Chapter

12

Advanced Administration

Contents

This chapter revisits some aspects we already described, with a different perspective: instead of installing one single computer, we will study mass-deployment systems; instead of creating RAID or LVM volumes at install time, we'll learn to do it by hand so we can later revise our initial choices. Finally, we will discuss monitoring tools and virtualization techniques. As a consequence, this chapter is more particularly targeting professional administrators, and focuses somewhat less on individuals responsible for their home network.

12.1. RAID and LVM

chapter 4, “Installation” page 52 presented these technologies from the point of view of the installer, and how it integrated them to make their deployment easy from the start. After the initial installation, an administrator must be able to handle evolving storage space needs without having to resort to an expensive reinstallation. They must therefore understand the required tools for manipulating RAID and LVM volumes.

RAID and LVM are both techniques to abstract the mounted volumes from their physical counterparts (actual hard-disk drives or partitions thereof); the former ensures the security and availability of the data in case of hardware failure by introducing redundancy, the latter makes volume management more flexible and independent of the actual size of the underlying disks. In both cases, the system ends up with new block devices, which can be used to create filesystems or swap space, without necessarily having them mapped to one physical disk. RAID and LVM come from quite different backgrounds, but their functionality can overlap somewhat, which is why they are often mentioned together.

PERSPECTIVE

Btrfs combines LVM and RAID

While LVM and RAID are two distinct kernel subsystems that come between the disk block devices and their filesystems, *btrfs* is a filesystem, initially developed at Oracle, that purports to combine the featuresets of LVM and RAID and much more.

➨ https://btrfs.wiki.kernel.org/index.php/Main_Page

Among the noteworthy features are the ability to take a snapshot of a filesystem tree at any point in time. This snapshot copy doesn’t initially use any disk space, the data only being duplicated when one of the copies is modified. The filesystem also handles transparent compression of files, and checksums ensure the integrity of all stored data.

In both the RAID and LVM cases, the kernel provides a block device file, similar to the ones corresponding to a hard disk drive or a partition. When an application, or another part of the kernel, requires access to a block of such a device, the appropriate subsystem routes the block to the relevant physical layer. Depending on the configuration, this block can be stored on one or several physical disks, and its physical location may not be directly correlated to the location of the block in the logical device.

12.1.1. Software RAID

RAID means *Redundant Array of Independent Disks*. The goal of this system is to prevent data loss and ensure availability in case of hard disk failure. The general principle is quite simple: data are stored on several physical disks instead of only one, with a configurable level of redundancy. Depending on this amount of redundancy, and even in the event of an unexpected disk failure, data can be losslessly reconstructed from the remaining disks.

CULTURE

Independent **or** *inexpensive*?

The I in RAID initially stood for *inexpensive*, because RAID allowed a drastic increase in data safety without requiring investing in expensive high-end disks. Probably due to image concerns, however, it is now more customarily considered to stand for *independent*, which doesn't have the unsavory flavor of cheapness.

RAID can be implemented either by dedicated hardware (RAID modules integrated into SCSI or SATA controller cards) or by software abstraction (the kernel). Whether hardware or software, a RAID system with enough redundancy can transparently stay operational when a disk fails; the upper layers of the stack (applications) can even keep accessing the data in spite of the failure. Of course, this "degraded mode" can have an impact on performance, and redundancy is reduced, so a further disk failure can lead to data loss. In practice, therefore, one will strive to only stay in this degraded mode for as long as it takes to replace the failed disk. Once the new disk is in place, the RAID system can reconstruct the required data so as to return to a safe mode. The applications won't notice anything, apart from potentially reduced access speed, while the array is in degraded mode or during the reconstruction phase.

When RAID is implemented by hardware, its configuration generally happens within the BIOS setup tool, and the kernel will consider a RAID array as a single disk, which will work as a standard physical disk, although the device name may be different (depending on the driver).

We only focus on software RAID in this book.

Different RAID Levels

RAID is actually not a single system, but a range of systems identified by their levels; the levels differ by their layout and the amount of redundancy they provide. The more redundant, the more failure-proof, since the system will be able to keep working with more failed disks. The counterpart is that the usable space shrinks for a given set of disks; seen the other way, more disks will be needed to store a given amount of data.

Linear RAID Even though the kernel's RAID subsystem allows creating "linear RAID", this is not proper RAID, since this setup doesn't involve any redundancy. The kernel merely aggregates several disks end-to-end and provides the resulting aggregated volume as one virtual disk (one block device). That is about its only function. This setup is rarely used by itself (see later for the exceptions), especially since the lack of redundancy means that one disk failing makes the whole aggregate, and therefore all the data, unavailable.

RAID-0 This level doesn't provide any redundancy either, but disks aren't simply stuck on end one after another: they are divided in *stripes*, and the blocks on the virtual device are stored on stripes on alternating physical disks. In a two-disk RAID-0 setup, for instance, even-numbered blocks of the virtual device will be stored on the first physical disk, while odd-numbered blocks will end up on the second physical disk.

This system doesn't aim at increasing reliability, since (as in the linear case) the availability of all the data is jeopardized as soon as one disk fails, but at increasing performance:

during sequential access to large amounts of contiguous data, the kernel will be able to read from both disks (or write to them) in parallel, which increases the data transfer rate. The disks are utilized entirely by the RAID device, so they should have the same size not to lose performance.

RAID-0 use is shrinking, its niche being filled by LVM (see later).

RAID-1 This level, also known as "RAID mirroring", is both the simplest and the most widely used setup. In its standard form, it uses two physical disks of the same size, and provides a logical volume of the same size again. Data are stored identically on both disks, hence the "mirror" nickname. When one disk fails, the data is still available on the other. For really critical data, RAID-1 can of course be set up on more than two disks, with a direct impact on the ratio of hardware cost versus available payload space.

NOTE

Disks and cluster sizes

If two disks of different sizes are set up in a mirror, the bigger one will not be fully used, since it will contain the same data as the smallest one and nothing more. The useful available space provided by a RAID-1 volume therefore matches the size of the smallest disk in the array. This still holds for RAID volumes with a higher RAID level, even though redundancy is stored differently.

It is therefore important, when setting up RAID arrays (except for RAID-0 and "linear RAID"), to only assemble disks of identical, or very close, sizes, to avoid wasting resources.

NOTE

Spare disks

RAID levels that include redundancy allow assigning more disks than required to an array. The extra disks are used as spares when one of the main disks fails. For instance, in a mirror of two disks plus one spare, if one of the first two disks fails, the kernel will automatically (and immediately) reconstruct the mirror using the spare disk, so that redundancy stays assured after the reconstruction time. This can be used as another kind of safeguard for critical data.

One would be forgiven for wondering how this is better than simply mirroring on three disks to start with. The advantage of the "spare disk" configuration is that the spare disk can be shared across several RAID volumes. For instance, one can have three mirrored volumes, with redundancy assured even in the event of one disk failure, with only seven disks (three pairs, plus one shared spare), instead of the nine disks that would be required by three triplets.

This RAID level, although expensive (since only half of the physical storage space, at best, is useful), is widely used in practice. It is simple to understand, and it allows very simple backups: since both disks have identical contents, one of them can be temporarily extracted with no impact on the working system. Read performance is often increased since the kernel can read half of the data on each disk in parallel, while write performance isn't too severely degraded. In case of a RAID-1 array of N disks, the data stays available even with N-1 disk failures.

RAID-4 This RAID level, not widely used, uses N disks to store useful data, and an extra disk to store redundancy information. If that disk fails, the system can reconstruct its contents from the other N. If one of the N data disks fails, the remaining N-1 combined with the "parity" disk contain enough information to reconstruct the required data.

RAID-4 isn't too expensive since it only involves a one-in-N increase in costs and has no noticeable impact on read performance, but writes are slowed down. Furthermore, since a write to any of the N disks also involves a write to the parity disk, the latter sees many more writes than the former, and its lifespan can shorten dramatically as a consequence. Data on a RAID-4 array is safe only up to one failed disk (of the N+1).

RAID-5 RAID-5 addresses the asymmetry issue of RAID-4: parity blocks are spread over all of the N+1 disks, with no single disk having a particular role.

Read and write performance are identical to RAID-4. Here again, the system stays functional with up to one failed disk (of the N+1), but no more.

RAID-6 RAID-6 can be considered an extension of RAID-5, where each series of N blocks involves two redundancy blocks, and each such series of N+2 blocks is spread over N+2 disks.

This RAID level is slightly more expensive than the previous two, but it brings some extra safety since up to two drives (of the N+2) can fail without compromising data availability. The counterpart is that write operations now involve writing one data block and two redundancy blocks, which makes them even slower.

RAID-1+0 This isn't strictly speaking, a RAID level, but a stacking of two RAID groupings. Starting from 2×N disks, one first sets them up by pairs into N RAID-1 volumes; these N volumes are then aggregated into one, either by "linear RAID" or (increasingly) by LVM. This last case goes farther than pure RAID, but there is no problem with that.

RAID-1+0 can survive multiple disk failures: up to N in the 2×N array described above, provided that at least one disk keeps working in each of the RAID-1 pairs.

GOING FURTHER

RAID-10

RAID-10 is generally considered a synonym of RAID-1+0, but a Linux specificity makes it actually a generalization. This setup allows a system where each block is stored on two different disks, even with an odd number of disks, the copies being spread out along a configurable model.

Performances will vary depending on the chosen repartition model and redundancy level, and of the workload of the logical volume.

Obviously, the RAID level will be chosen according to the constraints and requirements of each application. Note that a single computer can have several distinct RAID arrays with different configurations.

Setting up RAID

Setting up RAID volumes requires the *mdadm* package; it provides the `mdadm` command, which allows creating and manipulating RAID arrays, as well as scripts and tools integrating it to the rest of the system, including the monitoring system.

Our example will be a server with a number of disks, some of which are already used, the rest being available to setup RAID. We initially have the following disks and partitions:

- the sdb disk, 4 GB, is entirely available;
- the sdc disk, 4 GB, is also entirely available;
- on the sdd disk, only partition sdd2 (about 4 GB) is available;
- finally, a sde disk, still 4 GB, entirely available.

NOTE
Identifying existing RAID volumes

The /proc/mdstat file lists existing volumes and their states. When creating a new RAID volume, care should be taken not to name it the same as an existing volume.

We're going to use these physical elements to build two volumes, one RAID-0 and one mirror (RAID-1). Let's start with the RAID-0 volume:

```
# mdadm --create /dev/md0 --level=0 --raid-devices=2 /dev/sdb /dev/sdc
mdadm: Defaulting to version 1.2 metadata
mdadm: array /dev/md0 started.
# mdadm --query /dev/md0
/dev/md0: 8.00GiB raid0 2 devices, 0 spares. Use mdadm --detail for more detail.
# mdadm --detail /dev/md0
/dev/md0:
           Version : 1.2
     Creation Time : Tue Jun 25 08:47:49 2019
        Raid Level : raid0
        Array Size : 8378368 (7.99 GiB 8.58 GB)
      Raid Devices : 2
     Total Devices : 2
       Persistence : Superblock is persistent

       Update Time : Tue Jun 25 08:47:49 2019
             State : clean
    Active Devices : 2
   Working Devices : 2
    Failed Devices : 0
     Spare Devices : 0

        Chunk Size : 512K

Consistency Policy : none

              Name : mirwiz:0  (local to host debian)
              UUID : 146e104f:66ccc06d:71c262d7:9af1fbc7
            Events : 0

    Number   Major   Minor   RaidDevice State
       0       8       32        0      active sync   /dev/sdb
       1       8       48        1      active sync   /dev/sdc
```

```
# mkfs.ext4 /dev/md0
mke2fs 1.44.5 (15-Dec-2018)
Discarding device blocks: done
Creating filesystem with 2094592 4k blocks and 524288 inodes
Filesystem UUID: 413c3dff-ab5e-44e7-ad34-cf1a029cfe98
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

# mkdir /srv/raid-0
# mount /dev/md0 /srv/raid-0
# df -h /srv/raid-0
Filesystem      Size  Used Avail Use% Mounted on
/dev/md0        7.9G   36M  7.4G   1% /srv/raid-0
```

The `mdadm --create` command requires several parameters: the name of the volume to create (`/dev/md*`, with MD standing for *Multiple Device*), the RAID level, the number of disks (which is compulsory despite being mostly meaningful only with RAID-1 and above), and the physical drives to use. Once the device is created, we can use it like we'd use a normal partition, create a filesystem on it, mount that filesystem, and so on. Note that our creation of a RAID-0 volume on `md0` is nothing but coincidence, and the numbering of the array doesn't need to be correlated to the chosen amount of redundancy. It is also possible to create named RAID arrays, by giving `mdadm` parameters such as `/dev/md/linear` instead of `/dev/md0`.

Creation of a RAID-1 follows a similar fashion, the differences only being noticeable after the creation:

```
# mdadm --create /dev/md1 --level=1 --raid-devices=2 /dev/sdd2 /dev/sde
mdadm: Note: this array has metadata at the start and
    may not be suitable as a boot device.  If you plan to
    store '/boot' on this device please ensure that
    your boot-loader understands md/v1.x metadata, or use
    --metadata=0.90
mdadm: largest drive (/dev/sdd2) exceeds size (4192192K) by more than 1%
Continue creating array? y
mdadm: Defaulting to version 1.2 metadata
mdadm: array /dev/md1 started.
# mdadm --query /dev/md1
/dev/md1: 4.00GiB raid1 2 devices, 0 spares. Use mdadm --detail for more detail.
# mdadm --detail /dev/md1
/dev/md1:
           Version : 1.2
     Creation Time : Tue Jun 25 10:21:22 2019
        Raid Level : raid1
        Array Size : 4189184 (4.00 GiB 4.29 GB)
     Used Dev Size : 4189184 (4.00 GiB 4.29 GB)
      Raid Devices : 2
     Total Devices : 2
       Persistence : Superblock is persistent

       Update Time : Tue Jun 25 10:22:03 2019
             State : clean, resyncing
    Active Devices : 2
   Working Devices : 2
    Failed Devices : 0
     Spare Devices : 0

Consistency Policy : resync

     Resync Status : 93% complete

              Name : mirwiz:1  (local to host debian)
              UUID : 7d123734:9677b7d6:72194f7d:9050771c
            Events : 16

    Number   Major   Minor   RaidDevice State
       0       8      64         0      active sync   /dev/sdd2
       1       8      80         1      active sync   /dev/sde
# mdadm --detail /dev/md1
/dev/md1:
[...]
          State : clean
[...]
```

TIP

RAID, disks and partitions

As illustrated by our example, RAID devices can be constructed out of disk partitions, and do not require full disks.

A few remarks are in order. First, `mdadm` notices that the physical elements have different sizes; since this implies that some space will be lost on the bigger element, a confirmation is required.

More importantly, note the state of the mirror. The normal state of a RAID mirror is that both disks have exactly the same contents. However, nothing guarantees this is the case when the volume is first created. The RAID subsystem will therefore provide that guarantee itself, and there will be a synchronization phase as soon as the RAID device is created. After some time (the exact amount will depend on the actual size of the disks…), the RAID array switches to the “active” or “clean” state. Note that during this reconstruction phase, the mirror is in a degraded mode, and redundancy isn’t assured. A disk failing during that risk window could lead to losing all the data. Large amounts of critical data, however, are rarely stored on a freshly created RAID array before its initial synchronization. Note that even in degraded mode, the `/dev/md1` is usable, and a filesystem can be created on it, as well as some data copied on it.

TIP

Starting a mirror in degraded mode

Sometimes two disks are not immediately available when one wants to start a RAID-1 mirror, for instance because one of the disks one plans to include is already used to store the data one wants to move to the array. In such circumstances, it is possible to deliberately create a degraded RAID-1 array by passing `missing` instead of a device file as one of the arguments to `mdadm`. Once the data have been copied to the “mirror”, the old disk can be added to the array. A synchronization will then take place, giving us the redundancy that was wanted in the first place.

TIP

Setting up a mirror without synchronization

RAID-1 volumes are often created to be used as a new disk, often considered blank. The actual initial contents of the disk is therefore not very relevant, since one only needs to know that the data written after the creation of the volume, in particular the filesystem, can be accessed later.

One might therefore wonder about the point of synchronizing both disks at creation time. Why care whether the contents are identical on zones of the volume that we know will only be read after we have written to them?

Fortunately, this synchronization phase can be avoided by passing the `--assume-clean` option to `mdadm`. However, this option can lead to surprises in cases where the initial data will be read (for instance if a filesystem is already present on the physical disks), which is why it isn’t enabled by default.

Now let’s see what happens when one of the elements of the RAID-1 array fails. `mdadm`, in particular its --fail option, allows simulating such a disk failure:

```
# mdadm /dev/md1 --fail /dev/sde
mdadm: set /dev/sde faulty in /dev/md1
# mdadm --detail /dev/md1
/dev/md1:
[...]
       Update Time : Tue Jun 25 11:03:44 2019
             State : clean, degraded
    Active Devices : 1
   Working Devices : 1
    Failed Devices : 1
     Spare Devices : 0

Consistency Policy : resync

              Name : mirwiz:1  (local to host debian)
              UUID : 7d123734:9677b7d6:72194f7d:9050771c
            Events : 20

    Number   Major   Minor   RaidDevice State
       -       0        0        0      removed
       1       8       80        1      active sync   /dev/sdd2

       0       8       64        -      faulty   /dev/sde
```

The contents of the volume are still accessible (and, if it is mounted, the applications don't notice a thing), but the data safety isn't assured anymore: should the sdd disk fail in turn, the data would be lost. We want to avoid that risk, so we'll replace the failed disk with a new one, sdf:

```
# mdadm /dev/md1 --add /dev/sdf
mdadm: added /dev/sdf
# mdadm --detail /dev/md1
/dev/md1:
[...]
      Raid Devices : 2
     Total Devices : 3
       Persistence : Superblock is persistent

       Update Time : Tue Jun 25 11:09:42 2019
             State : clean, degraded, recovering
    Active Devices : 1
   Working Devices : 2
    Failed Devices : 1
     Spare Devices : 1

Consistency Policy : resync

    Rebuild Status : 27% complete

              Name : mirwiz:1  (local to host debian)
              UUID : 7d123734:9677b7d6:72194f7d:9050771c
            Events : 26

    Number   Major   Minor   RaidDevice State
       2       8       96        0      spare rebuilding   /dev/sdf
       1       8       80        1      active sync   /dev/sdd2

       0       8       64        -      faulty   /dev/sde
# [...]
[...]
# mdadm --detail /dev/md1
/dev/md1:
[...]
       Update Time : Tue Jun 25 11:10:47 2019
             State : clean
    Active Devices : 2
   Working Devices : 2
    Failed Devices : 1
     Spare Devices : 0

Consistency Policy : resync

              Name : mirwiz:1  (local to host debian)
              UUID : 7d123734:9677b7d6:72194f7d:9050771c
            Events : 39
    Number   Major   Minor   RaidDevice State
       2       8       96        0      active sync   /dev/sdd2
       1       8       80        1      active sync   /dev/sdf

       0       8       64        -      faulty   /dev/sde
```

Here again, the kernel automatically triggers a reconstruction phase during which the volume, although still accessible, is in a degraded mode. Once the reconstruction is over, the RAID array is back to a normal state. One can then tell the system that the `sde` disk is about to be removed from the array, so as to end up with a classical RAID mirror on two disks:

```
# mdadm /dev/md1 --remove /dev/sde
mdadm: hot removed /dev/sde from /dev/md1
# mdadm --detail /dev/md1
/dev/md1:
[...]
    Number   Major   Minor   RaidDevice State
       2       8       96        0      active sync   /dev/sdd2
       1       8       80        1      active sync   /dev/sdf
```

From then on, the drive can be physically removed when the server is next switched off, or even hot-removed when the hardware configuration allows hot-swap. Such configurations include some SCSI controllers, most SATA disks, and external drives operating on USB or Firewire.

Backing up the Configuration

Most of the meta-data concerning RAID volumes are saved directly on the disks that make up these arrays, so that the kernel can detect the arrays and their components and assemble them automatically when the system starts up. However, backing up this configuration is encouraged, because this detection isn't fail-proof, and it is only expected that it will fail precisely in sensitive circumstances. In our example, if the `sde` disk failure had been real (instead of simulated) and the system had been restarted without removing this `sde` disk, this disk could start working again due to having been probed during the reboot. The kernel would then have three physical elements, each claiming to contain half of the same RAID volume. Another source of confusion can come when RAID volumes from two servers are consolidated onto one server only. If these arrays were running normally before the disks were moved, the kernel would be able to detect and reassemble the pairs properly; but if the moved disks had been aggregated into an `md1` on the old server, and the new server already has an `md1`, one of the mirrors would be renamed.

Backing up the configuration is therefore important, if only for reference. The standard way to do it is by editing the `/etc/mdadm/mdadm.conf` file, an example of which is listed here:

Example 12.1 *mdadm configuration file*

```
# mdadm.conf
```

```
#
# !NB! Run update-initramfs -u after updating this file.
# !NB! This will ensure that initramfs has an uptodate copy.
#
# Please refer to mdadm.conf(5) for information about this file.
#

# by default (built-in), scan all partitions (/proc/partitions) and all
# containers for MD superblocks. alternatively, specify devices to scan, using
# wildcards if desired.
DEVICE /dev/sd*

# auto-create devices with Debian standard permissions
CREATE owner=root group=disk mode=0660 auto=yes

# automatically tag new arrays as belonging to the local system
HOMEHOST <system>

# instruct the monitoring daemon where to send mail alerts
MAILADDR root

# definitions of existing MD arrays
ARRAY /dev/md0 metadata=1.2 name=mirwiz:0 UUID=146e104f:66ccc06d:71c262d7:9af1fbc7
ARRAY /dev/md1 metadata=1.2 name=mirwiz:1 UUID=7d123734:9677b7d6:72194f7d:9050771c

# This configuration was auto-generated on Tue, 25 Jun 2019 07:54:35 -0400 by mkconf
```

One of the most useful details is the DEVICE option, which lists the devices where the system will automatically look for components of RAID volumes at start-up time. In our example, we replaced the default value, partitions containers, with an explicit list of device files, since we chose to use entire disks and not only partitions, for some volumes.

The last two lines in our example are those allowing the kernel to safely pick which volume number to assign to which array. The metadata stored on the disks themselves are enough to re-assemble the volumes, but not to determine the volume number (and the matching `/dev/md*` device name).

Fortunately, these lines can be generated automatically:

```
# mdadm --misc --detail --brief /dev/md?
ARRAY /dev/md0 metadata=1.2 name=mirwiz:0 UUID=146e104f:66ccc06d:71c262d7:9af1fbc7
ARRAY /dev/md1 metadata=1.2 name=mirwiz:1 UUID=7d123734:9677b7d6:72194f7d:9050771c
```

The contents of these last two lines doesn't depend on the list of disks included in the volume. It is therefore not necessary to regenerate these lines when replacing a failed disk with a new one. On the other hand, care must be taken to update the file when creating or deleting a RAID array.

12.1.2. LVM

LVM, the *Logical Volume Manager*, is another approach to abstracting logical volumes from their physical supports, which focuses on increasing flexibility rather than increasing reliability. LVM allows changing a logical volume transparently as far as the applications are concerned; for instance, it is possible to add new disks, migrate the data to them, and remove the old disks, without unmounting the volume.

LVM Concepts

This flexibility is attained by a level of abstraction involving three concepts.

First, the PV (*Physical Volume*) is the entity closest to the hardware: it can be partitions on a disk, or a full disk, or even any other block device (including, for instance, a RAID array). Note that when a physical element is set up to be a PV for LVM, it should only be accessed via LVM, otherwise the system will get confused.

A number of PVs can be clustered in a VG (*Volume Group*), which can be compared to disks both virtual and extensible. VGs are abstract, and don't appear in a device file in the `/dev` hierarchy, so there is no risk of using them directly.

The third kind of object is the LV (*Logical Volume*), which is a chunk of a VG; if we keep the VG-as-disk analogy, the LV compares to a partition. The LV appears as a block device with an entry in `/dev`, and it can be used as any other physical partition can be (most commonly, to host a filesystem or swap space).

The important thing is that the splitting of a VG into LVs is entirely independent of its physical components (the PVs). A VG with only a single physical component (a disk for instance) can be split into a dozen logical volumes; similarly, a VG can use several physical disks and appear as a single large logical volume. The only constraint, obviously, is that the total size allocated to LVs can't be bigger than the total capacity of the PVs in the volume group.

It often makes sense, however, to have some kind of homogeneity among the physical components of a VG, and to split the VG into logical volumes that will have similar usage patterns. For instance, if the available hardware includes fast disks and slower disks, the fast ones could be clustered into one VG and the slower ones into another; chunks of the first one can then be assigned to applications requiring fast data access, while the second one will be kept for less demanding tasks.

In any case, keep in mind that an LV isn't particularly attached to any one PV. It is possible to influence where the data from an LV are physically stored, but this possibility isn't required for day-to-day use. On the contrary: when the set of physical components of a VG evolves, the physical storage locations corresponding to a particular LV can be migrated across disks (while staying within the PVs assigned to the VG, of course).

Setting up LVM

Let us now follow, step by step, the process of setting up LVM for a typical use case: we want to simplify a complex storage situation. Such a situation usually happens after some long and convoluted history of accumulated temporary measures. For the purposes of illustration, we'll consider a server where the storage needs have changed over time, ending up in a maze of available partitions split over several partially used disks. In more concrete terms, the following partitions are available:

- on the `sdb` disk, a `sdb2` partition, 4 GB;
- on the `sdc` disk, a `sdc3` partition, 3 GB;
- the `sdd` disk, 4 GB, is fully available;
- on the `sdf` disk, a `sdf1` partition, 4 GB; and a `sdf2` partition, 5 GB.

In addition, let's assume that disks `sdb` and `sdf` are faster than the other two.

Our goal is to set up three logical volumes for three different applications: a file server requiring 5 GB of storage space, a database (1 GB) and some space for back-ups (12 GB). The first two need good performance, but back-ups are less critical in terms of access speed. All these constraints prevent the use of partitions on their own; using LVM can abstract the physical size of the devices, so the only limit is the total available space.

The required tools are in the *lvm2* package and its dependencies. When they're installed, setting up LVM takes three steps, matching the three levels of concepts.

First, we prepare the physical volumes using `pvcreate`:

```
# pvcreate /dev/sdb2
  Physical volume ”/dev/sdb2” successfully created.
# pvdisplay
  ”/dev/sdb2” is a new physical volume of ”4.00 GiB”
  --- NEW Physical volume ---
  PV Name               /dev/sdb2
  VG Name
  PV Size               4.00 GiB
  Allocatable           NO
  PE Size               0
  Total PE              0
  Free PE               0
  Allocated PE          0
  PV UUID               z4Clgk-T5a4-C27o-1P0E-lIAF-OeUM-e7EMwq

# for i in sdc3 sdd sdf1 sdf2 ; do pvcreate /dev/$i ; done
  Physical volume ”/dev/sdc3” successfully created.
  Physical volume ”/dev/sdd” successfully created.
  Physical volume ”/dev/sdf1” successfully created.
  Physical volume ”/dev/sdf2” successfully created.
# pvdisplay -C
  PV         VG Fmt  Attr PSize  PFree
  /dev/sdb2     lvm2 ---   4.00g  4.00g
  /dev/sdc3     lvm2 ---   3.00g  3.00g
  /dev/sdd      lvm2 ---   4.00g  4.00g
  /dev/sdf1     lvm2 ---   4.00g  4.00g
  /dev/sdf2     lvm2 ---  <5.00g <5.00g
```

So far, so good; note that a PV can be set up on a full disk as well as on individual partitions of it. As shown above, the `pvdisplay` command lists the existing PVs, with two possible output formats.

Now let's assemble these physical elements into VGs using `vgcreate`. We'll gather only PVs from the fast disks into a `vg_critical` VG; the other VG, `vg_normal`, will also include slower elements.

```
# vgcreate vg_critical /dev/sdb2 /dev/sdf1
  Volume group ”vg_critical” successfully created
# vgdisplay
  --- Volume group ---
  VG Name               vg_critical
  System ID
  Format                lvm2
  Metadata Areas        2
  Metadata Sequence No  1
  VG Access             read/write
  VG Status             resizable
  MAX LV                0
  Cur LV                0
  Open LV               0
  Max PV                0
  Cur PV                2
  Act PV                2
  VG Size               7.99 GiB
  PE Size               4.00 MiB
  Total PE              2046
  Alloc PE / Size       0 / 0
  Free  PE / Size       2046 / 7.99 GiB
  VG UUID               wAbBjx-d82B-q7St-0KFf-z40h-w5Mh-uAXkNZ

# vgcreate vg_normal /dev/sdc3 /dev/sdd /dev/sdf2
  Volume group ”vg_normal” successfully created
# vgdisplay -C
  VG          #PV #LV #SN Attr   VSize   VFree
  vg_critical   2   0   0 wz--n-   7.99g   7.99g
  vg_normal     3   0   0 wz--n- <11.99g <11.99g
```

Here again, commands are rather straightforward (and `vgdisplay` proposes two output formats). Note that it is quite possible to use two partitions of the same physical disk into two different VGs. Note also that we used a `vg_` prefix to name our VGs, but it is nothing more than a convention.

We now have two “virtual disks”, sized about 8 GB and 12 GB respectively. Let’s now carve them up into “virtual partitions” (LVs). This involves the `lvcreate` command, and a slightly more complex syntax:

```
# lvdisplay
# lvcreate -n lv_files -L 5G vg_critical
  Logical volume ”lv_files” created.
# lvdisplay
  --- Logical volume ---
  LV Path                /dev/vg_critical/lv_files
  LV Name                lv_files
  VG Name                vg_critical
  LV UUID                W6XT08-iBBx-Nrw2-f8F2-r2y4-Ltds-UrKogV
  LV Write Access        read/write
  LV Creation host, time debian, 2019-11-30 22:45:46 -0500
  LV Status              available
  # open                 0
  LV Size                5.00 GiB
  Current LE             1280
  Segments               2
  Allocation             inherit
  Read ahead sectors     auto
  - currently set to     256
  Block device           254:0

# lvcreate -n lv_base -L 1G vg_critical
  Logical volume ”lv_base” created.
# lvcreate -n lv_backups -L 11.98G vg_normal
  Rounding up size to full physical extent 11.98 GiB
  Logical volume ”lv_backups” created.
# lvdisplay -C
  LV         VG          Attr     LSize  Pool Origin Data%  Meta%  Move Log Cpy%Sync
          Convert
  lv_base    vg_critical -wi-a---  1.00g
  lv_files   vg_critical -wi-a---  5.00g
  lv_backups vg_normal   -wi-a--- 11.98g
```

Two parameters are required when creating logical volumes; they must be passed to the `lvcreate` as options. The name of the LV to be created is specified with the -n option, and its size is generally given using the -L option. We also need to tell the command what VG to operate on, of course, hence the last parameter on the command line.

GOING FURTHER

`lvcreate` options

The `lvcreate` command has several options to allow tweaking how the LV is created.

Let’s first describe the `-l` option, with which the LV’s size can be given as a number of blocks (as opposed to the “human” units we used above). These blocks (called PEs, *physical extents*, in LVM terms) are contiguous units of storage space in PVs, and they can’t be split across LVs. When one wants to define storage space for an

LV with some precision, for instance to use the full available space, the `-l` option will probably be preferred over `-L`.

It is also possible to hint at the physical location of an LV, so that its extents are stored on a particular PV (while staying within the ones assigned to the VG, of course). Since we know that `sdb` is faster than `sdf`, we may want to store the `lv_base` there if we want to give an advantage to the database server compared to the file server. The command line becomes: `lvcreate -n lv_base -L 1G vg_critical /dev/sdb2`. Note that this command can fail if the PV doesn't have enough free extents. In our example, we would probably have to create `lv_base` before `lv_files` to avoid this situation – or free up some space on `sdb2` with the `pvmove` command.

Logical volumes, once created, end up as block device files in `/dev/mapper/`:

```
# ls -l /dev/mapper
total 0
crw------- 1 root root 10, 236 Jun 10 16:52 control
lrwxrwxrwx 1 root root       7 Jun 10 17:05 vg_critical-lv_base -> ../dm-1
lrwxrwxrwx 1 root root       7 Jun 10 17:05 vg_critical-lv_files -> ../dm-0
lrwxrwxrwx 1 root root       7 Jun 10 17:05 vg_normal-lv_backups -> ../dm-2
# ls -l /dev/dm-*
brw-rw---T 1 root disk 253, 0 Jun 10 17:05 /dev/dm-0
brw-rw---- 1 root disk 253, 1 Jun 10 17:05 /dev/dm-1
brw-rw---- 1 root disk 253, 2 Jun 10 17:05 /dev/dm-2
```

NOTE

Auto-detecting LVM volumes

When the computer boots, the `lvm2-activation` systemd service unit executes `vgchange -aay` to "activate" the volume groups: it scans the available devices; those that have been initialized as physical volumes for LVM are registered into the LVM subsystem, those that belong to volume groups are assembled, and the relevant logical volumes are started and made available. There is therefore no need to edit configuration files when creating or modifying LVM volumes.

Note, however, that the layout of the LVM elements (physical and logical volumes, and volume groups) is backed up in `/etc/lvm/backup`, which can be useful in case of a problem (or just to sneak a peek under the hood).

To make things easier, convenience symbolic links are also created in directories matching the VGs:

```
# ls -l /dev/vg_critical
total 0
lrwxrwxrwx 1 root root 7 Jun 10 17:05 lv_base -> ../dm-1
lrwxrwxrwx 1 root root 7 Jun 10 17:05 lv_files -> ../dm-0
# ls -l /dev/vg_normal
total 0
lrwxrwxrwx 1 root root 7 Jun 10 17:05 lv_backups -> ../dm-2
```

The LVs can then be used exactly like standard partitions:

```
# mkfs.ext4 /dev/vg_normal/lv_backups
mke2fs 1.44.5 (15-Dec-2018)
Discarding device blocks: done
Creating filesystem with 3140608 4k blocks and 786432 inodes
Filesystem UUID: b9e6ed2f-cb37-43e9-87d8-e77568446225
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done

# mkdir /srv/backups
# mount /dev/vg_normal/lv_backups /srv/backups
# df -h /srv/backups
Filesystem                        Size  Used Avail Use% Mounted on
/dev/mapper/vg_normal-lv_backups   12G   41M   12G   1% /srv/backups
# [...]
[...]
# cat /etc/fstab
[...]
/dev/vg_critical/lv_base    /srv/base       ext4 defaults 0 2
/dev/vg_critical/lv_files   /srv/files      ext4 defaults 0 2
/dev/vg_normal/lv_backups   /srv/backups    ext4 defaults 0 2
```

From the applications' point of view, the myriad small partitions have now been abstracted into one large 12 GB volume, with a friendlier name.

LVM Over Time

Even though the ability to aggregate partitions or physical disks is convenient, this is not the main advantage brought by LVM. The flexibility it brings is especially noticed as time passes, when needs evolve. In our example, let's assume that new large files must be stored, and that the LV dedicated to the file server is too small to contain them. Since we haven't used the whole space available in `vg_critical`, we can grow `lv_files`. For that purpose, we'll use the `lvresize` command, then `resize2fs` to adapt the filesystem accordingly:

```
# df -h /srv/files/
Filesystem                        Size  Used Avail Use% Mounted on
/dev/mapper/vg_critical-lv_files  4.9G  4.2G  485M  90% /srv/files
# lvdisplay -C vg_critical/lv_files
  LV       VG          Attr     LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync
        Convert
  lv_files vg_critical -wi-ao-- 5.00g
# vgdisplay -C vg_critical
  VG          #PV #LV #SN Attr   VSize VFree
  vg_critical   2   2   0 wz--n- 7.99g 1.99g
# lvresize -L 6G vg_critical/lv_files
  Size of logical volume vg_critical/lv_files changed from 5.00 GiB (1280 extents) to
        6.00 GiB (1536 extents).
  Logical volume vg_critical/lv_files successfully resized.
# lvdisplay -C vg_critical/lv_files
  LV       VG          Attr       LSize Pool Origin Data%  Meta%  Move Log Cpy%Sync
       Convert
  lv_files vg_critical -wi-ao---- 6.00g
# resize2fs /dev/vg_critical/lv_files
resize2fs 1.44.5 (15-Dec-2018)
Filesystem at /dev/vg_critical/lv_files is mounted on /srv/files; on-line resizing
       required
old_desc_blocks = 1, new_desc_blocks = 1
The filesystem on /dev/vg_critical/lv_files is now 1572864 (4k) blocks long.

# df -h /srv/files/
Filesystem                        Size  Used Avail Use% Mounted on
/dev/mapper/vg_critical-lv_files  5.9G  4.2G  1.5G  75% /srv/files
```

CAUTION

Resizing filesystems

Not all filesystems can be resized online; resizing a volume can therefore require unmounting the filesystem first and remounting it afterwards. Of course, if one wants to shrink the space allocated to an LV, the filesystem must be shrunk first; the order is reversed when the resizing goes in the other direction: the logical volume must be grown before the filesystem on it. It is rather straightforward, since at no time must the filesystem size be larger than the block device where it resides (whether that device is a physical partition or a logical volume).

The ext3, ext4 and xfs filesystems can be grown online, without unmounting; shrinking requires an unmount. The reiserfs filesystem allows online resizing in both directions. The venerable ext2 allows neither, and always requires unmounting.

We could proceed in a similar fashion to extend the volume hosting the database, only we've reached the VG's available space limit:

```
# df -h /srv/base/
Filesystem                       Size  Used Avail Use% Mounted on
/dev/mapper/vg_critical-lv_base  976M  882M   28M  97% /srv/base
# vgdisplay -C vg_critical
  VG          #PV #LV #SN Attr   VSize VFree
  vg_critical   2   2   0 wz--n- 7.99g 1016.00m
```

No matter, since LVM allows adding physical volumes to existing volume groups. For instance, maybe we've noticed that the `sdb1` partition, which was so far used outside of LVM, only contained archives that could be moved to `lv_backups`. We can now recycle it and integrate it to the volume group, and thereby reclaim some available space. This is the purpose of the `vgextend` command. Of course, the partition must be prepared as a physical volume beforehand. Once the VG has been extended, we can use similar commands as previously to grow the logical volume then the filesystem:

```
# pvcreate /dev/sdb1
  Physical volume ”/dev/sdb1” successfully created.
# vgextend vg_critical /dev/sdb1
  Volume group ”vg_critical” successfully extended
# vgdisplay -C vg_critical
  VG          #PV #LV #SN Attr   VSize  VFree
  vg_critical   3   2   0 wz--n- <9.99g <1.99g
# [...]
[...]
# df -h /srv/base/
Filesystem                       Size  Used Avail Use% Mounted on
/dev/mapper/vg_critical-lv_base  2.0G  882M  994M  48% /srv/base
```

GOING FURTHER

Advanced LVM

LVM also caters for more advanced uses, where many details can be specified by hand. For instance, an administrator can tweak the size of the blocks that make up physical and logical volumes, as well as their physical layout. It is also possible to move blocks across PVs, for instance, to fine-tune performance or, in a more mundane way, to free a PV when one needs to extract the corresponding physical disk from the VG (whether to affect it to another VG or to remove it from LVM altogether). The manual pages describing the commands are generally clear and detailed. A good entry point is the `lvm(8)` manual page.

12.1.3. RAID or LVM?

RAID and LVM both bring indisputable advantages as soon as one leaves the simple case of a desktop computer with a single hard disk where the usage pattern doesn't change over time. However, RAID and LVM go in two different directions, with diverging goals, and it is legitimate to wonder which one should be adopted. The most appropriate answer will of course depend on current and foreseeable requirements.

There are a few simple cases where the question doesn't really arise. If the requirement is to safeguard data against hardware failures, then obviously RAID will be set up on a redundant array of disks, since LVM doesn't really address this problem. If, on the other hand, the need is for a flexible storage scheme where the volumes are made independent of the physical layout of the disks, RAID doesn't help much and LVM will be the natural choice.

NOTE

If performance matters…

If input/output speed is of the essence, especially in terms of access times, using LVM and/or RAID in one of the many combinations may have some impact on performances, and this may influence decisions as to which to pick. However, these differences in performance are really minor, and will only be measurable in a few use cases. If performance matters, the best gain to be obtained would be to use non-rotating storage media (*solid-state drives* or SSDs); their cost per megabyte is higher than that of standard hard disk drives, and their capacity is usually smaller, but they provide excellent performance for random accesses. If the usage pattern includes many input/output operations scattered all around the filesystem, for instance for databases where complex queries are routinely being run, then the advantage of running them on an SSD far outweigh whatever could be gained by picking LVM over RAID or the reverse. In these situations, the choice should be determined by other considerations than pure speed, since the performance aspect is most easily handled by using SSDs.

The third notable use case is when one just wants to aggregate two disks into one volume, either for performance reasons or to have a single filesystem that is larger than any of the available disks. This case can be addressed both by a RAID-0 (or even linear-RAID) and by an LVM volume. When in this situation, and barring extra constraints (for instance, keeping in line with the rest of the computers if they only use RAID), the configuration of choice will often be LVM. The initial set up is barely more complex, and that slight increase in complexity more than makes up for the extra flexibility that LVM brings if the requirements change or if new disks need to be added.

Then of course, there is the really interesting use case, where the storage system needs to be made both resistant to hardware failure and flexible when it comes to volume allocation. Neither RAID nor LVM can address both requirements on their own; no matter, this is where we use both at the same time — or rather, one on top of the other. The scheme that has all but become a standard since RAID and LVM have reached maturity is to ensure data redundancy first by grouping disks in a small number of large RAID arrays, and to use these RAID arrays as LVM physical volumes; logical partitions will then be carved from these LVs for filesystems. The selling point of this setup is that when a disk fails, only a small number of RAID arrays will need to be reconstructed, thereby limiting the time spent by the administrator for recovery.

Let's take a concrete example: the public relations department at Falcot Corp needs a workstation for video editing, but the department's budget doesn't allow investing in high-end hardware from the bottom up. A decision is made to favor the hardware that is specific to the graphic nature of the work (monitor and video card), and to stay with generic hardware for storage. However, as is widely known, digital video does have some particular requirements for its storage: the amount of data to store is large, and the throughput rate for reading and writing this data is important for the overall system performance (more than typical access time, for instance). These constraints need to be fulfilled with generic hardware, in this case two 300 GB SATA hard disk drives; the system data must also be made resistant to hardware failure, as well as some of the user data. Edited videoclips must indeed be safe, but video rushes pending editing are less critical, since they're still on the videotapes.

RAID-1 and LVM are combined to satisfy these constraints. The disks are attached to two different SATA controllers to optimize parallel access and reduce the risk of a simultaneous failure, and they therefore appear as `sda` and `sdc`. They are partitioned identically along the following scheme:

```
# fdisk -l /dev/sda

Disk /dev/sda: 300 GB, 300090728448 bytes, 586114704 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
```

```
Disk identifier: 0x00039a9f

Device    Boot     Start       End   Sectors Size Id Type
/dev/sda1 *         2048   1992060   1990012 1.0G fd Linux raid autodetect
/dev/sda2        1992061   3984120   1992059 1.0G 82 Linux swap / Solaris
/dev/sda3        4000185 586099395 582099210 298G 5  Extended
/dev/sda5        4000185 203977305 199977120 102G fd Linux raid autodetect
/dev/sda6      203977306 403970490 199993184 102G fd Linux raid autodetect
/dev/sda7      403970491 586099395 182128904  93G 8e Linux LVM
```

- The first partitions of both disks (about 1 GB) are assembled into a RAID-1 volume, md0. This mirror is directly used to store the root filesystem.
- The sda2 and sdc2 partitions are used as swap partitions, providing a total 2 GB of swap space. With 1 GB of RAM, the workstation has a comfortable amount of available memory.
- The sda5 and sdc5 partitions, as well as sda6 and sdc6, are assembled into two new RAID-1 volumes of about 100 GB each, md1 and md2. Both these mirrors are initialized as physical volumes for LVM, and assigned to the vg_raid volume group. This VG thus contains about 200 GB of safe space.
- The remaining partitions, sda7 and sdc7, are directly used as physical volumes, and assigned to another VG called vg_bulk, which therefore ends up with roughly 200 GB of space.

Once the VGs are created, they can be partitioned in a very flexible way. One must keep in mind that LVs created in vg_raid will be preserved even if one of the disks fails, which will not be the case for LVs created in vg_bulk; on the other hand, the latter will be allocated in parallel on both disks, which allows higher read or write speeds for large files.

We will therefore create the lv_var and lv_home LVs on vg_raid, to host the matching filesystems; another large LV, lv_movies, will be used to host the definitive versions of movies after editing. The other VG will be split into a large lv_rushes, for data straight out of the digital video cameras, and a lv_tmp for temporary files. The location of the work area is a less straightforward choice to make: while good performance is needed for that volume, is it worth risking losing work if a disk fails during an editing session? Depending on the answer to that question, the relevant LV will be created on one VG or the other.

We now have both some redundancy for important data and much flexibility in how the available space is split across the applications.

NOTE

Why three RAID-1 volumes?

We could have set up one RAID-1 volume only, to serve as a physical volume for vg_raid. Why create three of them, then?

The rationale for the first split (md0 vs. the others) is about data safety: data written to both elements of a RAID-1 mirror are exactly the same, and it is therefore possible to bypass the RAID layer and mount one of the disks directly. In case of a kernel bug, for instance, or if the LVM metadata become corrupted, it is still possible to boot a minimal system to access critical data such as the layout of disks in the RAID and LVM volumes; the metadata can then be reconstructed and the files can be accessed again, so that the system can be brought back to its nominal state.

The rationale for the second split (`md1` vs. `md2`) is less clear-cut, and more related to acknowledging that the future is uncertain. When the workstation is first assembled, the exact storage requirements are not necessarily known with perfect precision; they can also evolve over time. In our case, we can't know in advance the actual storage space requirements for video rushes and complete video clips. If one particular clip needs a very large amount of rushes, and the VG dedicated to redundant data is less than halfway full, we can re-use some of its unneeded space. We can remove one of the physical volumes, say `md2`, from `vg_raid` and either assign it to `vg_bulk` directly (if the expected duration of the operation is short enough that we can live with the temporary drop in performance), or undo the RAID setup on `md2` and integrate its components `sda6` and `sdc6` into the bulk VG (which grows by 200 GB instead of 100 GB); the `lv_rushes` logical volume can then be grown according to requirements.

12.2. Virtualization

Virtualization is one of the most major advances in the recent years of computing. The term covers various abstractions and techniques simulating virtual computers with a variable degree of independence on the actual hardware. One physical server can then host several systems working at the same time and in isolation. Applications are many, and often derive from this isolation: test environments with varying configurations for instance, or separation of hosted services across different virtual machines for security.

There are multiple virtualization solutions, each with its own pros and cons. This book will focus on Xen, LXC, and KVM, but other noteworthy implementations include the following:

- QEMU is a software emulator for a full computer; performances are far from the speed one could achieve running natively, but this allows running unmodified or experimental operating systems on the emulated hardware. It also allows emulating a different hardware architecture: for instance, an *amd64* system can emulate an *arm* computer. QEMU is free software.

  ➨ `https://www.qemu.org/`

- Bochs is another free virtual machine, but it only emulates the x86 architectures (i386 and amd64).

- VMWare is a proprietary virtual machine; being one of the oldest out there, it is also one of the most widely-known. It works on principles similar to QEMU. VMWare proposes advanced features such as snapshotting a running virtual machine.

  ➨ `https://www.vmware.com/`

- VirtualBox is a virtual machine that is mostly free software (some extra components are available under a proprietary license). Unfortunately it is in Debian's "contrib" section because it includes some precompiled files that cannot be rebuilt without a proprietary compiler and it currently only resides in Debian Unstable as Oracle's policies make it impossible to keep it secure in a Debian stable release (see #794466[1]). While younger than VMWare and restricted to the i386 and amd64 architectures, it still includes some snapshotting and other interesting features.

➨ https://www.virtualbox.org/

HARDWARE

Virtualization support

Some computers might not have hardware virtualization support; when they do, it should be enabled in the BIOS.

To know if you have virtualization support enabled, you can check if the relevant flag is enabled with `grep`. If the following command for your processor returns some text, you already have virtualization support enabled:

- For Intel processors you can execute `grep vmx /proc/cpuinfo`
- For AMD processors you can execute `grep svm /proc/cpuinfo`

12.2.1. Xen

Xen is a "paravirtualization" solution. It introduces a thin abstraction layer, called a "hypervisor", between the hardware and the upper systems; this acts as a referee that controls access to hardware from the virtual machines. However, it only handles a few of the instructions, the rest is directly executed by the hardware on behalf of the systems. The main advantage is that performances are not degraded, and systems run close to native speed; the drawback is that the kernels of the operating systems one wishes to use on a Xen hypervisor need to be adapted to run on Xen.

Let's spend some time on terms. The hypervisor is the lowest layer, that runs directly on the hardware, even below the kernel. This hypervisor can split the rest of the software across several *domains*, which can be seen as so many virtual machines. One of these domains (the first one that gets started) is known as *dom0*, and has a special role, since only this domain can control the hypervisor and the execution of other domains. These other domains are known as *domU*. In other words, and from a user point of view, the *dom0* matches the "host" of other virtualization systems, while a *domU* can be seen as a "guest".

CULTURE

Xen and the various versions of Linux

Xen was initially developed as a set of patches that lived out of the official tree, and not integrated to the Linux kernel. At the same time, several upcoming virtualization systems (including KVM) required some generic virtualization-related functions to facilitate their integration, and the Linux kernel gained this set of functions (known as the *paravirt_ops* or *pv_ops* interface). Since the Xen patches were duplicating some of the functionality of this interface, they couldn't be accepted officially.

Xensource, the company behind Xen, therefore had to port Xen to this new framework, so that the Xen patches could be merged into the official Linux kernel. That meant a lot of code rewrite, and although Xensource soon had a working version

[1] https://bugs.debian.org/794466

based on the paravirt_ops interface, the patches were only progressively merged into the official kernel. The merge was completed in Linux 3.0.

➨ https://wiki.xenproject.org/wiki/XenParavirtOps

Since *Jessie* is based on version 3.16 of the Linux kernel, the standard *linux-image-686-pae* and *linux-image-amd64* packages include the necessary code, and the distribution-specific patching that was required for *Squeeze* and earlier versions of Debian is no more.

➨ https://wiki.xenproject.org/wiki/Xen_Kernel_Feature_Matrix

CULTURE

Xen and non-Linux kernels

Xen requires modifications to all the operating systems one wants to run on it; not all kernels have the same level of maturity in this regard. Many are fully-functional, both as dom0 and domU: Linux 3.0 and later, NetBSD 4.0 and later, and OpenSolaris. Others only work as a domU. You can check the status of each operating system in the Xen wiki:

➨ https://wiki.xenproject.org/wiki/Dom0_Kernels_for_Xen

➨ https://wiki.xenproject.org/wiki/DomU_Support_for_Xen

However, if Xen can rely on the hardware functions dedicated to virtualization (which are only present in more recent processors), even non-modified operating systems can run as domU (including Windows).

NOTE

Architectures compatible with Xen

Xen is currently only available for the i386, amd64, arm64 and armhf architectures.

Using Xen under Debian requires three components:

- The hypervisor itself. According to the available hardware, the appropriate package will be either *xen-hypervisor-4.11-amd64*, *xen-hypervisor-4.11-armhf*, or *xen-hypervisor-4.11-arm64*.
- A kernel that runs on that hypervisor. Any kernel more recent than 3.0 will do, including the 4.19 version present in *Buster*.
- The i386 architecture also requires a standard library with the appropriate patches taking advantage of Xen; this is in the *libc6-xen* package.

The hypervisor also brings *xen-utils-4.11*, which contains tools to control the hypervisor from the dom0. This in turn brings the appropriate standard library. During the installation of all that, configuration scripts also create a new entry in the GRUB bootloader menu, so as to start the chosen kernel in a Xen dom0. Note, however, that this entry is not usually set to be the first one in the list, but it will be selected by default.

Once these prerequisites are installed, the next step is to test the behavior of the dom0 by itself; this involves a reboot to the hypervisor and the Xen kernel. The system should boot in its standard fashion, with a few extra messages on the console during the early initialization steps.

Now is the time to actually install useful systems on the domU systems, using the tools from *xen-tools*. This package provides the `xen-create-image` command, which largely automates the task. The only mandatory parameter is --hostname, giving a name to the domU; other options are important, but they can be stored in the /etc/xen-tools/xen-tools.conf configuration file, and their absence from the command line doesn't trigger an error. It is therefore important to either check the contents of this file before creating images, or to use extra parameters in the xen-create-image invocation. Important parameters of note include the following:

- --memory, to specify the amount of RAM dedicated to the newly created system;
- --size and --swap, to define the size of the "virtual disks" available to the domU;
- --debootstrap-cmd, to specify the which debootstrap command is used. The default is debootstrap if debootstrap and cdebootstrap are installed. In that case, the --dist option will also most often be used (with a distribution name such as *buster*).

GOING FURTHER

Installing a non-Debian system in a domU

In case of a non-Linux system, care should be taken to define the kernel the domU must use, using the --kernel option.

- --dhcp states that the domU's network configuration should be obtained by DHCP while --ip allows defining a static IP address.
- Lastly, a storage method must be chosen for the images to be created (those that will be seen as hard disk drives from the domU). The simplest method, corresponding to the --dir option, is to create one file on the dom0 for each device the domU should be provided. For systems using LVM, the alternative is to use the --lvm option, followed by the name of a volume group; xen-create-image will then create a new logical volume inside that group, and this logical volume will be made available to the domU as a hard disk drive.

NOTE

Storage in the domU

Entire hard disks can also be exported to the domU, as well as partitions, RAID arrays or pre-existing LVM logical volumes. These operations are not automated by xen-create-image, however, so editing the Xen image's configuration file is in order after its initial creation with xen-create-image.

Once these choices are made, we can create the image for our future Xen domU:

```
# xen-create-image --hostname testxen --dhcp --dir /srv/testxen --size=2G --dist=
       buster --role=udev

[...]
eneral Information
--------------------
Hostname       :  testxen
Distribution   :  buster
Mirror         :  http://deb.debian.org/debian
Partitions     :  swap            512M  (swap)
                  /               2G    (ext4)
Image type     :  sparse
Memory size    :  256M
Kernel path    :  /boot/vmlinuz-4.19.0-5-amd64
Initrd path    :  /boot/initrd.img-4.19.0-5-amd64
[...]
Logfile produced at:
         /var/log/xen-tools/testxen.log
```

```
Installation Summary
---------------------
Hostname        :  testxen
Distribution    :  buster
MAC Address     :  00:16:3E:0C:74:2F
IP Address(es)  :  dynamic
SSH Fingerprint :  SHA256:PuAGX4/4S07Xzh1u0Cl2tL04EL5udf9ajvvbufBrfvU (DSA)
SSH Fingerprint :  SHA256:ajFTX54eakzolyzmZku/ihq/BK6KYsz5MewJ98BM5co (ECDSA)
SSH Fingerprint :  SHA256:/sFov86b+rD/bRSJoHKbiMqzGFiwgZulEwpzsiw6aSc (ED25519)
SSH Fingerprint :  SHA256:/NJg/CcoVj+OLE/cL3yyJINStnla7YkHKe3/xEdVGqc (RSA)
Root Password   :  EwmQMHtywY9zsRBpqQuxZTb
```

We now have a virtual machine, but it is currently not running (and therefore only using space on the dom0's hard disk). Of course, we can create more images, possibly with different parameters.

Before turning these virtual machines on, we need to define how they'll be accessed. They can of course be considered as isolated machines, only accessed through their system console, but this rarely matches the usage pattern. Most of the time, a domU will be considered as a remote server, and accessed only through a network. However, it would be quite inconvenient to add a network card for each domU; which is why Xen allows creating virtual interfaces, that each domain can see and use in a standard way. Note that these cards, even though they're virtual, will only be useful once connected to a network, even a virtual one. Xen has several network models for that:

- The simplest model is the *bridge* model; all the eth0 network cards (both in the dom0 and the domU systems) behave as if they were directly plugged into an Ethernet switch.
- Then comes the *routing* model, where the dom0 behaves as a router that stands between the domU systems and the (physical) external network.
- Finally, in the *NAT* model, the dom0 is again between the domU systems and the rest of the network, but the domU systems are not directly accessible from outside, and traffic goes through some network address translation on the dom0.

These three networking nodes involve a number of interfaces with unusual names, such as `vif*`, `veth*`, `peth*` and `xenbr0`. The Xen hypervisor arranges them in whichever layout has been defined, under the control of the user-space tools. Since the NAT and routing models are only adapted to particular cases, we will only address the bridging model.

The standard configuration of the Xen packages does not change the system-wide network configuration. However, the `xend` daemon is configured to integrate virtual network interfaces into any pre-existing network bridge (with `xenbr0` taking precedence if several such bridges exist). We must therefore set up a bridge in `/etc/network/interfaces` (which requires installing the *bridge-utils* package, which is why the *xen-utils-4.11* package recommends it) to replace the existing eth0 entry:

```
auto xenbr0
iface xenbr0 inet dhcp
```

```
    bridge_ports eth0
    bridge_maxwait 0
```

After rebooting to make sure the bridge is automatically created, we can now start the domU with the Xen control tools, in particular the `xl` command. This command allows different manipulations on the domains, including listing them and, starting/stopping them. You might need to increase the default memory by editing the variable memory from configuration file (in this case, `/etc/xen/testxen.cfg`). Here we have set it to 1024 (megabytes).

```
# xl list
Name                                        ID   Mem VCPUs        State        Time(s
       )
Domain-0                                     0  1894     2     r-----      63.5
# xl create /etc/xen/testxen.cfg
Parsing config from /etc/xen/testxen.cfg
# xl list
Name                                        ID   Mem VCPUs        State        Time(s
       )
Domain-0                                     0  1505     2     r-----     100.0
testxen                                     13  1024     0     --p---       0.0
```

TOOL

Choice of toolstacks to manage Xen VM

In Debian 7 and older releases, `xm` was the reference command line tool to use to manage Xen virtual machines. It has now been replaced by `xl` which is mostly backwards compatible. But those are not the only available tools: `virsh` of libvirt and `xe` of XenServer's XAPI (commercial offering of Xen) are alternative tools.

CAUTION

Only one domU per image

While it is of course possible to have several domU systems running in parallel, they will all need to use their own image, since each domU is made to believe it runs on its own hardware (apart from the small slice of the kernel that talks to the hypervisor). In particular, it isn't possible for two domU systems running simultaneously to share storage space. If the domU systems are not run at the same time, it is, however, quite possible to reuse a single swap partition, or the partition hosting the `/home` filesystem.

Note that the `testxen` domU uses real memory taken from the RAM that would otherwise be available to the dom0, not simulated memory. Care should therefore be taken, when building a server meant to host Xen instances, to provision the physical RAM accordingly.

Voilà! Our virtual machine is starting up. We can access it in one of two modes. The usual way is to connect to it "remotely" through the network, as we would connect to a real machine; this will usually require setting up either a DHCP server or some DNS configuration. The other way, which may be the only way if the network configuration was incorrect, is to use the `hvc0` console, with the `xl console` command:

```
# xl console testxen
[...]
```

```
Debian GNU/Linux 10 testxen hvc0

testxen login:
```

One can then open a session, just like one would do if sitting at the virtual machine's keyboard. Detaching from this console is achieved through the Control+] key combination.

TIP

Getting the console straight away

Sometimes one wishes to start a domU system and get to its console straight away; this is why the `xl create` command takes a `-c` switch. Starting a domU with this switch will display all the messages as the system boots.

TOOL

OpenXenManager

OpenXenManager (in the *openxenmanager* package) is a graphical interface allowing remote management of Xen domains via Xen's API. It can thus control Xen domains remotely. It provides most of the features of the `xl` command.

Once the domU is up, it can be used just like any other server (since it is a GNU/Linux system after all). However, its virtual machine status allows some extra features. For instance, a domU can be temporarily paused then resumed, with the `xl pause` and `xl unpause` commands. Note that even though a paused domU does not use any processor power, its allocated memory is still in use. It may be interesting to consider the `xl save` and `xl restore` commands: saving a domU frees the resources that were previously used by this domU, including RAM. When restored (or unpaused, for that matter), a domU doesn't even notice anything beyond the passage of time. If a domU was running when the dom0 is shut down, the packaged scripts automatically save the domU, and restore it on the next boot. This will of course involve the standard inconvenience incurred when hibernating a laptop computer, for instance; in particular, if the domU is suspended for too long, network connections may expire. Note also that Xen is so far incompatible with a large part of ACPI power management, which precludes suspending the host (dom0) system.

DOCUMENTATION

`xl` options

Most of the `xl` subcommands expect one or more arguments, often a domU name. These arguments are well described in the `xl(1)` manual page.

Halting or rebooting a domU can be done either from within the domU (with the `shutdown` command) or from the dom0, with `xl shutdown` or `xl reboot`.

GOING FURTHER

Advanced Xen

Xen has many more features than we can describe in these few paragraphs. In particular, the system is very dynamic, and many parameters for one domain (such as the amount of allocated memory, the visible hard drives, the behavior of the task scheduler, and so on) can be adjusted even when that domain is running. A domU can even be migrated across servers without being shut down, and without losing its network connections□ For all these advanced aspects, the primary source of information is the official Xen documentation.

➨ https://xenproject.org/help/documentation/

12.2.2. LXC

Even though it is used to build “virtual machines”, LXC is not, strictly speaking, a virtualization system, but a system to isolate groups of processes from each other even though they all run on the same host. It takes advantage of a set of recent evolutions in the Linux kernel, collectively known as *control groups*, by which different sets of processes called “groups” have different views of certain aspects of the overall system. Most notable among these aspects are the process identifiers, the network configuration, and the mount points. Such a group of isolated processes will not have any access to the other processes in the system, and its accesses to the filesystem can be restricted to a specific subset. It can also have its own network interface and routing table, and it may be configured to only see a subset of the available devices present on the system.

These features can be combined to isolate a whole process family starting from the `init` process, and the resulting set looks very much like a virtual machine. The official name for such a setup is a “container” (hence the LXC moniker: *LinuX Containers*), but a rather important difference with “real” virtual machines such as provided by Xen or KVM is that there is no second kernel; the container uses the very same kernel as the host system. This has both pros and cons: advantages include excellent performance due to the total lack of overhead, and the fact that the kernel has a global vision of all the processes running on the system, so the scheduling can be more efficient than it would be if two independent kernels were to schedule different task sets. Chief among the inconveniences is the impossibility to run a different kernel in a container (whether a different Linux version or a different operating system altogether).

NOTE
LXC isolation limits

LXC containers do not provide the level of isolation achieved by heavier emulators or virtualizers. In particular:

- since the kernel is shared among the host system and the containers, processes constrained to containers can still access the kernel messages, which can lead to information leaks if messages are emitted by a container;
- for similar reasons, if a container is compromised and a kernel vulnerability is exploited, the other containers may be affected too;
- on the filesystem, the kernel checks permissions according to the numerical identifiers for users and groups; these identifiers may designate different users and groups depending on the container, which should be kept in mind if writable parts of the filesystem are shared among containers.

Since we are dealing with isolation and not plain virtualization, setting up LXC containers is more complex than just running debian-installer on a virtual machine. We will describe a few prerequisites, then go on to the network configuration; we will then be able to actually create the system to be run in the container.

Preliminary Steps

The *lxc* package contains the tools required to run LXC, and must therefore be installed.

LXC also requires the *control groups* configuration system, which is a virtual filesystem to be mounted on `/sys/fs/cgroup`. Since Debian 8 switched to systemd, which also relies on control groups, this is now done automatically at boot time without further configuration.

Network Configuration

The goal of installing LXC is to set up virtual machines; while we could, of course, keep them isolated from the network, and only communicate with them via the filesystem, most use cases involve giving at least minimal network access to the containers. In the typical case, each container will get a virtual network interface, connected to the real network through a bridge. This virtual interface can be plugged either directly onto the host's physical network interface (in which case the container is directly on the network), or onto another virtual interface defined on the host (and the host can then filter or route traffic). In both cases, the *bridge-utils* package will be required.

The simple case is just a matter of editing `/etc/network/interfaces`, moving the configuration for the physical interface (for instance, eth0) to a bridge interface (usually br0), and configuring the link between them. For instance, if the network interface configuration file initially contains entries such as the following:

```
auto eth0
iface eth0 inet dhcp
```

They should be disabled and replaced with the following:

```
#auto eth0
#iface eth0 inet dhcp

auto br0
iface br0 inet dhcp
  bridge-ports eth0
```

The effect of this configuration will be similar to what would be obtained if the containers were machines plugged into the same physical network as the host. The "bridge" configuration manages the transit of Ethernet frames between all the bridged interfaces, which includes the physical eth0 as well as the interfaces defined for the containers.

In cases where this configuration cannot be used (for instance, if no public IP addresses can be assigned to the containers), a virtual *tap* interface will be created and connected to the bridge. The equivalent network topology then becomes that of a host with a second network card plugged into a separate switch, with the containers also plugged into that switch. The host must then act as a gateway for the containers if they are meant to communicate with the outside world.

In addition to *bridge-utils*, this "rich" configuration requires the *vde2* package; the `/etc/network/interfaces` file then becomes:

```
# Interface eth0 is unchanged
auto eth0
```

```
iface eth0 inet dhcp

# Virtual interface
auto tap0
iface tap0 inet manual
  vde2-switch -t tap0

# Bridge for containers
auto br0
iface br0 inet static
  bridge-ports tap0
  address 10.0.0.1
  netmask 255.255.255.0
```

The network can then be set up either statically in the containers, or dynamically with DHCP server running on the host. Such a DHCP server will need to be configured to answer queries on the br0 interface.

Setting Up the System

Let us now set up the filesystem to be used by the container. Since this "virtual machine" will not run directly on the hardware, some tweaks are required when compared to a standard filesystem, especially as far as the kernel, devices and consoles are concerned. Fortunately, the *lxc* includes scripts that mostly automate this configuration. For instance, the following commands (which require the *debootstrap* and *rsync* packages) will install a Debian container:

```
root@mirwiz:~# lxc-create -n testlxc -t debian
debootstrap is /usr/sbin/debootstrap
Checking cache download in /var/cache/lxc/debian/rootfs-stable-amd64 ...
Downloading debian minimal ...
I: Retrieving Release
I: Retrieving Release.gpg
[...]
Download complete.
Copying rootfs to /var/lib/lxc/testlxc/rootfs...
[...]
root@mirwiz:~#
```

Note that the filesystem is initially created in `/var/cache/lxc`, then moved to its destination directory. This allows creating identical containers much more quickly, since only copying is then required.

Note that the Debian template creation script accepts an `--arch` option to specify the architecture of the system to be installed and a `--release` option if you want to install something else than the current stable release of Debian. You can also set the MIRROR environment variable to point to a local Debian mirror.

The newly-created filesystem now contains a minimal Debian system, and by default the container has no network interface (besides the loopback one). Since this is not really wanted, we will edit the container's configuration file (`/var/lib/lxc/testlxc/config`) and add a few lxc.network.* entries:

```
lxc.net.0.type = veth
lxc.net.0.flags = up
lxc.net.0.link = br0
lxc.net.0.hwaddr = 4a:49:43:49:79:20
```

These entries mean, respectively, that a virtual interface will be created in the container; that it will automatically be brought up when said container is started; that it will automatically be connected to the br0 bridge on the host; and that its MAC address will be as specified. Should this last entry be missing or disabled, a random MAC address will be generated.

Another useful entry in that file is the setting of the hostname:

```
lxc.uts.name = testlxc
```

Starting the Container

Now that our virtual machine image is ready, let's start the container with `lxc-start --daemon --name=testlxc`.

In LXC releases following 2.0.8, root passwords are not set by default. We can set one running `lxc-attach -n testlxc` *`passwd`*. Now we can login:

```
root@mirwiz:~# lxc-console -n testlxc
Debian GNU/Linux 9 testlxc console

testlxc login: root
Password:
Linux testlxc 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5 (2019-06-19) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
root@testlxc:~# ps auxwf
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2  56736  6608 ?        Ss   09:28   0:00 /sbin/init
root        32  0.0  0.1  46096  4680 ?        Ss   09:28   0:00 /lib/systemd/systemd-journald
root        75  0.0  0.1  67068  3328 console  Ss   09:28   0:00 /bin/login --
root        82  0.0  0.1  19812  3664 console  S    09:30   0:00  \_ -bash
root        88  0.0  0.1  38308  3176 console  R+   09:31   0:00      \_ ps auxwf
root        76  0.0  0.1  69956  5636 ?        Ss   09:28   0:00 /usr/sbin/sshd -D
root@testlxc:~#
```

We are now in the container; our access to the processes is restricted to only those started from the container itself, and our access to the filesystem is similarly restricted to the dedicated subset of the full filesystem (`/var/lib/lxc/testlxc/rootfs`). We can exit the console with Control+a q.

Note that we ran the container as a background process, thanks to the `--daemon` option of `lxc-start`. We can interrupt the container with a command such as `lxc-stop --name=testlxc`.

The *lxc* package contains an initialization script that can automatically start one or several containers when the host boots (it relies on `lxc-autostart` which starts containers whose lxc.start.auto option is set to 1). Finer-grained control of the startup order is possible with lxc.start.order and lxc.group: by default, the initialization script first starts containers which are part of the onboot group and then the containers which are not part of any group. In both cases, the order within a group is defined by the lxc.start.order option.

GOING FURTHER
Mass virtualization

Since LXC is a very lightweight isolation system, it can be particularly adapted to massive hosting of virtual servers. The network configuration will probably be a bit more advanced than what we described above, but the “rich” configuration using `tap` and `veth` interfaces should be enough in many cases.

It may also make sense to share part of the filesystem, such as the `/usr` and `/lib` subtrees, so as to avoid duplicating the software that may need to be common to several containers. This will usually be achieved with `lxc.mount.entry` entries in the containers configuration file. An interesting side-effect is that the processes will then use less physical memory, since the kernel is able to detect that the programs are shared. The marginal cost of one extra container can then be reduced to the disk space dedicated to its specific data, and a few extra processes that the kernel must schedule and manage.

We haven’t described all the available options, of course; more comprehensive information can be obtained from the `lxc(7)` and `lxc.container.conf(5)` manual pages and the ones they reference.

12.2.3. Virtualization with KVM

KVM, which stands for *Kernel-based Virtual Machine*, is first and foremost a kernel module providing most of the infrastructure that can be used by a virtualizer, but it is not a virtualizer by itself. Actual control for the virtualization is handled by a QEMU-based application. Don’t worry if this section mentions `qemu-*` commands: it is still about KVM.

Unlike other virtualization systems, KVM was merged into the Linux kernel right from the start. Its developers chose to take advantage of the processor instruction sets dedicated to virtualization (Intel-VT and AMD-V), which keeps KVM lightweight, elegant and not resource-hungry. The counterpart, of course, is that KVM doesn’t work on any computer but only on those with appropriate processors. For x86-based computers, you can verify that you have such a processor by looking for “vmx” or “svm” in the CPU flags listed in `/proc/cpuinfo`.

With Red Hat actively supporting its development, KVM has more or less become the reference for Linux virtualization.

Preliminary Steps

Unlike such tools as VirtualBox, KVM itself doesn't include any user-interface for creating and managing virtual machines. The *qemu-kvm* package only provides an executable able to start a virtual machine, as well as an initialization script that loads the appropriate kernel modules.

Fortunately, Red Hat also provides another set of tools to address that problem, by developing the *libvirt* library and the associated *virtual machine manager* tools. libvirt allows managing virtual machines in a uniform way, independently of the virtualization system involved behind the scenes (it currently supports QEMU, KVM, Xen, LXC, OpenVZ, VirtualBox, VMWare and UML). `virtual-manager` is a graphical interface that uses libvirt to create and manage virtual machines.

We first install the required packages, with `apt-get install libvirt-clients libvirt-daemon-system qemu-kvm virtinst virt-manager virt-viewer`. *libvirt-daemon-system* provides the `libvirtd` daemon, which allows (potentially remote) management of the virtual machines running of the host, and starts the required VMs when the host boots. *libvirt-clients* provides the `virsh` command-line tool, which allows controlling the `libvirtd`-managed machines.

The *virtinst* package provides `virt-install`, which allows creating virtual machines from the command line. Finally, *virt-viewer* allows accessing a VM's graphical console.

Network Configuration

Just as in Xen and LXC, the most frequent network configuration involves a bridge grouping the network interfaces of the virtual machines (see section 12.2.2.2, "Network Configuration" page 357).

Alternatively, and in the default configuration provided by KVM, the virtual machine is assigned a private address (in the 192.168.122.0/24 range), and NAT is set up so that the VM can access the outside network.

The rest of this section assumes that the host has an eth0 physical interface and a br0 bridge, and that the former is connected to the latter.

Installation with `virt-install`

Creating a virtual machine is very similar to installing a normal system, except that the virtual machine's characteristics are described in a seemingly endless command line.

Practically speaking, this means we will use the Debian installer, by booting the virtual machine on a virtual DVD-ROM drive that maps to a Debian DVD image stored on the host system. The VM will export its graphical console over the VNC protocol (see section 9.2.2, "Using Remote Graphical Desktops" page 212 for details), which will allow us to control the installation process.

We first need to tell libvirtd where to store the disk images, unless the default location (`/var/lib/libvirt/images/`) is fine.

```
root@mirwiz:~# mkdir /srv/kvm
root@mirwiz:~# virsh pool-create-as srv-kvm dir --target /srv/kvm
Pool srv-kvm created

root@mirwiz:~#
```

TIP

Add your user to the libvirt group

All samples in this section assume that you are running commands as root. Effectively, if you want to control a local libvirt daemon, you need either to be root or to be a member of the `libvirt` group (which is not the case by default). Thus if you want to avoid using root rights too often, you can add yourself to the `libvirt` group and run the various commands under your user identity.

Let us now start the installation process for the virtual machine, and have a closer look at `virt-install`'s most important options. This command registers the virtual machine and its parameters in libvirtd, then starts it so that its installation can proceed.

```
# virt-install --connect qemu:///system           1
               --virt-type kvm                    2
               --name testkvm                     3
               --memory 1024                      4
               --disk /srv/kvm/testkvm.qcow,format=qcow2,size=10    5
               --cdrom /srv/isos/debian-10.2.0-amd64-netinst.iso    6
               --network bridge=virbr0            7
               --graphics vnc                     8
               --os-type linux                    9
               --os-variant debian10

Starting install...
Allocating 'testkvm.qcow'             |  10 GB     00:00
```

The --connect option specifies the "hypervisor" to use. Its form is that of an URL containing a virtualization system (xen://, qemu://, lxc://, openvz://, vbox://, and so on) and the machine that should host the VM (this can be left empty in the case of the local host). In addition to that, and in the QEMU/KVM case, each user can manage virtual machines working with restricted permissions, and the URL path allows differentiating "system" machines (/system) from others (/session).

Since KVM is managed the same way as QEMU, the --virt-type kvm allows specifying the use of KVM even though the URL looks like QEMU.

The --name option defines a (unique) name for the virtual machine.

The --memory option allows specifying the amount of RAM (in MB) to allocate for the virtual machine.

The --disk specifies the location of the image file that is to represent our virtual machine's hard disk; that file is created, unless present, with a size (in GB) specified by the size parameter. The format parameter allows choosing among several ways of storing the image file. The default format (qcow2) allows starting with a small file that only grows when the virtual machine starts actually using space.

The --cdrom option is used to indicate where to find the optical disk to use for installation. The path can be either a local path for an ISO file, an URL where the file can be obtained, or the device file of a physical CD-ROM drive (i.e. /dev/cdrom).

The --network specifies how the virtual network card integrates in the host's network configuration. The default behavior (which we explicitly forced in our example) is to integrate it into any pre-existing network bridge. If no such bridge exists, the virtual machine will only reach the physical network through NAT, so it gets an address in a private subnet range (192.168.122.0/24).

--graphics vnc states that the graphical console should be made available using VNC. The default behavior for the associated VNC server is to only listen on the local interface; if the VNC client is to be run on a different host, establishing the connection will require setting up an SSH tunnel (see section 9.2.1.3, "Creating Encrypted Tunnels with Port Forwarding" page 210). Alternatively, --graphics vnc,listen=0.0.0.0 can be used so that the VNC server is accessible from all interfaces; note that if you do that, you really should design your firewall accordingly.

The --os-type and --os-variant options allow optimizing a few parameters of the virtual machine, based on some of the known features of the operating system mentioned there.

At this point, the virtual machine is running, and we need to connect to the graphical console to proceed with the installation process. If the previous operation was run from a graphical desktop environment, this connection should be automatically started. If not, or if we operate remotely, `virt-viewer` can be run from any graphical environment to open the graphical console (note that the root password of the remote host is asked twice because the operation requires 2 SSH connections):

```
$ virt-viewer --connect qemu+ssh://root@server/system testkvm
root@server's password:
root@server's password:
```

When the installation process ends, the virtual machine is restarted, now ready for use.

Managing Machines with `virsh`

Now that the installation is done, let us see how to handle the available virtual machines. The first thing to try is to ask `libvirtd` for the list of the virtual machines it manages:

```
# virsh -c qemu:///system list --all
 Id Name                 State
----------------------------------
  8 testkvm              shut off
```

Let’s start our test virtual machine:

```
# virsh -c qemu:///system start testkvm
Domain testkvm started
```

We can now get the connection instructions for the graphical console (the returned VNC display can be given as parameter to `vncviewer`):

```
# virsh -c qemu:///system vncdisplay testkvm
127.0.0.1:0
```

Other available `virsh` subcommands include:

- reboot to restart a virtual machine;
- shutdown to trigger a clean shutdown;
- destroy, to stop it brutally;
- suspend to pause it;
- resume to unpause it;
- autostart to enable (or disable, with the --disable option) starting the virtual machine automatically when the host starts;
- undefine to remove all traces of the virtual machine from `libvirtd`.

All these subcommands take a virtual machine identifier as a parameter.

Installing an RPM based system in Debian with yum

If the virtual machine is meant to run a Debian (or one of its derivatives), the system can be initialized with `debootstrap`, as described above. But if the virtual machine is to be installed with an RPM-based system (such as Fedora, CentOS or Scientific Linux), the setup will need to be done using the `yum` utility (available in the package of the same name).

The procedure requires using `rpm` to extract an initial set of files, including notably `yum` configuration files, and then calling `yum` to extract the remaining set of packages. But since we call `yum` from outside the chroot, we need to make some temporary changes. In the sample below, the target chroot is `/srv/centos`.

```
# rootdir=”/srv/centos”
# mkdir -p ”$rootdir” /etc/rpm
# echo ”%_dbpath /var/lib/rpm” > /etc/rpm/macros.dbpath
# wget http://mirror.centos.org/centos/7/os/x86_64/Packages/centos-release
       -7-6.1810.2.el7.centos.x86_64.rpm
# rpm --nodeps --root ”$rootdir” -i centos-release-7-6.1810.2.el7.centos.x86_64.rpm
rpm: RPM should not be used directly install RPM packages, use Alien instead!
rpm: However assuming you know what you are doing...
warning: centos-release-7-6.1810.2.el7.centos.x86_64.rpm: Header V3 RSA/SHA256
       Signature, key ID f4a80eb5: NOKEY
# sed -i -e ”s,gpgkey=file:///etc/,gpgkey=file://${rootdir}/etc/,g” $rootdir/etc/yum.
       repos.d/*.repo
# yum --assumeyes --installroot $rootdir groupinstall core
[...]
# sed -i -e ”s,gpgkey=file://${rootdir}/etc/,gpgkey=file:///etc/,g” $rootdir/etc/yum.
       repos.d/*.repo
```

12.3. Automated Installation

The Falcot Corp administrators, like many administrators of large IT services, need tools to install (or reinstall) quickly, and automatically if possible, their new machines.

These requirements can be met by a wide range of solutions. On the one hand, generic tools such as SystemImager handle this by creating an image based on a template machine, then deploy that image to the target systems; at the other end of the spectrum, the standard Debian installer can be preseeded with a configuration file giving the answers to the questions asked during the installation process. As a sort of middle ground, a hybrid tool such as FAI (*Fully Automatic Installer*) installs machines using the packaging system, but it also uses its own infrastructure for tasks that are more specific to massive deployments (such as starting, partitioning, configuration and so on).

Each of these solutions has its pros and cons: SystemImager works independently from any particular packaging system, which allows it to manage large sets of machines using several distinct Linux distributions. It also includes an update system that doesn't require a reinstallation, but this update system can only be reliable if the machines are not modified independently; in other words, the user must not update any software on their own, or install any other software. Similarly, security updates must not be automated, because they have to go through the centralized reference image maintained by SystemImager. This solution also requires the target machines to be homogeneous, otherwise many different images would have to be kept and managed (an i386 image won't fit on a powerpc machine, and so on).

On the other hand, an automated installation using debian-installer can adapt to the specifics of each machine: the installer will fetch the appropriate kernel and software packages from the relevant repositories, detect available hardware, partition the whole hard disk to take advantage of all the available space, install the corresponding Debian system, and set up an appropriate bootloader. However, the standard installer will only install standard Debian versions, with the base system and a set of pre-selected “tasks”; this precludes installing a particular system with non-packaged applications. Fulfilling this particular need requires customizing the installer…

Fortunately, the installer is very modular, and there are tools to automate most of the work required for this customization, most importantly simple-CDD (CDD being an acronym for *Custom Debian Derivative*). Even the simple-CDD solution, however, only handles initial installations; this is usually not a problem since the APT tools allow efficient deployment of updates later on.

We will only give a rough overview of FAI, and skip SystemImager altogether (which is no longer in Debian), in order to focus more intently on debian-installer and simple-CDD, which are more interesting in a Debian-only context.

12.3.1. Fully Automatic Installer (FAI)

Fully Automatic Installer is probably the oldest automated deployment system for Debian, which explains its status as a reference; but its very flexible nature only just compensates for the complexity it involves.

FAI requires a server system to store deployment information and allow target machines to boot from the network. This server requires the *fai-server* package (or *fai-quickstart*, which also brings the required elements for a standard configuration).

FAI uses a specific approach for defining the various installable profiles. Instead of simply duplicating a reference installation, FAI is a full-fledged installer, fully configurable via a set of files and scripts stored on the server; the default location `/srv/fai/config/` is not automatically created, so the administrator needs to create it along with the relevant files. Most of the times, these files will be customized from the example files available in the documentation for the *fai-doc* package, more particularly the `/usr/share/doc/fai-doc/examples/simple/` directory.

Once the profiles are defined, the `fai-setup` command generates the elements required to start a FAI installation; this mostly means preparing or updating a minimal system (NFS-root) used during installation. An alternative is to generate a dedicated boot CD with `fai-cd`.

Creating all these configuration files requires some understanding of the way FAI works. A typical installation process is made of the following steps:

- fetching a kernel from the network, and booting it;
- mounting the root filesystem from NFS;
- executing `/usr/sbin/fai`, which controls the rest of the process (the next steps are therefore initiated by this script);
- copying the configuration space from the server into `/fai/`;
- running `fai-class`. The `/fai/class/[0-9][0-9]*` scripts are executed in turn, and return names of “classes” that apply to the machine being installed; this information will serve as a base for the following steps. This allows for some flexibility in defining the services to be installed and configured.
- fetching a number of configuration variables, depending on the relevant classes;
- partitioning the disks and formatting the partitions, based on information provided in `/fai/disk_config/`*`class`*;

- mounting said partitions;
- installing the base system;
- preseeding the Debconf database with `fai-debconf`;
- fetching the list of available packages for APT;
- installing the packages listed in `/fai/package_config/`*`class`*;
- executing the post-configuration scripts, `/fai/scripts/`*`class`*`/[0-9][0-9]*`;
- recording the installation logs, unmounting the partitions, and rebooting.

12.3.2. Preseeding Debian-Installer

At the end of the day, the best tool to install Debian systems should logically be the official Debian installer. This is why, right from its inception, debian-installer has been designed for automated use, taking advantage of the infrastructure provided by *debconf*. The latter allows, on the one hand, to reduce the number of questions asked (hidden questions will use the provided default answer), and on the other hand, to provide the default answers separately, so that installation can be non-interactive. This last feature is known as *preseeding*.

GOING FURTHER

Debconf with a centralized database

Preseeding allows to provide a set of answers to Debconf questions at installation time, but these answers are static and do not evolve as time passes. Since already-installed machines may need upgrading, and new answers may become required, the `/etc/debconf.conf` configuration file can be set up so that Debconf uses external data sources (such as an LDAP directory server, or a remote file accessed via NFS or Samba). Several external data sources can be defined at the same time, and they complement one another. The local database is still used (for read-write access), but the remote databases are usually restricted to reading. The `debconf.conf(5)` manual page describes all the possibilities in detail (you need the *debconf-doc* package).

Using a Preseed File

There are several places where the installer can get a preseeding file:

- in the initrd used to start the machine; in this case, preseeding happens at the very beginning of the installation, and all questions can be avoided. The file just needs to be called `preseed.cfg` and stored in the initrd root.
- on the boot media (CD or USB key); preseeding then happens as soon as the media is mounted, which means right after the questions about language and keyboard layout. The preseed/file boot parameter can be used to indicate the location of the preseeding file (for instance, `/cdrom/preseed.cfg` when the installation is done off a CD-ROM, or `/hd-media/preseed.cfg` in the USB-key case).
- from the network; preseeding then only happens after the network is (automatically) configured; the relevant boot parameter is then preseed/url=http://*server*/preseed.cfg.

At a glance, including the preseeding file in the initrd looks like the most interesting solution; however, it is rarely used in practice, because generating an installer initrd is rather complex. The other two solutions are much more common, especially since boot parameters provide another way to preseed the answers to the first questions of the installation process. The usual way to save the bother of typing these boot parameters by hand at each installation is to save them into the configuration for `isolinux` (in the CD-ROM case) or `syslinux` (USB key).

Creating a Preseed File

A preseed file is a plain text file, where each line contains the answer to one Debconf question. A line is split across four fields separated by whitespace (spaces or tabs), as in, for instance, d-i mirror/suite string stable:

- the first field is the “owner” of the question; “d-i” is used for questions relevant to the installer, but it can also be a package name for questions coming from Debian packages;
- the second field is an identifier for the question;
- third, the type of question;
- the fourth and last field contains the value for the answer. Note that it must be separated from the third field with a single space; if there are more than one, the following space characters are considered part of the value.

The simplest way to write a preseed file is to install a system by hand. Then `debconf-get-selections --installer` will provide the answers concerning the installer. Answers about other packages can be obtained with `debconf-get-selections`. However, a cleaner solution is to write the preseed file by hand, starting from an example and the reference documentation: with such an approach, only questions where the default answer needs to be overridden can be preseeded; using the priority=critical boot parameter will instruct Debconf to only ask critical questions, and use the default answer for others.

DOCUMENTATION
Installation guide appendix

The installation guide, available online, includes detailed documentation on the use of a preseed file in an appendix. It also includes a detailed and commented sample file, which can serve as a base for local customizations.

➨ `https://www.debian.org/releases/stable/amd64/apb`

➨ `https://www.debian.org/releases/stable/example-preseed.txt`

Creating a Customized Boot Media

Knowing where to store the preseed file is all very well, but the location isn’t everything: one must, one way or another, alter the installation boot media to change the boot parameters and add the preseed file.

Booting From the Network When a computer is booted from the network, the server sending the initialization elements also defines the boot parameters. Thus, the change needs to be made in the PXE configuration for the boot server; more specifically, in its `/tftpboot/pxelinux.cfg/default` configuration file. Setting up network boot is a prerequisite; see the Installation Guide for details.

➨ https://www.debian.org/releases/stable/amd64/ch04s05

Preparing a Bootable USB Key Once a bootable key has been prepared (see section 4.1.2, "Booting from a USB Key" page 53), a few extra operations are needed. Assuming the key contents are available under `/media/usbdisk/`:

- copy the preseed file to `/media/usbdisk/preseed.cfg`
- edit `/media/usbdisk/syslinux.cfg` and add required boot parameters (see example below).

Example 12.2 *syslinux.cfg file and preseeding parameters*

```
default vmlinuz
append preseed/file=/hd-media/preseed.cfg locale=en_US.UTF-8 keymap=us language=us
       country=US vga=788 initrd=initrd.gz  --
```

Creating a CD-ROM Image A USB key is a read-write media, so it was easy for us to add a file there and change a few parameters. In the CD-ROM case, the operation is more complex, since we need to regenerate a full ISO image. This task is handled by *debian-cd*, but this tool is rather awkward to use: it needs a local mirror, and it requires an understanding of all the options provided by `/usr/share/debian-cd/CONF.sh`; even then, `make` must be invoked several times. `/usr/share/debian-cd/README` is therefore a very recommended read.

Having said that, debian-cd always operates in a similar way: an "image" directory with the exact contents of the CD-ROM is generated, then converted to an ISO file with a tool such as `genisoimage`, `mkisofs` or `xorriso`. The image directory is finalized after debian-cd's `make image-trees` step. At that point, we insert the preseed file into the appropriate directory (usually `$TDIR/$CODENAME/CD1/`, $TDIR and $CODENAME being parameters defined by the `CONF.sh` configuration file). The CD-ROM uses `isolinux` as its bootloader, and its configuration file must be adapted from what debian-cd generated, in order to insert the required boot parameters (the specific file is `$TDIR/$CODENAME/boot1/isolinux/isolinux.cfg`). Then the "normal" process can be resumed, and we can go on to generating the ISO image with `make image CD=1` (or `make images` if several CD-ROMs are generated).

12.3.3. Simple-CDD: The All-In-One Solution

Simply using a preseed file is not enough to fulfill all the requirements that may appear for large deployments. Even though it is possible to execute a few scripts at the end of the normal installation process, the selection of the set of packages to install is still not quite flexible (basically, only "tasks" can be selected); more important, this only allows installing official Debian packages, and precludes locally-generated ones.

On the other hand, debian-cd is able to integrate external packages, and debian-installer can be extended by inserting new steps in the installation process. By combining these capabilities, it should be possible to create a customized installer that fulfills our needs; it should even be able to configure some services after unpacking the required packages. Fortunately, this is not a mere hypothesis, since this is exactly what Simple-CDD (in the *simple-cdd* package) does.

The purpose of Simple-CDD is to allow anyone to easily create a distribution derived from Debian, by selecting a subset of the available packages, preconfiguring them with Debconf, adding specific software, and executing custom scripts at the end of the installation process. This matches the "universal operating system" philosophy, since anyone can adapt it to their own needs.

Creating Profiles

Simple-CDD defines "profiles" that match the FAI "classes" concept, and a machine can have several profiles (determined at installation time). A profile is defined by a set of `profiles/`*`profile`*`.*` files:

- the `.description` file contains a one-line description for the profile;
- the `.packages` file lists packages that will automatically be installed if the profile is selected;
- the `.downloads` file lists packages that will be stored onto the installation media, but not necessarily installed;
- the `.preseed` file contains preseeding information for Debconf questions (for the installer and/or for packages);
- the `.postinst` file contains a script that will be run at the end of the installation process;
- lastly, the `.conf` file allows changing some Simple-CDD parameters based on the profiles to be included in an image.

The default profile has a particular role, since it is always selected; it contains the bare minimum required for Simple-CDD to work. The only thing that is usually customized in this profile is the simple-cdd/profiles preseed parameter: this allows avoiding the question, introduced by Simple-CDD, about what profiles to install.

Note also that the commands will need to be invoked from the parent directory of the `profiles` directory.

Configuring and Using `build-simple-cdd`

QUICK LOOK
Detailed configuration file

An example of a Simple-CDD configuration file, with all possible parameters, is included in the package (`/usr/share/doc/simple-cdd/examples/simple-cdd.conf.detailed.gz`). This can be used as a starting point when creating a custom configuration file.

Simple-CDD requires many parameters to operate fully. They will most often be gathered in a configuration file, which `build-simple-cdd` can be pointed at with the --conf option, but they can also be specified via dedicated parameters given to `build-simple-cdd`. Here is an overview of how this command behaves, and how its parameters are used:

- the profiles parameter lists the profiles that will be included on the generated CD-ROM image;
- based on the list of required packages, Simple-CDD downloads the appropriate files from the server mentioned in server, and gathers them into a partial mirror (which will later be given to debian-cd);
- the custom packages mentioned in local_packages are also integrated into this local mirror;
- debian-cd is then executed (within a default location that can be configured with the debian_cd_dir variable), with the list of packages to integrate;
- once debian-cd has prepared its directory, Simple-CDD applies some changes to this directory:
  - files containing the profiles are added in a `simple-cdd` subdirectory (that will end up on the CD-ROM);
  - other files listed in the all_extras parameter are also added;
  - the boot parameters are adjusted so as to enable the preseeding. Questions concerning language and country can be avoided if the required information is stored in the language and country variables.
- debian-cd then generates the final ISO image.

Generating an ISO Image

Once we have written a configuration file and defined our profiles, the remaining step is to invoke `build-simple-cdd --conf simple-cdd.conf`. After a few minutes, we get the required image in `images/debian-10-amd64-CD-1.iso`.

12.4. Monitoring

Monitoring is a generic term, and the various involved activities have several goals: on the one hand, following usage of the resources provided by a machine allows anticipating saturation and the subsequent required upgrades; on the other hand, alerting the administrator as soon as a service is unavailable or not working properly means that the problems that do happen can be fixed sooner.

Munin covers the first area, by displaying graphical charts for historical values of a number of parameters (used RAM, occupied disk space, processor load, network traffic, Apache/MySQL load, and so on). *Nagios* covers the second area, by regularly checking that the services are working and available, and sending alerts through the appropriate channels (e-mails, text messages, and so on). Both have a modular design, which makes it easy to create new plug-ins to monitor specific parameters or services.

ALTERNATIVE
Zabbix, an integrated monitoring tool

Although Munin and Nagios are in very common use, they are not the only players in the monitoring field, and each of them only handles half of the task (graphing on one side, alerting on the other). Zabbix, on the other hand, integrates both parts of monitoring; it also has a web interface for configuring the most common aspects. It has grown by leaps and bounds during the last few years, and can now be considered a viable contender. On the monitoring server, you would install *zabbix-server-pgsql* (or *zabbix-server-mysql*), possibly together with *zabbix-frontend-php* to have a web interface. On the hosts to monitor you would install *zabbix-agent* feeding data back to the server.

➨ `https://www.zabbix.com/`

ALTERNATIVE
Icinga, a Nagios fork

Spurred by divergences in opinions concerning the development model for Nagios (which is controlled by a company), a number of developers forked Nagios and use Icinga as their new name. Icinga is still compatible — so far — with Nagios configurations and plugins, but it also adds extra features.

➨ `https://www.icinga.org/`

12.4.1. Setting Up Munin

The purpose of Munin is to monitor many machines; therefore, it quite naturally uses a client/server architecture. The central host — the grapher — collects data from all the monitored hosts, and generates historical graphs.

Configuring Hosts To Monitor

The first step is to install the *munin-node* package. The daemon installed by this package listens on port 4949 and sends back the data collected by all the active plugins. Each plugin is a simple program returning a description of the collected data as well as the latest measured value. Plugins are stored in `/usr/share/munin/plugins/`, but only those with a symbolic link in `/etc/munin/plugins/` are really used.

When the package is installed, a set of active plugins is determined based on the available software and the current configuration of the host. However, this autoconfiguration depends on a feature that each plugin must provide, and it is usually a good idea to review and tweak the results by hand. Browsing the Plugin Gallery[2] can be interesting even though not all plugins have comprehensive documentation. However, all plugins are scripts and most are rather simple and well-commented. Browsing `/etc/munin/plugins/` is therefore a good way of getting an idea of what each plugin is about and determining which should be removed. Similarly, enabling an interesting plugin found in `/usr/share/munin/plugins/` is a simple matter of setting up a symbolic link with `ln -sf /usr/share/munin/plugins/`*`plugin`*` /etc/munin/plugins/`. Note that when a plugin name ends with an underscore "_", the plugin requires a parameter. This parameter must be stored in the name of the symbolic link; for instance, the "if_" plugin must be enabled with a `if_eth0` symbolic link, and it will monitor network traffic on the eth0 interface.

Once all plugins are correctly set up, the daemon configuration must be updated to describe access control for the collected data. This involves allow directives in the `/etc/munin/munin-node.conf` file. The default configuration is allow ^127\.0\.0\.1$, and only allows access to the local host. An administrator will usually add a similar line containing the IP address of the grapher host, then restart the daemon with `systemctl restart munin-node`.

GOING FURTHER

Creating local plugins

Munin does include detailed documentation on how plugins should behave, and how to develop new plugins.

➨ http://guide.munin-monitoring.org/en/latest/plugin/writing.html

A plugin is best tested when run in the same conditions as it would be when triggered by munin-node; this can be simulated by running `munin-run `*`plugin`* as root. A potential second parameter given to this command (such as `config`) is passed to the plugin as a parameter.

When a plugin is invoked with the `config` parameter, it must describe itself by returning a set of fields:

```
$ sudo munin-run load config
graph_title Load average
graph_args --base 1000 -l 0
graph_vlabel load
graph_scale no
graph_category system
load.label load
graph_info The load average of the machine describes how
       many processes are in the run-queue (scheduled to run
        ”immediately”).
load.info 5 minute load average
```

The various available fields are described by the "Plugin reference" available as part of the "Munin guide".

➨ https://munin.readthedocs.org/en/latest/reference/plugin.html

[2] ➨ http://gallery.munin-monitoring.org

When invoked without a parameter, the plugin simply returns the last measured values; for instance, executing `sudo munin-run load` could return `load.value 0.12`.

Finally, when a plugin is invoked with the `autoconf` parameter, it should return “yes” (and a 0 exit status) or “no” (with a 1 exit status) according to whether the plugin should be enabled on this host.

Configuring the Grapher

The “grapher” is simply the computer that aggregates the data and generates the corresponding graphs. The required software is in the *munin* package. The standard configuration runs `munin-cron` (once every 5 minutes), which gathers data from all the hosts listed in `/etc/munin/munin.conf` (only the local host is listed by default), saves the historical data in RRD files (*Round Robin Database*, a file format designed to store data varying in time) stored under `/var/lib/munin/` and generates an HTML page with the graphs in `/var/cache/munin/www/`.

All monitored machines must therefore be listed in the `/etc/munin/munin.conf` configuration file. Each machine is listed as a full section with a name matching the machine and at least an address entry giving the corresponding IP address.

```
[ftp.falcot.com]
    address 192.168.0.12
    use_node_name yes
```

Sections can be more complex, and describe extra graphs that could be created by combining data coming from several machines. The samples provided in the configuration file are good starting points for customization.

The last step is to publish the generated pages; this involves configuring a web server so that the contents of `/var/cache/munin/www/` are made available on a website. Access to this website will often be restricted, using either an authentication mechanism or IP-based access control. See section 11.2, “Web Server (HTTP)” page 293 for the relevant details.

12.4.2. Setting Up Nagios

Unlike Munin, Nagios does not necessarily require installing anything on the monitored hosts; most of the time, Nagios is used to check the availability of network services. For instance, Nagios can connect to a web server and check that a given web page can be obtained within a given time.

Installing

The first step in setting up Nagios is to install the *nagios4* and *monitoring-plugins* packages. Installing the packages configures the web interface and the Apache server. The authz_groupfile and auth_digest Apache modules must be enabled, for that execute:

```
# a2enmod authz_groupfile
Considering dependency authz_core for authz_groupfile:
Module authz_core already enabled
Enabling module authz_groupfile.
To activate the new configuration, you need to run:
  systemctl restart apache2
# a2enmod auth_digest
Considering dependency authn_core for auth_digest:
Module authn_core already enabled
Enabling module auth_digest.
To activate the new configuration, you need to run:
  systemctl restart apache2
# systemctl restart apache2
```

Adding other users is a simple matter of inserting them in the `/etc/nagios4/hdigest.users` file.

Pointing a browser at http://*server*/nagios4/ displays the web interface; in particular, note that Nagios already monitors some parameters of the machine where it runs. However, some interactive features such as adding comments to a host do not work. These features are disabled in the default configuration for Nagios, which is very restrictive for security reasons.

Enabling some features involves editing `/etc/nagios4/nagios.cfg`. We also need to set up write permissions for the directory used by Nagios, with commands such as the following:

```
# systemctl stop nagios4
# dpkg-statoverride --update --add nagios www-data 2710 /var/lib/nagios4/rw
# dpkg-statoverride --update --add nagios nagios 751 /var/lib/nagios4
# systemctl start nagios4
```

Configuring

The Nagios web interface is rather nice, but it does not allow configuration, nor can it be used to add monitored hosts and services. The whole configuration is managed via files referenced in the central configuration file, `/etc/nagios4/nagios.cfg`.

These files should not be dived into without some understanding of the Nagios concepts. The configuration lists objects of the following types:

- a *host* is a machine to be monitored;
- a *hostgroup* is a set of hosts that should be grouped together for display, or to factor some common configuration elements;

- a *service* is a testable element related to a host or a host group. It will most often be a check for a network service, but it can also involve checking that some parameters are within an acceptable range (for instance, free disk space or processor load);
- a *servicegroup* is a set of services that should be grouped together for display;
- a *contact* is a person who can receive alerts;
- a *contactgroup* is a set of such contacts;
- a *timeperiod* is a range of time during which some services have to be checked;
- a *command* is the command line invoked to check a given service.

According to its type, each object has a number of properties that can be customized. A full list would be too long to include, but the most important properties are the relations between the objects.

A *service* uses a *command* to check the state of a feature on a *host* (or a *hostgroup*) within a *timeperiod*. In case of a problem, Nagios sends an alert to all members of the *contactgroup* linked to the service. Each member is sent the alert according to the channel described in the matching *contact* object.

An inheritance system allows easy sharing of a set of properties across many objects without duplicating information. Moreover, the initial configuration includes a number of standard objects; in many cases, defining new hosts, services and contacts is a simple matter of deriving from the provided generic objects. The files in `/etc/nagios4/conf.d/` are a good source of information on how they work.

The Falcot Corp administrators use the following configuration:

Example 12.3 *`/etc/nagios4/conf.d/falcot.cfg` file*

```
define contact{
    name                            generic-contact
    service_notification_period     24x7
    host_notification_period        24x7
    service_notification_options    w,u,c,r
    host_notification_options       d,u,r
    service_notification_commands   notify-service-by-email
    host_notification_commands      notify-host-by-email
    register                        0 ; Template only
}
define contact{
    use             generic-contact
    contact_name    rhertzog
    alias           Raphael Hertzog
    email           hertzog@debian.org
}
define contact{
    use             generic-contact
    contact_name    rmas
    alias           Roland Mas
    email           lolando@debian.org
}

define contactgroup{
    contactgroup_name     falcot-admins
    alias                 Falcot Administrators
    members               rhertzog,rmas
}

define host{
    use                   generic-host ; Name of host template to use
    host_name             www-host
    alias                 www.falcot.com
    address               192.168.0.5
    contact_groups        falcot-admins
    hostgroups            debian-servers,ssh-servers
}
define host{
    use                   generic-host ; Name of host template to use
    host_name             ftp-host
    alias                 ftp.falcot.com
    address               192.168.0.6
    contact_groups        falcot-admins
    hostgroups            debian-servers,ssh-servers
}

# ’check_ftp’ command with custom parameters
define command{
    command_name          check_ftp2
    command_line          /usr/lib/nagios/plugins/check_ftp -H $HOSTADDRESS$ -w 20 -c
            30 -t 35
}

# Generic Falcot service
define service{
    name                  falcot-service
    use                   generic-service
    contact_groups        falcot-admins
    register              0
}

# Services to check on www-host
define service{
    use                   falcot-service
    host_name             www-host
    service_description   HTTP
    check_command         check_http
```

```
}
define service{
    use                     falcot-service
    host_name               www-host
    service_description     HTTPS
    check_command           check_https
}
define service{
    use                     falcot-service
    host_name               www-host
    service_description     SMTP
    check_command           check_smtp
}

# Services to check on ftp-host
define service{
    use                     falcot-service
    host_name               ftp-host
    service_description     FTP
    check_command           check_ftp2
}
```

This configuration file describes two monitored hosts. The first one is the web server, and the checks are made on the HTTP (80) and secure-HTTP (443) ports. Nagios also checks that an SMTP server runs on port 25. The second host is the FTP server, and the check includes making sure that a reply comes within 20 seconds. Beyond this delay, a *warning* is emitted; beyond 30 seconds, the alert is deemed critical. The Nagios web interface also shows that the SSH service is monitored: this comes from the hosts belonging to the ssh-servers hostgroup. The matching standard service is defined in `/etc/nagios4/conf.d/services_nagios2.cfg`.

Note the use of inheritance: an object is made to inherit from another object with the "use *parent-name*". The parent object must be identifiable, which requires giving it a "name *identifier*" property. If the parent object is not meant to be a real object, but only to serve as a parent, giving it a "register 0" property tells Nagios not to consider it, and therefore to ignore the lack of some parameters that would otherwise be required.

DOCUMENTATION

List of object properties

A more in-depth understanding of the various ways in which Nagios can be configured can be obtained from the documentation hosted on https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/index.html. It includes a list of all object types, with all the properties they can have. It also explains how to create new plugins.

GOING FURTHER

Remote tests with NRPE

Many Nagios plugins allow checking some parameters local to a host; if many machines need these checks while a central installation gathers them, the NRPE (*Nagios Remote Plugin Executor*) plugin needs to be deployed. The *nagios-nrpe-plugin* package needs to be installed on the Nagios server, and *nagios-nrpe-server* on the hosts where local tests need to run. The latter gets its configuration from

`/etc/nagios/nrpe.cfg`. This file should list the tests that can be started remotely, and the IP addresses of the machines allowed to trigger them. On the Nagios side, enabling these remote tests is a simple matter of adding matching services using the new *check_nrpe* command.

Keywords

Workstation
Graphical desktop
Office work
X.org