

PROUHD : RAID for the end-user.

Pierre Vignéras
pierre@vignerass.name



April 14, 2010

Résumé

RAID has still not been adopted by most end-users despite its inherent quality such as performance and reliability. Reasons such as complexity of RAID technology (levels, hard/soft), set-up, or support may be given. We believe the main reason is that most end-users own a vast amount of heterogeneous storage devices (USB stick, IDE/SATA/SCSI internal/external hard drives, SD/XD Card, SSD, ...), and that RAID-based systems are mostly designed for homogenous (in size and technology) hard disks. Therefore, there is currently no storage solution that manages heterogeneous storage devices efficiently.

In this article, we propose such a solution and we call it PROUHD (Pool of RAID Over User Heterogeneous Devices). This solution supports heterogeneous (in size and technology) storage devices, maximizes the available storage space consumption, is tolerant to device failure up to a customizable degree, still makes automatic addition, removal and replacement of storage devices possible and remains performant in the face of average end-user workflow.

Although this article makes some references to Linux, the algorithms described are independent of the operating system and thus may be implemented on any of them.

Copyrights

This document is licensed under a *Creative Commons Attribution-Share Alike 2.0 France License*. Please, see for details : <http://creativecommons.org/licenses/by-sa/2.0/>

Disclaimer

The information contained in this document is for general information purposes only. The information is provided by Pierre Vignéras and while I endeavor to keep the information up to date and correct, I make no representations or warranties of any kind, express or implied, about the completeness, accuracy, reliability, suitability or availability with respect to the document or the information, products, services, or related graphics contained in the document for any purpose.

Any reliance you place on such information is therefore strictly at your own risk. In no event I will be liable for any loss or damage including without limitation, indirect or consequential loss or damage, or any loss or damage whatsoever arising from loss of data or profits arising out of, or in connection with, the use of this document.

Through this document you are able to link to other documents which are not under the control of Pierre Vignéras. I have no control over the nature, content and availability of those sites. The inclusion of any links does not necessarily imply a recommendation or endorse the views expressed within them.

Table des matières

1 Introduction	3
2 Algorithm	3
2.1 Description	3
2.2 Analysis	5
2.3 Implementation (layout-disks)	8
2.4 Performance	8
3 Partitionning drives	9
4 Handling Disk Failure	9
4.1 Replacement Procedure	11
4.1.1 Replacing a failed device with a same-size one.	11
4.1.2 Replacing a failed device with a larger one.	11
4.1.3 Replacing a failed drive with a smaller one	14
4.1.4 RAID array reconstruction	17
5 Adding/removing a device to/from a PROUHD	17
6 Forecasting: Storage Box for Average End-Users	18
7 Alternatives	19
8 Questions, Comments & Suggestions	19
9 Note	19
10 Acknowledgment	19

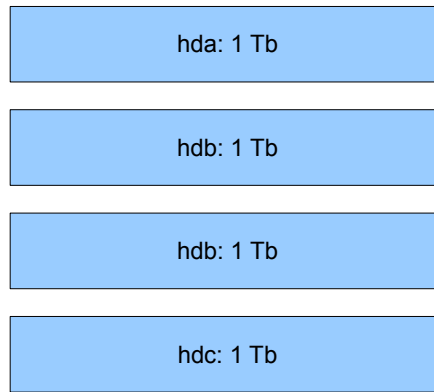


FIGURE 1 – Stacking storage devices (same size, ideal RAID case).

1 Introduction

Whereas RAID¹ has been massively adopted by the industry, it is still not common on end-users desktop. Complexity of RAID system might be one reason... among many others. Actually, in a state-of-the-art data center, the storage is designed according to some requirements (the "top-bottom" approach already discussed in a previous article²). Therefore, from a RAID perspective, the storage is usually composed of a pool of disks of same size and characteristics including spares³. The focus is often on performance. The global storage capacity is usually not a big deal.

The average end-user case is rather different in that their global storage capacity is composed of various storage devices such as :

- Hard drives (internal IDE, internal/external SATA, external USB, external Firewire) ;
- USB Sticks ;
- Flash Memory such as SDCard, XDCard, ... ;
- SSD.

On the opposite, performance is not the big deal for the end-user : most usage does not require very high throughput. Cost and capacity are main important factors along with ease of use. By the way, the end-user does not usually have any spare devices.

We propose in this paper an algorithm for disk layout using (software) RAID that has the following characteristics :

- it supports heterogeneous storage devices (size and technology) ;
- it maximizes storage space ;
- it is tolerant to device failure up to a certain degree that depends on the number of available devices and on the RAID level chosen ;
- it still makes automatic addition, removal and replacement of storage devices possible under certain conditions ;
- it remains performant in the face of average end-user workflow.

2 Algorithm

2.1 Description

Conceptually, we first stack storage devices one over the other as shown in figure 1.

1. For an introduction on RAID technology, please refer to online articles such as :

http://en.wikipedia.org/wiki/Standard_RAID_levels

2. <http://www.vigneras.org/pierre/wp/2009/07/21/choosing-the-right-file-system-layout-under-linux/>

3. By the way, since similar disks may fail at similar time, it may be better to create storage pools from disks of different model or even vendor.

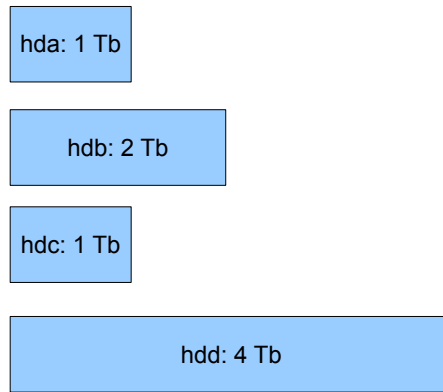


FIGURE 2 – Stacking storage devices (different size = usual end-user case).

On that example with $n = 4$ devices, each of capacity $c = 1 Tb$ (terabytes), we end up with a global storage capacity of $G = n * c = 4 Tb$. From that global storage space, using RAID, you can get :

- a 4 Tb ($n * c$) virtual storage devices (called PV for Physical Volume⁴ in the following) using RAID0 (level 0), but then you have no fault tolerancy (if a physical device fail, the whole virtual device is lost).
- a 1 Tb (c) PV using RAID1 ; in that case, you have a fault tolerancy degree of 3 (the PV remains valid in the face of 3 drives failure, and this is the maximum).
- a 3 Tb ($(n - 1) * c$) PV using RAID5 ; in that case, you have a fault tolerancy degree of 1 ;
- a 2 Tb ($M * c$) PV using RAID10 ; it that case, the fault tolerancy degree is also 1⁵ (M is the number of mirrored sets, 2 in our case).

The previous example hardly represents a real (end-user) case. Figure 2 represents such a scenario, with 4 disks also (though listed capacities does not represent common use cases, they ease mental capacity calculation for the algorithm description). In this case, we face $n = 4$ devices d , of respective capacity c_d : 1 Tb, 2 Tb, 1 Tb, and 4 Tb. Hence the global storage capacity is : $G = \sum c_d = 1 + 2 + 1 + 4 = 8 Tb$. Since traditional RAID array requires same device size, in that case, the minimum device capacity is used : $c_{min} = 1 Tb$. Therefore, we can have :

- 4 Tb, using RAID0 ;
- 1 Tb, using RAID1 ;
- 3 Tb, using RAID5 ;
- 2 Tb, using RAID10.

Thus, exactly the same possibilities than in the previous example. The main difference however, is the wasted storage space — defined as the storage space unused from each disk neither for storage nor for fault tolerancy⁶.

In our example, the 1 Tb capacity of both devices hda and hdc are fortunately fully used. But only 1 Tb out of 2 Tb of device hdb and 1 Tb out of 4 Tb of device hdd is really used. Therefore in this case, the wasted storage space is given by the formula :

$$W = \sum_d (c_d - c_{min}) = (1 - 1) + (2 - 1) + (1 - 1) + (4 - 1) = 4 Tb$$

In this example, $W = 4 Tb$ out of $G = 8 Tb$, *i.e.* 50% of the global storage space is actually unused. For an end-user, such an amount of wasted space is definitely an argument against using RAID, despite all the other advantages RAID provides (flexibility for adding/removing devices, fault tolerancy and performance).

4. This comes from the LVM terminology which is often used with RAID on Linux.

5. This is the worst case and the one that should be taken into account. Of course, disks hda and hdc may fail, for example, and the PV will remain available, but the best case is not the one that represents the fault tolerancy degree.

6. Note that this is independent on the actual RAID level chosen : each byte in a RAID array is used, either for storage or for fault tolerance. In the example, using RAID1, we only get 1 Tb out of 8 Tb and it may look like a waste. But if RAID1 is chosen for such an array, it actually means that the fault tolerancy degree of 3 is required. And such a fault tolerancy degree has a storage cost !

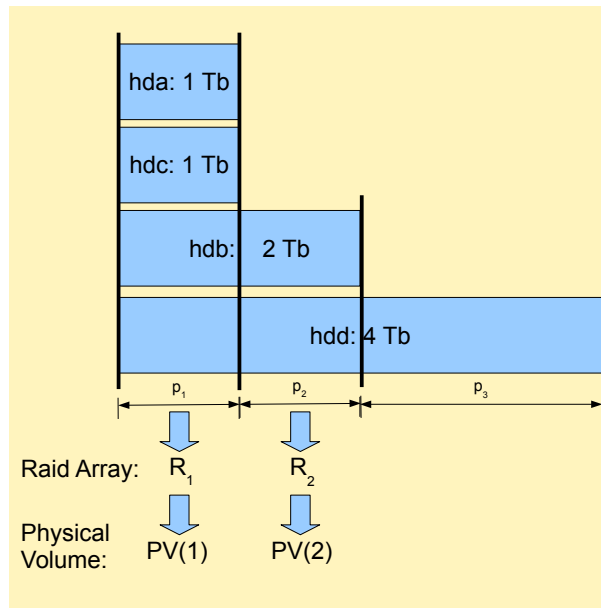


FIGURE 3 – Illustration of the vertical RAID layout.

The algorithm we propose is very simple indeed. First, we sort the device list in ascending capacity order. Then, we partition each disk in such a way that an array with the maximum number of other partitions of the same size can be made. Figure 3 shows the process in our preceding example with 4 disks.

A first partition p_1 is made on all disks. The size of that partition is the size of the first disk, hda, which is the minimum — 1 Tb in our case. Since the second disk in our sorted list, named hdc is also of 1 Tb capacity, no room is available for making a new partition. Therefore, it is skipped. Next disk is hdb in our sorted list. Its capacity is of 2 Tb. The first p_1 partition takes 1 Tb already. Another 1 Tb is available for partitioning and it becomes p_2 . Note that this other 1 Tb partition p_2 is also made on each following disk in our sorted list. Therefore, our last device, hdd has already 2 partitions : p_1 and p_2 . Since it is the last disk, the remaining storage space (2 Tb) will be wasted. Now, a RAID array can be made from each partition of the same size from different disks. In this case, we have the following choices :

- making a RAID array R_1 using 4 p_1 partitions, we can get :
 - 4 Tb in RAID0 ;
 - 1 Tb in RAID1 ;
 - 3 Tb in RAID5 ;
 - 2 Tb in RAID10 ;
- making another array R_2 using 2 p_2 partitions, we can get :
 - 2 Tb in RAID0 ;
 - 1 Tb in RAID1.

Therefore, we maximized the storage space we can get from multiple devices. Actually, we minimized the wasted space which is given — with this algorithm — by the last partition of the last drive, in this case : $W = 2 Tb$. Only 20% of the global storage space is wasted, and this is the minimum we can get. Said otherwise, 80% of the global storage space is used either for storage or fault tolerancy and this is the maximum we can get using RAID technology.

The amount of available storage space depends on the RAID level chosen for each PV from vertical partitions $\{p_1, p_2\}$. It can vary from 2 Tb {RAID1, RAID1} up to 6 Tb {RAID0, RAID0}. The maximum storage space available with a fault tolerancy degree of 1 is 4 Tb {RAID5, RAID1}.

2.2 Analysis

In this section, we will give an analysis of our algorithm. We consider n storage devices of respective capacity c_i for $i \in [1, n]$ where $\forall i \in [1, n-1] c_i \leq c_{i+1}$. Said otherwise, the n drives are sorted by their capacity in ascending order as illustrated on figure 4. We also define $c_0 = 0$ for simplification purposes.

We also define :

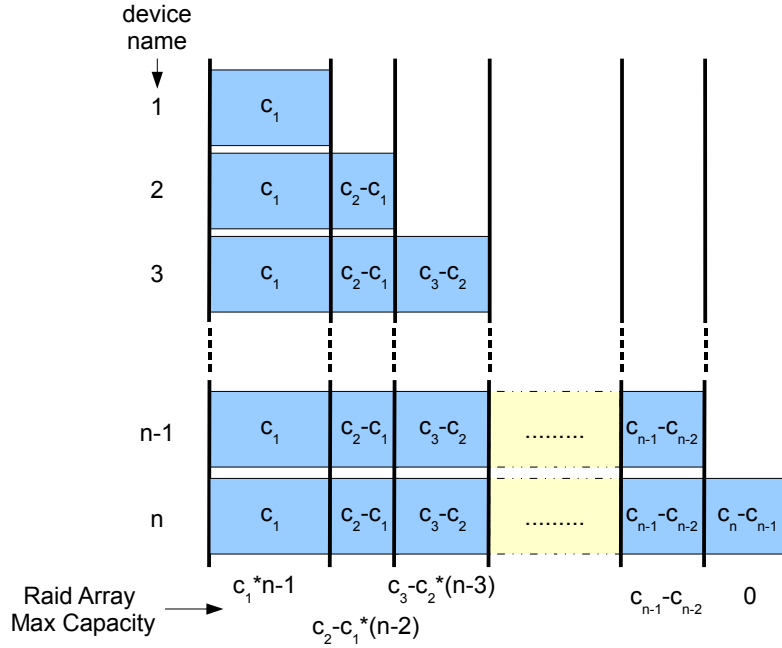


FIGURE 4 – Illustration of the general algorithm.

- the global storage space :

$$G(n) = \sum_{i=1}^n c_i = c_1 + c_2 + \dots + c_n$$

naturally, we also define $G(0) = 0$ (no device gives no storage) ;

- the wasted storage space $W(n) = c_n - c_{n-1}$; we also define $W(0) = 0$ (no device gives no waste) ; note anyway that $W(1) = c_1$ (with only one device you cannot make any RAID array and therefore, the wasted space is maximum !)
- the maximum (safe) available storage space (using RAID5⁷) :

$$\begin{aligned} C_{max}(n) &= c_1 \cdot (n-1) + (c_2 - c_1) \cdot (n-2) + \dots + (c_{n-1} - c_{n-2}) \cdot 1 \\ &= \sum_{i=1}^{n-1} (c_i - c_{i-1}) \cdot (n-i) \\ &= \sum_{i=1}^{n-1} W(i) \cdot (n-i) \end{aligned}$$

we also define $C_{max}(0) = 0$, and $C_{max}(1) = 0$ (you need at least 2 drives to make a RAID array).

- the lost storage space defined as $P(n) = G(n) - C_{max}(n) = W(n) + c_{n-1} = c_n$; it represents the amount of space not used for storage (it includes both space used for fault tolerancy and the wasted space) ; note that $P(0) = 0$ and that $P(1) = c_1 = W(1)$ (with one drive, the wasted space is maximum, and is equal to the lost space).

We also have, $C_{max}(n) = G(n) - c_n = G(n-1)$: the maximum storage space at level n is the global storage space at previous level $n-1$. By the way, when a new storage device is added, with a capacity of c_{n+1} we have :

- the new global storage space : $G(n+1) = G(n) + c_{n+1}$;
- the new maximum available storage space : $C_{max}(n+1) = C_{max}(n) + c_n$;

7. From the available storage space point of view, RAID5 consumes one partition for fault tolerancy. When only 2 partitions are available, RAID1 is the only option available with fault tolerancy, and it also consumes one partition for that purpose. Therefore, from a maximum available storage space perspective, a 2 devices RAID1 array is considered a RAID5 array.

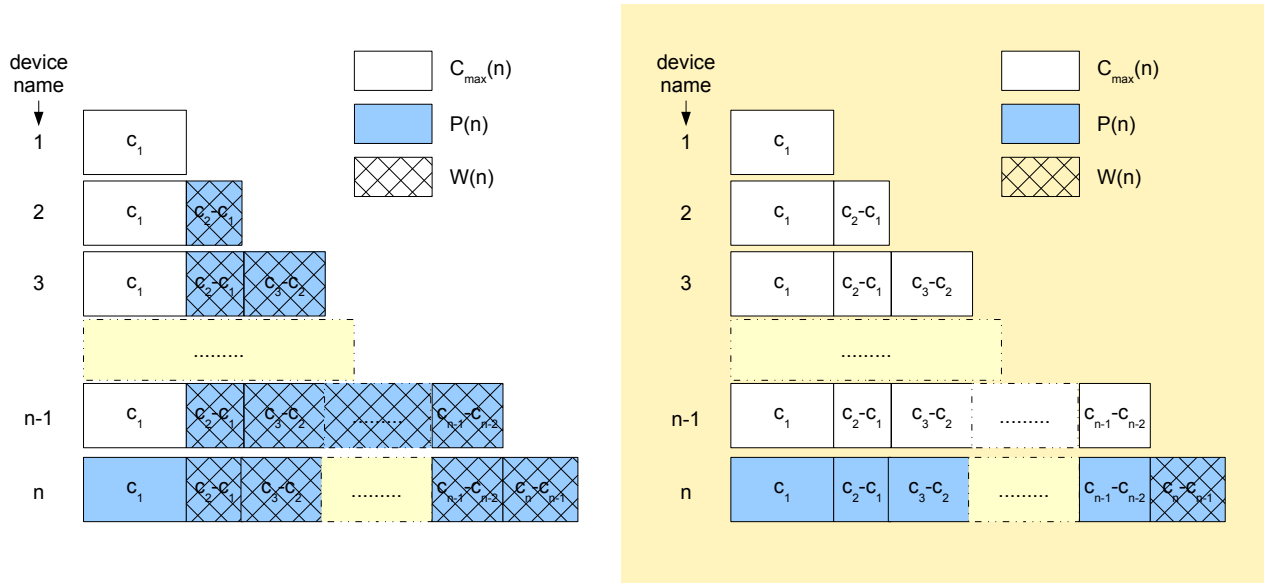


FIGURE 5 – Graphical representation of quantities $P(n)$, $W(n)$ and $C_{max}(n)$ for the traditional RAID algorithm (left) and the PROUHD algorithm (right).

- the new wasted space is : $W(n+1) = c_{n+1} - c_n$;
- the new lost space : $P(n+1) = c_{n+1}$.

When a new storage device bigger than any other in the configuration is added, the maximum available storage space is increased by an amount equal to the last device in the previous configuration without the new device. Moreover, the new lost space is exactly equal to the size of that new device.

As a conclusion, purchasing a much bigger device than the last one in the configuration is not a big win in the first place, since it mainly increases the wasted space! That wasted space will be used when a new drive of a higher capacity will get introduced.

You may compare our algorithm with the usual RAID layout (*i.e.* using same device size $c_{min} = c_1$) on the same set of devices :

- the global storage space remains unchanged : $G'(n) = \sum_1^n c_i$;
- the maximum storage becomes : $C'_{max}(n) = c_1 \cdot (n-1)$;
- the wasted space becomes :

$$W'(n) = \sum_2^n (c_i - c_1) = G'(n) - n \cdot c_1$$

- the lost space becomes : $P'(n) = G'(n) - C'_{max}(n) = c_1 + W'(n)$

When a new device of capacity c_{n+1} is added to the device set, we get :

- $C'_{max}(n+1) = c_1 \cdot n = C'_{max}(n) + c_1$ (the available storage space is increased by c_1 only) ;
- $W'(n+1) = W'(n) + (c_{n+1} - c_1)$ (whereas the wasted space is increased by $(c_{n+1} - c_1)$) ;
- $P'(n+1) = W'(n) + c_{n+1} = P'(n) + (c_{n+1} - c_1)$ (and the lost space is increased by the same amount) ;

As seen formally, the traditional algorithm is very weak in the handling of heterogeneous storage device size. When you add a new device, in the configuration of a higher capacity, you increase both the wasted space and the lost space by an amount that is the difference in size between that new device and the first one. Figure 5 gives a graphical comparison of $P(n)$, $W(n)$ and $C_{max}(n)$ on the whole set of devices for traditional RAID algorithm (left) and for PROUHD (right).

By the way, formally, since $c_n > c_{n-1} > c_1$, it is clear that $c_n - c_1 > c_n - c_{n-1} > 0$. Thus, $c_n - c_1 + c_{n-1} - c_1 + \dots + c_2 - c_1 = \sum_2^n (c_i - c_1) = W'(n) > c_n - c_{n-1} = W(n)$. Therefore the heterogeneous algorithm always gives a better result in terms of wasted space, as expected. It can be shown easily that the heterogeneous algorithm also gives systematically a better result for the lost space $P(n) < P'(n)$.

On the opposite, our algorithm can be seen as an extension of traditional layout where all devices are of same size. This translates formally to $c_i = c_1, \forall i \in [1, n]$, and we have :

- for a global storage space of : $G(n) = \sum_1^n c_i = n \cdot c_1$;
- a maximum storage space of : $C_{max}(n) = (n-1) \cdot c_1$ (RAID5) ;

- a wasted space of: $W(n) = \sum_2^n (c_i - c_1) = 0$;
- a lost space of: $P(n) = G(n) - C_{max}(n) = c_1$;

And we get back to what we are used to where only one disk is lost for n drives of same size (using RAID5).

2.3 Implementation (layout-disks)

We propose an open-source python software — called *layout-disks* and available at <http://www.sf.net/layout-disks>— that given a list of devices label and size, returns the possible layout using this algorithm. As an example, with 4 disks taken from illustration 3, the software proposes the following :

```
$ layout-disks hda :1 hdb :2 hdc :1 hdd :4
From ['hda', 'hdc', 'hdb', 'hdd'] create a partition of 1.0 to get :
  Size  | RAID Level  | Tolerancy  | Storage Efficiency (%)
  1.0   | RAID1       | 3          | 25.0
  3.0   | RAID5       | 1          | 75.0
  2.0   | RAID10      | 1          | 50.0
From ['hdb', 'hdd'] create a partition of 1.0 to get :
  Size  | RAID Level  | Tolerancy  | Storage Efficiency (%)
  1.0   | RAID1       | 1          | 50.0
-- Global overview --
-- Global storage space :      G =      8.0
-- Maximum (safe) storage :    C_max = 4.0 ( 50.0 %)
-- Wasted storage space :      W =      2.0 ( 25.0 %)
-- Lost storage space :        P =      4.0 ( 50.0 %)
Enjoy ! ;-)
```

The software tells that from the first partition of each 4 drives, several RAID level options are available (from RAID1 up to RAID5)⁸. From the second partition on devices hdb and hdd, only RAID1 is available.

2.4 Performance

From a performance point of view, this layout is definitely not optimal for every usage. Traditionally, in the enterprise case, two different virtual RAID devices map to different physical storage devices. On the opposite here, any distinct PROUHD devices share some of their physical storage devices. If no care is taken, this can lead to very poor performance as any request made to a PROUHD device may be queued by the kernel until other requests made to other PROUHD device have been served. Note however that this is not different from the single disk case except from a strict performance point of view : the throughput of a RAID array — especially on reads — may well outperform the throughput of a single disk thanks to parallelism.

For most end-user cases, this layout is perfectly fine from a performance point of view, especially for storing multimedia files such as photo, audio or video files where most of the time, files are written once, and read multiple times, sequentially. A file server with such a PROUHD disk layout will easily serve multiple end-user clients simultaneously. Such a layout may also be used for backup storage. The only reason such a configuration should not be used is where you have strong performance requirements. On the other side, if your main concern is storage space management, such a configuration is very sound.

By the way, you may combine such a layout with the Linux Volume Manager (LVM). For example, if your main concern is storage space with a tolerance level of 1, you may combine, the 3.0Gb RAID5 region with the 1.0Gb RAID1 region in the previous example as a volume group resulting in a virtual device of 4.0 Gb, from which you can define logical volumes (LV) at will.

Advantages of such a combined RAID/LVM layout versus a strict LVM layout (without any RAID array in between), is that you can benefit advantages of RAID levels (all levels 0, 1, 5, 10, 50, or 6) whereas LVM provide, as far as I know, a "poor" (compared to RAID) mirroring and stripping implementation. By the way,

8. RAID0 is only presented if option `--unsafe` is specified. RAID6 and other RAID levels are not implemented currently. Any help is welcome!;-)

note that specifying mirror or stripe options at logical volume creation will not give the expected performance and/or tolerancy improvement since physical volumes are (already) RAID arrays sharing real physical devices.

SSD special case Our solution makes good use of available storage space at the expense of raw performance penalty in some cases : when concurrent access are made to distincts RAID arrays sharing same physical devices. Concurrent accesses usually imply random access to data.

Hard drives have a hard limit on their I/O througput with random access pattern due to their mecanical constraints : after data has been located, the reading (or writing) head should seek to the correct cylinder and wait until the correct sector passes under it thanks to plate rotation. Obviously, reading or writing to hard disks is mainly a sequential process. A read/write request is pushed onto a queue (in software or in hardware), and it should just wait previous ones. Of course, many improvements were made to speed up the reading/writing process (for example, using buffer and cache, smart queue managements, bulk operations, data locality computation among others), but performance of hard drives are physically limited anyhow, especially on random accesses. In some ways, this random (concurrent) access problems is the reason why RAID has been introduced in the first place.

SSDs are very different from hard disks. In particular, they do not have such mecanical constraints. They handle random accesses much better than hard disks. Therefore, the performance penalty of PROUHD discussed above may not be so true with SSD. Concurrent accesses made to distincts RAID arrays sharing physical SSDs will result in several requests with a random access pattern made to each underlying SSD. But as we have seen, SSDs handles random request quite well. Some investigations should be made to compare performance of PROUHD over hard disks versus PROUHD over SSDs. Any help in this regard will be appreciated.

3 Partitionning drives

PROUHD requires that storage devices are properly partitionned into slices of same size. Depending on the number of different sized storage devices, the algorithm may lead to the creation of a vast number of partitions on each device. Fortunately, it is not required to use primary partitions which are limited to 4 by PC BIOS for legacy reasons. Logical partitions can be used in order to create all the required slices : there are almost no limit to their numbers. On the other side, if you need partitions of more than 2 TeraBytes, then logical partitions are no more an option.

For this specific case (partition size of more than 2TB), GUID Partition Table (GPT) might be an option. As far as I know, only parted⁹ supports them.

It might be tempting to use LVM for partitionning purpose. If this is a perfect choice in the usual case of partitionning, I would not recommend it for PROUHD anyway. Actually, the other way round is the good option : RAID arrays are perfect choice for LVM Physical Volume (PV). I mean, each RAID array becomes a PV. From some PVs, you create Volume Group (VG). From those VGs, you create Logical Volumes (LV) that you finally format and mount into your filesystem. Therefore, the chain of layers is as follow :

Device -> RAID -> PV -> VG -> LV -> FS.

If you use LVM for partitionning drives, you end up with a huge number of layers that kill performance (probably) and design :

Device -> PV -> VG -> LV -> RAID -> PV -> VG -> LV -> FS.

Honestly, I have not tested such a complex configuration. I would be interested on feedbacks though. ;-)

4 Handling Disk Failure

Of course, any disk will fail, one day or another. The later, the better. But, planning disk replacement is not something that can be postponed until failure, it is usually not at the good time (the murphy's law!). Thanks to RAID (for level 1 and above), a disk failure does not prevent the whole system to work normally. This is a problem since you may not even notice that something went wrong. Again, if nothing is planned,

9. See <http://www.gnu.org/software/parted/index.shtml>

you will discover it the hard way, when a second disk actually fail, and when you have no way to recover your RAID arrays. First thing is to monitor your storage devices. You have (at least) 2 tools for that purpose :

smartmontools : SMART is a standard implemented in most IDE and SATA drives that monitor the health of a disk, performing some tests (online and offline), and that can send reports by email, especially when one or many tests went wrong. Note that SMART does not give any guarantee that it will anticipate failure, nor that its failure forecasts are accurate. Anyway, when SMART tells that something is wrong, it is better to plan for a disk replacement very soon. By the way, in such a case, do not stop the drive unless you have a spare, they usually dislike being re-started, especially after such forecasted failures. Configuring smartmontools is quite simple. Install that software and look at the file `smartd.conf` usually in `/etc`.

mdadm : mdadm is the linux tool for (software) RAID management. When something happens to a RAID array, an email can be sent. See the file `mdadm.conf` usually in `/etc` for details.

In traditionnal RAID, when one device from a RAID array fail, the array is in a so called "degraded" mode. In such a mode, the array is still working, data remains accessible, but the whole system may suffer a performance penalty. When you replace the faulty device, the array is reconstructed. Depending on the RAID level, this operation is either very simple (mirroring requires only a single copy) or very complex (RAID5 and 6 requires CRC computation). In either case, the time required to complete this reconstruction is usually quite huge (depending on the array size). But the system is normally able to perform this operation online. It can even limit the overhead as much as possible when the RAID array is serving clients. Note that RAID5 and RAID6 levels can stress a file server quite well during array reconstructions.

In the case of PROUHD, the effect on the whole system is worse since one drive failure impact many RAID arrays. Traditionnaly, degraded RAID arrays can get reconstructed all at the same time. The main point is to reduce the time spent in degraded mode minimizing the probability of data loss globally (the more time in degraded mode, the more probable data loss can occur). But parallel reconstruction is not a good idea in the PROUHD case because RAID arrays share storage devices. Therefore, any reconstruction impact all arrays. Parallel reconstructions will just stress more all storage devices, and thus, the global reconstruction will probably not recover sooner than a simpler sequential one.

Fortunately, linux mdadm is smart enough to prevent parallel reconstructions of arrays that share same storage devices as shown by the following examples :

```
Sep 6 00:57:02 phobos kernel: md: syncing RAID array md0
Sep 6 00:57:02 phobos kernel: md: minimum _guaranteed_ reconstruction speed: 1000
KB/sec/disc .
Sep 6 00:57:02 phobos kernel: md: using maximum available idle IO bandwidth (but
not more than 200000 KB/sec) for reconstruction .
Sep 6 00:57:02 phobos kernel: md: using 128k window, over a total of 96256 blocks.
Sep 6 00:57:02 phobos kernel: md: delaying resync of md1 until md0 has finished
resync (they share one or more physical units)
Sep 6 00:57:02 phobos kernel: md: syncing RAID array md2
Sep 6 00:57:02 phobos kernel: md: minimum _guaranteed_ reconstruction speed: 1000
KB/sec/disc .
Sep 6 00:57:02 phobos kernel: md: using maximum available idle IO bandwidth (but
not more than 200000 KB/sec) for reconstruction .
Sep 6 00:57:02 phobos kernel: md: using 128k window, over a total of 625137152
blocks .
Sep 6 00:57:02 phobos kernel: md: delaying resync of md3 until md2 has finished
resync (they share one or more physical units)
Sep 6 00:57:02 phobos kernel: md: delaying resync of md1 until md0 has finished
resync (they share one or more physical units)
Sep 6 00:57:02 phobos kernel: md: delaying resync of md4 until md2 has finished
resync (they share one or more physical units)
Sep 6 00:57:02 phobos kernel: md: delaying resync of md1 until md0 has finished
resync (they share one or more physical units)
```

```

Sep  6 00:57:02 phobos kernel: md: delaying resync of md3 until md4 has finished
resync (they share one or more physical units)
Sep  6 00:57:25 phobos kernel: md: md0: sync done.
Sep  6 00:57:26 phobos kernel: md: delaying resync of md3 until md4 has finished
resync (they share one or more physical units)
Sep  6 00:57:26 phobos kernel: md: syncing RAID array md1
Sep  6 00:57:26 phobos kernel: md: minimum _guaranteed_ reconstruction speed: 1000
KB/sec/disc .
Sep  6 00:57:26 phobos kernel: md: using maximum available idle IO bandwidth (but
not more than 200000 KB/sec) for reconstruction .
Sep  6 00:57:26 phobos kernel: md: using 128k window, over a total of 2016064
blocks .
Sep  6 00:57:26 phobos kernel: md: delaying resync of md4 until md2 has finished
resync (they share one or more physical units)
Sep  6 00:57:26 phobos kernel: RAID1 conf printout:
Sep  6 00:57:26 phobos kernel: --- wd:2 rd:2

```

Therefore, we can rely on mdadm to do the right thing with RAID, whether it is an homogeneous, an heterogeneous configuration or a combination of both.

4.1 Replacement Procedure

4.1.1 Replacing a failed device with a same-size one.

This is the ideal situation and it mostly follows the traditional RAID approach except that you now have more than one RAID array to manage for each device. Let's take our example (figure 6 left), and let's suppose that a failure has been detected on hdb. Note that a failure may have been detected locally on hdb2, and not on hdb1 for example. Anyway, the whole disk will have to be replaced and therefore, all arrays are concerned. In our example, we have setup the storage with the following PROUHD configuration :

```

/dev/md0 : hda1, hdb1, hdc1, hdd1 (RAID5, (4-1)*1Tb = 3 Tb)
/dev/md1 : hdb2, hdd2 (RAID1, (2*1Tb)/2 = 1Tb)

```

1. Logically remove each faulty device partition from its corresponding RAID array :

```

mdadm /dev/md0 --faulty /dev/hdb1 --remove /dev/hdb1
mdadm /dev/md1 --faulty /dev/hdb2 --remove /dev/hdb2

```
2. Physically remove the faulty device — unless you have a hot-plug system such as USB you will have to power off the whole system ;
3. Physically add a new device — unless you have a hot-plug system such as USB, you will have to power on the whole system ;
4. Partition the new device (let say /dev/sda) with the exact same layout than the failed device : 2 partitions of 1Tb each /dev/sda1 and /dev/sda2 ;
5. Logically add each new partition to its corresponding RAID array :

```

mdadm /dev/md0 --add /dev/sda1
mdadm /dev/md1 --add /dev/sda2

```

After a while, all your RAID arrays will get re-constructed.

4.1.2 Replacing a failed device with a larger one.

This case is not so simple indeed. The main issue is that the whole layout is not at all related to the old one. Let's take the previous example, and see what happened if /dev/hdb fail. If we replace that 2Tb device with a 3Tb new device, we should end up with the layout of figure 6 (right).

Notice that partition p_2 is now of 2Tb and not of 1Tb as it was the case previously (see figure 3). This means that the previous RAID array made from /dev/hdb2 :1Tb and /dev/hdd2 :1Tb is no more relevant after the replacement : it does not appear in the layout algorithm. Instead, we have a RAID array made of /dev/sda2 :2Tb and /dev/hdd2 :2Tb.

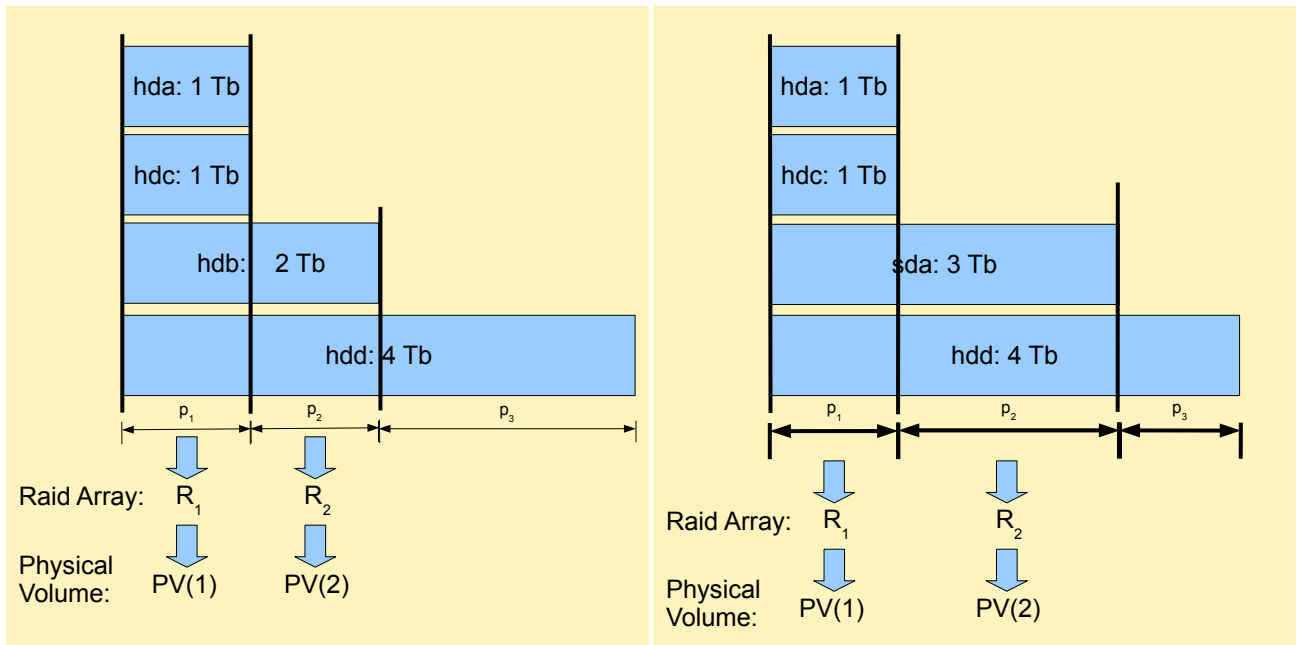


FIGURE 6 – Replacing a failed device by a larger one. Layout before (left) and after (right) the replacement of /dev/hdb :2 with /dev/sda :3.

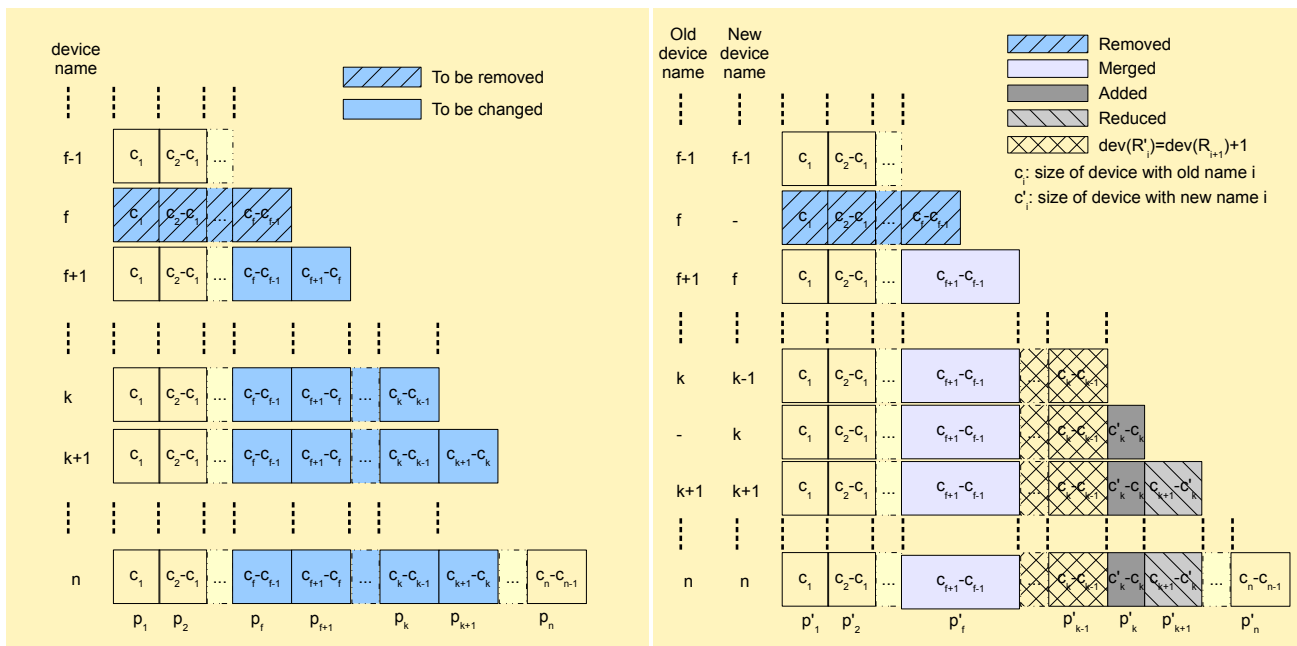


FIGURE 7 – Replacing a failed device (f) by a larger one (k), general case before (left) and after (right).

In the general case, as shown on figure 7, the last partition of the failed device f , is no more relevant. Therefore, the whole RAID array labeled R_f of size $(c_f - c_{f-1}) \cdot (n - f)$, made from partitions p_f of devices $f, f + 1, \dots, n$ should be removed. The following array, R_{f+1} , that was made from the last partition of the following disk, $f + 1$, should be resized according to the new layout. Partitions p_{f+1} were having a size of $c_{f+1} - c_f$. These partitions can now be "merged" since there is no "in-between" c_{f-1} and c_{f+1} . Therefore, new "merged" partitions become p'_f with a size of $c_{f+1} - c_{f-1}$.

Finally, the new device is inserted between devices at rank k and $k + 1$ because its capacity c'_k is so that $c_k \leq c'_k < c_{k+1}$. (Note that all devices $i, f < i \leq k$ will shift to rank $i - 1$ because new device is added *after* failed device f). The new device should be partitionned so all partitions from 1 up to $f - 1$ are of same size than in the previous layout : $p'_i = p_i, \forall i \in [1, f - 1]$. Size of partition f is given by : $p'_f = c_{f+1} - c_{f-1}$ as we have seen previously. Finally, all following partitions, up to k are of the same size than in the old layout : $p'_i = p_{i+1}, \forall i \in [f + 1, k]$. This new device, adds its own modification in the new layout according to the difference between its size c'_k and the size of the previous device c'_{k-1} which is the k device in the old layout ($c'_{k-1} = c_k$). Therefore, in the new layout, partition k has a size given by $p'_k = c'_k - c_k$. Finally, next partition should be modified. It was previously of size $c_{k+1} - c_k$, but this is no more relevant in the new layout. It should be reduced to $p'_{k+1} = c_{k+1} - c'_k$. The following partitions should not be changed. Note that the new device replaces failed partitions $p_i, i \in [1, f]$ from the failed device, but adds up 1 more partition to RAID arrays $R'_i, i \in [f + 1, k - 1]$. We note $dev(R_i)$ the number of partitions that made up RAID array R_i . Therefore, we have : $dev(R'_i) = dev(R_{i+1}) + 1$. Fortunately, it is possible to grow a RAID array under Linux thanks to the great `mdadm grow` command.

In summary, old layout :

$$p_1, p_2, \dots, p_f, \dots, p_k, \dots, p_n$$

becomes new layout :

$$p'_1, p'_2, \dots, p'_f, \dots, p'_k, \dots, p'_n$$

with :

$$\begin{aligned} p'_i &= p_i, \forall i \in [1, f - 1] \\ p'_f &= c_{f+1} - c_{f-1} = c'_f - c'_{f-1} \\ p'_i &= p_{i+1}, \forall i \in [f + 1, k] \\ p'_k &= c'_k - c_k = c'_k - c'_{k-1} \\ p'_{k+1} &= c_{k+1} - c'_k = c'_{k+1} - c'_k \\ p'_i &= p_i, \forall i \in [k + 2, n] \\ dev(R'_i) &= dev(R_{i+1}) + 1, \forall i \in [f + 1, k - 1] \end{aligned}$$

As we see, replacing a faulty device by a larger one leads to quite a lot of modifications. Fortunately, they are somewhat local : in a large set of devices, modifications happens only to a bounded number of devices and partitions. Anyway, the whole operation is obviously very time consuming and error prone if done without proper tools.

Hopefully, the whole process can be automated. The algorithm presented below uses LVM advanced volume management. It supposes that RAID arrays are physical volumes that belongs to some virtual groups (VG) from which logical volumes (LV) are created for the making of filesystems. As such, we note $PV(i)$ the LVM physical volume backed by RAID array R_i .

We suppose disk f is dead. We thus have f degraded RAID arrays, and $n - (f + 1)$ safe RAID arrays. An automatic replacement procedure is defined step-by-step below.

1. Backup your data (this should be obvious, we are playing with degraded arrays since one disk is out of order, therefore any mistake will eventually lead to data loss! For that purpose, you may use any storage space available that do not belong to the failed disk. Next RAID arrays in the layout are fine for example.
2. Mark all partitions $p_i, \forall i \in [1, f]$ of broken device as faulty, in their corresponding RAID arrays R_i and remove them (`mdadm -fail -remove`).

3. Remove the failed storage device f .
4. Insert the new storage device k .
5. Partition new device k according to the new layout (fdisk). In particular, last failed device partition and last new device partition should have correct sizes : $p'_f = c_{f+1} - c_{f-1} = c'_f - c'_{f-1}$ and $p'_k = c'_k - c_k = c'_k - c'_{k-1}$. At that stage, will still have f degraded arrays : $R_i, \forall i \in [1, f]$.
6. Replace failed partition by adding new device partition $p'_i, i \in [1, f-1]$ to its corresponding raid array R'_i (mdadm -add). After this step, only R'_f is a degraded RAID array.
7. Remove $PV(f)$, and $PV(f+1)$ from their corresponding VG (pvmove). LVM will handle that situation quite well, but it requires enough free space in the VG (and time!). It will actually copy data to other PV in the (same) VG.
8. Stop both RAID arrays R_f and R_{f+1} corresponding to $PV(f)$ and $PV(f+1)$ (mdadm stop).
9. Merge (fdisk) partition f and $f+1$ into one single partition $p'_f = c_{f+1} - c_{f-1} = c'_f - c'_{f-1}$. This should work fine, since other partitions are not impacted by that. It should be done on each device following failed device f : that is $n - f$ storage devices in total (device k was already partitionned in step 5).
10. Create a new raid array R'_f from the merged partition p'_f (mdadm create).
11. Create the corresponding $PV(f)$ (pvcreate), and add it to the previous VG (vgextend). At that step, we are back to a safe global storage space : all RAID arrays are now safe. But the layout is not optimal : partition $p'_i, i \in [f+1, k]$ are still unused for example.
12. Remove $PV(k+1)$ from its corresponding VG (pvmove). Again, you will need some available storage space.
13. Stop the corresponding RAID array (mdadm stop).
14. Split old partition $p_{k+1} = c_{k+1} - c_k$ into new $p'_k = c'_k - c'_{k-1}$ and $p'_{k+1} = c'_{k+1} - c'_k$ (fdisk) ; This should be done on each device following k , that is $n - k$ devices in total. This should not cause any problem, other partitions are not impacted.
15. Create two new RAID arrays R'_k and R'_{k+1} from thus 2 new partitions p'_k and p'_{k+1} (mdadm create).
16. Create $PV'(k)$ and $PV'(k+1)$ accordingly (pvcreate). Insert them back into VG (vgextend).
17. Finally, add each new device partition $p'_i, i \in [f+1, k-1]$ to its corresponding raid array R'_i . You will have to grow RAID arrays R'_i so that $dev(R'_i) = dev(R_{i+1}) + 1, i \in [f+1, k-1]$ (mdadm grow).
18. We are back with the new correct layout, with n safe RAID arrays.

Note that this process focuses on the end-user : it makes the replacement as convenient as possible, preventing the user a long wait between failed device removal and new one replacement. All is done at the beginning. Of course, the time required before the whole pool of RAID arrays runs non-degraded can be quite huge. But it is somewhat transparent from the end-user point of view.

4.1.3 Replacing a failed drive with a smaller one

This case is the worst one, for two reasons. First, the global capacity is obviously reduced : $G'(n) < G(n)$. Second, since some bytes of the failed larger drives were used for fault tolerancy¹⁰, some of those bytes are no more present in the new device. This will have quite a consequence on the practical algorithm as we will see.

When a device f fail, all RAID arrays R_i , where $i \leq f$ becomes degraded. When we replace the failed device f by a new device $k+1$ where $k+1 < f$, $c_k \leq c'_{k+1} < c_{k+1} \leq c_f$, then RAID arrays $R'_i, i \leq k$ becomes repaired, but RAID arrays $R'_i, k+2 \leq i \leq f$ remains degraded (see figure 8) because there is not enough storage space in the new device for taking over failed ones. (Note that all devices $i, k+1 \leq i \leq f-1$ will shift to rank $i+1$ because new device is added *before* failed device f).

As in the previous case, the solution requires the merging of partitions p_{f-1} with the one from p_{f+1} since there is no more p_f . Hence, $p'_{f+1} = c_{f+1} - c_{f-1} = c'_{f+1} - c'_f$ on all devices $i \geq f+1$. Also, the new device $k+1$, should be partitionned correctly. In particular, its last partition $p'_{k+1} = c'_{k+1} - c_k = c'_{k+1} - c'_k$. Devices

10. Unless RAID0 was used, but in that case, the situation is even worse !

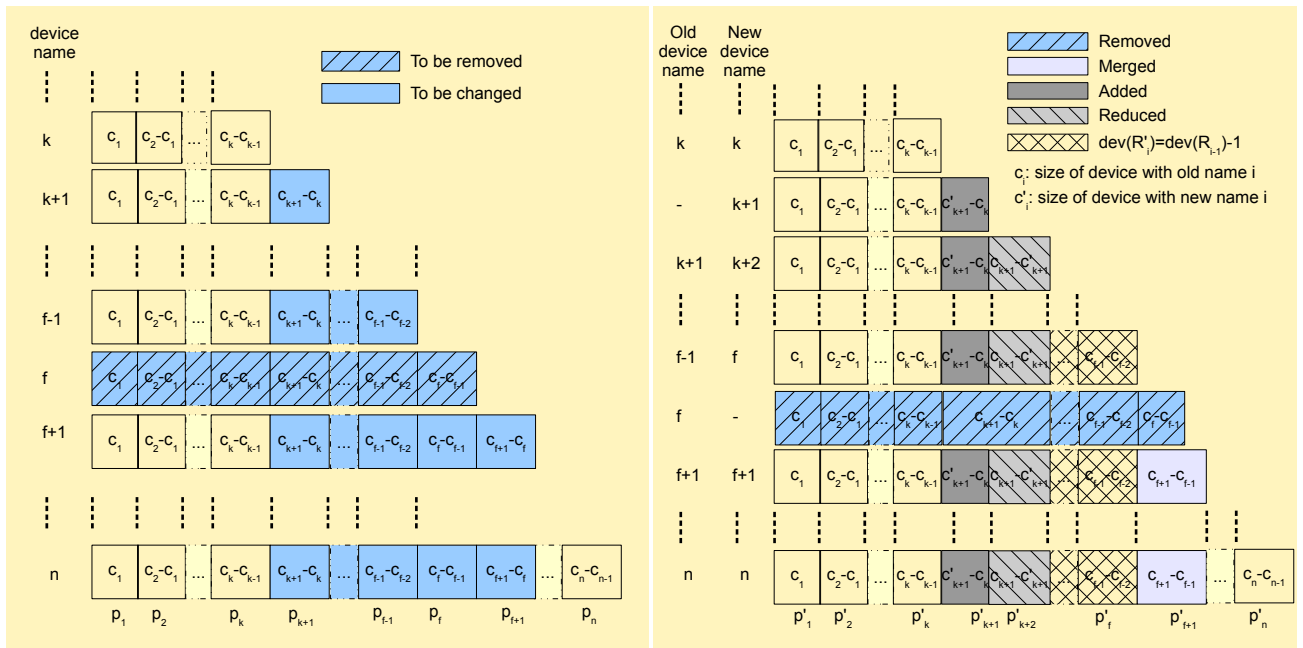


FIGURE 8 – Replacing a failed device (f) by a smaller one (k), general case before (left) and after (right).

$i \geq k + 2$ should change their partitioning according to new partition p'_{k+1} . For those devices, partition p'_{k+2} should also be changed : $p'_{k+2} = c_{k+1} - c'_{k+1} = c'_{k+2} - c'_{k+1}$. Most important modifications concerns all RAID arrays R'_i , $k + 2 \leq i \leq f$ since they are still degraded. For all of them, their number of (virtual) devices should be decreased by one : for example, R_{k+2} was made of $(n - k - 1)$ "vertical" partitions p_{k+2} from device $k + 2$ up to device n since device f was wide enough to support a partition p_{k+2} . It is no more the case for R'_{k+2} since the new device does not provide sufficient storage space to support a p_{k+1} partition. Therefore, $dev(R'_i) = dev(R_{i-1}) - 1$, $k + 2 \leq i \leq f$.

In summary, old layout :

$$p_1, p_2, \dots, p_k, \dots, p_f, \dots, p_n$$

becomes new layout :

$$p'_1, p'_2, \dots, p'_k, \dots, p'_f, \dots, p'_n$$

with

$$\begin{aligned} p'_i &= p_i, \forall i \in [1, k] \\ p'_{k+1} &= c'_{k+1} - c_k = c'_{k+1} - c'_k \\ p'_{k+2} &= c_{k+1} - c'_{k+1} = c'_{k+2} - c'_{k+1} \\ p'_i &= p_{i-1}, \forall i \in [k+3, f] \\ p'_{f+1} &= c_{f+1} - c_{f-1} = c'_{f+1} - c'_f \\ p'_i &= p_i, \forall i \in [f+2, n] \\ dev(R'_i) &= dev(R_{i-1}) - 1, \forall i \in [k+2, f] \end{aligned}$$

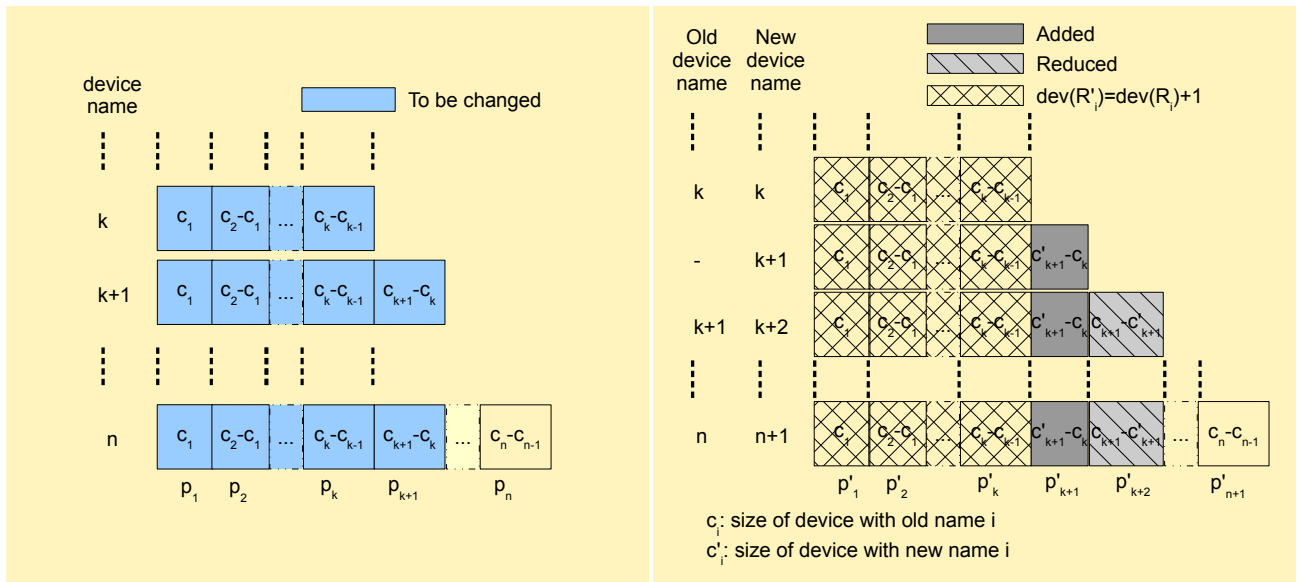
Unfortunately, as far as we know, it is not (currently) possible to shrink a RAID device using Linux RAID. The only option is to remove the whole set of arrays R_{i-1} , $\forall i \in [k+2, f]$ entirely, and to create new ones with the correct number of devices. An automatic replacement procedure is therefore defined step-by-step below :

1. Backup your data !;-)
2. Mark all partitions p_i , $\forall i \in [1, f]$ of broken device as faulty, in their corresponding RAID arrays R_i and remove them (`mdadm -fail -remove`).
3. Remove failed storage device f .

4. Insert the new storage device $k + 1$.
5. Partition new device according to the new layout (fdisk). In particular, last partition should have correct size : $p'_{k+1} = c'_{k+1} - c_k = c'_{k+1} - c'_k$. At that stage we still have f degraded RAID arrays : $R_i, \forall i \in [1, f]$.
6. Replace faulty partitions by adding new device ones p'_i and add them to their respective arrays $R'_i, \forall i \in [1, k]$. After this step, $R_i, \forall i \in [k + 1, f]$ are still old degraded arrays, that is $f - k$ RAID arrays in total. Two RAID arrays are still made of wrong sized partitions : R_{k+1} and R_{f+1} .
7. For each array $R'_i, \forall i \in [k + 3, f]$:
 - (a) Move the data corresponding to R'_i to other devices (pvmove on the related LVM volume $PV'(i)$);
 - (b) Remove the corresponding LVM volume $PV'(i)$ from its volume group VG' (pvremove);
 - (c) Stop related array R'_i (mdadm stop);
 - (d) Create a new RAID array R'_i from partition p'_i . Note that there is now one less partition in R'_i : $dev(R'_i) = dev(R_{i-1}) - 1$;
 - (e) Create the corresponding LVM volume $PV'(i)$ (pvcreate);
 - (f) Add that new LVM volume to its related volume group VG' .
8. At this step, R_{k+1} and R_{f+1} are still made of wrong sized old p_{k+1} and p_{f+1} .
9. Move the data corresponding to R_{f+1} to other devices (pvmove on the related LVM volume $PV(f + 1)$);
10. Remove the corresponding LVM volume $PV(f + 1)$ from its volume group VG (pvremove);
11. Stop the related array R_{f+1} (mdadm stop);
12. Merge (fdisk) old partitions p_f and p_{f+1} into one single partition $p'_{f+1} = c_{f+1} - c_{f-1} = c'_{f+1} - c'_f$. This should work fine, since other partitions are not impacted by that. It should be done on each device following failed device f : that is $n - f$ storage devices in total.
13. Create a new raid array R'_{f+1} from the merged partition p'_{f+1} (mdadm create).
14. Create the corresponding $PV'(f + 1)$ (pvcreate), and add it to the previous VG (vgextend). At that step, only R_{k+1} remains wrong and degraded.
15. Move the data corresponding to R_{k+1} to other devices (pvmove on the related LVM volume $PV(k + 1)$).
16. Remove the corresponding LVM volume $PV(k + 1)$ from its volume group VG (pvremove);
17. Stop the related array R_{k+1} (mdadm stop);
18. Split (fdisk) old partitions p_{k+1} into new partitions $p'_{k+1} = c'_{k+1} - c_k = c'_{k+1} - c'_k$ and $p'_{k+2} = c_{k+1} - c'_{k+1} = c'_{k+2} - c'_{k+1}$. This should be done on all following devices, that is $n - k - 1$ devices in total.
19. Create (mdadm -create) new RAID arrays R'_{k+1} and R'_{k+2} from partitions p'_{k+1} and p'_{k+2} ;
20. Create (pvcreate) the corresponding $PV'(k + 1)$ and $PV'(k + 2)$ and add (vgextend) them to their corresponding VG' .
21. You are back with the new correct layout, with n safe RAID arrays.

Note that step 7 is done one array per one array. The main idea is to reduce the amount of available storage space required by the algorithm. Another option is to remove all LVM volumes (PV) at the same time from their related VG, then, to remove their corresponding RAID arrays, and then to recreate them with the correct number of partitions (it should be reduced by one). Removing all those arrays in one turn may result in a big reduction of available storage space that might block the whole process while removing PV from their corresponding VG. Since such a removal results in the move of the data from one PV to others (in the same VG), it also requires that there is enough free space in that VG to accommodate the full copy.

On the other side, the algorithm described may result in a vast amount of data transfert. For example, suppose that all PVs are actually in a single VG. The removal of the first PV in the list ($PV(k+3)$ therefore) may result in the move of its data to $PV(k+4)$. Unfortunately, on next iteration, $PV(k+4)$ will be also removed resulting in the transfert of same data to $PV(k+5)$ and so on. Investigation on a smarter algorithm for that specific step 7 is therefore a must.


 FIGURE 9 – Adding a device (k) to the pool, general case before (left) and after (right).

4.1.4 RAID array reconstruction

Given the size of current hard drives, and the Unrecoverable Bit Error (UBE) — $\frac{1}{10^{15}}$ for enterprise class disk drives (SCSI, FC, SAS) and $\frac{1}{10^{14}}$ for desktop class disk drives (IDE/ATA/PATA, SATA), the reconstruction of a disk array after the failure of a device can be quite challenging. When the array is in a degraded mode, during reconstruction, it tries to get data from remaining devices. But with today large device capacity, the probability of an error during that step becomes significant. Especially, there is a trend with large RAID5 groups to be unrecoverable after a single disk failure. Hence the design of RAID6 that can handle 2 simultaneous disk failures but with a very high write performance hit.

Instead of setting up large RAID5 groups, it might be preferable to setup large set of RAID10 arrays. This gives better result both in terms of reliability (RAID1 is far easier to recover than RAID5), and performance. But the high storage cost — 50% of space lost — often makes this choice irrelevant despite the cheap price of the MB today.

With PROUHD, given that the wasted space is minimum, the RAID10 option might be an acceptable compromise (over traditional RAID layout of course).

Moreover, in PROUHD, RAID components do not cover entire drives but only a portion of it (a partition). Therefore, the probability of other sector errors is reduced.

5 Adding/removing a device to/from a PROUHD

As shown by figure 9, adding a new device k in the pool is much simpler than previous replacement cases. The last partition of the new device impacts the previous layout :

$$\begin{aligned} p'_{k+1} &= c'_{k+1} - c_k = c'_{k+1} - c'_k \\ p'_{k+2} &= c_{k+1} - c'_{k+1} = c'_{k+2} - c'_{k+1} \end{aligned}$$

And all raid arrays up to k should see their number of devices increased by one :

$$dev(R'_i) = dev(R_i) + 1, \forall i \in [1, k]$$

The reverse is also much simpler than any replacement procedure as shown by figure 10. Removing a device k from the pool leads also to a modification of its related partition p_k :

$$p'_k = c_{k+1} - c_{k-1} = c'_k - c'_{k-1}$$

- maximize performance (choose RAID10 when possible, then RAID1) ;
- safe config (choose RAID10 when possible, RAID5, then RAID1) ;
- custom config.

Presenting those configurations graphically, enabling configuration comparisons, proposing pre-defined configurations for well-known workloads (multimedia files, system files, log files and so on) will add up to the initial solution.

Finally, the main performance (and cost) of such storage boxes will come from the actual number of controllers. Concurrent requests (RAID naturally increases them) are best served when they come from different controllers.

7 Alternatives

One of the problem of the adoption of RAID technology is the lack of support for heterogeneous sized storage devices. An alternative to our proposition does exist anyway. If failed device are always replaced by larger ones in a RAID array, then there will be a time where all devices have been replaced. At that time, the whole RAID array can be expanded using the RAID mdadm grow comand. Note that this only works for RAID1 and RAID5.

8 Questions, Comments & Suggestions

If you have any question, comment, and/or suggestion on this document, feel free to contact me at the following address : pierre@vigneras.name.

9 Note

This article has been published in HTML format at :

<http://www.linuxconfig.org/prouhd-raid-for-the-end-user> (ISSN 1836-5930)

thanks to Lubos Rendek.

10 Acknowledgment

The author would like to thanks Lubos Rendek and Pascal Grange for their valuable comments and suggestions.