CHAPTER 8: DUAL DATA CENTERS

by Volker Herminghaus

Since the attacks of September 11, 2001, many companies have started to move their data to two or more data centers located a few kilometers away from each other to account for possible threats by terrorists, warfare, natural disaster etc. General awareness has risen to the fact that a whole data center could fail. Surveys taken toward the end of the last century and valid until today have shown that more than 90% of all companies that suffered a total data loss did not survive the following years. They could neither reach, inform, nor bill their customers, who then walked away to the competition. The risk of not surviving a major blow to the informational infrastructure has shown to be so high as to warrant the expense of running a second, redundant data center just for availability purposes.

This chapter does not contain any Easy Sailing part. That it because dual data center setups simply are **not** easy.

8.1 Volume Management in Dual Data Centers

It is obvious that in order to ensure operability in case of a disaster all data must be copied to the remote data center and be as up-to-data as possible. There are three basic ways to do this:

1) **Offline snapshotting** - copy a mirror to the remote site at regular intervals, e.g. every night
2) **Online replication** - use one of several host or array-based replication techniques
3) **Online mirroring** - use standard mirroring across geographically close data centers

While the first one, **offline snapshotting** is viable, it takes a great deal of coordina-

V. Herminghaus and A. Sriba, *Storage Management in Data Centers*,
DOI: 10.1007/978-3-540-85023-6_8, © Springer-Verlag Berlin Heidelberg 2009

tion of the hundreds, maybe thousands of different volumes that many companies use. For instance, you would have to create snapshots or other point-in-time copies of the data every night. This could be done either by VxVM-based snapshot volumes which are then split into new disk groups and taken over by the remote host to import in case of a disaster. These snapshots would have to reside in the remote site (we will call that the disaster recovery site, or DR site, from now on). Alternatively, the storage arrays could replicate point-in-time copies of their LUNs to the remote site. If bandwidth between main site and DR site is limited and the complexity of the data center is not too high, offline snapshotting is a possibility which is relatively cheap in terms of hardware. But obviously, the data at the remote site my be somewhat stale so you will need extensive roll-forward mechanisms in place to bring the data up to date in case of a actual failure or switch. If you plan this setup properly, then you get a cheap and quick backup solution for free: the most recent snapshot constitutes a backup which can be used to restore the most recent saved state of a volume or application without resorting to slow tape backups.

Online replication is relatively widely used. Unfortunately many data centers use a storage array-based approach, which is almost guaranteed to either fail or be incredibly slow if the distance between the sites exceeds a few kilometers. The latter is true no matter what the array vendors will say! There are physical constants (like the speed of light, for instance - you may have heard of that one ;-), that preclude efficient replication of database traffic to a remote site unless the operating system handles at least part of the replication. Therefore, all purely storage array-based replication methods can do either of two things: They will be **either correct or fast**, but never both at the same time, unless they employ a device driver for the operating system (which they usually do not). Read more about online replication and its limits as well as how to do it right (in software) in the section beginning on page 213.

Online mirroring is a very good way to keep your data safe in case of a disaster. But it only works with sufficient performance across relatively short distances. While a setup over 3 or 5 kilometers is well feasible, latency starts to kick in pretty badly when 10 kilometers or more are reached. This can slow down your volume performance significantly. And again, there is not much that you can do about it: it's all physics and it's natural constants, especially the speed of light. It may be surprising, but you can trust us: light speed is much too slow for efficient computing over great distances! We will go into more detail later, in the discussion of protocols and wire speed beginning on page 213.

8.1.1 Growing a Mirrored Volume Across Sites

With data centers that are online mirrored across sites there begun to appear a deficiency of VxVM. Volume sizes are usually not static, and volumes are frequently resized to make room for the ever-growing databases. When **growing volumes across sites** or even just across enclosures at a single site, an important distinction between man and machine becomes obvious: Computers do not think along with their human operators. They just do whatever the operator tells them.

While any human in his right mind would not spend a second thinking about crossing mirror sides in the middle of a volume, there is no reason why a computer might not choose to do this. Unless, of course, the operator tells it to do it right, namely extend the mirror on site A with space from site A, and extend the mirror at site B using space allocated

from site B.

But what if there is no way of telling the computer, or rather VxVM, what we want? This is exactly the reason for many misconfigured volumes and some outages: There used to be no way of telling VxVM to allocate storage in an enclosure-specific way **when resizing**. There were all kinds of ways of specifying allocation, even very fine-grained allocation, when **creating** a volume, but not when **growing** one. We will look at the current state of the VxVM art (as of 5.0MP1 Solaris) and find some workarounds for existing volumes on older VxVMs as well as show you how it is done correctly in later versions of VxVM..

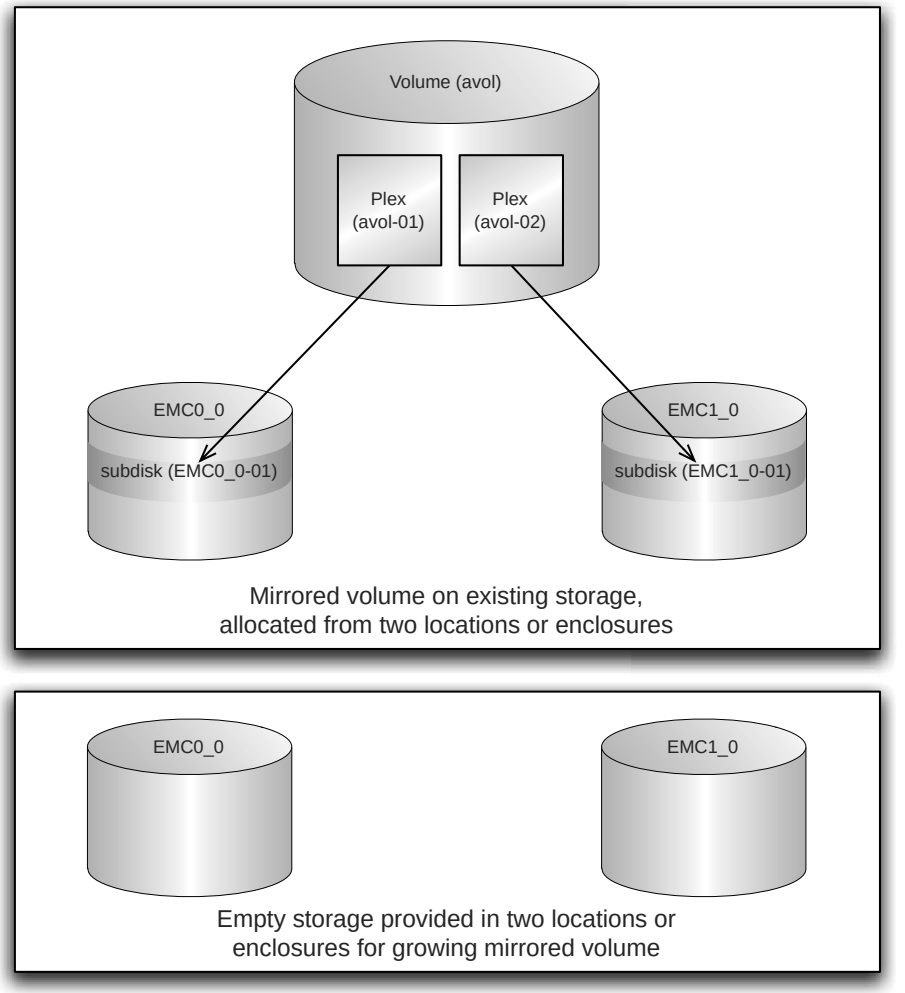

Figure 8-1: Growing a volume that is mirrored across two enclosures (or data centers) can be tricky, as you will see in the next pictures.

The Problem of Cross-Site Plexes

If you are using VxVM across data centers you may already have experienced one of the main issues concerning storage for dual data center setups, which is this: If you are using online mirroring, and you want to grow a mirrored volume, then unfortunately there is a good chance that the plexes of the volume will be grown using storage from the wrong, i.e. the "other", remote data center. A plex which begins in data center A will be extended by storage from data center B and vice versa. If any of the data centers fails, then because each plex has subdisks in each data center, both plexes will be disabled and thus the volume will be inaccessible. This is **not** high availability computing!

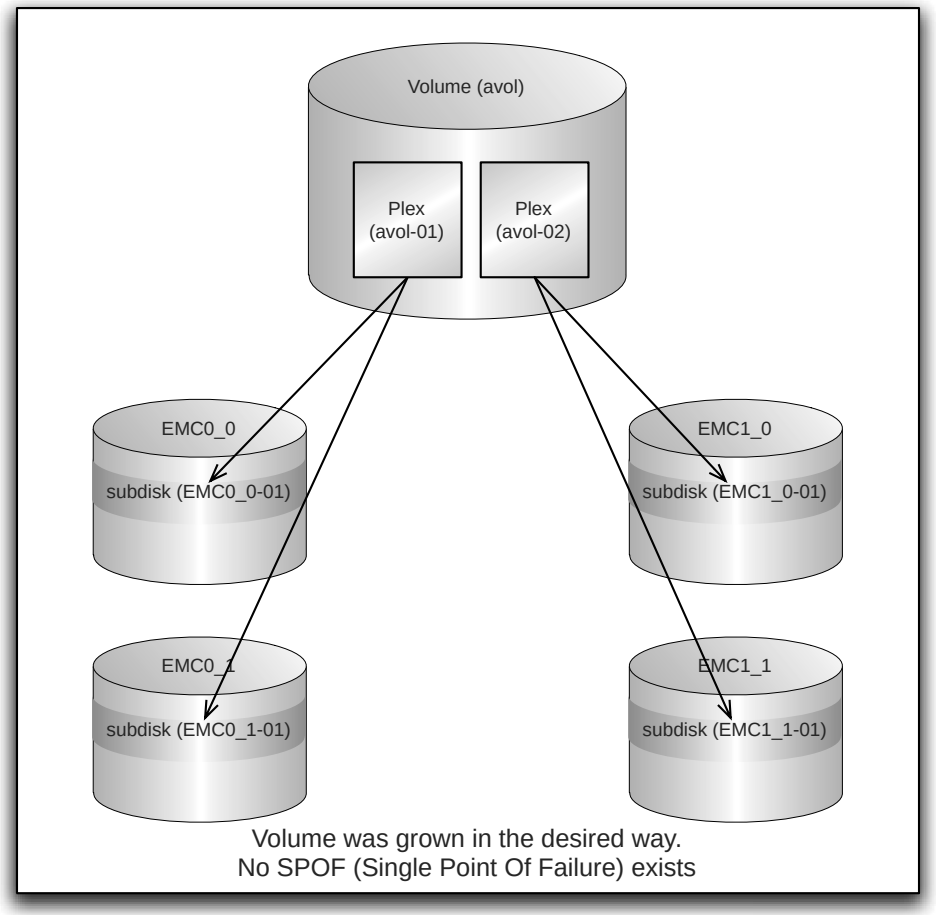

Figure 8-2: This is what we want, and sometimes get.

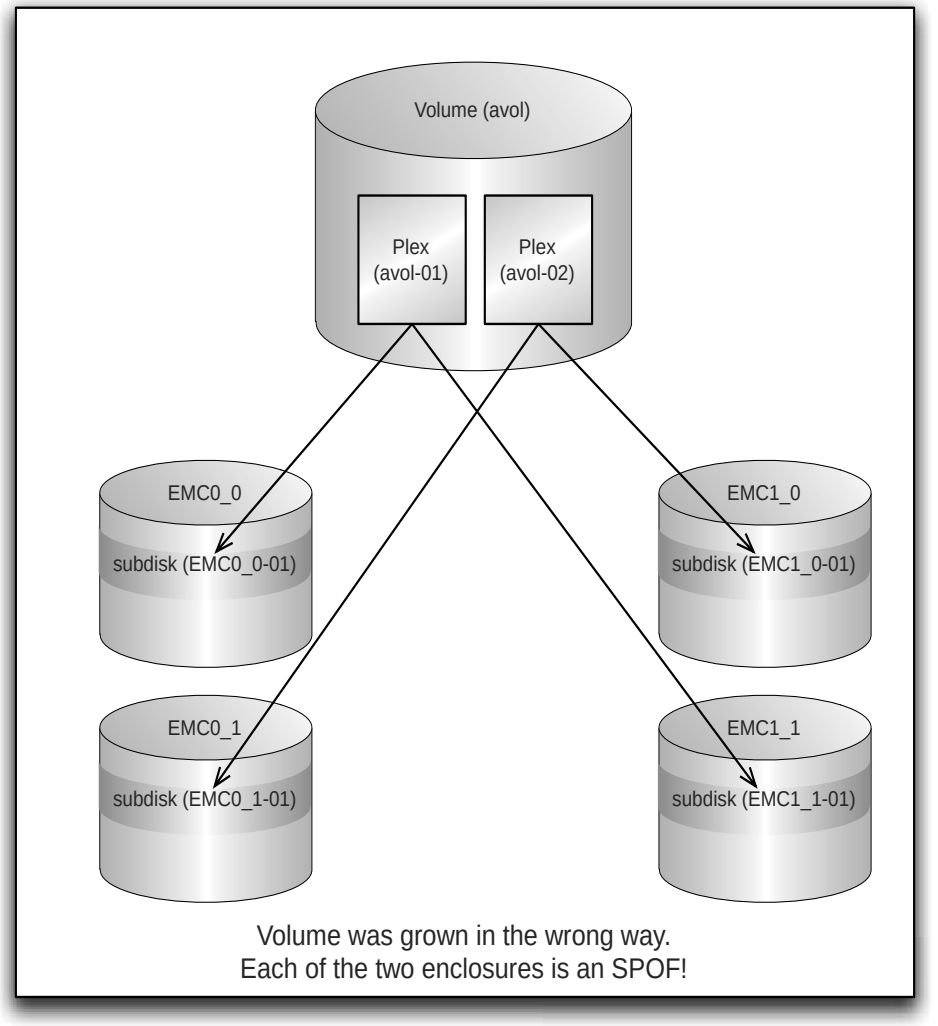

Figure 8-3: **But often we end up with this result, which is highly undesirable! Failure of one of the sites (or enclosures) will cause both plexes to be disabled and render the volume inaccessible.**

So how do we grow a volume which is allocated across data centers (or across two enclosures)? The answer is that it is not easy if the volumes already exist, but VxVM does have support for creating new volumes which respect locality and will always deliver the correct result. We will first cover the case of existing volumes. Later in this chapter, on page 204, we will explain how to create new volumes that respect locality.

8.1.2 Growing Existing Volumes Across Sites

This section describes workarounds for growing **existing** volumes in **existing** disk groups correctly across enclosures and sites without the use of the site-awareness feature introduced recently. Be warned that this is not always easy and may not work under some circumstances. If you want to have full reliability for growing volumes cross-site and can afford to recreate your volumes from scratch or if you are setting up new volumes anyway, then it is probably better to skip forward to page 204, where the topic of VxVM's **site awareness** feature is discussed.

Simple Solution with High Load and Limited Redundancy

There is no inherent VxVM logic to help us with growing "old" volumes across enclosures. We are basically on our own. Having said that, there are actually a few ways to try and enforce it. One is to create and allocate your own subdisks using **vxmake**., then map them into the appropriate plexes manually as well. This gives you full control over even the tiniest aspects of storage allocation. Previous chapters contained enough descriptive examples to illustrate how that can be done. But it is a rather low-level procedure indeed, and far would it be from us to blame anyone for not wanting to do their subdisks manually. So what alternatives are there?

One alternative is to temporarily break your mirror, grow it using storage allocation parameters to limit subdisk allocation to the local data center. Both are simple, standard commands.

For example, to grow the volume avol by 2 GB using storage on the array EMC0, you may use commands like these:

```
# export VXVM_DEFAULTDG=adg      # set default disk group for this session
# vxplex -o rm dis avol-02       # Split off the mirror and throw it away
# vxassist growby avol 2g enclr:EMC0     # Extend volume using storage from EMC0
```

In the next step, you would then re-mirror the volume using storage allocation parameters that limit subdisk allocation to the remote site, also using common VxVM commands like the following:

```
# vxassist mirror avol enclr:EMC1        # mirror volume to EMC1 storage
```

But that leaves your volume unmirrored for a while, and it introduces a large quantity of extraneous I/O. Mostly due to the heavy I/O caused by re-mirroring, and due to the lack of redundancy during the process it is not a generally accepted solution.

More Complication Solution, Low Load, Full Redundancy

A very viable way of making sure the volume remains mirrored across enclosures when growing it is tricking VxVM's storage allocation into doing the correct thing. In order to do that we first have to know what VxVM's basic procedure for allocating storage is.

How VxVM Allocates Storage

1) In the first step VxVM limits the number of disks to match whatever storage allocation you passed on the **vxassist** command line. For instance, if you specified **ctlr:c3,c5** then it will limit itself to use only targets that can be reached over controllers 3 or 5. If you specified **enclr:HDS9500_0** it will only use LUNs from that enclosure etc.

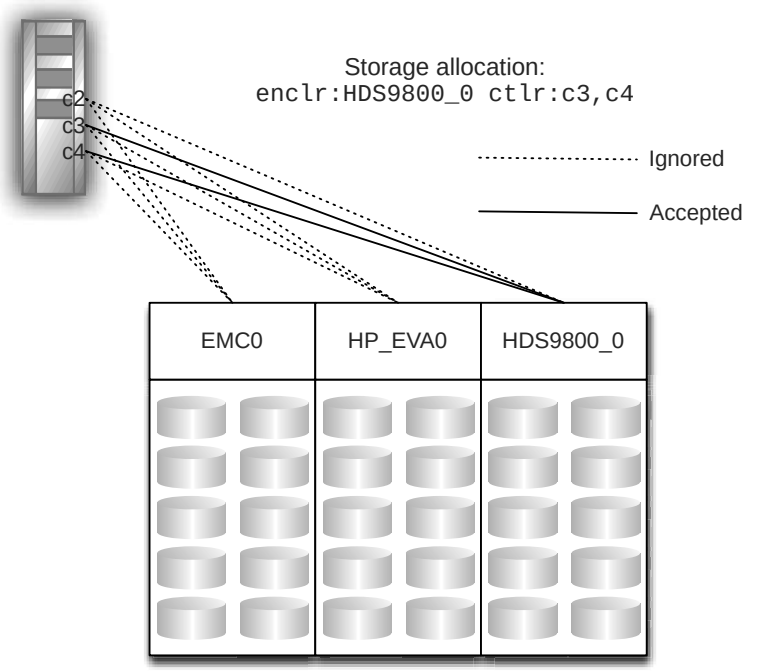

Figure 8-4: Step one of the storage selection algorithm employed by VxVM: Only LUNs or disks that match all criteria given in the storage allocation are accepted, the rest is ignored. In this case, all those disks which are visible via controllers c3 or c4 on enclosure HDS9800_0 are accepted, while the rest is ignored.

2) In the second step VxVM searches for disks that allow the required extent to be allocated in one piece rather than concatenated. If it finds any, then it limits further searches to that subset of the disks. If it does not find any, then it just uses concatenation of smaller subdisks, but we have not further investigated into VxVM's behavior for that case.

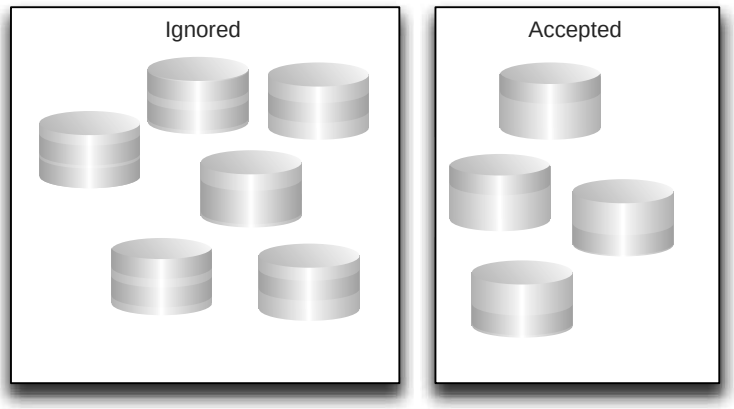

Figure 8-5: **Step two of the storage selection algorithm: If there are enough LUNs that allow contiguous allocation of the complete space, then all other LUNs are ignored.**

3) In the third step VxVM will look for the disk where the extent in question can be allocated at the smallest offset from block 0 (leading to VxVM's preference for empty disks). If it finds more than one disk with the smallest offset from block 0 then again it has a subset of disks that enter the next step.

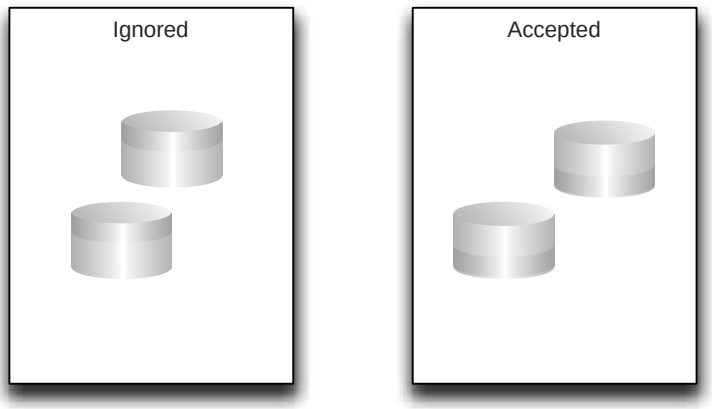

Figure 8-6: **Step three of the storage selection algorithm: LUNs that have enough contiguous free space at the lowest block offset from block zero are preferred (empty disks are typically preferred as a result). The other LUNs, which are ignored here, would be accepted if more space was needed. In the given case, the two LUNs to the right suffice so the other ones are ignored.**

4) In the last step VxVM allocates "top to bottom" from the remaining subset of disks by sorting the accessnames of the disks (correctly distinguishing between alphabetical and numerical parts of the accessname) in ascending order and preferring the resulting disks in that order.
5) When a volume is extended, VxVM allocates storage plex by plex. It sorts the plex names in alphanumerically ascending order and begins with the first plex of the result, i.e. typically the plex that has a name like **<volname-01>**, then proceeds with **<volname-02>** and so on.

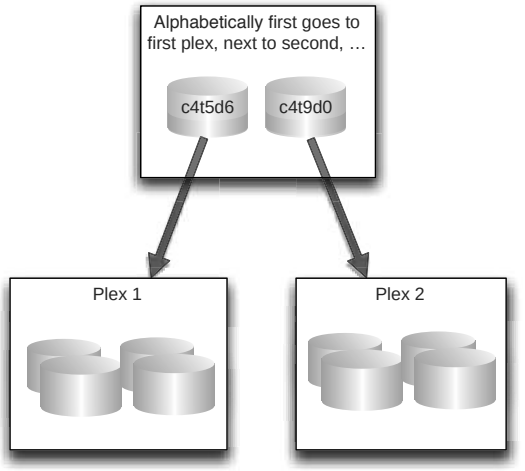

Figure 8-7: Steps four and five of the storage selection algorithm: The LUNs are ordered alphanumerically and assigned to the plexes in ascending order.

Tricking VxVM's Allocation Strategy

If we find that VxVM uses disks on the wrong enclosure for the plexes then all we need to do is put a volume on the disks that it allocates first (in the current implementation the ones for the first plex). This volume needs to be large enough that the offset of the free extent to be used for extending the plex is at least one block larger than the offset on the disks intended for the other plex.

For example, if you are using two new, empty LUNs to extend a mirrored concat volume across enclosures, then because the free extents start at block 0 on both LUNs it would be sufficient to create a volume of length 1 block (only 512 bytes!) on the LUN that you want VxVM to allocate to the second data plex. Because VxVM will prefer the LUN that has the offset of 0 blocks it will allocate this LUN to the first plex, then use whatever is left for the second plex (which will happen to be our second disk; the one with the micro-volume on it). After successful extension you can then throw the micro-volume away.

In the more complicated case of adding, say, ten LUNs to either plex (total twenty) you

can force creation of little subdisks by creating a 10-column stripe of minimal size on the disks that you want to allocate to the second plex, and nothing on the disks that you want to allocate to the first plex. This will make VxVM prefer the empty disks for the first plex and defer usage of the other disks (the ones with the tiny striped volume on them) until space for the second plex is allocated.

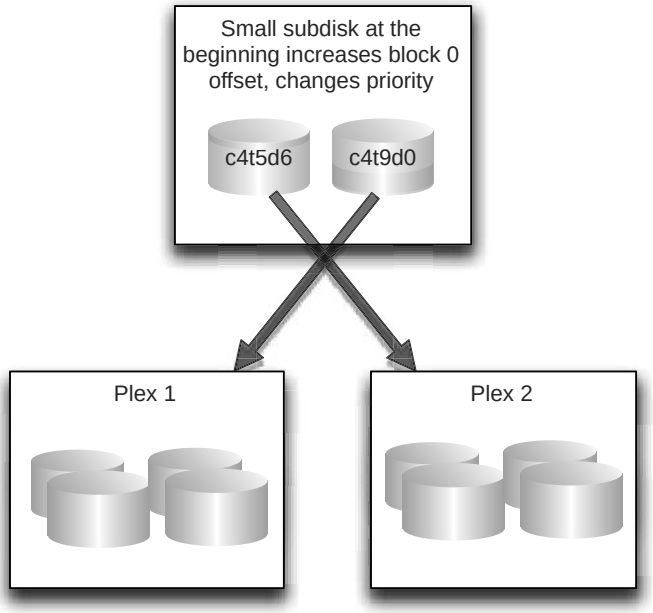

Figure 8-8: **Tricking VxVM's allocation strategy into doing the desired thing can be done by adding a tiny volume to the LUN that would be allocated to the first plex. This makes VxVM defer allocation from the LUN; the other one is used first, effectively swapping the allocation.**

Do not worry about the minimal volumes taking up space. Even if you had to throw 10 MB at the problem (which in general you do not), that would be nothing compared to an outage due to cross-allocation, right? If you do not want to use this scheme because your LUNs are not always empty or it would be too much of a hassle, then the authors are sorry but cannot help you any further except point you to the end of the section, where a slightly oversized solution is presented. But if you refuse to use this scheme because you think that having a "hole" in your disk where there is no subdisk for a volume is unacceptable for any reason then be assured that there is no technical reason why you should not "waste" 1MB or so per LUN to tweak VxVM to do your stuff right. I.e there is no reasonable complaint against this procedure (but admittedly several unreasonable ones, which we choose to disregard – life is hard enough without unreasonable demands).

Volume Layouts: stripe-mirror-sd vs. stripe-mirror-col

If you are using volumes that are striped via VxVM, then there is a solution that you might not be aware of. Most admins will use the layered volume type of "stripe-mirror" for striped, redundant volumes across enclosures. But that is indeed not the optimal layout in terms of redundancy, as it does in fact allow the crossing of enclosure boundaries inside a single column. For instance, it might happen that you create a stripe-mirror using LUNs from the Bern and Zurich data centers, but inside a single column, both Bern and Zurich LUNs will be used in both plexes. This causes an undesired, unacceptable and often undetected single point of failure. The reason that this is allowed to happen is that VxVM maps the layout specification of **stripe-mirror** to the more specific layout of **stripe-mirror-col**, which means a stripe-mirror that mirrors its sub-entities by column, i.e. mirroring is done column by column. This bad point about this layout is that its requirements are satisfied even if both plexes in a column contain LUNs from the same enclosure! What you probably want to do is mirror at the subdisk level, i.e. have each subdisk mirrored on its own. That could mean you are ending up having more sub-volumes in your layered volume, but they will be mirrored piece by piece. A column can no longer have plexes containing storage from more than one enclosure, since that would imply more than one subdisk in the plex, which is forbidden in this layout. If this sounds good to you, then you are invited to try the layout specification **stripe-mirror-sd**, which stands for stripe-mirror on a subdisk level. Note that you can also relayout or convert you existing volumes to stripe-mirror-sd. In the case of a conversion, i.e. if your previous layout is stripe-mirror, it takes only a very short time and does no I/O to the public region; only the private region database is altered.

As it always seems to be the case, however, even this layout is not without drawbacks: while it works fine for storage systems that consist only of same-sized LUNs, and where each LUN is used for only one volume, it does not universally work for any kind of storage. This is due to the way that storage allocation works in the **stripe-mirror-sd** layout: It creates a subvolume over the **greatest** of the available subdisks rather than the **smallest**. While this behavior does limit the creation of a possibly excessive number of subvolumes, it unfortunately does not prevent creation of a single point of failure. Look at the following example, where we try to salvage a volume. The volume has acquired a single point of failure by a **vxassist growby** command and we try to undo the SPOF problem by converting it to the **stripe-mirror-sd** layout. We start from scratch, with eight LUNs, four each from the bern and Zurich site. They are marked by using appropriate disk names:

```
# vxdisk list # We got four disks from each site: bern and zurich
DEVICE       TYPE            DISK         GROUP        STATUS
[...]
c0t2d0s2     auto:cdsdisk    bern01       adg          online
c0t3d0s2     auto:cdsdisk    zurich01     adg          online
c0t4d0s2     auto:cdsdisk    zurich02     adg          online
c0t10d0s2    auto:cdsdisk    bern02       adg          online
c0t11d0s2    auto:cdsdisk    bern03       adg          online
c0t12d0s2    auto:cdsdisk    zurich03     adg          online
c0t13d0s2    auto:cdsdisk    bern04       adg          online
c0t14d0s2    auto:cdsdisk    zurich04     adg          online
```

We create a volume on the disks using a standard **vxassist make** command with a standard layout. But first, we check how large we can make the volume on half the disks.

```
# vxassist maxsize layout=stripe-mirror ncol=2 alloc=bern01,bern02,\
  zurich01,zurich02
Maximum volume size: 35358720 (17265Mb)
# vxassist make r10vol 17265m layout=stripe-mirror alloc=bern01,bern02,\
  zurich01,zurich02 init=active
# vxprint -qrtLg adg
[...]
v  r10vol       -            ENABLED  ACTIVE   35358720 SELECT    r10vol-03
fsgen
pl r10vol-03    r10vol       ENABLED  ACTIVE   35358720 STRIPE    2/128    RW

sv r10vol-S01   r10vol-03    r10vol-L01 1      17679360 0/0       2/2      ENA
v2 r10vol-L01   -            ENABLED  ACTIVE   17679360 SELECT    -        fsgen
p2 r10vol-P01   r10vol-L01   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 bern01-02    r10vol-P01   bern01   0        17679360 0         c0t2d0   ENA
p2 r10vol-P02   r10vol-L01   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 zurich02-02  r10vol-P02   zurich02 0        17679360 0         c0t4d0   ENA

sv r10vol-S02   r10vol-03    r10vol-L02 1      17679360 1/0       2/2      ENA
v2 r10vol-L02   -            ENABLED  ACTIVE   17679360 SELECT    -        fsgen
p2 r10vol-P03   r10vol-L02   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 zurich01-02  r10vol-P03   zurich01 0        17679360 0         c0t3d0   ENA
p2 r10vol-P04   r10vol-L02   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 bern02-02    r10vol-P04   bern02   0        17679360 0         c0t10d0  ENA
```

The volume looks fine. Even if the first plex is not always on the same side in all subvolumes (the plexes using LUNs from Bern have been emphasized), this does not harm resiliency at all; it is a purely cosmetic matter.

Now we grow the volume using only the extra LUNs that are still free.

```
# vxassist growby r10vol 1g alloc=bern03,bern04,zurich03,zurich04
# vxprint -qrtLg adg # and see what happens:
[...]
r10vol       -            ENABLED  ACTIVE   37455872 SELECT    r10vol-03 fsgen
pl r10vol-03    r10vol       ENABLED  ACTIVE   37455872 STRIPE    2/128    RW

sv r10vol-S01   r10vol-03    r10vol-L01 1      18727936 0/0       2/2      ENA
v2 r10vol-L01   -            ENABLED  ACTIVE   18727936 SELECT    -        fsgen
p2 r10vol-P01   r10vol-L01   ENABLED  ACTIVE   18727936 CONCAT    -        RW
s2 bern01-02    r10vol-P01   bern01   0        17679360 0         c0t2d0   ENA
s2 bern03-02    r10vol-P01   bern03   0        1048576  17679360  c0t11d0  ENA
p2 r10vol-P02   r10vol-L01   ENABLED  ACTIVE   18727936 CONCAT    -        RW
s2 zurich02-02  r10vol-P02   zurich02 0        17679360 0         c0t4d0   ENA
s2 bern04-02    r10vol-P02   bern04   0        1048576  17679360  c0t13d0  ENA
```

```
sv r10vol-S02   r10vol-03    r10vol-L02 1      18727936 1/0       2/2      ENA
v2 r10vol-L02   -            ENABLED  ACTIVE   18727936 SELECT    -        fsgen
p2 r10vol-P03   r10vol-L02   ENABLED  ACTIVE   18727936 CONCAT    -        RW
s2 zurich01-02  r10vol-P03   zurich01 0        17679360 0         c0t3d0   ENA
s2 zurich03-02  r10vol-P03   zurich03 0        1048576  17679360  c0t12d0  ENA
p2 r10vol-P04   r10vol-L02   ENABLED  ACTIVE   18727936 CONCAT    -        RW
s2 bern02-02    r10vol-P04   bern02   0        17679360 0         c0t10d0  ENA
s2 zurich04-02  r10vol-P04   zurich04 0        1048576  17679360  c0t14d0  ENA
```

As you can see in the highlighted parts above, a something bad has happened: some of the plexes are using storage from both locations, leading to SPOF susceptibility. We try to remedy the situation by mirroring each individual subdisk. This is done by converting the volume to a layout of **stripe-mirror-sd**.

```
# vxassist convert r10vol layout=stripe-mirror-sd # Takes only one second...
# vxprint -qrtLg adg
[...]
# vxassist convert r10vol layout=stripe-mirror-sd
# vxprint -qrtLg adg
v  r10vol       -            ENABLED  ACTIVE   37455872 SELECT    r10vol-01
fsgen
pl r10vol-01    r10vol       ENABLED  ACTIVE   37455872 STRIPE    2/128    RW

sv r10vol-S03   r10vol-01    r10vol-L03 1      17679360 0/0       2/2      ENA
v2 r10vol-L03   -            ENABLED  ACTIVE   17679360 SELECT    -        fsgen
p2 r10vol-P05   r10vol-L03   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 bern01-01    r10vol-P05   bern01   0        17679360 0         c0t2d0   ENA
p2 r10vol-P06   r10vol-L03   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 zurich02-01  r10vol-P06   zurich02 0        17679360 0         c0t4d0   ENA

sv r10vol-S04   r10vol-01    r10vol-L04 1      1048576  0/17679360 2/2      ENA
v2 r10vol-L04   -            ENABLED  ACTIVE   1048576  SELECT    -        fsgen
p2 r10vol-P07   r10vol-L04   ENABLED  ACTIVE   1048576  CONCAT    -        RW
s2 bern03-01    r10vol-P07   bern03   0        1048576  0         c0t11d0  ENA
p2 r10vol-P08   r10vol-L04   ENABLED  ACTIVE   1048576  CONCAT    -        RW
s2 bern04-01    r10vol-P08   bern04   0        1048576  0         c0t13d0  ENA

sv r10vol-S05   r10vol-01    r10vol-L05 1      17679360 1/0       2/2      ENA
v2 r10vol-L05   -            ENABLED  ACTIVE   17679360 SELECT    -        fsgen
p2 r10vol-P09   r10vol-L05   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 zurich01-01  r10vol-P09   zurich01 0        17679360 0         c0t3d0   ENA
p2 r10vol-P10   r10vol-L05   ENABLED  ACTIVE   17679360 CONCAT    -        RW
s2 bern02-01    r10vol-P10   bern02   0        17679360 0         c0t10d0  ENA

sv r10vol-S06   r10vol-01    r10vol-L06 1      1048576  1/17679360 2/2      ENA
v2 r10vol-L06   -            ENABLED  ACTIVE   1048576  SELECT    -        fsgen
```

```
p2 r10vol-P11   r10vol-L06   ENABLED  ACTIVE   1048576  CONCAT    -        RW
s2 zurich03-01  r10vol-P11   zurich03 0        1048576  0         c0t12d0  ENA
p2 r10vol-P12   r10vol-L06   ENABLED  ACTIVE   1048576  CONCAT    -        RW
s2 zurich04-01  r10vol-P12   zurich04 0        1048576  0         c0t14d0  ENA
```

As you can see, now all plexes are rearranged in such a way that no single plex holds storage from both locations. But they are rearranged in a way that there still is a SPOF: Two of the subvolumes are mirrored inside a location! So this is not a general solution, but if you know what you're doing, and if you carefully inspect the volume after growing it you may get lucky.

8.1.3 Mirroring Site-Aware Volumes Across Sites

Site Awareness: a Solution for New Disk Groups and Volumes

With VxVM 5.0 there is a new feature called "site awareness". This feature was created expressly with dual or even multiple data centers in mind. To reduce read latency and WAN traffic when reading from a mirror you can, with VxVM 5.0, define which site the host and the LUNs are located in. Then, if you set the appropriate read policy for the volumes, VxVM will only read from the local storage by preferring those LUNs that bear the same name tag as the host. It will also use automatic cross-site mirroring if you tell the disk group which sites there are, and VxVM 5.0 will even extend the mirrors correctly in that case! The downside is that as of SF5.0MP1 **existing** volume do not profit from this new capabilities, so unless you are setting up a new system or you are willing to accept some serious downtime, it may be better to stick with the workarounds mentioned before.

Note: the day after the last version of this book was finished and sent off to the press we were exposed to a newer release of SF: 5.0MP3 (Linux). In this version, the procedures discussed below do seem to work even with existing volumes. Due to the lack of testing time, however, this cannot be guaranteed. The book was delayed enough to integrate this extra paragraph but not enough to change the structure of this whole chapter.

Having said that, this is how site awareness is set up on the physical level

```
# vxdctl set site=<sitename>     # set the site for this server
# vxdctl list                    # check the site for this server
# vxdisk settag <accessname> site=<sitename>     # set the site for a disk
# vxdisk listtag        # check the site for all disks
# vxdg addsite <firstsite>
# vxdg addsite <secondsite>
```

A disk group that is prepared in the way outlined above will have different defaults than usual: it will, by default, mirror all new volumes across all sites added to the disk group using **vxdg addsite <sitename>**. Growing a mirror that was created in a disk group like this will automatically grow such that each plex remains confined to its site.

While the above sounds very good, VxVM will also add dirty region logs to all newly created volumes. To reiterate: DRLs are logs that remember which regions are currently undergoing write-I/O and must be resynchronised using **RDWRBACK** after a crash. It therefore speeds up crash recovery. Adding a DRL may not be the ideal solution since you may want to use **vxsnap prepare** instead, which creates a combined DRL and DCO log to cover all possible resynchronisation events: Plex resynchronisation using **RDWRBACK** and plex resynchronisation using **ATCOPY**. If you want to use the combined DRL/DCO log, specify **layout=mirror,nolog** to **vxassist** when you create the volume, then use **vxsnap prepare** to add the DRL/DCO log (DCO version 20). Specifying the **nolog** attribute keeps VxVM from creating the DRL that is unnecessary f you want to use **vxsnap**.

While site awareness is not the same as enclosure-awareness, it can still be used to keep the plexes confined to their enclosures. In order for that to work we need to define a "site" for each enclosure, and make VxVM handle each enclosure as an individual site. There is a little downside to that because VxVM will only read from the "local" site, but you can set the volume read policy to round (**vxvol rdpol round $VOLNAME**) to allow round-robin access. The other downside is that if your volumes span multiple enclosures at each site, then this procedure does not work because you would have to specify multiple enclosures acting as "locations" for your host, too.

That said, let's look at the following walkthrough of a disk group containing ten LUNs being set up for automatic site awareness, and subsequent growing of a volume in this disk group.

The general setup here is that there are two data centers, and we wish to mirror across these. The data centers are located in KEL (which stands for Kelsterbach; the location of a large data center in Germany) and FRA (Frankfurt, a location about 30km away from Kelsterbach, and home of many data centers for banks).

First we want to inform VxVM that our machine is located in **KEL**., then check if it was set successfully:

```
# vxdctl set site=KEL
# vxdctl list
Volboot file
version: 3/1
seqno:   0.9
cluster protocol version: 70
[...]
siteid:  KEL    # OK, looks good. The site info is persisted in /etc/vx/volboot
```

The following are our disks (with the boot disks omitted because they do not play a role here and just take up space)

```
# vxdisk list
DEVICE       TYPE            DISK         GROUP        STATUS
[...]
c1t1d0s2     auto:cdsdisk    adg00        adg          online
c1t1d1s2     auto:cdsdisk    adg01        adg          online
c1t1d2s2     auto:cdsdisk    adg02        adg          online
c1t1d3s2     auto:cdsdisk    adg03        adg          online
c1t1d4s2     auto:cdsdisk    adg04        adg          online
```

```
c1t1d5s2     auto:cdsdisk    adg05        adg          online
c1t1d6s2     auto:cdsdisk    adg06        adg          online
c1t1d7s2     auto:cdsdisk    adg07        adg          online
c1t1d8s2     auto:cdsdisk    adg08        adg          online
c1t1d9s2     auto:cdsdisk    adg09        adg          online
```

Let's check if this disk group knows anything about sites at all. Remember the **vxprint -m** command outputs every single bit of persistent information about a disk group, so we'll use it and **grep** for "site".

```
# vxprint -m -g adg | grep site
        siteconsistent=off # Aha! Interesting, but turned off...
        site=   # All the disks have a site tag, but it's empty
        site=
        site=
        site=
        site=
        site=
        site=
        site=
        site=
        site=
```

In the next step we will identify which disks reside in which location. If you are emulating sites for the sake of keeping your volumes confined to an enclosure, then you need to make sure not to mix up disks here. If you do, then VxVM will be forced to consistently allocate storage from the wrong disks, and we can certainly live without that.
This is the general syntax of the appropriate vxdisk command to set the location tag on a disk:

```
# vxdisk -g adg settag c1t1d0s2 site=FRA
```

But we do not want to type so much, and loops scale much better than repeating individual commands, so we'll put it all in a loop:

```
# for disk in 0 1 2 3 4; do vxdisk settag c1t1d${disk}s2 site=FRA; done
# vxdisk listtag
DEVICE          NAME                        VALUE
c1t1d0s2        site                        FRA
c1t1d1s2        site                        FRA
c1t1d2s2        site                        FRA
c1t1d3s2        site                        FRA
c1t1d4s2        site                        FRA
[...]
```

As you can see above the first five disks are now flagged with site **FRA**. Now we'll flag the other ones with **KEL**.

```
# for disk in 5 6 7 8 9; do vxdisk settag c1t1d${disk}s2 site=KEL; done
# vxdisk listtag
DEVICE          NAME                    VALUE
c1t1d0s2        site                    FRA
c1t1d1s2        site                    FRA
c1t1d2s2        site                    FRA
c1t1d3s2        site                    FRA
c1t1d4s2        site                    FRA
c1t1d5s2        site                    KEL
c1t1d6s2        site                    KEL
c1t1d7s2        site                    KEL
c1t1d8s2        site                    KEL
c1t1d9s2        site                    KEL
# vxprint -m -g adg | grep site
        siteconsistent=off
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=KEL
        site=KEL
        site=KEL
        site=KEL
        site=KEL
```

All the disks are flagged with their appropriate site. Let's have a look at the default volume parameters that VxVM is going to use when creating a new volume now:

```
# vxassist -g adg help showattrs
#Attributes:
layout=nomirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-
col,nostripe-mirror-sd,noconcat-mirror,nomirror-concat,span,nocontig,raid5log,no
regionlog,diskalign,nostorage
 mirrors=2 columns=0 regionlogs=1 raid5logs=1 dcmlogs=0 dcologs 0
[...]
```

Looks like it is not going to mirror, and not going to add a log to the volumes. Note that this will change quite dramatically as we tell the disk group about sites:

```
# vxdg -g adg addsite FRA       # We add a site to the disk group
# vxassist -g adg help showattrs
#Attributes:
 layout=mirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-
col,nostripe-mirror-sd,noconcat-mirror,nomirror-concat,span,nocontig,raid5log,-
regionlog,diskalign,nostorage
```

```
 mirrors=1 columns=0 regionlogs=1 raid5logs=1 dcmlogs=1 dcologs 1
[...]
```

Already, VxVM's default has changed to mirroring by default (albeit with only one plex, so it is not really mirroring). But it also now defaults to adding a **DRL** (see emphasized type above). Note that wile in the last line above it does say "regionlogs=1 [...] **dcmlogs=1 dcologs=1**" that does not mean these logs are added by default. Only a DRL (dirty region log) is added, and that is because the "**regionlog**" attribute (emphasized in the second line) is set. The line containing the **DCM** and **DCO** parameters just specifies the number of logs to use if logging is requested. It does not actually specify whether or not logging is default; that is done in the first part, where it says **layout=mirror,[...],regionlog,[...]** etc.

If we add the second site, the setup is complete:

```
# vxdg -g adg addsite KEL
# vxassist -g adg help showattrs
#Attributes:
 layout=mirror,nostripe,nomirror-stripe,nostripe-mirror,nostripe-mirror-
col,nostripe-mirror-sd,noconcat-mirror,nomirror-concat,span,nocontig,raid5log,-
regionlog,diskalign,nostorage
 mirrors=2 columns=0 regionlogs=2 raid5logs=2 dcmlogs=2 dcologs 2
[...]
```

As you can see the layout has not changed at all; mirror and regionlog are the only positive settings in the default layout line. But the number of elements has changed in the second line: mirrors=2 and all the count of all the logs is two also. That means that now VxVM will mirror (with two plexes), and add two DRLs unless we specify otherwise:

```
# vxprint -m -g adg | grep site
        siteconsistent=off
site FRA
site KEL
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=KEL
        site=KEL
        site=KEL
        site=KEL
        site=KEL
```

Note the two additional lines in the output that specify the FRA and KEL sites. These are from the disk group record, which now knows about the two sites because we added them to the disk group. Let's create a volume with the (new) default layout and see what we get:

```
# vxassist make avol 100m
# vxprint -qrtg adg
[...]
sr FRA          ACTIVE  # Site records now show up - one per site
sr KEL          ACTIVE  # They are part of the disk group
[...]
v  avol         -            ENABLED  ACTIVE   204800   SITEREAD  -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg00-01     avol-01      adg00    528      204800   0         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg05-01     avol-02      adg05    528      204800   0         c1t1d5   ENA
pl avol-03      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg00-02     avol-03      adg00    0        528      LOG       c1t1d0   ENA
pl avol-04      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg05-02     avol-04      adg05    0        528      LOG       c1t1d5   ENA
```

The volume consists of two data plexes – one at each site (**adg00** is in **FRA** and **adg05** is in **KEL**) – and two **LOGONLY** plexes, i.e. DRLs, on the same disks as the data plexes. Now look at what site-specific information is added to the volume:

```
# vxprint -m -g adg | grep site
        siteconsistent=off
site FRA
site KEL
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=FRA
        site=KEL
        site=KEL
        site=KEL
        site=KEL
        site=KEL
        site=FRA
        site=KEL
        site=FRA
        site=KEL
        siteconsistent=on
        allsites=on
```

Apparently the volume inherits the **siteconsistent** attribute from the disk group, but for the volume it is set to **on**, while it remains **off** on the disk group itself. The volume also has another attribute, **allsites**, which is set to on. The disk group does not know such a attribute. So far the volume has been very small – a mere 100 MegaBytes – because this was just for demonstrating the principle. But when we grow the volume beyond the

boundaries of our 50GB-LUNs, we see that VxVM stops using its more primitive, or at least site-agnostic, allocation strategy and switch to site-aware allocation:

```
# vxassist -b growby avol 100g
# vxprint -qrtg adg
[...]
sr FRA          ACTIVE  # Site records now show up - one per site
sr KEL          ACTIVE  # They are part of the disk group
[...]
v  avol         -            ENABLED  SYNC     209920000 SITEREAD -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   209920000 CONCAT   -        RW
sd adg00-01     avol-01      adg00    528      122107120 0        c1t1d0   ENA
sd adg01-01     avol-01      adg01    0        87812880 122107120 c1t1d1   ENA
pl avol-02      avol         ENABLED  ACTIVE   209920000 CONCAT   -        RW
sd adg05-01     avol-02      adg05    528      122107120 0        c1t1d5   ENA
sd adg06-01     avol-02      adg06    0        87812880 122107120 c1t1d6   ENA
pl avol-03      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg00-02     avol-03      adg00    0        528      LOG       c1t1d0   ENA
pl avol-04      avol         ENABLED  ACTIVE   LOGONLY  CONCAT    -        RW
sd adg05-02     avol-04      adg05    0        528      LOG       c1t1d5   ENA
```

So this is finally a working procedure of volume-growth where all plexes remain confined to their respective enclosures (emulated by defining them as sites).

Additional Site–Specific Commands

There are more subcommand to vxdg for handling sites than just **vxdg addsite**. Anything that can be added can also be removed again, so there is **vxdg rmsite**, too. While this was obvious, there are two more commands that are designed to temporarily detach and then reattach a site. these are named accordingly (see below). Detaching a site disables all devices in the disk group that carry the tag of the given site. These devices then carry a new flag: **detached**. Reattaching the site removes the **detached** flag from the disks in the disk group that carry the appropriate site tag. Those volumes that were in use in the meantime must then be resynchronised by stopping and restarting them, or online by issuing the command **vxrecover**. Detaching a site can be useful in case of scheduled maintenance on an enclosure, path, or actual site. Here is the synopsis for the appropriate **vxdg** commands:

```
vxdg [-g diskgroup] [-o addmirror] addsite site
vxdg [-g diskgroup] [-o rmmirror] rmsite site
vxdg [-g diskgroup] [-f] detachsite site
vxdg [-g diskgroup] [-o overridessb] reattachsite site
```

Using Snapshots with Sites

If we wish to create snapshots of a volume that has been created with the default layout of a site-aware disk group (mirroring with DRL), then we need to prepare the volume first by adding a **DCO version 20** log to it (see more about this in the chapter about snapshots).

Unfortunately, we get an error message when we actually try that because the volume still has the DRL attached:

```
# vxsnap prepare avol
VxVM vxassist ERROR V-5-1-8807 volume avol has DRL log attached. Prepare disal-
lowed.
```

In this case we need to remove the DRL before we can prepare it for snapshotting. So we could do:

```
# vxassist remove log avol nlogs=0       # Removes logs until 0 logs are left
# vxsnap prepare avol
[...]
# [vxedit -rf rm avol]  # Get rid of the volume for the next exercise
```

Instead of removing the DRL after volume creation we could deviate from the default layout when we create the volume. From an empty disk group, the whole process of creating a volume cross-site and then snapshotting it is done like this:

```
# vxassist make avol 100m layout=mirror,nolog
# vxsnap prepare avol
# vxsnap addmir avol alloc=site:FRA
# vxprint -qrtg adg
[...]
sr FRA          ACTIVE  # Site records now show up - one per site
sr KEL          ACTIVE  # They are part of the disk group
[...]
v  avol         -            ENABLED  ACTIVE   204800   SITEREAD  -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg00-01     avol-01      adg00    0        204800   0         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg05-01     avol-02      adg05    0        204800   0         c1t1d5   ENA
pl avol-03      avol         ENABLED  SNAPDONE 204800   CONCAT    -        WO
sd adg02-01     avol-03      adg02    0        204800   0         c1t1d2   ENA
dc avol_dco     avol         avol_dcl
v  avol_dcl     -            ENABLED  ACTIVE   544      SITEREAD  -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg06-01     avol_dcl-01  adg06    0        544      0         c1t1d6   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-01     avol_dcl-02  adg01    0        544      0         c1t1d1   ENA
pl avol_dcl-03  avol_dcl     DISABLED DCOSNP   544      CONCAT    -        RW
sd adg03-01     avol_dcl-03  adg03    0        544      0         c1t1d3   ENA
```

Note that VxVM used disks from the specified location (**FRA**) both the disk for the plex that is in state **SNAPDONE** (i.e. the one destined to become the snapshot volume) and the disk for the currently disabled **DCO** plex that is in state **DCOSNP** (the plex that is destined to become the **DCO** volume for the snapshot volume). So now the rest is standard snapshot

syntax as it has been covered extensively in the appropriate chapter. For completeness, it is included here without comment:

```
# vxsnap make source=avol/new=snap_avol/plex=avol-03
# vxprint -qrtg adg
[...]
sr FRA          ACTIVE   # Site records now show up - one per site
sr KEL          ACTIVE   # They are part of the disk group
[...]
v  avol         -            ENABLED  ACTIVE   204800   SITEREAD  -        fsgen
pl avol-01      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg00-01     avol-01      adg00    0        204800   0         c1t1d0   ENA
pl avol-02      avol         ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg05-01     avol-02      adg05    0        204800   0         c1t1d5   ENA
dc avol_dco     avol         avol_dcl
v  avol_dcl     -            ENABLED  ACTIVE   544      SITEREAD  -        gen
pl avol_dcl-01  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg06-01     avol_dcl-01  adg06    0        544      0         c1t1d6   ENA
pl avol_dcl-02  avol_dcl     ENABLED  ACTIVE   544      CONCAT    -        RW
sd adg01-01     avol_dcl-02  adg01    0        544      0         c1t1d1   ENA
sp snap_avol_snp avol        avol_dco

v  snap_avol    -            ENABLED  ACTIVE   204800   ROUND     -        fsgen
pl avol-03      snap_avol    ENABLED  ACTIVE   204800   CONCAT    -        RW
sd adg02-01     avol-03      adg02    0        204800   0         c1t1d2   ENA
dc snap_avol_dco snap_avol   snap_avol_dcl
v  snap_avol_dcl -           ENABLED  ACTIVE   544      ROUND     -        gen
pl avol_dcl-03  snap_avol_dcl ENABLED ACTIVE   544      CONCAT    -        RW
sd adg03-01     avol_dcl-03  adg03    0        544      0         c1t1d3   ENA
sp avol_snp     snap_avol    snap_avol_dco
```

8.1.4 Summary

While there still is no real solution to growing **existing** mirrors across enclosures or sites we hope to have given a viable approach for most cases, i.e. the ones where new, empty LUNs are provided for the purpose of extending the existing volume. The best solution is to create new disk groups and build new volumes inside the new disk groups because in that case, VxVM can actually be made aware of what you are trying to achieve and will assist you rather than stand in your way.

But because this approach cannot always be used, resorting to the workaround for the time being is often preferred. By using or knowledge of the internal storage allocation scheme of VxVM we can still outmaneuver the VxVM allocator to use the disks the way we want, and make it allocate in a way that leaves no SPOFs.

8.2 Replication Across Data Centers

When the distance between the data centers exceeds a threshold, then latency becomes a dominant problem. Write I/Os that require synchronous acknowledgement from the storage array have to wait for a longer time before they return. Read I/Os incur the extra delay of having to travel several kilometers before they reach the destination host. This can make remote mirroring no longer viable in many cases, because of the many cases where immediate acknowledgement from the storage array is required.

8.2.1 Replication vs. Mirroring

When a volume is mirrored then all write I/Os are sent to all mirrors immediately, while read I/Os are satisfied from one of the mirrors, usually in a round-robin fashion. In VxVM 5.0 you can define the site for your own host as well as for your disks so that VxVM can prefer reading locally, but it will still write to all mirrors instantaneously. Mirrored volumes are considered to be consistent in all regular situations, and all sides are usually treated exactly equally.

If the distance between location becomes too great, or if bandwidth between the locations is too limited, then updating the remote mirror may take too long to guarantee trouble-free and high-performance operation. In these cases replication can be a good option.

Replication can be described as a time-lagged mirroring, with the remote side being allowed to become out of sync until it catches up. Because of this, and because of the general issues with bandwidth in WANs the remote site is write-only and accordingly, no read I/O can be serviced by the replica site to replicating host.

In case of a disaster that destroys the active host (or even data center) or renders it unusable a host at the remote site is employed to take over the replicated data (which has been read-only for that host so far). The remote hosts talks to the array, has the array turn around the direction of replication, or put it in failover (standalone) mode, in which this side of the replication couple becomes read-write. The remote host then proceeds to import and use the disk groups in the remote location just as the failed host used to with its set of the data (which has now turned read-only, or is unreachable).

Replication, if done correctly, can indeed bridge greater distances than mirroring can. But the ultimate barrier that both schemes face is their suitability to great distances of the low level block transfer protocol employed.

It is important to understand that the speed of light, although it is generally considered extremely fast, is a very real barrier to remote mirroring and remote replication. In order to understand the parameters and what they lead to, we need to become familiar with some of the fundamentals of physics. Don't worry, there will be no difficult mathematics involved, it is all relatively simple.

8.2.2 The Speed of Light and Latency

The speed of light is almost 300 thousand kilometers per second. This sure sounds comfortably fast for all applications except maybe interstellar travel, doesn't it?

Actually it is much too slow for computing across several kilometers of distance, unless great care is taken in using the right protocols. Why is light speed to slow for us?

Why the Speed of Light is Too Slow

The problem about light speed is that while 300,000 kilometers per second may seem like a whole lot, a gigahertz is also a whole lot. In order to find out how far a bit travels when it is transmitted via a 1 gigahertz link, we need to divide the 300,000 kilometers by the amount of cycles per second for one gigahertz. This gives the simple formula

```
300,000,000 (metres per second) : 1,000,000,000 (1 gigahertz) = 0.3
```

which yields the surprising result that a bit is only 0,3 metres (one foot) long at 1 gigahertz.

Efficiency: Why Wire Speeds are Mostly Irrelevant

The length of 1 foot per bit at 1 gigahertz is valid only in vacuum. In a fibre cable we need to divide this further by the refractive index of the medium, which is roughly 3/2, so actually a bit that runs along a 1 gigahertz fibre channel link is only about 20 centimeters (cm) long! At 2 gigahertz it would be only 10 cm, at 4 GHz it's 5 cm and so on. So basically with increasing transmission frequency the packet that we are transmitting is just getting smaller, but it isn't actually transferred any faster over the fibre. The first bit of a packet reaches its endpoint just as quickly at 1 GHz as it does at 4 GHz and as it would at 1 terahertz. The only speed advantage of using higher GHz on FC is that because the bits are shorter the **end** of the packet is reached earlier, and that more packets can be put on the fibre because a packet occupies a shorter range of the cable. Being able to put more packets onto the fibre per second is very important for increasing **total** bandwidth, especially in local SANs. But if the length of a packet is small relative to the distance over which the packet has to be transmitted, and especially if only one packet can be transferred at a time, then it really **does not make any difference** whether we are using 1 GHz or 4GHz fibre channel, or even just 100MHz Ethernet. Let us look at an example:

Using fibre channel arbitrated loop (a self-organizing topology and protocol from the early days of fibre channel) we transmit data to a remote site 25 kilometers away. Naturally, data is organized into blocks. In an arbitrated loop these blocks are usually transmitted in single transfers. Each transfer involves waiting for a token to come by, setting a flag in the token that says we would like to transfer data (requesting the bus), sending it off to the next station, which repeats the same, unaltered token unless it wants to send data itself (but we will assume the best case: that other node wants to send any data). After one whole round trip the token reappears at our host and it still contains our original request so we now know that our data transfer has been permitted by all participating nodes. We now put the data packet (which has been waiting the whole time for the token to come

back) onto the channel where it, too, makes a round trip: to the receiver and back. Once it is back the loop is freed for the next packet transfer etc.

How high is the efficiency of this transfer?

Well, that depends on how long the data packet is. Data packets in fibre channel are limited to 2112 bytes, which are 8 x 2112 bits which is equivalent to 8 x 2112 x 0,2 metres or roughly 3380 metres.

There is only one packet of ~3.4 km on a transport medium. The total distance travelled is (remember the token has to travel a full round trip before the data round-trip) four times the distance between sender and receiver (which we said was 25 km). So the total distance travelled for the transfer (without any overhead, of which there can actually be quite a lot) is 100km. 3,4 km of these 100 km are used for data. The efficiency is immediately obvious: it is merely 3.4%, because only 3.4 km out of 100km are used.

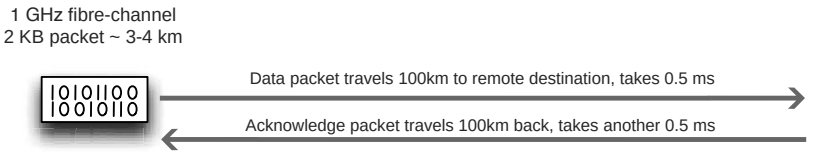

Figure 8-9: **There is practically no influence of wire speed on synchronous replication, as the total transfer time is dominated by the distance if that is more than a few times the packet size. At wire-speeds of about 200,000 km/sec, a block takes one millisecond to travel to a location 100 km away, and back. That is a very long time. It is in the range of hard disk access times, and we know that they are the limiting factor today!**

If you used 4GHz then you would have four times the theoretical bandwidth but because the bulk of the delay is not caused by the length of the packet but by the (unchanged) distance it does not really improve the situation at all. We get four times higher bandwidth but in turn we also get almost four times lower efficiency. The packet is not 3.4 km now, but only a quarter of that: 0.85 km. But the distance is still 100km. so we end up with 0.85% efficiency on the 4 GHz fibre channel.

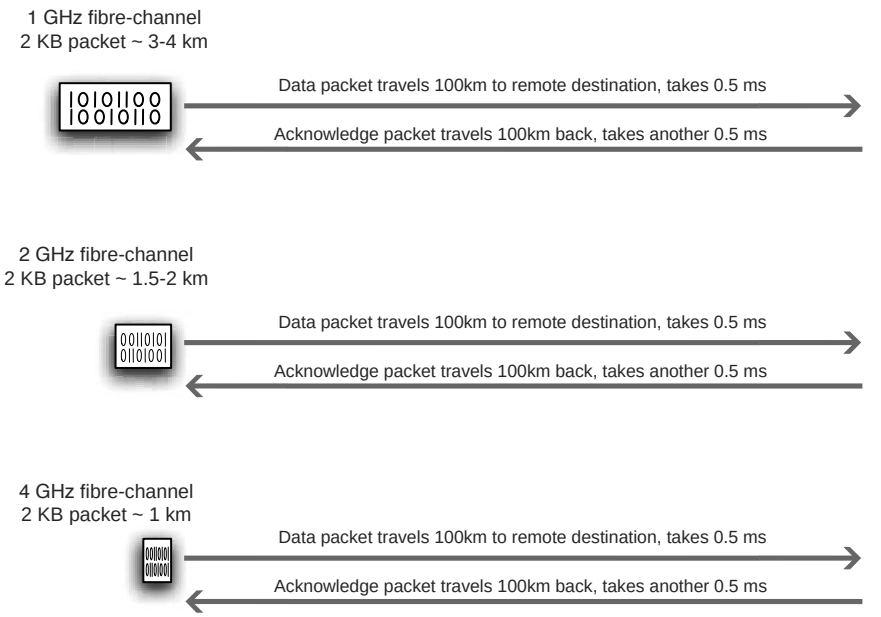

Figure 8-10: Using a higher wire speed means more packets can be transmitted per second, but that is purely because the packets are shorter, not because they travel more quickly. Light speed is the same for everyone!

Of course the distance is actually 100 km **plus 3.4 km** in the first case and 100 km **plus 0.85 km** in the second case, but that would have complicated the formula and the small total difference is not worth the trouble.

Why Protocols Make All the Difference

The low efficiency comes almost solely from the fact that in the example given above there is only a single packet on the wire at one time. This is a dictated by the protocol we chose. For instance, switched fabric fibre channel architectures do allow multiple packets under way at any time, as long as the buffer credits do not run out.

If we had used a switched fabric in the above example the numbers would look much more friendly. The sending host would initiate block transfers without having to send a token out and wait for it to return first. And it would not have to stop transmitting new blocks until either of the following conditions occur:

- Two fabric nodes run out of buffer credits
- The host application's protocol requires it to wait for an acknowledgement of some sort.
- The host runs out of data to send.

So in the case of 1 GHz transport medium we could just line up packet after packet and will the whole distance with data. Obviously, with a higher medium speed (like 4 GHz) because of the shorter length of each packet we could actually fit in four times as many packets as we could with 1 GHz without losing efficiency.

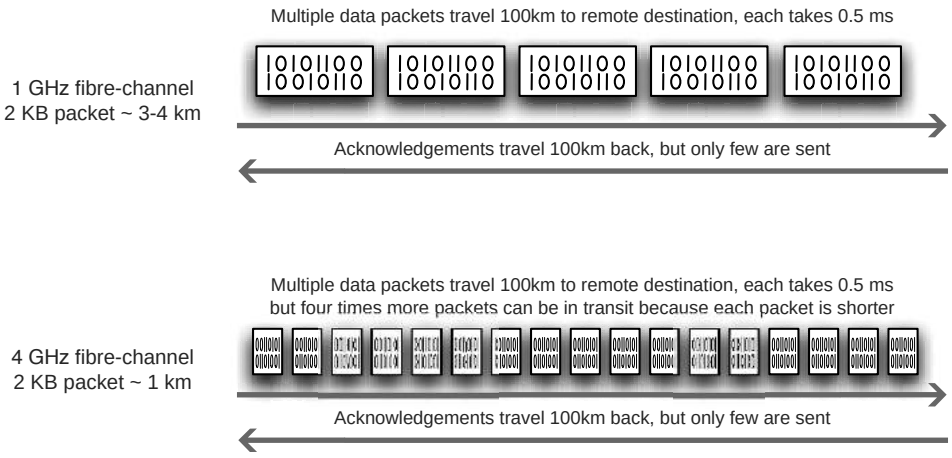

Figure 8-11: Higher frequencies help only in putting more data packets on the wire at the same time. the actual travel time is unchanged. In order to get any kind of performance out of long-distance replication it is mandatory to use asynchronous protocols which do not require immediate acknowledgement from the remote storage. But asynchronous replication cannot be done by the storage array alone and still be consistent: the operating system must be involved (see below).

Obviously it all depends on which protocol you choose and how you configure the basic transport parameters (like buffer credits, which we will discuss in detail in the technical deep dive section)

8.2.3 Replication Using Storage Array Logic

Most storage array manufacturers offer some way of replicating data in a synchronous or asynchronous way. Many times they will offer the synchronous variant for short ranges and the asynchronous variant for long range. When asked, they may occasionally agree that asynchronous replication will not create a consistent, usable copy at the remote side, but typically the importance of this is down played enough that it is ultimately used and goes into production. When disaster strikes, and only 70% of the applications at the remote site actually come up this is often hailed as a great success rather than the utter failure that it

actually is. Let us look a little deeper into the two variants of replication: synchronous and asynchronous, and discuss their relative merits.

Asynchronous Replication Using Storage Array Logic

This kind of replication can only be used for replicating data which does not require transaction fidelity. It is therefore in order to use asynchronous replication using storage array logic for file system data and for other, usually low profile data. If contents of databases are replicated using asynchronous replication from storage array to remote storage array without the participation of an operating system resident driver to coordinate the transfers, then the contents on the remote side are practically guaranteed to be unusable after an actual disaster has happened on the primary side. This is because of the way asynchronous replication works. In the Symmetrix SRDF/A implementation by EMC, for instance, a checkpoint is taken at regular intervals (e.g. 30 seconds) and all tracks that have changed between the current checkpoint and the previous checkpoint are transferred as quickly as possible. This means the changed data is generally transferred out of order and, if there were multiple changes to the same region, only the last of those changes is transferred. The only conceivable way in which storage arrays might handle asynchronous transfers correctly is if every single write I/O was sequentially numbered and then transferred in exactly this sequence to the remote storage array. However, this would introduce a single bottleneck into an otherwise highly parallel architecture and it is doubtful if any array vendors have implemented such a scheme. Apart from that, transmission error recovery would be a big challenge.

So generally speaking, in asynchronous mode, the storage array's internal logic simply has no means of knowing which blocks belong together to form a single, atomic transaction. So there is a high probability of transactions being literally ripped apart: one part is transferred while the other one is not transferred until the next checkpoint, which may be 30 seconds away.

This obviously leads to problems with transactional data, so you cannot use asynchronous storage array-based replication with transactional data. It will turn out to be unusable once the remote host tries to start the database after a disaster.

Note that **the corruption never happens if the direction of replication is voluntarily switched during the usual disaster recovery (DR) tests**. In these tests the operators are of course unwilling to actually simulate a hard shut down of the storage array because those are usually shared with many other, often productive, servers. So they just shut down the LUNs in software, or induce a replication direction switch using command line or GUI tools of the storage array. But you **must** be aware that this is not really a DR test! It may serve a vendor very well as a demonstration that their array works great, but it will not help you in case of an actual disaster. It is pure smoke-screening! If an array actually fails and you were using asynchronous replication then your transactional data **will be** broken!

Synchronous Replication Using Storage Array Logic

There is a "nice try" kind of approach of the array vendors to do replication of transactional data right, which is called synchronous replication, but it is extremely sensitive to latency induced by distance. Basically, every single block must be acknowledged from the remote site before the next block can be sent, i.e. there can only be one block under way

at any one time per transfer channel. That is why often several channels are used between storage arrays; that allows for some degree of concurrency (although it does not help consistency).

The reason why this has to be is not at all easy to understand. We will try to give you a coherent explanation, but the matter is rather complex. For this reason, we will just pick a single example of a transaction that will fail using a storage array, and allow ourselves to deduct from this example that they are flawed in principle **even if they tried to maintain write order** (which normally they don't).

Imagine the following setup:

- A storage array that replicates each individual write in the same write order, but does not wait for acknowledgements, but instead keeps grinding along;
- Host A: a database machine running some kind of financial database application that is connected (of course) to a network of other machines interacting with host A;
- Host B: the remote replica machine for host A;
- Host X: a machine interacting remotely via an outside network with host A, feeding financial transactions to host A.

The problem is **write order fidelity**, as you will see in the following example:

Host X has just fed a large financial transaction to host A that changed, let's say, first block 5000 and then block 6000 in a volume. Host A has processed the transaction locally, and has committed it to replicated storage. The storage system is busily replicating the data (synchronously) to the remote site(s), but is not waiting for he acknowledgement for block 5000 before sending block 6000. When the remote site acknowledges the receipt of both blocks, that data is deemed secure and host A acknowledges the successful transaction to host X. This means that e.g. some large amount of money has just been received or sent away, or that some large amount of stock has just been bought or sold.

Host A is now receiving the next transaction, which changes first block 7000 and then block 6000 of the same volume. It commits the transaction to local storage, and the storage array begins replicating the two blocks over to the remote site, but does not wait for individual acknowledgements before transferring each following block. It sends block 7000 on its way and block 6000 immediately thereafter. Block 7000 is not acknowledged because of a bad checksum or some other trivial error. Block 6000, however, is successfully transferred. After successful transfer of block 6000, the data center is destroyed by a disaster. Because it is not integrated into the operating system, the storage array's replication mechanism has no means of knowing that the changes to block 5000 and 6000 belong together, as well as the changes to block 6000 and 7000. For this reason, the remote storage array will indeed use the overwritten block 6000 in conjunction with the previously acknowledged block 5000 when host B comes up to take over after host A's catastrophic failure. But obviously this would lead to trouble! There is a full, possibly very important or very expensive transaction, that is half covered by a newer transaction, which has not yet completed and which may be rolled back, or may be reissued on the remote host, leading to write I/O on possibly quite different portions of the volume. The result is that the transaction is either not done at all (although it had been acknowledged to the business partner), or executed twice (although only one execution had been requested). Additionally, the database will have trouble starting up and running because of logical inconsistencies in the table spaced. In any case, there is a principal problem with the fact that a storage array has no intrinsic

knowledge about which individual block writes belong to the same I/O operation, while an operating system-resident device driver does have this intrinsic knowledge. To illustrate the point: if you think the above is a very constructed example that is unlikely to happen in the real world, then think again: When there are several thousand write I/Os outstanding on a storage array the probability for such a scenario becomes several thousand times more likely! Also, please keep in mind that a disaster recovery solution only ever gets to do its work in such extraordinary situations, and it better work really well then. To emphasize the degree to which a user of current storage-array based replication is exposed, please ponder at least the following points:

Using anything but a flat concat volume will completely destroy the connection between you storage array's view of the data and your operating system's view of the data. For instance, a write I/O to a striped volume looks to the storage array like several completely independent write I/Os, each going to a different LUN (actually just columns of the same striped volume and therefore likely to be highly interdependent). A mirrored stripe is not any better, it is even worse because now there are two times the number of I/Os, whose interdependence is unclear.

Remembering Moore's law and current throughput numbers, you can estimate the number of outstanding I/O operations at any time. This number is typically a high one, not a low one. It is also increasing year over year. It is not at all unusual, but in fact very likely, that in the case of an actual disaster many transactions are indeed corrupted and/or rendered unusable **after** they have already been acknowledged to your business partners. This could lead to serious risks to the business. It is important to know that when picking the replication method. We know it is much easier to pass the job on to the SAN department, who claim to know what they are doing, and to ignore the risk because it's no longer one's own bailiwick. But it may not be such a good idea after all if the survival of your company is at stake.

8.2.4 Replication Using Kernel Mode Logic

If we were to use an operating system based solution then the information about coherency or interdependency of data blocks in not yet lost. Replication therefore has a chance to actually work, and indeed work both quickly and reliably. There is a very well working solution embedded in VxVM which is called VVR (Veritas Volume Replicator). This feature is already part of VxVM; it is activated by purchasing the appropriate license and adding the license key to the system.

Overview of VVR

VVR works by replicating volume changes (i.e. write I/O) via a normal IP connection to a similar volume on the remote hosts. A remote disk group must be imported on the remote host, and its volumes must be started, but they must not be mounted because the volume's data can change at the block level due to the data being replicated from the active site. A multitude of destination hosts is allowed, although using just one destination host is the usual setup. The IP address of the source and destination, along with some other meta–info like replication type etc. is contained in a data structure called an **RLINK**, which stands for **remote link**. The RLINK is created in the disk group that contains the volumes which

are to be replicated. It also contains other global replication information pertaining to the disk group, such as the type of synchronisation and type of error handling, the state of the connection and such.

The actual data that must be replicated are not held in the RLINK itself (there is no persistent memory in an RLINK), but rather in a separate log volume, called the **SRL** (for Serial Replication Log) which is attached to the original volume. a similar log volume is attached to the remote, recipient volume. Any write-I/O is first persisted to the log volume, and a descriptor with information about the size and type of I/O is stored along with the data. While the data is being persisted to the log it is also transmitted to the recipient host(s), where it is likewise stored to the log volume. Once the data is written to the log it cannot be lost and is therefore considered safe. It will be written to the actual volume as soon as possible, but if anything happened in the meantime, the log will simply be replayed (both locally and remotely) and therefore it does not really matter so much whether the data is in the log volume or in the data volume.

The log volume is organized as a circular buffer: In the event of an extended downtime of the IP connection, or in case the bandwidth is overused for an extended time, the buffer will eventually fill up so it must be sized intelligently. But even in the event of a log volume filling up VVR will still remember all of the initial I/Os and replay them in order, while using a persistent log bitmap to remember the regions that have changed since the time that the log volume filled up.

It is in this final, exceptional state that VVR behaves only as well as a storage array replication mechanism: it forgets about write order and write coherence and block interdependency and just flags regions to replay later, when bandwidth is available again.

In the regular state VVR behaves much better that storage array based replication because it transmits and replays each I/O as a single entity, maintaining what is called write order fidelity. No matter if the I/O was small or huge it will always be replicated atomically, either in total or not at all. It will also be transmitted in the correct order, and in the case of subsequent writes to the same block, all writes are transmitted instead of just the most recent one. That is what all storage array based replication mechanisms fail to do correctly, because they have no chance of knowing about the write order in which the host has issued the I/Os. All they know about is independent blocks.

Asynchronous Replication Using VVR

When VVR is set up for a disk group with an RLINK indicating **asynchronous** replication, then all write I/O is acknowledged to the host as soon as the data is persisted to the local log. Loss of data is now considered impossible (SRL logs are usually mirrored) unless a disaster destroys the site. Data is always replicated as quickly as possible via the network, but the latency incurred from waiting for a remote machine to reply to a replicated I/O is saved. Because the SRL log is sequential, the local log write I/O is therefore usually even faster (and therefore subsequent I/Os may be dispatched more rapidly) than in the case of actually writing to the data volume, although with today's storage arrays the difference is diminishing. If disaster strikes, the remote site may not have received all transactions that have been locally acknowledged, but at least the database table space is not corrupt and all transactions are either complete or do not exist at all, rather than being ripped apart because a storage array knows nothing about the interdependence of blocks.

Synchronous Replication Using VVR

When VVR is set up for a disk group with an RLINK indicating **synchronous** replication, then all write I/O is acknowledged to the host only when the data is acknowledged by the remote host. Loss of data is now considered impossible even if a disaster destroys the site, as it is unlikely that the remote site fails at the same time. The latency from waiting for a remote machine to reply to the replicated I/O is incurred. But because VVR transmits all write I/Os transactionally and guarantees write order fidelity, there is no general need to wait for an acknowledgement of the first block before transmitting the next, the way that storage arrays need to in the synchronous transfer case. VVR can instead put as many transactions on the wire as possible and saturate the remote connection, provided the database and the application that drives it is sufficiently parallelized.

8.3 Estimating Replication Speed

In order to calculate the maximum speed of a synchronous replication process over any significant distance it is important to first gain some understanding of the physical principles that underlie the process as well as a little more about transactions and storage arrays. We have learned above that storage array replication has no means of knowing which I/Os (from the storage array's perspective) belong together, so the array must transfer all blocks strictly in order or it will break the transaction paradigm (and you do **not** want that)! Now here comes another stepping stone to high-speed synchronous storage array replication: There is the possibility that the transfer of a block is unsuccessful and must be repeated, i.e. a block that had been sent before must be sent again because there was some kind of error happening on the way.

If it were the case that a storage array which synchronously replicates its LUNs replicated write I/Os – in the correct order because they would be useless otherwise – at full speed without waiting for an acknowledgement from the remote storage array, then the following might happen:

A block that had been transmitted towards the beginning turns out to have a bad checksum, so it must be resent by the originating storage array, The recipient storage array accordingly sends a message to the active storage array requesting a retransmission of said block. However, before that request is fulfilled, disaster strikes and the request is lost. The recipient site is now stuck with a stream of transactional data which may already have been persisted to disk, but of which a part is missing! This is a completely inacceptable situation! It has falsified our data; the database might not start up, or in could be corrupt in such a way that some large financial transaction was undone or repeated.

It would be theoretically possible to implement a protocol on the storage array level that kept track of not only write sequence, but also transmission state, and that would persist only those blocks to the remote site that have been successfully received, and persist them in the right order. Then, when a block must be retransmitted, the protocol on the recipient would no longer continue persisting data to disk until the block has been successfully resent, and then it would persist all of the blocks it has received so far – again, in the right order. But that protocol is loaded with overhead and it is not immediately clear what cleanup procedures would be necessary in the case of multiple transmission and retransmission errors etc. In short, as far as we know no array vendor has so far come up with a protocol that actually works synchronously without having to wait for acknowledgement from the remote site before sending the next block for the same LUN or LUN group.

We can therefore safely assume that a storage array will only put one write I/O per LUN and per connection on the wire at any one time.

Given this background, one may ask what the latency and the resulting bandwidth my turn out to be. The answer to this question is usually quite devastating, at least when replication is done across distances that are worthwhile, i.e. in the range of tens or hundreds of kilometers or more. This is where the speed of light really becomes an unexpected

bottleneck, and you will see – in a few simple calculations – why this is the case.

Basic Physical Constants and Laws

The speed of light is just a little below 300 000 km/sec in vacuum. In a glass fibre the speed must be adjusted by the refractive index of the medium, yielding about two-thirds that, or 200 000 km/sec. While this sounds like a lot, hundreds of thousands of kilometers every second, keep in mind how many cycles a processor running at a measly 1 GHz goes through in a second: 1 billion! So on one hand there is light passing us at two hundred million metres per second through a fibre link, while the processor works so fast that it executes several cycles for each metre of the passing stream of light.

This comparison may seem a little weird at first, but you will understand why we chose it: It has enabled us to make a useful comparison between cycle speed and distance! If you consider an optical light module (OLM, or GBIC) that modulates its bit stream at one GHz onto the laser light for transfer to a remote site, then it will actually create bits that have a length of one-fifth of a metre. This is because at one GHz there are five cycles executed (i.e. five bits processed) while light is travelling one metre. So in effect, one can say that a bit in a fibre-channel has a length of 20cm.

Now comes the surprising and depressing part: if at one GHz a bit is 20cm, then a byte is eight times that (160cm), and a fibre-channel packet is 2112 times the length of a byte. One byte at 1.6 metres times 2112 bytes per fibre-channel packet is just shy of 3.4 km.

If you are using a synchronous storage array replication protocol and your remote site happens to be 34 km away, and you remember the fact that a packet must first be acknowledged before its successor can be sent (see The Full Battleship above for an explanation of this) then that means that a packet must in effect travel twice the 34 km (once forth, once back as an ACK), and the data is only 3.4 km long. This means your theoretical maximum speed is 3.4 km divided by 68 km, which is exactly 5% of the burst transfer rate.

And increasing the channel speed does not help either: It does not make the packet cross the distance any faster, it just shortens the length of the packet. So while the burst speed goes up, your efficiency drops by almost exactly the same amount!

There is no way to speed this up but to use a protocol that does not need to wait for acknowledgement, and such a protocol is inherently instable when implemented only inside storage arrays because they do not know about write order and block interdependence.

However, there is a workaround that alleviates the situation to some extent. It does not go all the way to deliver a perfect solution, but it at least abates some of the worst problems: the use of so-called Buffer-to-Buffer Credits, or more shortly Buffer Credits.

Buffer Credits vs. TCP–type Sliding Windows

We have outlined above that acknowledging every single packet on the application layer is not an option when crossing large distances. But large distances are a requirement for many data centers, so there must be a way to bridge them more efficiently. That is what the system of using buffer credits has been developed for. It is a little like TCP/IP's sliding window protocol, but is both more efficient and more error-prone than that. We will refer to the TCP sliding window occasionally for comparison, so let's reiterate shortly what it does: At the initiation of a connection the participants negotiate how many packets will be allowed to be sent without waiting for acknowledgement from the recipient. The

number of packets is called the size of the sliding window. A solid value would be 64, for instance. During communication the sender numbers the packets and sends them away until the sliding window is exhausted. Normally the recipient will acknowledge packets as they arrive. But for performance reasons (all this must be handled in software inside the TCP stack) the recipient batches the acknowledgements, so that the sender will see, for example, an ACK after (let's say) every ten packets. The ACK packet contains the sequence number of the last packet that was received, so that the sender, in turn, can now send up to the sliding window size above that packet sequence number.

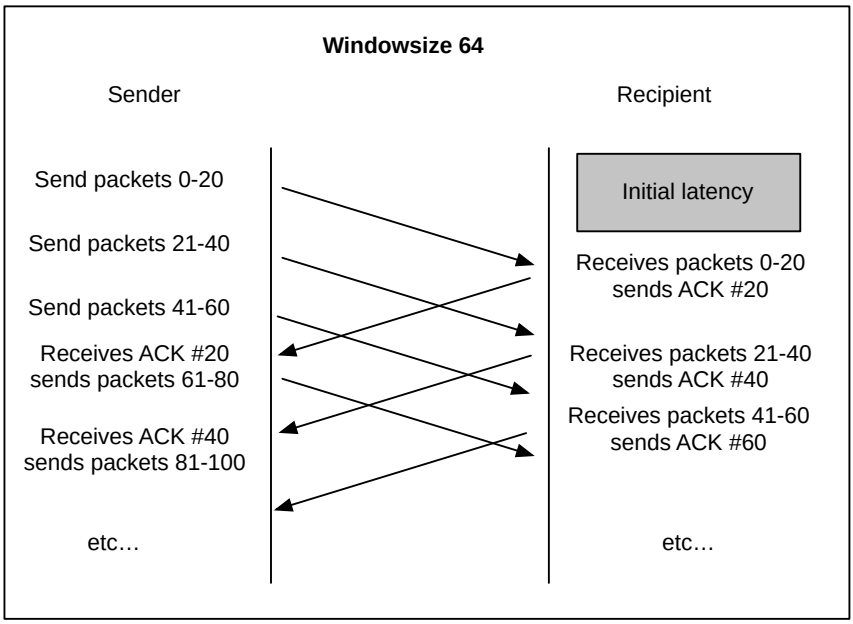

Figure 8-12: Sliding windows as used in TCP/IP are endpoint-to-endpoint protocol features. They are slower than node-to-node protocol features, but provide reliable end-to-end communication.

This protocol is very efficient for large transfers, because once it has overcome its initial latency the data can be transmitted at close to wire speed. The braking factor here is the protocol processing overhead, which may be OK in end-to-end communication, but would be prohibitive to have inside the switching nodes of a SAN fabric. Too much state to keep track of, too much processing and interpreting sequence numbers in ACK packets etc.

Fibre-channel uses a more simplistic approach, because it must be implemented in hardware. In the FC approach, every block is acknowledged, but because it must be done as rapidly as possible (the ports need to be freed to handle "real" data) the acknowledgement only signals reception of "one block". No sequence number is transmitted in the ACK; that

would require inspection of the packet contents, which would cause too much overhead.

Acknowledgements are called R_RDY for "receiver ready" in FC.

In order to enable the sender to put a number of packets on the wire at once, the ports of a fabric are given so-called buffer credits. The remaining buffer credits are decremented every time a packet is sent, and incremented every time an ACK is received. So a sender can send as many packets as he own buffer credits; eight buffer credits is a typical number today. Going back to our calculation above, this correlates to a stretch of eight times 3.4km at one GHz. But because the ACK must travel the same distance, the actual distance must be halved. So we end up at roughly 14km distance that could be traversed at full speed with eight buffer credits. Longer distances require more buffer credits and sometimes extended licenses from the FC vendors. The initial latency is the same as in TCP.

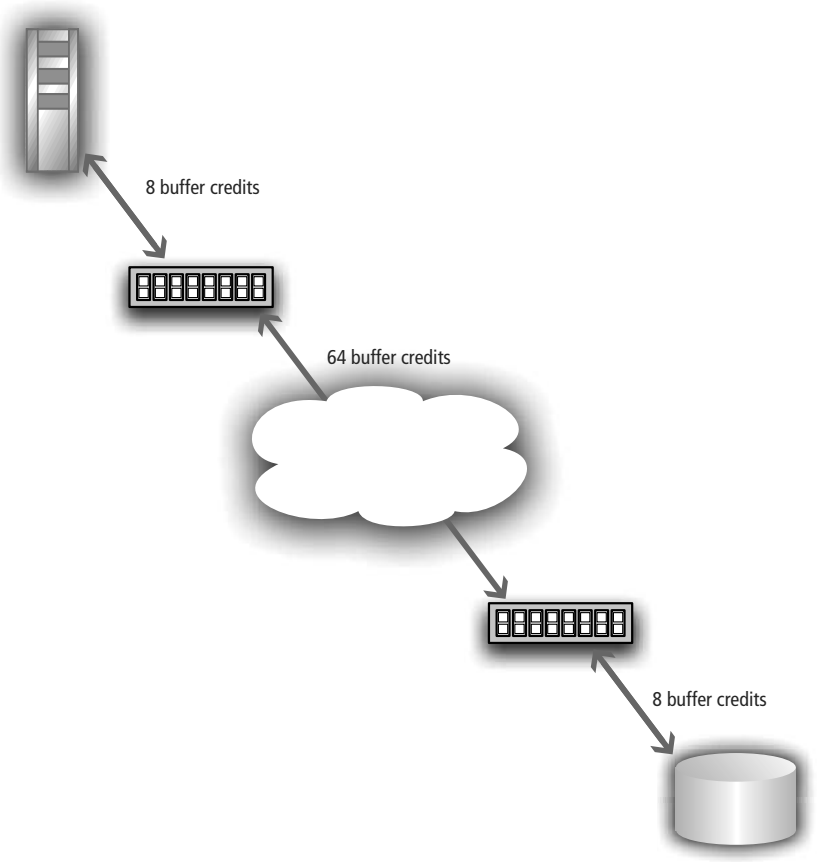

Figure 8-13: **This picture shows a wide-area connection between host and disk. The connection consists of several legs: host HBA to SAN switch, SAN switch to remote SAN switch, and remote SAN switch to storage array port. Buffer credits are assigned port-to-port rather than endpoint-to-endpoint. They can be implemented with very little overhead, but do not offer good protection because the endpoints are not notified when credits are lost between two intermediate ports.**

With this fine protocol, what can go wrong? Actually, a lot can go wrong! For instance, let's look at a SCSI I/O transmitted over an FC link. The total transfer is supposed to be 1 MB, which will be appropriately split into 512 packets of 2 KB each. Within any infrastructure there is a certain probability that individual packets get lost. That probability is normally very low, so that we do not generally need to think about it. But let's, for the sake of the argument, assume that one of the 512 packets gets lost and is therefore not acknowledged to the sender. The result is two-fold:

1) On the sender side, the available buffer credits are reduced by one. Because only 511 ACKs were sent instead of 512, the sender has successively decremented the buffer credits 512 times, but only incremented it 511 times. I.e. one buffer credit is permanently lost. This leads to reduced performance because one less packet can be under way at once. (This problem can be remedied under some circumstances)
2) On the receiver side, the missing block is not detected as such by the FC layer. Because buffer credits are maintained only as a counter, the identity of lost packets is unknown, leaving error recovery to the higher protocol layers. Buffer credits (BCs) are defined for port-to-port connections in a SAN, not for endpoint-to-endpoint connections. Imagine a SAN where the host is talking to the FC switch using 8 BCs, the switch is talking to a remote switch using 64 BCs, and the remote switch is talking to the LUN using 8 BCs. I.e. there is a switch-to-switch line in the middle of the communication channel. In this case neither the host nor the array serving the LUN will detect if a block was lost between the two switches. What they will detect is that the SCSI transfer did not complete. When will they detect that? They will detect it when the expected amount of data has not been transmitted after the usual timeout, which is typically in the range of one minute!

To sum it up: Losing packets (or R_RDYs for that matter) is a very bad thing in SANs because they permanently decrement your buffer credits and they cause long time-outs and retransmits on the SCSI layer. It's a good thing that FC is so reliable and does not lose packets very often.

Or does it? There is a non-imaginary customer that uses a 30 km FC connection between locations. The admins for this customer have to reset the buffer credits several times a day because too many packets get lost and performance is severely reduced. The problem is that over greater distances FC is not very reliable indeed. It loses packets at a much higher rate than inside a typical data center.

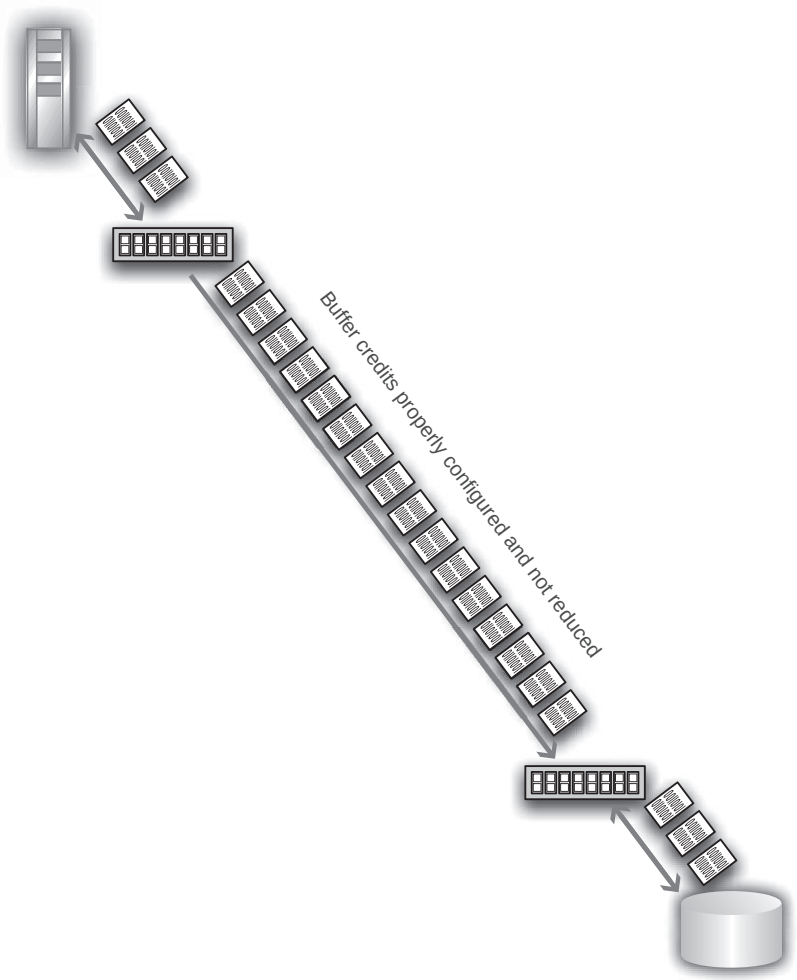

Figure 8-14: If buffer credits are configured for the appropriate number of packets that fit onto each leg, then bandwidth can be saturated.

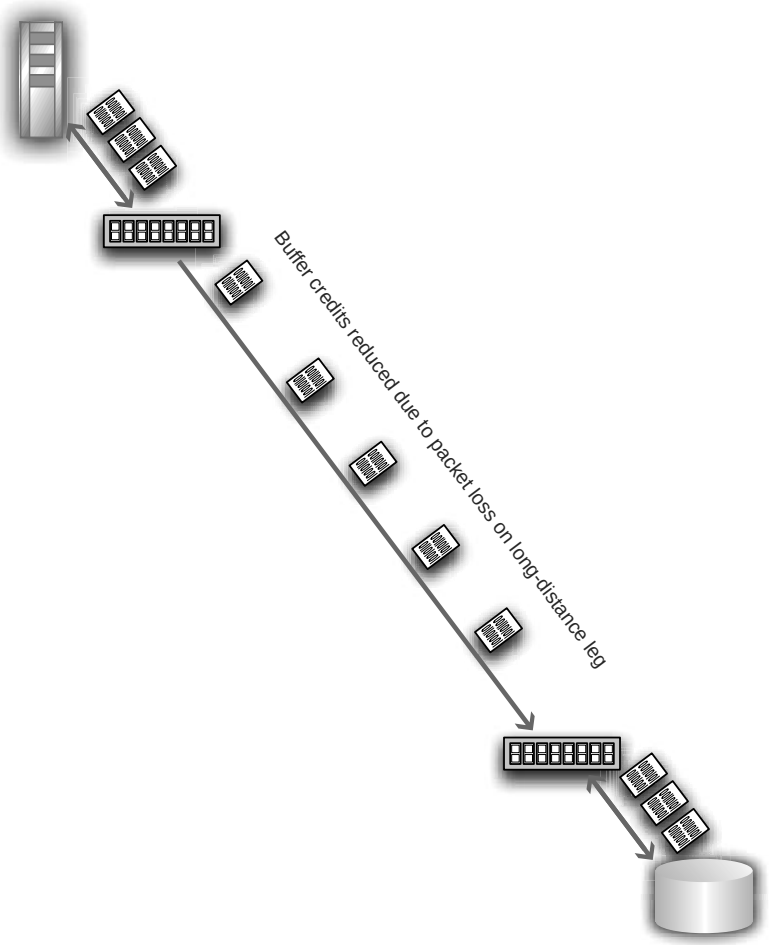

Figure 8-15: **If buffer credits are lost in an intermediate leg, the endpoints will not notice. Bandwidth on the leg that is affected by the reduced buffer credit count is reduced because less packets can be put onto the wire before buffer credits are exhausted.**

We hinted towards a remedy for permanently losing buffer credits above. The FC standard actually has a credit recovery mechanism for lost R_RDYs and lost frames. But both ports on the link must support that standard before it can be enabled. The protocol for lost buffer credit recovery consists of special primitive frames that are sent after every n-th packet. The number of packets between two primitives is a fixed power of two that can be set between 2^0 and 2^15. If a port receives such a primitive frame it checks if the appropriate number of R_RDYs (or packets) have been sent, and if not, transmits them or simply

increases its own buffer credits accordingly. It thus fixes the first part of the problem: the persistence of decreased buffer credits. It does not fix the SCSI timeout problem. And the first part is also not fixed unless compatible components are deployed, which (as you can see from the example customer) is not always the case. The worst problem may be the one that occurs when such special frame gets lost (low probability, but possibly high impact); we have not investigated any further into this matter yet.

More on buffer credits and buffer credit extenders for long-distance traffic can be found in several US patents, one of the more readable ones is 7352701. It can be accessed from many public patent access sites, e.g. here (check out the date of issue: it is not a joke):

```
http://www.patentstorm.us/patents/7352701/description.html
```