CHAPTER 4: DISK GROUPS

by Volker Herminghaus

4.1 Overview

In the previous chapter we discussed how disks or LUNs get incorporated into VxVM. That was admittedly a complicated chapter. From here on it is relatively straightforward, at least until you reach the technical deep dive section. Veritas had some very bright developers when they designed VxVM, and it shows. Some of the concepts are not easy to grasp because of the genius behind them. But once you understand them and get the hang of them, you will never want anything else. Guaranteed.

4.1.1 What is a Disk Group?

How would a single disk be useful in the RAID context? If we used individual disks, we would still be limited to a certain limited and fixed size, to a certain limited and fixed number of IOPS, to zero redundancy and so on. It does not make any sense to work with individual disks if you are running a volume management software. The least you need is a number of disks working together to overcome the physical limitations of individual disks. Preferably working together in such a way that control over the whole "disk group" (as we will call it from here on) can be easily transferred from one host to another in order to enable easy and reliable failover for cluster configurations. This is **exactly** what Veritas implemented with their VxVM disk groups, no less, no more.

To put it quite shortly: A DG is to a SAN what a partition is to a disk: it splices the available storage into separate items for independent use. When laying out disk groups care must be taken to balance the amount of independence against space constraints. As an analogy: you could layout your root disk with just one file system and keep all of **/var**, **/usr**, **/home** and **/opt** in the root (/) file system. The advantage is that very little space is lost to boundary effects: All directories share the same free disk blocks. Whereas if root (/), **/var**, **/usr**, **/home** and **/opt** are all separate file systems it would be possible for an application that is creating a lot of e.g. log data to let the **/var** file system run out of space while **/usr** still has a lot of free space left. What a waste! On the other hand, if all of the root disk was in a single file system, then the application that used to fill up just the **/var** file system may eventually fill up the whole, unified file system, eventually rendering the system unusable.

The same principle applies to DGs: You place a few disks into a DG destined for a database, a few others into a DG for a web server, and again others are used in a separate DG for your mirrored boot disks. How many DG should you create? This question is easily answered once you think of what kind of software VxVM is. Its purpose is to serve as a basis for high-availability cluster environments. An HA cluster uses multiple servers connected together and attached to shared storage to run a service on one server. In case of failure, another server automatically takes control and starts running the service. In order to do so it first has to take control over the storage, which it does by reserving the set of disks that are required in order to run the service. That set of disks is what is called a disk group in VxVM terminology. In the case outlined above, the cluster software would have one service for the database and another, independent service for the web server. Each service stores its data on the volumes contained in one disk group. When the host running the web server fails, the other server takes control of the web server DG by a process called **importing** the DG

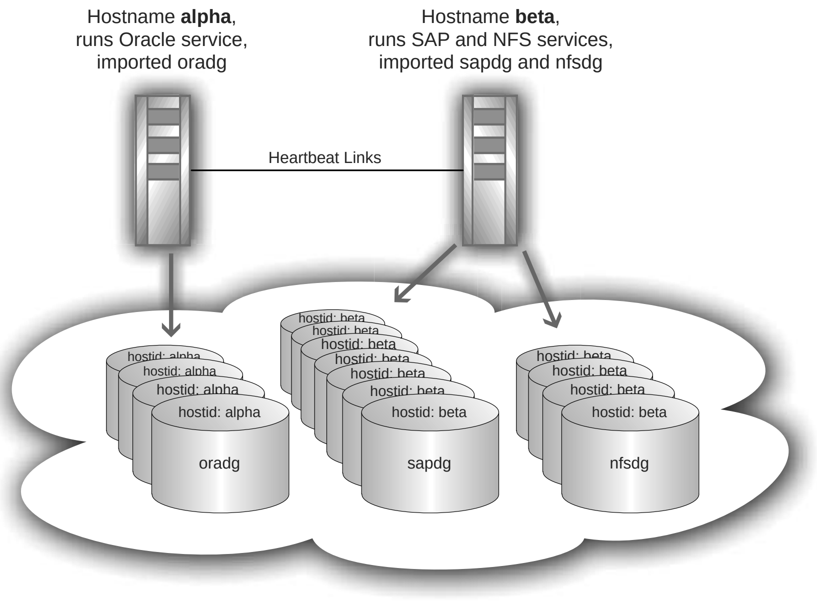
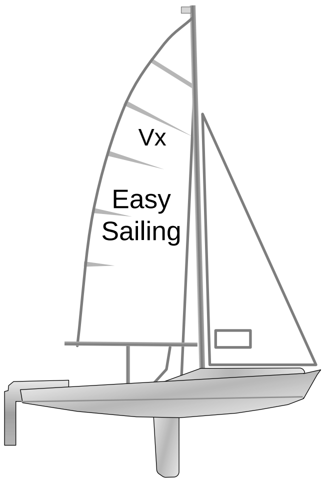


Figure 4-1: Disk groups are designed with high-availability clusters in mind. A disk group contains all storage resources for a certain service group and is dynamically imported and deported by the cluster software. During import the hostname of the importing host is written into the private region database of the disks in the disk group to prevent concurrent imports by other hosts.

Basically .you would use one DG for your boot environment (although that is not mandatory) and a DG for every service which the machine is running and which might eventually be taken over by or migrate to another host. Migration would be done by simply issuing a command to relinquish control of the disk group (called deporting) on this host and another command to gain control over the DG on the target host (importing).

As a rule of thumb: If you are absolutely sure that you will never ever move data between hosts then you might stick with just one DG. In a cluster, use one DG per service group. If you want to make sure that I/O from one application does not affect another application then it is a good idea to use DGs to separate them, too.



4.2 Simple Disk Group Operations

Creating and Displaying a Disk Group

The command for DG operations is called **vxdg**. It takes a command word and one or several disk group names and will then work on these objects. To create a DG you pass the **init** command word to **vxdg**, followed by the name of the DG you want
created as well as at least on disk medium that will become part of the DG. For example, to create a DG consisting of just one disk, you could type the following:

```
# vxdg init mydg c0t2d0
```

This command will create a new disk group, make disk **c0t2d0** a member of it and flush this metadata to the Private Region of **c0t2d0**.

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    c0t2d0       mydg         online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online
```

As you can see **c0t2d0** now carries the name **mydg** in the **GROUP** columns of the **vxdisk list** output. The **DISK** column of **c0t2d0** says **c0t2d0,** which is redundant with the access name. The disk name was initialized with the access name because we didn't supply a different name. Having the access name as the disk name can be annoying because eventually the access name may change (e.g. because you attach the disk to a different controller). So when we add the next disk to the DG we will take care to supply a name for it.

Adding Disks to and Removing Disks from a Disk Group

Adding a disk to a DG is done by supplying the **adddisk** command word to **vxdg**. We also need to tell **vxdg** which DG we mean by specifying the switch **-g <DG-name>**.

```
# vxdg -g mydg adddisk mydg02=c0t10d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    c0t2d0       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    -            -            online
```

Looking at the DISK column we find the name mydg02 that we supplied. The suggested naming convention when naming disks is to use the DG-name followed by a two-digit counter. We started at 02 because we know that we are going to change the first disk to mydg01 later on. How can we do that? There are several possibilities. For instance we could remove the disk from the DG again, then put it back in and supply the right name at that time. Removing disks from a DG is done by supplying the **rmdisk** command word to **vxdg**.

```
# vxdg -g mydg rmdisk c0t2d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    -            -            online

# vxdg -g mydg adddisk mydg01=c0t2d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    -            -            online
```

Now we add a third disk:

```
# vxdg -g mydg adddisk c0t11d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    c0t11d0      mydg         online
```

Oops, looks like we forgot to specify a name again! Let's fix this without removing the disk this time.

Renaming Virtual Objects

The command **vxedit** can rename almost any VxVM object at any time. Exceptions are disk group names (which can only be renamed during the import or deport process) and objects in DGs that are not currently imported (because that is when we have no control over them).

```
# vxedit -g mydg rename c0t11d0 mydg03
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -           online invalid
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online
```

Now let's look at where the disk group appears in our UNIX file system, particularly in the **/dev/vx** directory structure:

```
# ls -ld /dev/vx/*dsk
drwxr-xr-x   4 root     root         512 May 14 16:57 /dev/vx/dsk
drwxr-xr-x   4 root     root         512 May 14 16:57 /dev/vx/rdsk
# ls -l /dev/vx/dsk
total 2
drwxr-xr-x   2 root     root         512 May 14 16:57 mydg
```

Deporting and Importing Disk Groups

What happens when we relinquish control over the DG by a process called **deporting**? We pass the **deport** command word to **vxdg** and see what happens, then import the DG again and check again:

```
# vxdg deport mydg
# ls -l /dev/vx/dsk
total 0
# vxdg import mydg
# ls -l /dev/vx/dsk
total 2
drwxr-xr-x   2 root     root         512 May 14 16:57 mydg
```

Obviously when a DG is imported it is represented in the file system of the computer as a directory bearing the DG's name located in **/dev/vx/dsk** and **/dev/vx/rdsk**. Deporting the DG also removed the file system representation of the DG by removing these two directories. Thus, the objects in the DG can no longer be accessed. Whichever host imports the DG gets the appropriate directory created in **/dev/vx/dsk** and **/dev/vx/rdsk** and can access the DG objects.

You can get a listing of all DGs currently imported on a system by issuing the command:

```
# vxdg list
NAME         STATE           ID
mydg         enabled,cds           1210795596.81.infra0
```

What you see here is the name of our disk group (**mydg**), its state (**enabled** and **cds**) and the internal ID by which VxVM addresses it. This ID is generated by the timestamp (number of seconds since the UNIX epoch – January 01, 1970, 00:00:00 UTC), a short random number and the hostname of the machine that created the DG. All of this together makes it highly unlikely that two IDs are ever the same except in the case of a hardware copy from a storage array (which can indeed be a problem, but at least VxVM 5.0 is prepared for such cases).

What we know as the name of the DG is merely the human-readable representation for us poor mortals. Internally, VxVM uses unique IDs for all objects and sometimes we can make use of them, too, e.g. for reviving disk groups that had been accidentally destroyed.

Destroying Disk Groups

Speaking about destroying, we haven't destroyed a DG yet, so let's do it:

```
# vxdg destroy mydg
# vxdg list
NAME         STATE           ID
```

OK, all gone. Let's see what disks we have...

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none      -            -            online invalid
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online
```

Now we create the whole DG with all three disks in one single command:

```
# vxdg init mydg mydg01=c0t2d0 mydg02=c0t10d0 mydg03=c0t11d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none      -            -            online invalid
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online

# vxdg list
NAME         STATE           ID
```

```
mydg         enabled,cds          1210802401.96.infra0
```

Just as a side note: If you are really smart then you can use **perl** to find out exactly what day it was when this example disk group was created. Remember the DG ID holds the number of seconds since the UNIX epoch? This is the command you need:

```
# perl -e 'print scalar localtime (1210802401),"\n"'
Thu May 15 00:00:01 2008
```

Nice, isn't it? OK, since creating a DG with multiple disks in a single step was so easy (you can do this when adding disks, too: just supply multiple **name=<accessname>** pairs), let's try removing multiple disks. Does that work, too?

```
# vxdg -g mydg rmdisk mydg01 mydg02 mydg03
VxVM vxdg ERROR V-5-1-10127 disassociating disk-media mydg03:
        Cannot remove last disk in disk group
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online
```

What do we learn from this? First of all, actions in VxVM are transactional, i.e. they either work or not; they do not start and then break off leaving garbage behind. We also learn that specifying multiple arguments to rmdisk is valid syntax, since the command was obviously preparing to remove the third of the three disks when it failed. So it should be possible to at least remove the two others in one go:

```
# vxdg -g mydg rmdisk mydg01 mydg02
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online
```

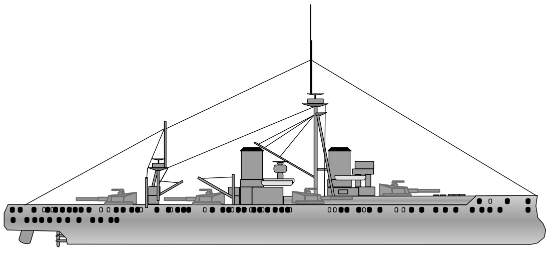
That looks good. Now let's try to get rid of the last disk:

```
# vxdg -g mydg rmdisk mydg03
VxVM vxdg ERROR V-5-1-10127 disassociating disk-media mydg03:
        Cannot remove last disk in disk group
```

We get the same error message. And it should indeed be obvious why VxVM must give us that error message. Remember that all data about a disk group is exclusively stored inside the DG itself! In other words, when you remove the last disk from a DG then there is no space left where VxVM can store information about this DG, like its name, state, flags, volumes etc. There is no storage medium to hold this information. In other words, the DG cannot exist without at least one member to hold the information about the DG. So removing the last disk from a DG is equivalent to destroying the DG, and this action should not be implicitly executed by VxVM for convenience, because it could potentially destroy important information. Maybe you just wanted to replace the old disks in the DG with new ones but you wanted to keep the complicated structure of its virtual objects. So you just remove the old disks from the DG and plan to insert the new ones in the next step. Only to find out that the DG was implicitly deleted when you removed the old disks, so that when you try to add the new ones VxVM informs you that there is no such DG. That is surely not desirable, so VxVM will always ask you to explicitly destroy a DG if that is what you really want. So let's do it:

```
# vxdg destroy mydg
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t0d0s2     auto:none       -            -            online invalid
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online
# vxdg list
NAME         STATE           ID
# ls -l /dev/vx/dsk
total 0
```

As we can see, nothing is left of our disk group after destroying it. In the technical deep dive section at the end of the chapter you will learn how to revive an accidentally destroyed DG!



The Full Battleship

4.3 Advanced Disk Group Operations

In addition to creating and destroying, importing and deporting, and adding and removing disks from DGs there are more actions and options that we should get into. They are not normally necessary for day-to-day operations, but may come in handy now and then, so let's go through them one by one:

Starting All Volumes in a DG

When a DG is imported its volumes are not started. Starting a volume needs to be done before access is allowed because VxVM needs to check the volume's status and consistency in order to guarantee data integrity. You can start volumes individually or you can start all volumes in a DG at once. This is done by executing one of the following commands, respectively:

```
# vxvol -g <DGname> start volname
# vxvol -g <DGname> startall
```

Listing DGs that are Not Imported

You can see the names of DGs even if they are not imported on your machine. You use the command **vxdisk -o alldgs** list for that. It will (kind of) do a physical I/O to each non-imported disk and read its DG name from the Private Region. The name is then displayed in parentheses in the normal output format.

```
# vxdisk list # three DGs are imported
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    firstdg01    firstdg      online
c0t10d0s2    auto:cdsdisk    seconddg01   seconddg     online
c0t11d0s2    auto:cdsdisk    thirddg01    thirddg      online
# vxdg deport firstdg seconddg # two are going away
# vxdisk list # How are we going to import them if we forgot their names?
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
```

```
c0t11d0s2    auto:cdsdisk    thirddg01    thirddg      online
# vxdisk -o alldgs list # That's how: Here are the names!
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            (firstdg)    online
c0t10d0s2    auto:cdsdisk    -            (seconddg)   online
c0t11d0s2    auto:cdsdisk    thirddg01    thirddg      online
```

4.3.1 Options for Importing or Deporting a DG

A DG is normally imported by the **vxdg import <DG>** command. In some cases it may be necessary to supply additional options to the **vxdg** command to modify its behavior. The following options are available:

Forced Import if Disks are Missing

If an import fails because not all of the disk in the DG can be accessed, then you can use the **-f** flag (forced import) to import the DG regardless. The default is for **vxdg import** to refuse import of a DG if not all of its disks can be accessed. The rationale behind that is that if you were to import such a DG and start its volumes, then if a redundant plex is located on a missing disk that plex would be marked STALE and would have to be resynchronised when the disk later comes online. Since it could happen that one of two storage boxes that hold a DG's disks is unavailable (powered off or disconnected) when we import the DG we do not want that to cause a full resynchronisation of potentially large amounts of data. Therefore, the import of such a DG must be manually forced by the administrator.

```
# vxdg -f import mydg
```

Forced Import of a DG that is Imported Somewhere Else

If an import fails because the DG is already marked as imported by another host then **if and only if we are really sure that the DG is not actually imported by another host** we can specify the **-C** flag (clear hostid) to undo the other host's reservation. A host that imports a DG will write its hostname into the Private Region of the DG's disks and thus mark the DG as imported. The VxVM running on another host trying to import the DG will see that the DG is imported and normally refuse the import. This is a reservation technique based on goodwill and cooperation. However, it is just as good as any hard reservation technique, for instance using SCSI-2 reservation commands. The reason for it being just as good is that in any reservation technique that is designed as a basis for automated failover there **must** be a way to break the reservation in software. Otherwise no automated failover would be possible if the host that holds the reservation crashes. Even the most elaborate reservation scheme, using SCSI-3 PGR (persistent group reservation) can break the reservation in software. Of course, care must be taken not to break the reservation lightly or out of convenience. I have once been called to a customer that was having problems with data consistency and crashes, and saw them routinely checking the "Clear Hostid" button in VxVM's GUI. When I asked why they were doing that they replied: "because if we don't then sometimes the import doesn't work". They were obviously unaware that their data became corrupted exactly because of what they were doing!

If you want to know more about the hostid field and about "who writes what where when" then read the technical deep dive section of this chapter beginning at page 89.

```
# vxdg -C import mydg
```

Renaming a DG

Changing the name of a DG is not as easy for VxVM as changing any other object. You may remember that the **vxedit rename** command can be used to alter the name of almost any virtual object. disk groups are different because they are also connected to the physical layer: their names appear as directory names in **/dev/vx/*dsk**. So if **vxedit** did change the name of the DG object it would also have to change the name of the DG's directory in **/dev/vx/*dsk**. But this directory is not under VxVM control but under the administrator's control, so we might not have the permissions required to rename the directory. In addition, existing mount points to volumes inside this directory may become confused if part of the path changes while the volume is still mounted. You see that renaming a DG that is already imported is difficult to do right. On the other hand, we need to have the DG imported before we can make any changes to it. If we did not import the DG then we did not reserve it and any other host might change the DG's name also.

The solution to this is that renaming can be done while importing or deporting a DG. In order to do so, we supply the -n flag (new name), followed by the new name, to vxdg. Depending on whether or not the DG is already imported, use either of the following to change the name of the DG from **old_dg** to **new_dg**:

```
# vxdg -n new_dg import old_dg
# vxdg -n new_dg deport old_dg
```

Temporary Changes to a DG

Some changes to a DG can be made only temporarily. These are importing a DG and renaming a DG upon import. If you supply the **-t** flag (temporary) when importing a DG then two things will be different from a normal import: A DG name change will not be persistent, i.e. the name will only be temporarily changed and will revert to the original when the DG is imported, or even when the system crashes. This is done by simply omitting to write the new name to the Private Regions of the DG's disks. The new name resides in memory only and will be forgotten when as the DG is no longer imported.

One more thing happens: Upon normal import a DG is flagged as both "**imported**" and "**autoimport**" and the **hostid** field is filled with the host's system name. If the host was to crash and restart, then when VxVM comes up the DGs are scanned to find ones that bear the right hostid and are flagged **autoimport**. Those DGs are then automatically imported by **vxconfigd**. The reason for this is to provide similar behavior for DGs as for plain disks, which do not need to be explicitly prepared for use like DGs. Basically, a DG that was imported before a reboot will automatically appear imported again after the reboot. Temporarily importing a DG will flag the DG **noautoimport**, which makes the system ignore the DG when it reboots. This is the way that cluster software is supposed to import DGs.

Because when crashed system comes back up, then another node may forcefully import the DG while the rebooting host's VxVM is autoimporting the DG simultaneously, leading to a race condition.

```
# vxdg -t import mydg
# vxdg -t -n new_dg import old_dg # temporarily rename a DG
```

Deporting a DG to a specific host

If you want to deport a DG so that it can only be imported by a specific host then you can do so by specifying the **-h** flag (host) followed by the name of the target host to the vxdg deport command. This functionality is rarely needed so we will no go into great detail here. What happens when you do this is that VxVM writes the new target host name into the hostid field of the DG when it deports the DG. Any host that then tries to import the DG will find that the DG appears to be already imported and refuse to import the DG unless the import is forced using the **-C** (clear hostid) flag. See the technical deep dive section for more info on the **hostid** field.

```
# vxdg -h targethost deport mydg
```

4.3.2 Disk Group Operations for Off-Host Processing

Splitting a Disk Group in Two

If you use so-called snapshots, or point-in-time copies, of volumes for backups and you do not want you production host to do the backing up of the snapshot contents itself, then you may want to give the snaphshot to a backup media server which can then read the data on the snapshot and copy it to backup media. This is called off-host processing. Off-host processing is not limited to making backups. It can also be used to optimize a file system, or for creating test data from a live production database. More will have to be said about snapshots in a later chapter before you can fully understand what is going on in this context (because we need to cover volumes first), but because this is the disk group chapter we will cover it here, and you can come back later from the snapshot chapter.

For now, just imagine that we have a disk group consisting of two disks: mydg01 and mydg02. We can split the DG by specifying which disks (or other virtual objects) we want to split out of the DG, and supplying a name for the new DG. This is called DG split and is executed by using the **vxdg split** command. The syntax is **vxdg split <sourceDG> <destinationDG> <object>**:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    mydg01       mydg         online
c0t11d0s2    auto:cdsdisk    mydg02       mydg         online
# vxdg split mydg newdg mydg02
# vxdisk list
```

```
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    mydg01       mydg         online
c0t11d0s2    auto:cdsdisk    mydg02       newdg        online
# vxdg list
NAME         STATE           ID
mydg         enabled,cds          1210850191.114.infra0
newdg        enabled,cds          1210850431.115.infra0
```

Note that while the DG name has changed for c0t11d0, its disk name is unchanged. It is still **mydg02**. We could change that using e.g. **vxedit -g newdg rename mydg02 newdg01**, but in most cases the DGs are rejoined after off-host processing, so it is not normally done.

Joining Two Disk Groups Together

To join two DGs together (typically after off-host processing) just specify them to the vxdg join command. The syntax is **vxdg join <sourceDG> <destinationDG>**. For example, to join the two DGs that were split in the section above, you would use the following command:

```
# vxdg join newdg mydg
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    mydg01       mydg         online
c0t11d0s2    auto:cdsdisk    mydg02       mydg         online
```

Moving Objects Between Disk Groups

You can also move objects, like disks or volumes, between two existing DGs by using the **vxdg move** command. The syntax is **vxdg move <sourceDG> <destinationDG> <object>**:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    otherdg01    otherdg      online
c0t10d0s2    auto:cdsdisk    mydg01       mydg         online
c0t11d0s2    auto:cdsdisk    mydg02       mydg         online
# vxdg move mydg otherdg mydg01
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    otherdg01    otherdg      online
c0t10d0s2    auto:cdsdisk    mydg01       otherdg      online
c0t11d0s2    auto:cdsdisk    mydg02       mydg         online
```

4.3.3 Miscellaneous Disk Group Operations

Setting, Inquiring, and Resetting the Default-DG

Having to specify a **-g** flag every time we issue a command can become rather unnerving. Pre-4.0 versions of VxVM used to require the existence of a DG with the name **rootdg**. This DG was intended for use with the host's internal disks, where deporting would not make any sense anyway. Without the **rootdg** VxVM would not be able to import any other disk group. In other words, where there was VxVM, there was also a **rootdg**. This made it a rather convenient default location for all kinds of objects. It was the default disk group for all actions. If you did not specify a DG for a command then VxVM would try to find a DG that contained all of the named objects, starting with the **rootdg**. Once it found a match it would use this DG and create, delete or modify the objects as requested. This made it very easy for beginners to work with VxVM.

Unfortunately it also made it rather easy for administrators to do horrible mistakes, like unwillingly delete an important volume whose name was not unique. Veritas therefore decided to prefer safety over convenience and not have VxVM search for the specified virtual objects automatically any more, but always require the user to explicitly name the DG. Because the default **rootdg** is no longer a requirement we can now set and check the default-DG as we like, using the following commands:

```
# vxdctl defaultdg mydg # to set the default-DG to mydg
# vxdg defaultdg # to inquire the current default
mydg
```

We now no longer need to pass the disk group name for every command. However, this is a global setting that is valid for the whole machine. It is entered into VxVM's boot info file, **/etc/vx/volboot**, where VxVM keeps important information so it can find its own identity (the hostid that it writes to the DGs is in here, too), the root disk etc. BTW: **never** alter this file manually. **Never**! It is created by the **vxdctl init** command and is formatted in a special way. If you mess with it your system may become unbootable!

If you just want to override a system-wide default DG, or set a default DG just for one session, or if every user wants their own default, then you can export a shell variable named **VXVM_DEFAULTDG** and set it to the desired disk group name. You can double-check if you used the correct variable name by first setting it to a nonexistent DG name and checking for an error message when you try to create an object. If you get an error message, then it was indeed the right name and you can then use command line repetition to set the same variable to the correct DG name.

```
# export VXVM_DEFAULTDG=wrongdg
# vxassist make testvol 100m
VxVM vxassist ERROR V-5-1-607 Diskgroup wrongdg not found
```

Enclosure-based and OS-based naming

If you use SAN disks then VxVM will by default use what are called enclosure-based names. Their access names are derived not from the physical paths by which the disk can be reached, but from the type and instance of the storage array they reside in. Depending on the type of storage you use you may see access names such as the ones below (which are admittedly simulated for lack of big storage array hardware). The first one is a JBOD disk, the second one is disk number 8 from IBM shark storage array number 0, and the third is disk number 4 from EMC Symmetrix number 1:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
Disk_0       auto:cdsdisk    -            -            online
IBM_SHARK0_8 auto:cdsdisk    -             -            online
EMC1_4       auto:cdsdisk    -            -            online
```

If you prefer the controller paths as access names then you can instruct VxVM to switch the naming scheme to either **ebn** (enclosure based naming) or **osn** (OS based naming). The change is immediate and only requires you to run the following command:

```
# /usr/sbin/vxddladm set namingscheme=ebn # for enclosure based names
# /usr/sbin/vxddladm set namingscheme=osn # for OS based names
```

Be aware that neither OS-based names nor enclosure-based names are guaranteed to be identical across hosts! Never write scripts that use access names together with any kind of force-option, as this may inadvertently destroy data. Don't forget that the words "it used to work" do not mean "it always works"!

In version 5.0 of VxVM a new namingscheme was introduced: **gdn**, which stands for Global Device Naming. It was meant to be a way of naming disks the same on all hosts accessing the disks. This is not currently possible unless a private region is put on the disk to hold the access name information. So **gdn** sounded like a nice idea, but cycling through several iterations of **ebn**, **osn**, and **gdn** shows that the **gdn** device names change all the time, even when we stay on the same host. Look at the following example of what happens when you switch between **gdn** and any other naming scheme:

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
d_1         auto:cdsdisk    mydg01       mydg         online
d_2          auto:cdsdisk    mydg03       mydg         online
d_3          auto:cdsdisk    mydg02       mydg         online
# vxddladm set namingscheme=osn
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online
# vxddladm set namingscheme=gdn
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
d_4          auto:cdsdisk    mydg01       mydg         online
d_5          auto:cdsdisk    mydg03       mydg         online
d_6          auto:cdsdisk    mydg02       mydg         online
# vxddladm set namingscheme=ebn
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
Disk_1       auto:cdsdisk    mydg01       mydg         online
Disk_2       auto:cdsdisk    mydg03       mydg         online
Disk_3       auto:cdsdisk    mydg02       mydg         online
# vxddladm set namingscheme=gdn
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
d_7          auto:cdsdisk    mydg01       mydg         online
d_8          auto:cdsdisk    mydg03       mydg         online
d_9          auto:cdsdisk    mydg02       mydg         online
```

As you see the **gdn** names change every time. It may just be a bug in this version, but I suggest it's better not to speak any more about **gdn** in order to protect the incompetent.

Listing Disks with their OS Native Access Names

If you use enclosure-based names then you may occasionally want to see their actual controller paths. There are two ways to do that without having to reset the naming scheme for the whole VxVM installation. The first is to do **vxdisk list <accessname>** and check the multipathing info in the last few lines. The second is much better but less commonly used because not everybody knows it:

```
# vxdisk -e list
DEVICE      TYPE      DISK          GROUP         STATUS        OS_NATIVE_NAME
Disk_0       auto      -             -             online     c0t2d0s2
IBM_SHARK0_8 auto      -             -             online     c0t10d0s2
EMC1_4       auto      -             -             online     c0t11d0s2
```

4.3.4 Summary

In this section you learned about disk groups: what the reasoning is behind them, how they are created and destroyed as well as deported and imported. You saw how to add disks to a DG and how to take disks away from a DG. We explained where the meta data of a DG must be stored (100% of it inside the DG itself!) and why this is the case. You know where a DG object is represented in the UNIX file system and that it will appear on import and disappear on deport.

If you want to know more, feel free to take the technical deep dive in the next section. If you want to get on to the next chapter, skip over the deep dive and come back later for more. It's worth it.

Technical Deep Dive

4.4 Disk Group Implementation Details

In the UNIX implementations of VxVM there is exactly one type of disk group. In the Windows implementation of VxVM there are no less than five types of disk groups, and none of them really seems to improve on the UNIX type of disk group (the five DG types are: basic, primary dynamic, secondary dynamic with no protection, secondary dynamic with SCSI reservation protection, and secondary dynamic for cluster, i.e. with a quorum acquisition scheme). This illustrates quite nicely how thoroughly a way of thinking or the environment of an operating system can influence software design. The UNIX VxVM disk group object fulfills all the needs for reliable enterprise computing. The Windows versions makes things much more complicated with its five different types of disk groups and adds nothing noteworthy to the UNIX VxVM's disk group functionality. You will see how easily a perfect disk group object can be constructed in the following paragraphs.

First of all we need to remember that a disk group's functions include moving the whole DG from one host to another. This rules out once and for all storing any information about a DG in the local file system. Why is that? It is simply because if there was (meta) data about the disk group stored in the local file system, then after the host that holds the disk group becomes unavailable (e.g. due to crash or power failure) the disk group could not be easily transferred to the failover host because the meta data cannot be extracted from the failed host's file system. Alternatively, the DG metadata could be stored in some kind of repository on all of the hosts that could potentially gain access to it

So where does VxVM store all the meta information about a disk group? It stores it inside the disk group itself. All of it. Is that hard to do? Well, not really: there is the Private Region which is destined for VxVM meta data, so there is room for the disk group meta data. Plus there are access methods inside VxVM to read and write the Private Region so it can't be too hard. What we would like to know is the concrete data structures that get written onto a disk when it joins a disk group. Or more precisely: the data structures that get written to **all of the disks** in the disk group when a new disk joins.

Representation of Disk Groups in the Private Region

In order to understand exactly how a disk group works let's have a look at what changes in the Private Region when we use one. To look at the most prominent information of a disk group we use the **vxdisk list <accessname>** command, which will reveal quite a lot. Let's pick our disk c0t2d0 and look at it before it has been treated by vxdisksetup:

```
# vxdisk list c0t2d0
Device:    c0t2d0s2
devicetag: c0t2d0
type:      auto
info:      format=none
```

```
flags:     online ready private autoconfig invalid
pubpaths:  block=/dev/vx/dmp/c0t2d0s2 char=/dev/vx/rdmp/c0t2d0s2
guid:      -
udid:      IBM%5FDNES-309170Y%5FDISKS%5FAJF18581%20%20%20%20%20%20%20%20
site:      -
Multipathing information:
numpaths:   1
c0t2d0s2        state=enabled
```

This is what VxVM knows about the disk without even initializing it. For instance, look at the flags section: The disk is flagged as **online** (this same **online** status turns up in the common **vxdisk list** output). It is also marked as **autoconfig** (which corresponds to the **auto:...** type in **vxdisk list**, and because it has no Private Region it is flagged as **invalid** (just like the **invalid** status in **vxdisk list**). Note also the multipathing information in the last three lines. This is where you see the actual number and details of the controller paths used by DMP to access the device. Now let's compare this to what the disk looks like once it is initialized with a Private Region (for our referential convenience the output was numbered by piping it through **cat -n**):

```
# vxdisksetup -i c0t2d0
# vxdisk list c0t2d0 | cat -n
 1  Device:    c0t2d0s2
 2  devicetag: c0t2d0
 3  type:      auto
 4  hostid:
 5  disk:      name= id=1210805068.106.infra0
 6  group:     name= id=
 7  info:      format=cdsdisk,privoffset=256,pubslice=2,privslice=2
 8  flags:     online ready private autoconfig autoimport
 9  pubpaths:  block=/dev/vx/dmp/c0t2d0s2 char=/dev/vx/rdmp/c0t2d0s2
10  guid:      {4dfc8d52-1dd2-11b2-8dfa-080020c28592}
11  udid:      IBM%5FDNES-309170Y%5FDISKS%5FAJF18581%20%20%20%20%20%20%20%20
12  site:      -
13  version:   3.1
14  iosize:    min=512 (bytes) max=2048 (blocks)
15  public:    slice=2 offset=65792 len=17846208 disk_offset=0
16  private:   slice=2 offset=256 len=65536 disk_offset=0
17  update:    time=1210805069 seqno=0.2
18  ssb:       actual_seqno=0.0
19  headers:   0 240
20  configs:   count=1 len=48144
21  logs:      count=1 len=7296
22  Defined regions:
23   config   priv 000048-000239[000192]: copy=01 offset=000000 disabled
24   config   priv 000256-048207[047952]: copy=01 offset=000192 disabled
25   log      priv 048208-055503[007296]: copy=01 offset=000000 disabled
26   lockrgn  priv 055504-055647[000144]: part=00 offset=000000
```

```
27  Multipathing information:
28  numpaths:   1
29  c0t2d0s2        state=enabled
```

OK, that is clearly too much information. Let us just go through the most important lines and see what we can make of them:

- Line 4 shows an empty **hostid**. This means the disks is available to all systems because it is not dedicated to a single host or cluster.
- Line 5 shows a disk ID (but no human readable disk name yet). Like the disk group ID, the disk ID also contains the number of seconds since the UNIX epoch.
- Line 6 shows that there is no disk group name or ID (no wonder; the disk in not a DG member yet)
- Line 7 shows the format (cdsdisk) as well as Public Region offset and slices for Pub/ Priv
- Line 8 shows a new flag, **autoimport**, which will be explained shortly
- Line 12 lists a site, which is uninitialized. This is actually a highly interesting feature that you can use in stretched dual data centers to optimize performance by defining the location of the disks as well as the location of the servers and have them read from the physically closest mirrors. We will not go into further detail on that here, but now you know where to look for optimization in case of stretched clusters.
- Line 17 lists the timestamp of the last change to the disk's configuration. If you know **perl** well then you can find out that it has gotten rather late by now (the computer's time zone is Germany, which is UTC plus 1 hour).

Low-Level Observations of Disk Group Behavior

Now let us reduce the output to the most interesting lines and only check those while we go through creating, importing, deporting, renaming and destroying a disk group. We do so by using egrep with an extended search pattern:

```
# vxdisk list c0t2d0 | egrep '^hostid: |^disk: |^group: |^flags: '
hostid:
disk:      name= id=1210805068.106.infra0
group:     name= id=
flags:     online ready private autoconfig autoimport
```

OK, now we create a disk group from this disk and check to see what has changed:

```
# vxdg init mydg mydg01=c0t2d0
# vxdisk list c0t2d0 | egrep '^hostid: |^disk: |^group: |^flags: '
hostid:    infra0
disk:      name=mydg01 id=1210805068.106.infra0
group:     name=mydg id=1210806514.110.infra0
flags:     online ready private autoconfig autoimport imported
```

OK, that makes a lot of sense, actually: The **hostid** field now holds our system name (**infra0**), the disk name is filled in with the name we provided (**mydg01**), the disk group information (line 3: **group**) is updated with name and a newly generated ID (oh my gosh it's late. Look at the time stamp in the disk group ID!). And there is one more flag that says the DG is currently imported. We just created the DG, and since we know that only the server that has imported the DG can alter its contents it makes a lot of sense that VxVM automatically keeps a newly created DG imported.

OK, what happens when we deport the DG from our host, thus relinquishing control of it and freeing the DG for anyone on the SAN to take it?

```
# vxdg deport mydg
# vxdisk list c0t2d0 | egrep '^hostid: |^disk: |^group: |^flags: '
hostid:
disk:      name= id=1210805068.106.infra0
group:     name=mydg id=1210806514.110.infra0
flags:     online ready private autoconfig
```

What we see is that the **hostid** is returned to an empty field, which means that the disk (or rather: the disk group) is no longer reserved by our host. The **imported** flag has gone away, which is rather obvious since the disk group is indeed not imported any more. The disk's ID has survived but its readable name has disappeared (which is also OK because whoever has imported the disk group by now might have changed the disk's name using **vxedit rename**). And the disk group's ID and name have survived. The reason for this is not immediately obvious. But consider that the disk group will eventually be re-imported by our host by passing its name to the **vxdg import** command. If the DG name was lost upon deporting we would have a hard time importing the DG back. Let's re-import the DG back onto our system and see if we get exactly the same output as we got before we deported it:

```
# vxdg import mydg
# vxdisk list c0t2d0 | egrep '^hostid: |^disk: |^group: |^flags: |^update'
hostid:    infra0
disk:      name=mydg01 id=1210805068.106.infra0
group:     name=mydg id=1210806514.110.infra0
flags:     online ready private autoconfig autoimport imported
update:    time=1210807801 seqno=0.11
```

Yes, everything is exactly the same as before: our host as written its name into the hostid field to prevent other VxVM-based systems from trying to import the DG, the disk name is back, and the DG is marked as imported again. Only the **update** time stamp shows the author that it really is time to continue writing this book tomorrow ;)

Identifying Free Extents in a DG

You normally do not care about where the individual free extents in a DG are, because you do not allocate extents for volumes yourself using vxmake. Instead, you use vxassist as a high-level command to help you gather space for your volume. Therefore you would normally ask vxassist how large a volume you can make given the required layout constraints.

But if you still want to check for individual free extents, out of fun or interest, you can do so by issuing the **vxdg free** command:

```
# vxdg -g mydg free
GROUP        DISK       DEVICE     TAG        OFFSET     LENGTH     FLAGS
mydg         mydg01     c0t2d0s2   c0t2d0     0          17846208   -
```

Setting and Inquiring Private Region Redundancy

Normally VxVM will not write to all Private Regions in a DG. It will only activate the first five disks in any DG and store the Private Region data on those If the disk group is spread across more than one controller, then the default value is six: three on each controller. They are called the config disks because they hold the configuration database. The configuration database is the database of all user-initiated changes to a DG. For instance, when a volume is created, modified, or deleted this is written into the configuration database of all config disks. The configuration database (or config-DB) hold much more information than just disk and volume information. It contains everything that VxVM knows about the DG.

There is another database, which is called the log-DB. The log-DB holds information about configuration changes that happened without user initiation. E.g. if a disk fails then the VxVM kernel will detach all plexes that have subdisks on the failed disk. This is effectively a change of the VxVM configuration, yet it was not user-initiated.

You can check which disks are config disks and which are log disks with the **vxdg list** command if you supply the disk group names as parameters:

```
# vxdg list mydg | cat -n
 1  Group:     mydg
 2  dgid:      1210850191.114.infra0
 3  import-id: 1024.113
 4  flags:     cds
 5  version:   140
 6  alignment: 8192 (bytes)
 7  ssb:            on
 8  detach-policy: global
 9  dg-fail-policy: dgdisable
10  copies:    nconfig=default nlog=default
11  config:    seqno=0.1066 permlen=48144 free=48137 templen=3 loglen=7296
12  config disk c0t2d0s2 copy 1 len=48144 state=clean online
13  config disk c0t10d0s2 copy 1 len=48144 state=clean online
14  config disk c0t11d0s2 copy 1 len=48144 state=clean online
15  log disk c0t2d0s2 copy 1 len=7296
16  log disk c0t10d0s2 copy 1 len=7296
17  log disk c0t11d0s2 copy 1 len=7296
```

Looking at the output you can see in line 10 that the number of copies for the config and log databases are at their default value. Lines 12-14 list the config disks, 15-17 list the log disks. Now let's change the number of configs and logs to 1 and 2, respectively. This is a tricky command line as you will see. It seems that the developers didn't really think deeply about how to implement it:

```
# vxedit set nconfig=1 mydg # fails because -g <DG> is missing
VxVM vxedit ERROR V-5-1-5455 Operation requires a disk group
# vxedit -g mydg set nconfig=1 # fails because no DG passed as a parameter
VxVM vxedit ERROR V-5-1-1670 must specify record names or a search pattern
# vxedit -g mydg set nconfig=1 mydg # works (ugly and redundant syntax)
# vxedit -g mydg set nconfig=2 mydg # same with the log disks
# vxdg list mydg | cat -n
 1  Group:     mydg
 2  dgid:      1210850191.114.infra0
 3  import-id: 1024.113
 4  flags:     cds
 5  version:   140
 6  alignment: 8192 (bytes)
 7  ssb:  on
 8  detach-policy: global
 9  dg-fail-policy: dgdisable
10  copies:    nconfig=1 nlog=2
11  config:    seqno=0.1071 permlen=48144 free=48137 templen=3 loglen=7296
12  config disk c0t2d0s2 copy 1 len=48144 state=clean online
13  config disk c0t10d0s2 copy 1 len=48144 disabled
14  config disk c0t11d0s2 copy 1 len=48144 disabled
15  log disk c0t2d0s2 copy 1 len=7296 disabled
16  log disk c0t10d0s2 copy 1 len=7296
17  log disk c0t11d0s2 copy 1 len=7296
```

We can see in line 10 that the number of config copies is now one and the number of log copies is two. lines 13 and 14 list the former config disks as disabled, and line 15 shows that one of the former log disks is now disabled. We can set the number of config or log copies to any value greater than zero. There are also the special values **default** and **all**, with their obvious meaning.

Resurrecting an Accidentally Destroyed DG

If we accidentally destroyed a DG and want it back, then here is a neat and undocumented trick: We cannot pass the name of the DG to the vxdg import command any more. That name is gone. It was invalidated when the DG was destroyed. But we can specify the old disk group's ID. All we need is find it. And it's actually easy to find: just use **vxdisk list <accessname>** on any of the disks that were in the DG, and the DG-ID will be listed in line 6 of the output:

```
# vxdisk -o alldgs list # DG is completely gone
DEVICE       TYPE         DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk -            -            online
```

```
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online

# vxdisk list c0t11d0 | grep ^group
group:     name= id=1210864274.146.infra0
# vxdg import 1210864274.146.infra0 # Just use DG-ID instead of DG name
# vxdisk list # The DG is back, and all volumes with it!
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    mydg01       mydg         online
c0t10d0s2    auto:cdsdisk    mydg02       mydg         online
c0t11d0s2    auto:cdsdisk    mydg03       mydg         online
```

Importing Multiple DGs of the Same Name

Because VxVM does not keep any information about deported DGs it is possible to create several DGs of the same name by deporting the first DG before creating the next one. For instance, we will create three independent DGs, all of the same name (**adg**, which stands for "a disk group").

```
# vxdg init adg adg01=c0t2d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    adg01        adg          online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    -            -            online
# vxdg deport adg
# vxdg init adg adg01=c0t10d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    adg01        adg          online
c0t11d0s2    auto:cdsdisk    -            -            online
# vxdg deport adg
# vxdg init adg adg01=c0t11d0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    -            -            online
c0t11d0s2    auto:cdsdisk    adg01        adg          online
```

Now let's make sure the other adg DGs are still there. We use the command **vxdisk -o alldgs list** to show all DGs, even the ones that are not currently imported.

```
# vxdisk -o alldgs list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            (adg)        online
c0t10d0s2    auto:cdsdisk    -            (adg)        online
```

```
c0t11d0s2    auto:cdsdisk    adg01        adg          online
```

Now how do we import another **adg**? When we try vxdg import adg we get an error saying it is already imported:

```
# vxdg import adg
VxVM vxdg ERROR V-5-1-10978 Disk group adg: import failed:
Disk group exists and is imported
```

If we try deporting our adg and then importing another adg it fails because VxVM keeps timestamps on each DG and **vxdg import** favors the most recent DG. so it will always re-import the last **adg** we had. But remember the trick for reviving an accidentally destroyed DG? We can import and DG using its ID instead of its name. so let's find the ID of the adg that resides on c0t10d0 (the second one):

```
# vxdisk list c0t10d0|grep ^group
group:     name=adg id=1210858812.122.infra0
# vxdg import 1210858812.122.infra0
VxVM vxdg ERROR V-5-1-10978 Disk group 1210858812.122.infra0: import failed:
Disk group exists and is imported
```

OK, it doesn't work without changing the name because then we would have two DGs with the same name imported at the same time. But we can temporarily rename the new **adg**:

```
# vxdg -t -n xdg import 1210858812.122.infra0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    -            -            online
c0t10d0s2    auto:cdsdisk    adg01        xdg          online
c0t11d0s2    auto:cdsdisk    adg01        adg          online
```

And the third one, too:

```
# vxdisk list c0t2d0|grep ^group
group:     name=adg id=1210858684.120.infra0
# vxdg -t -n ydg import 1210858684.120.infra0
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
c0t2d0s2     auto:cdsdisk    adg01        ydg          online
c0t10d0s2    auto:cdsdisk    adg01        xdg          online
c0t11d0s2    auto:cdsdisk    adg01        adg          online
```

So now we have three DGs with identical names imported under alias, but as soon as we deport them, because we supplied the **-t** flag their names will revert to their original names. Note that VxVM does not complain about duplicate disk names. They are in separate name spaces (each DG has its own name space, i.e. its own subdirectory in

/etc/vx/*dsk).

4.4.1 Major and Minor Numbers for Volumes and Partitions

Let's look at the volume. First, check what the volume looks like in the file system:

```
# ls -l /dev/vx/*dsk/adg/simplevol
brw-------   1 root     root     270, 62000 May 16 21:15 /dev/vx/dsk/adg/
simplevol
crw-------   1 root     root     270, 62000 May 16 21:16 /dev/vx/rdsk/adg/
simplevol
```

OK, we see a block" and a "character" device. Compare these to normal disk partitions:

```
# ls -lL /dev/*dsk/c0t10d0s2
brw-r-----   1 root     sys       32, 74 May 17 01:48 /dev/dsk/c0t10d0s2
crw-r-----   1 root     sys       32, 74 May 17 01:48 /dev/rdsk/c0t10d0s2
```

As you know, ls outputs the major and minor number instead of the file size when it displays device files. So the major number for the volume is 270, while the major number for a normal partition is 32. What do these major numbers correspond to?

```
# egrep " 32$| 270$" /etc/name_to_major
sd 32
vxio 270
```

As we can see partition I/O is done by the sd (SCSI disk, major number 32) driver, while volume I/O is passed to the vxio driver (major number 270).

Changing the Minor Numbers for Volume Devices

The minor number identifies the instance that the driver has to deal with. it is different for each device but identical for character or block device of the same instance. Each DG carries a base minor number that serves as the bottom value for device minor numbers for the volumes that reside in the DG. The base minor number is generated randomly along with the DG in a way that it does not conflict with any existing volume minor numbers. It is stored as the variable **base_minor**, which you can inquire using **vxprint -F** with the appropriate variable names (there will be more about this specific flag in a later section):

```
# vxprint -g adg -F %name,%base_minor
adg,62000 # base_minor of the DG adg
adg01,-
adg02,-
(...)
```

But of course it can always happen that we import a DG from another host, or that we create a new DG while some of our other DGs are deported, and thus VxVM does not know that the base minor number it picks may result in a conflict.

If a DG is imported that has the same minor numbers as the volumes in a DG that is already imported, then a warning is output and the volumes are automatically adjusted to non-conflicting values. However, the DG is not permanently changed. But we can also manually reminor the DG permanently. This is done by the **vxdg reminor** command:

```
# vxdg reminor adg 10000
# ls -l /dev/vx/*dsk/adg/simplevol
brw-------   1 root     root     270, 10000 May 17 01:57 /dev/vx/dsk/adg/
simplevol
crw-------   1 root     root     270, 10000 May 17 01:57 /dev/vx/rdsk/adg/
simplevol
```

As you can see, the minor number has changed from 62000 to 10000.

Why Change the Base Minor Number on a DG?

The base_minor will, as we said before, adjust itself in case of a conflict. However, there is an important case where this is not a good idea: If you are running an NFS server, then the clients identify the files on the NFS server by the NFS handle, which is a cookie comprised of several values. Among these values are the major and minor number of the device and the inode number of the file. If a failover occurs in a highly available NFS environment, and the failover host changes the minor numbers of the volumes that it takes control of, then the clients' NFS handles will become stale and cannot be used any more. NFS failover will not be transparent any more (which it would otherwise be).

In this case it would indeed be advisable to reminor the DG permanently in order to avoid future problems. Of course you should also make sure the **major** numbers are identical. This can be done by editing the **/etc/name_to_major** files to make the major numbers for **vxdmp** and **vxio** match on all cluster machines, or by using the **haremajor** script supplied with Veritas Cluster Server.