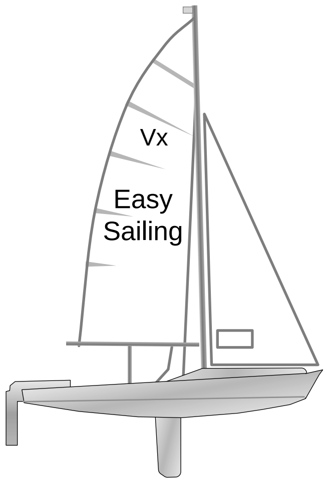
CHAPTER 15: STORAGE FOUNDATION SOFTWARE STACK

by Volker Herminghaus

15.1 Software Overview

Veritas Volume Manager operates in both user space and kernel space: User mode programs like **`vxassist`** interact with other user space programs like **`vxconfigd`** to create, modify, and delete volumes and other virtual objects. This is done during preparation of the volumes (i.e. creating them or starting them after importing a disk group) as well as during maintenance on the volumes (like resizing them or handling snapshots) or diagnosis (like running the **`vxprint`** and **`vxdisk list`** commands). In a running system where all necessary volumes have been started the only parts of VxVM actually needed are the device drivers that map volume regions to physical disk regions by applying the plex mapping table to I/O request (**`vxio`**) and for multiplexing disk paths (**`vxdmp`**). Of course it would be a very radical approach to try it, and full functionality could not be guaranteed, but it is in fact possible to run a UNIX machine using VxVM volumes without a single Volume Manager process running. It would not be possible to do any kind of maintenance on the volumes, like starting or stopping them, or importing or deporting a disk group, but those volumes that have already been started would be running perfectly well. This was a deliberate design decision by the developers because it eliminates a possible single point of failure: If a user process was necessary to enable volume I/O, then a system might be rendered unusable if this process crashed or was stopped by the user. Device drivers, unlike user processes, cannot be killed or unconfigured "against their will". Having the system be independent from user space processes for the bulk of the work (user I/O) was therefore a wise thing to do.

DOI: 10.1007/978-3-540-85023-6_15, © Springer-Verlag Berlin Heidelberg 2009



15.1.1 Structure of Storage Foundation Components

The graphic depiction shown below outlines the main components of storage foundation and their interaction. It serves as the basis for understanding the more complex environment of a full storage foundation installation. Depicting all the interdependencies would require a much larger canvas, and it is not really necessary to understand every aspect of the software stack; one can do a lot of useful work with just the basics.

The upper part of the diagram shows the user level programs, while the lower part shows the kernel level drivers, devices, and memory regions. The black arrows identify user-I/O and the grey arrows stand for control or metadata I/O. For instance, reading the diagram top down from the left we can see that a **vxassist** command to create a new volume contacts **vxconfigd**, which in turn contacts the **config** device. The **config** device is implemented by the **vxspec** device driver. The **config** device creates a new volume with its associated I/O mapping, stores it in the kernel memory region reserved for such information (the **volume mapping** cloud), and persists it to the private regions of the affected disk group via the **vxdmp** driver. The **vxdmp** driver alternates between all the available **sd** paths that are visible to **vxdmp**. Those paths are routed via fibre-channel (**fc**) drivers to their ultimate goal, the LUN. The information about the paths is gathered upon enabling the **vxconfigd**. This takes place automatically at boot time.

On the way back, I/O errors that become visible to **vxio** because they cannot be remedied by the **vxdmp** driver will be reported back to the **vxspec** driver, which passes them on to the **vcevent** device for notification to all connected **vxnotify** clients

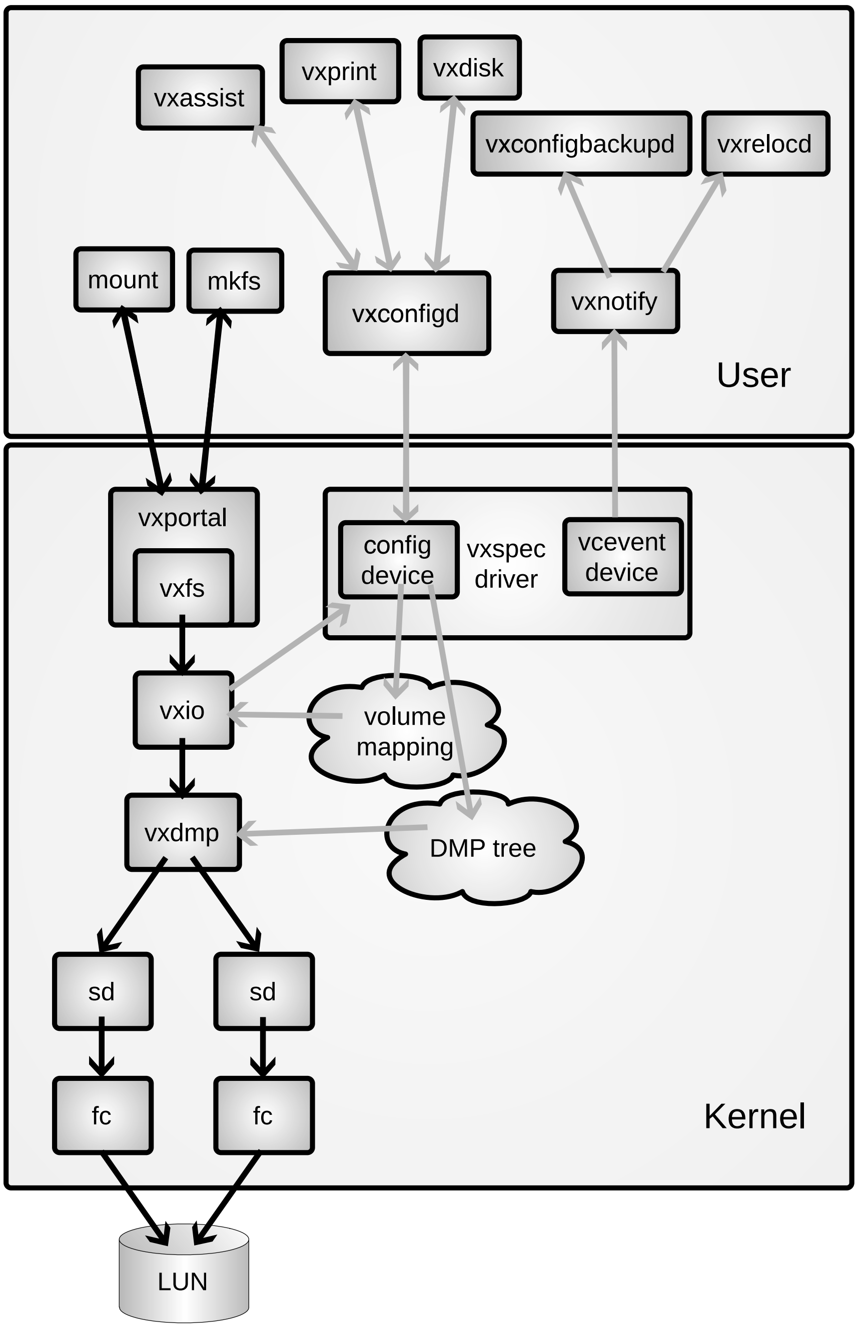


Figure 15-1: Storage Foundation's main components and their mutual relationships

On the file system side, **`mkfs, mount`**, and any other user I/O is routed via the **`vxportal`** or **`vxfs`** device drivers to **`vxio`** and on to **`vxdmp`**. It follows that as long as **`vxdmp`** is still waiting for completion of a request, the **`vxio`** driver will never be informed about the problem because **`vxdmp`** acts like an opaque wall between **`vxio`** and the lower device driver layers. In the setup shown in the diagram, with two **`sd`** paths routed via two **`fc`** paths, one might encounter the following a common problem:

- There is an underlying problem with the SAN infrastructure that keeps a LUN from responding
- The **`fc`** driver will typically give the LUN about one minute to reply to a request, after which the request is timed out. Note that the timeout can not be made arbitrarily short because we are, after all, dealing with a network here, and networks sometimes do have benign delays.
- A "non-fatal" error is reported after the timeout by the **`fc`** driver to the **`sd`** driver
- The **`sd`** driver, seeing only a non-fatal, i.e. recoverable error, retries up to the default of five times. Note that the number of retries can not be arbitrarily reduced. We certainly do not want a machine to crash just because of a bad CRC in a block on e.g. the swap device, do we? We would want the system to try hard and get the correct data from the disk, then possibly revector the bad block to prevent further mishap.
- After five retries, each timed out after one minute, **`vxdmp`** will consider the path that it was using for this particular I/O to be faulted and will retry the I/O on the other path.
- The same thing happens on the second path, and after ten minutes vxio finally gets the message that something is wrong in the I/O subsystem.

This sounds bad, but storage foundation is actually not to blame for any of this. The problem comes from the very simplistic interface between **`fc`** and **`sd`** drivers, which traditionally only pass "OK" or errors like "non-fatal" or "fatal" to their parent. A much more sophisticated driver interface is needed, and manufacturers have recently begun delivering such drivers. We recommend moving to the new generation of drivers as soon as they are viable for your enterprise. They really make a big difference.

Technical Deep Dive

15.2 Kernel Space Drivers

Veritas Volume Manager operates in both user space and kernel space: User mode programs are run to create, modify, and delete virtual objects, which are used by kernel drivers as replacements for physical disk media. The glue between user space programs and kernel space device drivers is the **ioctl** system call interface, via which the user mode programs can communicate their requests to the device drivers and thus effect changes to kernel mode variables.

Several device drivers are added to the system when installing storage foundation. We will discuss these drivers below. You can find them by searching for the string **"^vx"** in the **/etc/name_to_major** file which maps the device drivers' major numbers to their respective names. You can also run the **modinfo** command and search for **" vx"** in its output.

Finding all device drivers in a fully installed Storage Foundation 5.0 on Solaris 10 SPARC will get you a similar output to the following:

```
# grep vx /etc/name_to_major
vxportal 267
vxdmp 269
vxio 270
vxspec 271
vxfen 274

# modinfo|grep " vx"
 26  12b2b78  37f28 269   1  vxdmp (VxVM 5.0-2006-05-11a: DMP Drive)
 28 7c002000 337840 270   1  vxio (VxVM 5.0-2006-05-11a I/O driver)
 30  12e6ce0    d48 271   1  vxspec (VxVM 5.0-2006-05-11a control/st)
156 7bffce58    c30 267   1  vxportal (VxFS 5.0_REV-5.0A55_sol portal )
157 7b200000 1ba6d0  20   1  vxfs (VxFS 5.0_REV-5.0A55_sol SunOS 5)

# ls -lL /dev/vx|grep "^c"
crw-------   1 root     sys      271,  6 Aug  2 16:12 clust
crw-------   1 root     sys      271,  0 Aug  2 16:10 config
crw-r-----   1 root     sys      269, 262143 Aug  2 16:12 dmpconfig
crw-------   1 root     sys      271,  3 Aug  2 16:12 info
crw-------   1 root     sys      271,  2 Aug  2 16:18 iod
crw-------   1 root     sys      271,  7 Aug  2 16:18 netiod
crw-------   1 root     sys      271,  4 Aug  2 16:18 task
crw-------   1 root     sys      271,  5 Aug  2 16:18 taskmon
crw-------   1 root     sys      271,  1 Aug  2 16:18 trace
crw-------   1 root     sys      271,  8 Aug  2 16:18 vcevent
```

As you can see, most of the device files in the **/dev/vx** directory are instances of the **vxspec** device driver, which in our case has the major number 271. The only other driver is called **dmpconfig**, and it has the same major number as the **vxdmp** driver (269).

Let's look at the individual drivers one by one:

VXPORTAL

The **vxportal** device driver is used by **vxfs** file system utilities such as **mkfs**, **mount**, and **fsadm**, by the storage checkpoint administration utilities **fsckptadm** and **fsckpt_restore** and the quota utilities **vxquotaon** and **vxquotaoff**. Some commands may need to issue an **ioctl** to the VxFS modules even if no VxFS file systems are currently mounted. This attempt would fail because the **vxfs** device driver would not be active. For these cases the **vxportal** driver supplies the required interface to talk to the **vxfs** modules regardless of actual file systems being mounted.

VXDMP

The **vxdmp** driver has been discussed before and we would like to keep redundancy limited to volumes. Suffice it to say that the **/dev/vx/dmpconfig** device file is used to send **ioctl**s to the **vxdmp** device driver to get and set operating modes etc. The individual devices reside in the **/dev/vx/dmp** and **/dev/vx/rdmp** directories and are (of course) also instances of the **vxdmp** device driver.

VXFEN

This driver is used by the Veritas Cluster Server in order to ensure that data is unharmed in split-brain situations (split-brain is a situation in which members of a cluster are unaware of the other members' existence and erroneously take over their services, leading to uncoordinated write I/O on the shared storage. Because this corrupts data as well as metadata, even disk group private regions, it is generally considered a Very Bad Thing™ and VCS goes to great lengths to avoid this situation). It is not normally used in plain Volume Manger setups without clustering, and is not further discussed here. We are planning a new book about Veritas Cluster Server which will discuss this topic in detail.

VXSPEC

This device driver has multiple instances which all do quite different tasks, as you can see from the multiple instances of its major number (271) in the output of the **ls** command above. These instances are:

CLUST (VXCLUST)

This instance is used for clustered Volume Manager setups. It is used by **vxconfigd**, **vxclust**, **vxdctl** and **vxdg**.

config (vxconfig)

This device file is used for creating and modifying virtual objects. It was designed to be used solely by the **vxconfigd** process and can only ever be opened by one process at a time. The configuration daemon, **vxconfigd**, downloads volume configuration and address mapping into the kernel, and creates, deletes, and modifies virtual objects via this device.

info (vxinfo)

This device gathers and clears performance statistics for volume, plex, subdisk, and disk media objects from the kernel. It is used by very many of the VxVM utilities.

iod (vxiod)

This device is used to control the number and the behavior of the vxiod daemons. These daemons are discussed elsewhere in this book. They are mostly used for increasing the degree to which I/O can be done asynchronously.

netiod (vxnetiod)

This device is used by **/usr/sbin/vxnetd** in Veritas Volume Replicator (VVR) setups.

task (vxtask) / taskmon (vxtaskmon)

These device files are used to create and query kernel level tasks in VxVM.

trace (vxtrace)

This device is used by **/usr/sbin/vxtrace** to read all pre- and post-volume mapping I/Os, error records etc. from the kernel. It is not used by any other process.

vcevent (vxvcevent)

This instance of **vxspec** delivers events to the process that opens the device and reads from it. The most common user of this device file is the **vxnotify** command, which in turn is used by the relocation daemon (**vxrelocd**) to relocate subdisks from failed disks to free storage. In order to find failed disks **vxrelocd** spawns a **vxnotify** program which opens the **/dev/vx/vcevent** driver file. As soon as the Volume Manager kernel notices any configuration change, the **vcevent** driver passes that event to the listening process which can then act on the data supplied with the event in whichever way it deems appropriate. For instance, the **vxnotify** program will output the formatted data on its **stdout** channel.

Other clients for the vcevent device file are the **vxvcvol** and **vxibc** commands, which are used in Veritas Volume Replicator (VVR) setups and are not discussed here.

VXIO

This is probably the most important driver of VxVM, since it is this driver that does the actual volume I/O mapping function. It reads the volume configuration that **vxconfigd** has previously downloaded into the kernel using the **vxconfig** device, and uses it to map the volume regions to their physical disk region counterparts. This driver uses the **vxdmp** driver to actually do read and write I/Os instead of the plain **sd** drivers. In addition, the **vxio** driver supports **ioctl**s to inquire the process-ID of the configuration daemon **vxconfigd**, to detach a plex from a volume, to obtain an information record about a volume and to initiate **atomic copy** and **verify read** or **verify write** commands.

All volume block and character devices use the **vxio** driver and therefore the major number of all files addressed by **/dev/vx/*dsk/*/*** show up as (in our case) 270.

15.3 User Space Processes

Finding all Veritas Volume Manager processes in a running system is easy if you use the **ptools** located in **/usr/proc/bin**: For instance, the **pgrep** command searches the process table much more efficiently and much more precisely than any of the clumsy command chains that look like this:

```
ps -ef | grep -v grep | grep "vx" | awk '{ print $1 }'
```

The latter is a truly horrible, yet nearly ubiquitous construct, which will not only fail to find any process that happens to have the string "grep" anywhere in its command line, but it also spawns three extra processes (two **grep** and one **awk**). This is especially nasty if you recall that **awk** does in fact a much better job at **grep**'ing than **grep** does, so the two **grep** commands are really just overhead. But the biggest performance hog is the **ps** command, which uses the ancient approach of scanning the kernel's internal process table structure instead of resorting to the modern and fast **/proc** file system. We have found the disgusting contraption shown above in so many places that we cannot restrain ourselves from taking the opportunity to show you a better way. If you never use this kind of command chain to find a process then please consider yourself complimented by the authors; you are one out of a thousand that does it right!

After this short rant, here's how you really get all Veritas Volume Manager processes. The example below is from our Solaris 10 SPARC host running SF5.0 after a default installation:

```
# pgrep -lf vx
52 vxconfigd -x syslog -m boot
1649 /sbin/sh - /usr/lib/vxvm/bin/vxconfigbackupd
1021 /sbin/sh - /usr/lib/vxvm/bin/vxrelocd root
204 /sbin/vxesd
851 /opt/VRTSsmf/bin/vxsmf.bin -p RootSMF -B
915 /sbin/sh - /usr/lib/vxvm/bin/vxrelocd root
928 /sbin/sh - /usr/lib/vxvm/bin/vxconfigbackupd
791 /opt/VRTSob/bin/vxsvc -r /etc/vx/isis/Registry -e
```

```
693 /opt/VRTSobc/pal33/bin/vxpal -a VAILAgent -x
797 /opt/VRTSobc/pal33/bin/vxpal -a StorageAgent -x
877 /opt/VRTSsmf/bin/vxsmf.bin -p ICS -c /etc/vx/VxSMF/VxSMF.cfg --parentversion
1.
1000 /sbin/sh - /usr/lib/vxvm/bin/vxcached root
1001 vxnotify -C -w 15
916 /sbin/sh - /usr/lib/vxvm/bin/vxcached root
1650 vxnotify
1022 vxnotify -f -w 15
```

Here's the more structured output from a **ptree** command:

```
# ptree | grep vx
52    vxconfigd -x syslog -m boot
204   /sbin/vxesd
693   /opt/VRTSobc/pal33/bin/vxpal -a VAILAgent -x
791   /opt/VRTSob/bin/vxsvc -r /etc/vx/isis/Registry -e
797   /opt/VRTSobc/pal33/bin/vxpal -a StorageAgent -x
851   /opt/VRTSsmf/bin/vxsmf.bin -p RootSMF -B
  877   /opt/VRTSsmf/bin/vxsmf.bin -p ICS -c /etc/vx/VxSMF/VxSMF.cfg --parentve
915   /sbin/sh - /usr/lib/vxvm/bin/vxrelocd root
  1021  /sbin/sh - /usr/lib/vxvm/bin/vxrelocd root
    1022  vxnotify -f -w 15
916   /sbin/sh - /usr/lib/vxvm/bin/vxcached root
  1000  /sbin/sh - /usr/lib/vxvm/bin/vxcached root
    1001  vxnotify -C -w 15
928   /sbin/sh - /usr/lib/vxvm/bin/vxconfigbackupd
  1649  /sbin/sh - /usr/lib/vxvm/bin/vxconfigbackupd
    1650  vxnotify
```

There are many VxVM processes running — sixteen altogether, usually even more (some of the agents failed to start on our system: **actionagent** and **gridnode**) — of which many are not really necessary. The important ones are highlighted above. Being the frugal sysadmin type, we should try to get rid of as many unnecessary processes as possible, for the obvious reasons. But we need to be sure we know what we are doing, so what exactly are all these processes doing?

15.4 Reducing VxVM's Footprint

Let's take a look at all the processes in the output above and see what they do and if we could just get rid of them without affecting operations. Note that you may lose support from the vendor by doing so. Consider this a theoretical exploration rather than a hands-on guide; we obviously do not take responsibility for anyone messing with the VxVM processes on production machines!

15.4.1 Essential VxVM Processes

vxconfigd

Do not stop this process. If it is stopped you can not use any of the other vx* commands any more, since they all communicate with the VxVM kernel via this daemon. Note that I/O to those volumes which are already started is not affected, but configuration changes — even those of the simplest kind — cannot be initiated any more.

vxesd

This process (called the event source daemon) tracks events on the fibre-channel fabric and triggers updates to the **vxdmp** device tree in case of a fabric reconfiguration. It is probably wise to keep it running if you are allocating your storage from a SAN, but you can certainly turn it off if you use pure SCSI disks (but then: who does?). The event source daemon uses the Storage Networks Industry Association's (SNIA) HBA API to gather fabric topology information and to correlate LUN paths information with World Wide Names and array port IDs, so it is only useful if your array and/or HBA driver vendor supports this protocol.

vxrelocd with vxnotify

The relocation daemon, important for automatic restoration of the redundancy of volumes which encountered I/O errors. The relocation daemon is actually a shell script which spawns a **vxnotify** process with parameters that make **vxnotify** return only faults, and batch all events together that are not separated by at least a 15-second interval without any events. This **vxnotify** process connects to the **vcevent** driver (see above) to be informed whenever any kind of fault is incurred. It then passes the fault as plain text to the calling **vxrelocd** process, which operates on the output, finds the faulted disks and relocates the subdisks from redundant volumes if the corresponding regions are still readable from another plex. It also sends e-mail to the administrator during all stages of its recovery attempts. Some administrators prefer to turn hot relocation off because they want to keep full control over storage allocation at all times. This is best done by uncommenting the appropriate line in the boot script (**/lib/svc/method/vxvm-recover** in Solaris 10).

15.4.2 Unessential VxVM Processes

vxsvc

This is the server process for the **vea** GUI. It is based on the antique and arcane CORBA (common object request broker architecture) and frequently takes several minutes to shut down, resulting in excessive delays when a machine running it must be rebooted cleanly (i.e. using **init 6** rather than **reboot**). Unless your admins rely on the **vea** GUI for maintenance it may be a good idea to shut this server process down. While the GUI does enable the novice user to handle VxVM tasks in a relatively simple fashion, it often comes as a negative surprise to most UNIX users that the vxsvc process uses a registry file (/etc/vx/isis/Registry) that tries to emulate the much-hated Windows registry. Here are some lines from the beginning of the file:

```
KEY „HKEY_LOCAL_MACHINE" (
        KEY „Software" (
                KEY „VERITAS" (
[...]
                 [REG_SZ] „DomainController" = "<aHostname>";
```

That's right: **HKEY_LOCAL_MACHINE**! **DomainController**! **REG_SZ**! It is not immediately obvious what kind of advantage the addition of a registry would bring to a UNIX machine. But that is not the only Windows-like thing that **vxsvc** brought to UNIX. Its developers also added a ".ini"-file to Storage Foundation. This is what it looks like on our host:

```
# cat /etc/vx/isis/types.ini
[ALLOWED_MERGE_FAILURE]
vrts_vail* = "";
[…]
```

A true ".ini"-file, with the original Windows-ini-file syntax! On UNIX. No comment.

To summarize, the **vxsvc** process brings a lot of Windows to your UNIX machine. Whether you will think twice about deploying it, or whether you will leap into the air overjoyed about having The Power Of Windows™ on your UNIX system is of course up to you.

Keep in mind, however, that without **vxsvc** the new feature called Intelligent Storage Provisioning (ISP) will not work, so if you actually need that you will have to accept **vxsvc** on your machine.

15.4.3 Potentially Undesirable VxVM Processes

vxsmf

Called the system management framework and very poorly documented, this process "is primarily intended for developers and technical support staff. Customers should use this utility only under the guidance of a technical support person" (from the man page). It starts the initial Symantec Service Management Framework root process and all subordinate processes that are configured for the root process. The system management framework is a layer of communication and action daemons designed to enable inquiries and actions from a central management workstation to and from all machines in a data center. For instance, one could use the central management workstation to get a quick overview of the Storage Foundation licensing situation. Another interesting feature is the ability to query all storage layers as to how much of the storage arrays' total available storage is actually used for file system data. And storage arrays may be easily migrated from the management workstation because all connections from hosts to storage arrays are known to the central management workstation and can be replicated on the target array, the volumes could be mirrored to the target array, then the old mirrors removed from the source array etc.

This sounds truly great, if it works. What's not so great about it is that in order to keep this communication infrastructure secure, a whole authentication and authorization infrastructure must be built and maintained, a lot of processes are running on each host, and a number of ports must be opened for the communication to take place. Because the whole framework is (as of mid-2008, at least) rather poorly documented, it is not easy to convince anyone to actually use this additional layer. If you are not using it, then it is probably a good idea to shut it down and disable it from starting upon reboot.

vxpal with StorageAgent, gridnode, actionagent, VAILagent etc.

This daemon runs or issues commands to Veritas Provider Access Layer agents ("**pal**" stands for Provider Access Layer). The Provider Access Layer controls access to the software bus that is used by the so-called "providers" to interoperate with remote management processes like the GUI or the system management workstation outlined above. You can find the list of providers using the following command:

```
# pkginfo|grep "VRTS[^ ]*pr"
application VRTSddlpr   VERITAS Device Discovery Layer Services Provider
application VRTSfspro   VxFS Management Services Provider by Symantec
DataStorage VRTSmapro   Veritas Storage Mapping Provider from Symantec
application VRTSvmpro   VxVM Management Services Provider by Symantec
application VRTSvrpro   VVR Management Services Provider by Symantec
```

vxconfigbackupd

The configuration backup daemon. It uses a **vxnotify** process, similar to the way that **vxrelocd** does, to track changes to configurations in disk groups. Upon receiving a notification of a change it dumps all of the configuration information for the disk group affected to a backup directory (**/etc/vx/cbr/bk/$DGID**).

Originally a good idea, this feature is of rather dubious value today, as the concept has been altered (as a workaround for a serious flaw) to the point of making it almost worthless. The idea was to provide a means of "going back in time" several versions if something bad happened to your disk group, like if you accidentally deleted the wrong volume. Using the configuration backup that was previously created by the **vxconfigbackupd** one could reapply the private region contents of an earlier state of the disk group when the volume still existed. For that reason, earlier versions of Storage Foundation (4.x) kept up to six generations of disk group configuration backups. That turned out to be a problem, because with the size of the configuration data being about 24MB (at the time of writing this), having a lot of disk groups on a system would fill up the root file system very rapidly. For instance, a system with 30 disk groups would hold 6 generations of 30 disk group configuration backup copies at 24MB each, totalling over 4GB! This is less of a problem now but is certainly was a few years ago. That was the flaw mentioned above.

So the number of generations was cut from six to one. That was the workaround.

The reason why it is almost useless is the following: Any configuration change is immediately passed to the **vxconfigbackupd**, which will immediately overwrite the sole existing backup of the configuration with the current configuration. I.e. the backup is overwritten at the same time as the data is. There are only to ways to actually use the configuration backup data: One is to recover the most recent version of the configuration backup from tape. But that may not be up to data enough to use it. The other way — and that is what the vendor now officially announces its purpose to be — is to use it as a backup solely in case of unintended physical damage to the private region database, not for error recovery.

How does the **vxconfigbackupd** create its configuration backup? When **vxconfigbackupd** determines that the configuration must be backup up it runs **vxconfigbackup**, which in turn executes a number of commands, like **vxdisk list, vxdctl support, vxprivutil dumpconfig** etc., and saves the output in flat files. These commands can take a nontrivial amount of time and resources to execute, and generate about 24MB of data. Because the whole process is done every time the configuration of a disk group changes it can be very wasteful to keep this daemon running. Some examples of when a configuration backup is triggered include: creating a volume, resizing a volume, starting a volume, stopping a volume, importing a disk group, deporting a disk group, adding a disk to or removing a disk from a disk group. Weighing the likelihood of the configuration backup being able to help you out versus the amount of overhead created by generating of the configuration backup is up of course to you.

VXCACHED

The cache daemon vxcached catches notifications about cache volumes that are close to being full, and resizes them automatically for those cache volumes where such behavior was configured. A cache volume is a volume destined to hold the original data for incremental snapshots of several volumes. Setting up a group of incremental snapshots to write into a single cache volume is rather complicated and does not offer a huge advantage over more conventional approaches so we have not discussed the use of shared cache snapshots in this book.