

7.9. Networking with Linux Containers

The guests created with **virsh** can by default reach all network interfaces available on host. If the container configuration file does not list any network interfaces (such as in Example 7.1, “Creating Libvirt Configuration File”), the network namespace is not activated, allowing the containerized applications to bind to TCP or UDP addresses and ports from the host operating system. It also allows applications to access UNIX domain sockets associated with the host. To forbid the container an access to UNIX domains sockets, add the ***<privnet/>*** flag to the ***<features>*** parameter of the container configuration file.

With network namespace, you can dedicate a virtual network to the container. This network has to be previously defined with a configuration file in XML format stored in the **/etc/libvirt/qemu/networks/** directory. Also, the virtual network must be started with the **virsh net-start** command. To find more detailed instructions on how to create and manage virtual networks, refer to chapters *Network*

configuration and *Managing virtual networks* in Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide. To learn about general concepts and scenarios of virtual networking, see the chapter called *Virtual Networking* in the aforementioned guide.

To connect your container to a predefined virtual network, type as **root**:

```
virsh attach-interface domain type source --mac mac --config
```

- Replace *domain* with the name of the container that will use the network interface.
- Replace *type* with either **network** to indicate a physical network interface or with **bridge** if using a network bridge.
- With *source* you specify the name of the source network interface.
- Specify a mac address for your network interface with *mac*.
- Add the **--config** option option if you want to make the network attachment persistent. If not specified, your settings will not be applied after the system reboot.

Find the complete list of **virsh attach-interface** parameters in the virsh manual page.

To disconnect the container from the virtual network, type as **root**:

```
virsh detach-interface domain type --config
```

Here, *domain* stands for the name of the container, *type* identifies the type of the network as with the **virsh attach-interface** command above. The **--config** option makes the detachment persistent.

Virtual network can either use a Dynamic Host Configuration Protocol (DHCP) that automatically assigns TCP/IP information to client machines, or it can have manually assigned static IP address. The *httpd.service* is used in examples of container usage in this section; however, you can use *sshd.service* in the same manner without complications.

Note

Configuration file for a virtual network named *default* is installed as part of the *libvirt* package, and is configured to start automatically when **libvirtd** is started. The default network uses dynamic IP address assignment and operates in the NAT mode. Network Access Translation (NAT) protocol allows only outbound connections, so the virtual machines and containers using the default network are not directly visible from the network. Refer to Example 7.8, “Connecting httpd-container-001 to the Default Network”

Example 7.8. Connecting httpd-container-001 to the Default Network

As mentioned above, the *libvirt* package provides a default virtual network that is started automatically with **libvirtd**. To see the exact configuration, open the **/etc/libvirt/qemu/networks/default.xml** file, or use the **virsh net-edit *default*** command. The default configuration file can look as follows:

```
<network>
  <name>default</name>
  <bridge name="virbr0" />
  <forward/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254" />
    </dhcp>
  </ip>
</network>
```

To check if the network is running, type:

```
~]# virsh net-list
 Name                 State      Autostart     Persistent
----------------------------------------------------------
 default              active     yes           yes
```

With defined and running virtual network, use the **virsh attach-interface** command to connect a container to this network. For example, to persistently connect *httpd-coontainer-001* to the default virtual network, type:

```
~]# virsh attach-interface httpd-container-001 network default --config
```

To verify if the network is working correctly, connect to the container and execute the usual network-monitoring commands such as **ping** or **ip route**.

The default virtual network provided by **libvirt** operates in NAT mode, which makes it suitable mainly for testing purposes or for hosts that have dynamically changing network connectivity switching between ethernet, wifi and mobile connectivity. To expose your container to LAN or WAN, connect it to a network bridge.

A **network bridge** is a link-layer layer device which forwards traffic between networks based on MAC addresses. It makes forwarding decisions based on a table of MAC addresses which it builds by listening to network traffic and thereby learning what hosts are connected to each network. A software bridge can be used within a Linux host in order to emulate a hardware bridge, especially in virtualization applications for sharing a NIC with one or more virtual NICs. For more information on network bridging, see the chapter called *Configure Network Bridging* in Red Hat Enterprise Linux 7 Networking Guide.

Example 7.9. Connecting httpd-container-002 to LAN

Ethernet bridging is useful for machines with permanent wired LAN connection. Once the host networking is configured to have a bridge device, you can use this bridge for a virtual network. This requires creating a configuration file and then loading it into **libvirt**.

Imagine you have prepared a network bridge device called **br0** on your host operating system (see the chapter called *Configure Network Bridging* in Red Hat Enterprise Linux 7 Networking Guide). To use this device to create a virtual network, create the **lan.xml** file with the following content:

```
<network>
    <name>lan</name>
    <forward mode="bridge" />
    <bridge name="br0" />
</network>
```

After creating a valid configuration file, you can enable the virtual network. Type as **root**:

```
~]# virsh net-define lan.xml
```

If the network was successfully defined, the following message is displayed:

```
Network lan defined from lan.xml
```

Start the network and set it to be started automatically:

```
~]# virsh net-start lan
```

```
~]# virsh net-autostart lan
```

To check if the network is running type:

```
~]# virsh net-list
 Name                 State      Autostart     Persistent
----------------------------------------------------------
 default              active     yes           yes
 lan                  active     yes           yes
```

With prepared virtual network, attach it to the previously created container **httpd-container-002**:

```
~]# virsh attach-interface httpd-container-002 bridge lan --config
```

To verify if the network is working correctly, connect to the container and execute the usual network-monitoring commands such as **ping** or **ip route**.

The Linux **macvtap** driver provides an alternative way to configure a network bridge. It does not require any changes in network configuration on host, but on the other hand, it does not allow for connectivity between the host and guest operating system, only between the guest and other non-local machines. To set up the network using **macvtap**, follow the same steps as in the above example. The only difference is in the network configuration file, where you need to specify an interface device.

Example 7.10. Bridge Connectivity with MacVTap

The network configuration file for a **macvtap** bridge can look as follows:

```
<network>
    <name>lan02</name>
    <forward mode="bridge" />
        <interface dev="eth0" />
</network>
```

After creating the configuration file, start the network and connect the container to it as shown in Example 7.9, “Connecting httpd-container-002 to LAN”

You can find more information on **macvtap** in the section called *Network interfaces* in Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide.

7.10. Mounting Devices to a Container

To mount a device to the guest file system, use the general mounting syntax provided by **virsh**. The following command requires a definition of the device in an XML format. See the section called *PCI devices* in Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide to learn more about libvirt device configuration files. Type as **root**:

```
virsh attach-device domain file --config
```

Replace *domain* with the name of the container you wish to attach the device to, *file* stands for a libvirt configuration file for this device. Add **--config** to make this change persistent.

To detach a previously mounted device, type:

```
virsh detach-device domain file --config
```

where *domain*, *file*, and **--config** have the same meaning as with **virsh attach-device**.

In many cases, there is a need to attach an additional disk device to the container or to connect it to a virtual network. Therefore, **libvirt** provides more specific commands for mounting these types of devices. To learn about connecting the container to network interfaces see Section 7.9, “Networking with Linux Containers”. To attach a disk to the container, type as **root**:

```
virsh attach-disk domain source target --config
```

Replace *domain* with the name of the container, *source* stands for the path to the device to be mounted, while *target* defines how is the mounted device exposed to the guest. Add **--config** to make this change persistent. There are several other parameters you can define with **virsh attach-disk**, to see the complete list, refer to the virsh manual page.

To detach a previously mounted disk, type:

```
virsh detach-disk domain target --config
```

Here, *domain*, *target*, and **--config** have the same meaning as with **virsh attach-disk** described above.

7.11. Additional Resources

To learn more about using Linux Containers in Red Hat Enterprise Linux 7, refer to the following resources.

Installed Documentation

- **`virsh(1)`** — The manual page of the **`virsh`** command.
- **`systemd-cgls(1)`** — The manual page lists options for the **`systemd-cgls`** command.
- **`systemd-cgtop(1)`** — The manual page lists options for the **`systemd-cgtop`** command.
- **`machinectl(1)`** — The manual page describes the capabilities of the **`machinectl`** utility.

Online Documentation

- Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide — This guide instructs how to configure a Red Hat Enterprise Linux 7 host physical machine and how to install and configure guest virtual machines with different distributions, using the KVM hypervisor. Also included PCI device configuration, SR-IOV, networking, storage, device and guest virtual machine management, as well as troubleshooting, compatibility and restrictions.
- Red Hat Enterprise Linux 7 Networking Guide — The *Networking Guide* documents relevant information regarding the configuration and administration of network interfaces, networks and network services in Red Hat Enterprise Linux 7.

Revision History

Revision 0.0-0.13	Mon May 13 2013	Peter Ondrejka
Version for 7.0 GA release		