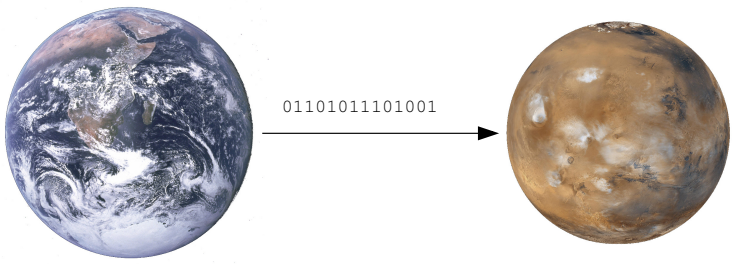
MARS Manual

Multiversion Asynchronous Replicated Storage



Thomas Schöbel-Theuer (tst@1und1.de)

Version 0.1-59

Abstract

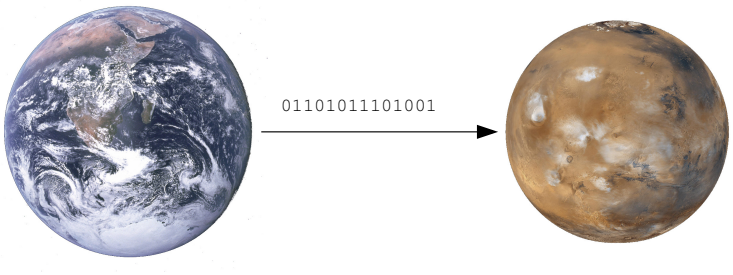
MARS is a block-level storage replication system for long distances / flaky networks under GPL. It runs as a Linux kernel module. The sysadmin interface is similar to DRBD[1], but its internal engine is completely different from DRBD: it works with **transaction logging**, similar to some database systems.

Therefore, MARS can provide stronger **consistency guarantees**. Even in case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but never become inconsistent. In contrast to DRBD, MARS preserves the **order of write operations** even when the network is flaky (**Anytime Consistency**).

The current version of MARS supports $k > 2$ replicas and works **asynchronously**. Therefore, application performance is completely decoupled from any network problems. Future versions are planned to also support synchronous or near-synchronous modes.

MARS supports a new method for building Cloud Storage / Software Defined Storage, called **LV Football**.

It comes with some automation scripts, leading to a similar functionality than Kubernetes, but devoted to stateful LVs over **virtual LVM pools** in the petabytes range.



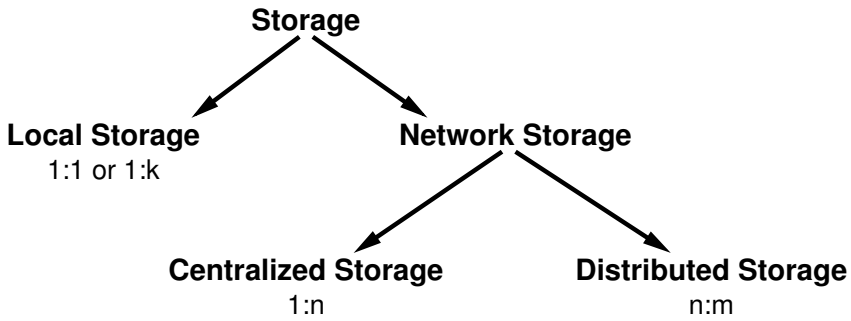
[1] Registered trademarks are the property of their respective owner.

Contents

1. Architectures of Cloud Storage / Software Defined Storage / Big Data

Datacenter architects have no easy job. Building up some petabytes of data in the wrong way can easily endanger a company, as will be shown later. There are some architectural laws to know and some rules to follow.

First, we need to take a look at the most general possibilities how storage can be architecturally designed:



The topmost question is: do we always need to access bigger masses of (typically unstructured) data over a network?

There is a common belief that both reliability and scalability could be only achieved this way. In the past, local storage has often been viewed as "too simple" to provide both enterprise grade reliability, and scalability. In the past, this was sometimes true.

However, with the advent of a new method called "LV Football" this picture has changed, see chapter 8. We will later review what level of reliability and scalability can be achieved with each of the fundamental models mentioned here.

1.1. What is *Cloud Storage*

According to a popular definition from https://en.wikipedia.org/wiki/Cloud_storage (retrieved June 2018), cloud storage is

(1) Made up of many **distributed resources**, but still **act as one**.

(2) Highly **fault tolerant** through redundancy and distribution of data.

(3) Highly **durable** through the creation of versioned copies.

(4) Typically **eventually consistent** with regard to data replicas.

There are some consequences from this definition:

1. Distributed Storage, in particular BigCluster architectures (see section 1.4): many of them (with few exceptions) are conforming to all of these requirements. Typical granularity are objects, or chunks, or other relatively small units of data.

2. Centralized Storage: does not conform to (1) and to (4) by definition[1]. By introduction of synchronous or asynchronous replication, it can be made to *almost* conform, except for (1) where some concept mismatches remain (probably resolvable by going to a RemoteSharding model on top of CentralStorage, where CentralStorage is only a *sub-component*). Typical granularity is replication of whole storage pools, or of LVs, or of filesystem data.

[1] Notice that sharding on top of CentralStorage is no longer a CentralStorage model by definition, but a RemoteSharding model according to section 1.4.1.

3. LocalStorage, and some further models like RemoteSharding (see section 1.4.1):
   (1) can be achieved at LV granularity with Football (see chapter 8), which creates a **Big Virtual LVM Pool**.
   (2) can be achieved at disk granularity with local RAID, and at LV granularity with DRBD or MARS.
   (3) can be achieved at LV granularity with LVM snapshots, and/or ZFS (or other filesystem) snapshots, and/or above filesystem layer by addition of classical backup.
   (4) can be achieved by MARS, which provides two different consistency guarantees at different levels, *both at the same time*:
   locally: Strict local consistency at LV granularity, also *within* any LV replica.
   globally: Eventually consistent *between* different LV replicas.

1.2. Granularity at Architecture

Here are the most important architectural differences between object-based storages and LV-based (Logical Volume) storages:

	Objects	LVs
Granularity	small (typically KiB)	huge (several TiB)
Number of instances	very high	low to medium
Typical access	random keys	named
Update in place	no	yes
Resize during operation	no	yes
Object support	native	on top of
LV support	on top of	native
Filesystem support	on top of	on top of
Scalable	at cluster	both cluster and grid
Location distances	per datacenter / on campus	long distances possible
Centralized pool management	per cluster	Football uniting clusters
Easy sharding support	cumbersome	yes

1.3. Local vs Centralized Storage

There is some old-fashioned belief that only centralized storage systems, as typically sold by commercial storage vendors, could achieve a high degree of reliability, while local storage were inferior by far. In the following, we will see that this is only true for an ***unfair*** comparison involving different classes of storage systems.

1.3.1. Internal Redundancy Degree

Centralized commerical storage systems are typically built up from highly redundant *internal* components:

1. Redundant power supplies with UPS.
2. Redundancy at the storage HDDs / SSDs.
3. Redandancy at internal transport busses.
4. Redundant RAM / SSD caches.
5. Redundant network interfaces.
6. Redundant compute heads.
7. Redundancy at control heads / management interfaces.

What about local hardware RAID controllers? Many people think that these relatively cheap units were massively inferior at practically each of these points. However, please take a *really deep* look at what classical RAID chip manufacturers like LSI / Avago / Broadcom and their competitors are offering as configuration variants of their top notch models. The following enumeration is in the same order as above (item by item):

1. Redundant hardware RAID cards with BBU caches, each with local goldcaps surviving power outages, their BBU caches cross-coupled via high-speed interconnects.
2. HDD / SSD redundancy: almost any RAID level you can think of.
3. Redundant SAS cross-cabling: any head can access any device.
4. BBU caches are redundant and cross-coupled, similarly to RDMA. When SSD caches are added to both cards, you also get redundancy there.
5. When using cross-coupled redundant cards, you automatically get redundant host bus interfaces (HBAs).
6. The same story: you also get two independent RAID controller instances which can do RAID computations independently from each other. Some implementations do this even in hardware (ASICs).
7. Dito: both cards may be plugged into two different servers, thereby creating redundancy at control level. As a side effect, you may also get a similar functionality than DRBD.

If you compare typical prices for both competing systems, you will notice a huge difference. See also section 1.5.

1.3.2. Capacity Differences

There is another hard-to-die myth: commercial storage would provide higher capacity. Please read the data sheets. It is *possible* (but not generally recommended) to put several hundreds of spindles into several external HDD enclosures, and then connect them to a redundant cross-coupled pair of RAID controllers via several types of SAS busses. By filling a rack this way, you can easily reach similar, if not higher capacities than commercial storage boxes, for a *fraction* of the price.

However, this is not the recommended way for general use cases (but could be an option for low demands like archieving). The big advantage of RAID-based local storage is **massive scale-out by sharding,** as explained in section 1.4.

1.3.3. Caching Differences

A frequent argument is that centralized storage systems had bigger caches than local RAID systems. While this argument is often true, it neglects an important point:

Local RAID systems often *don't need* bigger caches, because they are typically located at the *bottom* of a cache hierarchy, playing only a *particular* role in that hierarchy. There exist *further* caches which are **erronously not considered** by such an argument!

Example, see also section 1.7 for more details: At 1&1 Shared Hosting Linux (ShaHoLin), a typical LXC container containing several thousands to tenthousands of customer home directories, creates a long-term *average(!)* IOPS load at block layer of about 70 IOPS. No, this isn't a typo. It is not 70,000 IOPS. It is only 70 IOPS.

Linux kernel experts know why I am not kidding. The standard Linux kernel has two main caches, the Page Cache for file content, and the Dentry Cache (plus Inode slave cache) for metadata. Both caches are residing in **RAM**, which is the *fastest* type of cache you can get.

Nowadays, typical servers have several hundreds of gigabytes of RAM, sometimes even up to terabytes, resulting in an incredible caching behaviour which can be measured by those people who know how to do it (caution: it can be easily done wrongly).

Many people are neglecting these caches, sometimes not knowing of their existence, and are falsely assuming that 1 application `read()` or `write()` operation will also lead to 1 IOPS at block layer. As a consequence, they are demanding 50,000 IOPS or 100,000 or even 1,000,000 IOPS.

Some (but not all) commercial storage systems can deliver similar IOPS rates, because they have internal RAM caches in the same order of magnitude. People who are buying such systems are typically falling into some of the following classes (list is probably incomplete):

- some people know this, but price does not matter - the more caches, the better. Wasted money for doubled caches does not count for them, or is even viewed as an advantage to them (personally). Original citation of an anonymous person: "only the best and the most expensive storage is good enough for us".

- using NFS, which has extremely poor filesystem caching behaviour because the Linux nfs client implementation does not take full advantage of the dentry cache. Sometimes people know this, sometimes not. It seems that few people have read an important paper on the Linux implementation of nfs. Please search the internet for "Why nfs sucks" from Olaf Kirch (who is one of the original Linux nfs implementors), and *read* it. Your opinion about nfs might change.

- have transactional databases, where high IOPS may be *really* needed, but ***exceptionally***(!) for this class of application. For very big enterprise databases like big SAP installations, there may be a very valid justification for big RAM caches at storage layers. However: smaller transactional loads, as in webhosting, are *often* (not always) hammering a *low* number of **hot spots**, where *big* caches are not really needed. Relatively small BBU caches of RAID cards will do it also. Often people don't notice this because they don't measure the **workingset behaviour** of their application, as could be done for example with `blkreplay` (see https://blkreplay.org).

- do not notice that *well-tuned* filesystem caches over iSCSI are typically demanding much less IOPS, sometimes by several orders of magnitude, and are wasting money with caches at commercial boxes they don't need (classical **over-engineering**).

Anyway, local storage can be augmented with various types of local caches with various dimensioning.

However, there is no point in accessing the fastest possible type of RAM cache remotely over a network. Even expensive hardware-based RDMA cannot deliver the same performance as **directly caching** your data in the ***same* RAM** where your application is running. The Dentry Cache in the Linux kernel provides highly optimized **shared metadata** in SMP and NUMA systems (nowadays scaling to more than 100 processor cores), while the Page Cache provides **shared memory** via hardware MMU. This is crucial for the performance of classical local filesystems.

The physical laws of Einstein and others are telling us that neither this type of caching, nor its shared memory behaviour, can be transported over whatever type of network without causing performance degradation.

1.3.4. Latencies and Throughput

First of all: today there exist only a small number of HDD manufacturers on the world. The number of SSD manufacturers will likely decline in the long run. Essentially, commercial storage vendors are more or less selling you the same HDDs or SSDs as you could buy and deploy yourself. If at all, there are only some minor technical differences.

In the meantime, many people agree to a Google paper that the *ratio* of market prices (price per terabyte) between HDDs and SSDs are unlikely to change in a fundamental[2] way during the next 10 years. Thus, most large-capacity enterprise storage systems are built on top of HDDs.

Typically, HDDs and their mechanics are forming the overall bottleneck.

[2] In folklore, there exists a **fundamental empirical law**, fuzzily called "Storage Pyramid" or "Memory Hierarchy Law" or similar, which is well-known at least in German OS academic circles. The empirical law (extrapolated from **observations**, similarly to Moore's law) tells us that faster storage technology is always **more expensive** than slower storage technology, and that capacities of faster storage are typically always lesser than capacity of slower storage. This observation has been roughly valid for more than 50 years now. You can find it in several German lecture scripts. Unfortunately, the Wikipedia article https://en.wikipedia.org/wiki/Memory_hierarchy (retrieved in June 2018) does not cite this very important fundamental law about **costs**. In contrast, the German article https://de.wikipedia.org/wiki/Speicherhierarchie about roughly the same subject is mentioning "Kosten" which means "cost", and "teuer" which means "expensive".

- by construction, a *local* HDD attached via HBAs or a hardware RAID controller will show the least *additional* overhead in terms of *additional* latencies and throughput degradation caused by the attachment.
- When the *same* HDD is *indirectly* attached via Ethernet or Infiniband or another rack-to-rack transport, both latencies and throughput will become worse. Depending on further factors and influences, the overall bottleneck may shift to the network.

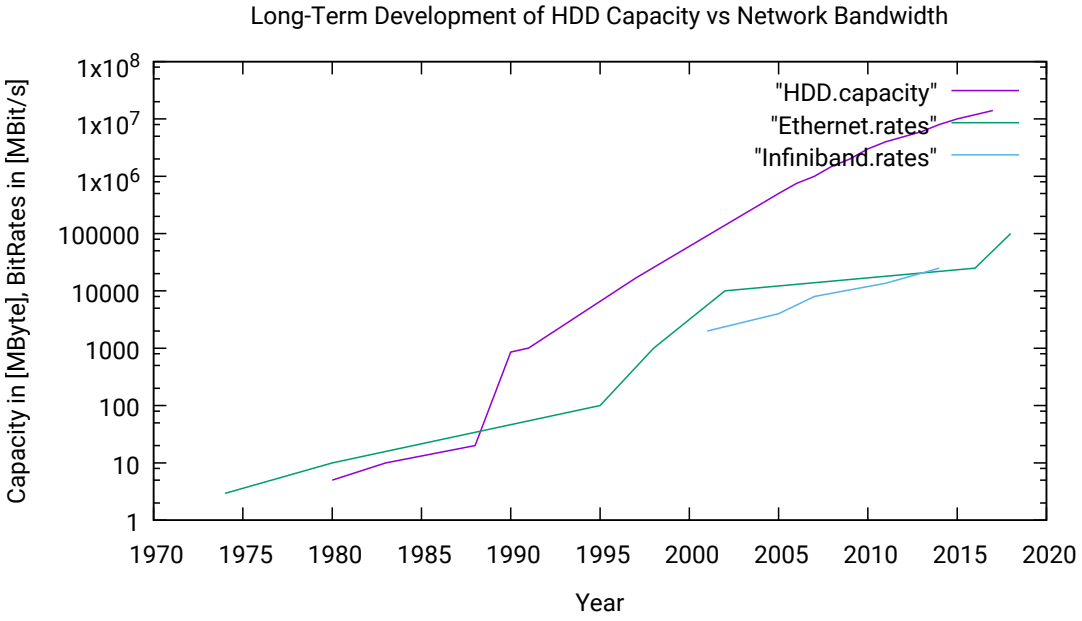
The laws of information transfer are telling us: with increasing distance, both latencies (laws of Einstein) and throughput (laws of energy needed for compensation of SNR = signal to noise ratio) are becoming worse. Distance matters. And the number of intermediate components, like routers / switches and their **queuing**, matters too.

This means that local storage has *always* an advantage in front of any attachment via network. Centralized storages are bound to some network, and thus suffer from disadvantages in terms of latencies and throughput.

What is the expected long-term future? Will additional latencies and throughput of centralized storages become better over time?

It is difficult to predict the future. Let us first look at the past evolution. The following graphics has taken its numbers from Wikipedia articles https://en.wikipedia.org/wiki/List_of_device_bit_rates and https://en.wikipedia.org/wiki/History_of_hard_disk_drives, showing that HDD capacities have grown **over-proportionally** by about 2 orders of magnitude over about 30 years, when compared to the relative growth of network bandwidth.

In the following graphics, effects caused by decreasing form factors have been neglected, which would even *amplify* the trend. For fairness, bundling of parallel disks or parallel communication channels[3] have been ignored. All comparisons are in logarithmic y axis scale:



What does this mean when extrapolated into the future?

It means that concentrating more and more capacity into a single rack due to increasing data density will likely lead to more problems in future. Accessing more and more data over the network will become increasingly more difficult when concentrating high-capacity HDDs or SSDs[4] into the same space volume as before.

In other words: centralized storages are no good idea yet, and will likely become an even worse idea in the future.

[3] It is easy to see that the slopes of `HDD.capacity` vs `Infiniband.rates` are different. Parallelizing by bundling of Infiniband wires will only lift the line a little upwards, but will not alter its slope in logarithmic scale. For extrapolated time $t \to \infty$, the extrapolated empirical long-term behaviour is rather striking.

[4] It is difficult to compare the space density of contemporary SSDs in a fair way. There are too many different form factors. For example, M2 cards are typically consuming even less cm^3/TB than classical 2.5 inch form factors. This trend is likely to continue in future.

Example: there was a major incident at a German web hosting company at the beginning of the 2000's. Their entire webhosting main business was running on a single proprietary highly redundant CentralStorage solution, which failed. Restore from backup took way too long from the viewpoint of a huge number of customers, leading to major press attention. Before this incident, they were the #1 webhoster in Germany. A few years later, 1&1 was the #1 instead. You can speculate whether this has to do with the incident. But anyway, the later geo-redundancy strategy of 1&1 basing on a sharding model (originally using DRBD, later MARS) was motivated by conclusions drawn from this incident.

Another example: in the 1980s, a CentralStorage "dinosaur[5]" architecture called SLED = Single Large Expensive Disk was propagated with huge marketing noise and effort, but its historic fate was predictable for real experts not bound to particular interests: SLED finally lost against their contemporary RAID competition. Nowadays, many people don't even remember the term SLED.

Today's future is likely dominated by **scaling-out architectures** like sharding, as explained in section 1.4.

1.3.5. Reliability Differences CentralStorage vs Sharding

In this section, we look at *fatal* failures only, ignoring temporary failures. A fatal failure of a storage is an incident which needs to be corrected by **restore from backup**.

By definition, even a *highly redundant* CentralStorage is *nevertheless* a SPOF = Single Point of Failure. This also applies to fatal failures.

Some people are incorrectly arguing with redundancy. However, the problem is that *any* system, even a highly redundant one, can fail fatally. There exists no perfect system on earth. One of the biggest known sources of fatal failure is **human error**.

In contrast, sharded storage (for example the LocalSharding model, see also section 1.4.1) has MPOF = Multiple Points Of Failure. It is unlikely that many shards are failing fatally at the same time, because shards are *independent*[6] from each other by definition.

What is the difference from the viewpoint of customers of the services?

When a CentralStorage fails fatally, a *huge* number of customers will be affected for a *long* time (see the example German webhoster mentioned in section 1.3.4). Reason: restore from backup will take extremely long because huge masses of data have to be restored. MTBF = Mean Time Between Failures is (hopefully) longer thanks to redundancy, but MTTR = Mean Time To Repair is also very long.

With (Local)Sharding, the risk of *some* fatal incident *somewhere* in the sharding pool is higher, but the ***size*** of such an incident is smaller in three dimensions at the same time:

1. There are much **less customers affected** (typically only 1 shard out of n shards).
2. **MTTR** = Mean Time To Repair is typically much better because there is much less data to be restored.
3. **Residual risk** plus resulting fatal damage by **un-repairable problems** is thus lower.

What does this mean from the viewpoint of an investor of a big "global player" company?

As is promised by the vendors, let us assume that failure of CentralStorage might be occurring less frequently. But *when* it happens on **enterprise-critical mass data**, the stock exchange value of the affected company will be exposed to a **hazard**. This is not bearable from the viewpoint of an investor.

In contrast, the (Local)Sharding model is *distributing* the **indispensible incidents** (because **perfect systems do not exist**, and **perfect humans do not exist**) to a lower number of

[5] With the advent of NVME, SSDs are almost directly driven by DMA. Accessing any high-speed DMA devices by default via network is a foolish idea, similarly foolish than playing games via an expensive high-end gamer graphics cards which is then *indirectly* attached via RDMA, or even via Ethernet. Probably no serious gamer would ever *try* to do that. But some storage vendors do, for strategic reasons. Probably for their own survival, their customers are to be misguided to overlook the blinking red indicators that centralized SSD storage is likely nothing but an expensive dead end in the history of dinosaur architectures.

[6] When all shards are residing in the same datacenter, there exists a SPOF by power loss or other impacts onto the whole datacenter. However, this applies to both the CentralStorage and to the LocalSharding model. In contrast to CentralStorage, LocalSharding can be more easily distributed over multiple datacenters.

customers with higher frequency, such that the **total impact onto the business** becomes bearable.

Risk analysis of enterprise-critical use cases is summarized in the following table:

	CentralStorage	(Local)Sharding
Probability of *some* fatal incident	lower	higher
# Customers affected	very high	very low
MTBF per storage	higher	lower
MTTR per storage	higher	lower
Unrepairable residual risk	higher	lower
Total impact	higher	lower
Investor's risk	**unbearable**	stock exchange compatible

Summary: CentralStorage is something for

- small to medium-sized companies which don't have the **manpower** and the **skills** for professionally building and operating a (Local)Sharding (or similar) system for their enterprise-critical mass data their business is relying upon.
- ***monolithic* enterprise applications** like classical SAP which are anyway bound to a specific vendor, where you cannot select a different solution (so-called **Vendor Lock-In**).
- when your application **is neither shardable** by construction (c.f. section 1.4), or when doing so would be a too high effort, **nor going to BigCluster**[7] (e.g. Ceph / Swift / etc, see secion 1.6 on page 22) is an option.

   If you have an *already sharded* system, e.g. in webhosting, don't convert it to a non-shardable one, and don't introduce SPOFs needlessly. You will introduce **technical debts** which are likely to hurt back somewhen in future!

As a real big "global player", or as a company being part of such a structure, you should be careful when listening to "marketing drones" of proprietary CentralStorage vendors. Always check your *concrete* use case. Never believe in wrongly generalized claims, which are only valid in some specific context, but do not really apply to your use case. It could be about your *life*.

1.3.6. Proprietary vs OpenSource

In theory, the following dimensions are orthogonal to each other:

Architecture: LocalStorage vs CentralStorage vs DistributedStorage

Licensing: Proprietary vs OpenSource

In practice, however, many vendors of proprietary storage systems are selecting the CentralStorage model. This way, they can avoid inter-operability with their competitors. This opens the door for the so-called **Vendor Lock-In**.

In contrast, the OpenSource community is based on *cooperation*. Opting for OpenSource means that you can **combine and exchange** numerous **components** with each other.

Key OpenSource players are *basing* their business on the **usefulness** of their software components for you, their customer. Please search the internet for further explanations from Eric S. Raymond.

Therefore **interoperability** is a *must* in the opensource business. For example, you can relatively easily migrate between DRBD and MARS, forth and backwards, see section 3.2. The

[7] Theoretically, BigCluster can be used to create 1 single huge remote LV (or 1 single huge remote FS instance) out of a pool of storage machines. Double-check, better triple-check that such a **big *logical* SPOF** is *really* needed, and cannot be circumvented by any means. Only in such a case, the current version of MARS cannot help (yet), because its *current focus* is on a big number of machines each having relatively small LVs. At 1&1 ShaHoLin, the biggest LVs are 40TiB at the moment, running for years now, and bigger ones are certainly possible. Only when current local RAID technology with external enclosures cannot easily create a single LV in the petabyte scale, BigCluster is probably the better solution (c.f. section 1.6 on page 22).

generic block devices provided by both DRBD and MARS (and by the kernel LVM2 implementation, and many others ...) can interact with zillions of filesystems, VMs, applications, and so forth.

Summary: **genericity** is a highly desired property in OpenSource communities, while proprietary products often try to control their usage by limiting either technical interoperability at certain layers, and/or legally by contracts. Trying to do so with OpenSource would make no sense, because *you*, the customer, are the *real* king who can *really* select and combine components. You can form a **really customized system** to your ***real needs***, not as just promised but not always actually delivered by so-called "marketing drones" from commercial vendors who are actually prefering the needs of their employer in front of yours.

There is another fundamental difference between proprietary software and OpenSource: the former is bound to some company, which may *vanish* from the market. Commercial storage systems may be **discontinued**.

This can be a serious threat to your business relying on the value of your data. In particular, buying storage systems from *small* vendors may increase this risk[8].

OpenSource is different: it cannot die, even if the individual, or the (small) company which produced it, does no longer exist. The sourcecode is in the **public**. It just could get *outdated* over time. However, as long as there is enough public interest, you will always find somebody who is willing to adapt and to *maintain* it. Even if you would be the only one having such an interest, you can *hire* a maintainer for it, specifically for your needs. You aren't **helpless**.

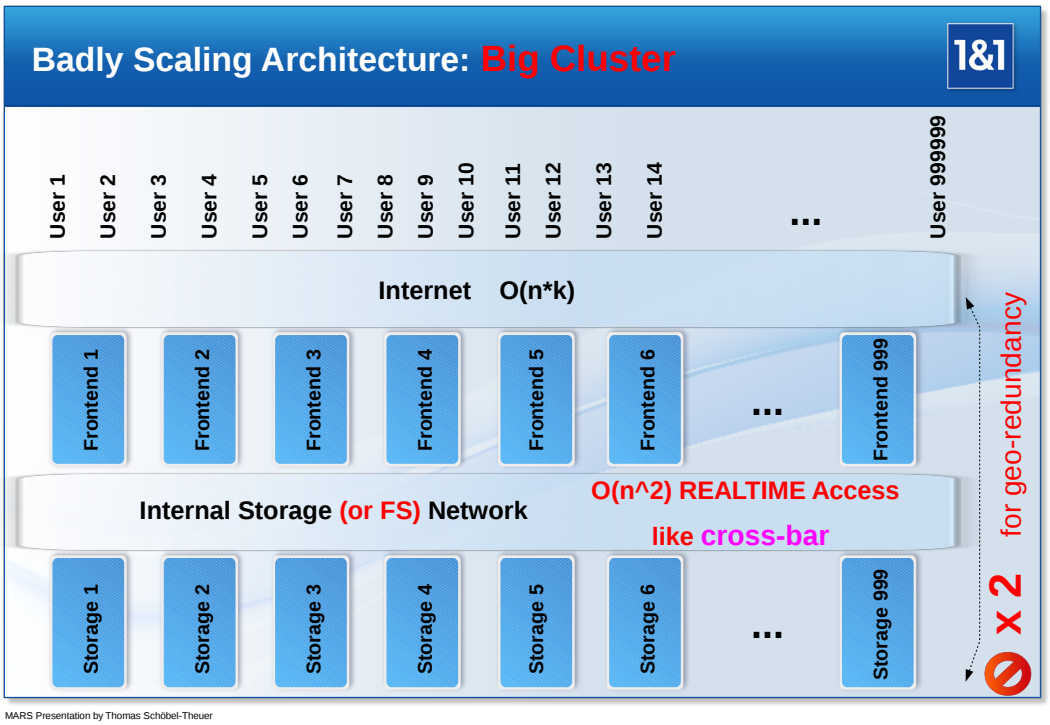
1.4. Distributed vs Local: Scalability Arguments from Architecture

Datacenters aren't usually operated for fun or for hobby. Scalability of an *architecture* is very important, because it can seriously limit your business. Overcoming architectural ill-designs can grow extremely cumbersome and costly.

Many enterprise system architects are starting with a particular architecture in mind, called "Big Cluster". There is a common belief that otherwise **scalability** could not be achieved:

[8] There is a risk of a *domino effect*: once there is a critical incident on highly redundant CentralStorage boxes from a particular (smaller) vendor, this may lead to major public media attention. This may form the *root cause* for such a vendor to vanish from the market. Thus you may be left alone with a buggy system, even if you aren't the victim of the concrete incident.

In contrast, bugs in an OpenSource component can be fixed by a larger community of interested people, or by yourself if you hire somebody for this.



The crucial point is the **storage network** here: n storageservers are interconnected with $m = O(n)$ frontend servers, in order to achieve properties like scalability, failure tolerance, etc.

Since *any* of the m frontends must be able to access *any* of the n storages in realtime, the storage network must be dimensioned for $O(n \cdot m) = O(n^2)$ network connections running in parallel. Even if the total network throughput is scaling only with $O(n)$, nevertheless $O(n^2)$ network connections have to be maintained at connection oriented protocols and at various layers of the operating software. The network has to *switch* the packets from n sources to m destinations (and their opposite way back) in **realtime**.

This **cross-bar functionality** in realtime makes the storage network complicated and expensive. Some further factors are increasing the costs of storage networks:

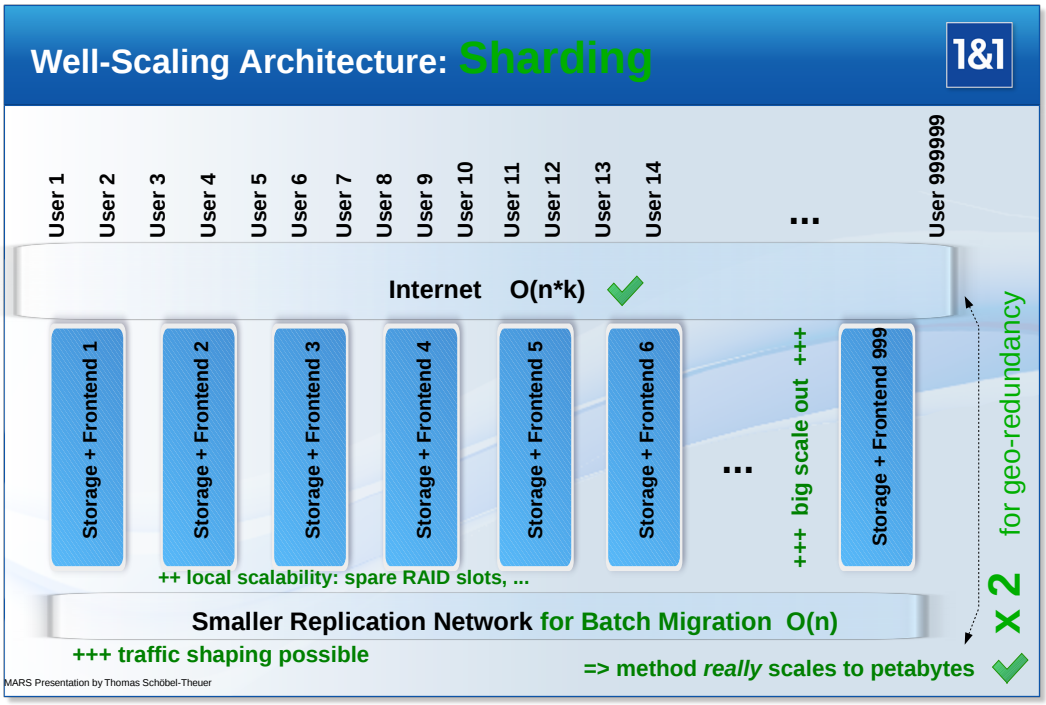
- In order to limit error propagation from other networks, the storage network is often built as a *physically separate* = *dedicated* network.

- Because storage networks are heavily reacting to high latencies and packet loss, they often need to be dimensioned for the **worst case** (load peaks, packet storms, etc), needing one of the best = typically most expensive components for reducing latency and increasing throughput. Dimensioning to the worst case instead of an average case plus some safety margins is nothing but an expensive **overdimensioning** / **over-engineering**.

- When **multipathing** is required for improving fault tolerance of the storage network itself, these efforts will even *double*.

- When geo-redundancy is required, the total effort may easily more than double another time because in cases of disasters like terrorist attacks the backup datacenter must be prepared for taking over for multiple days or weeks.

Fortunately, there is an alternative called "Sharding Architecture" which does not need a dedicated storage network at all, at least when built and dimensioned properly. Instead, it *should have* (but not always needs) a so-called **replication network** which can, when present, be dimensioned much smaller because it does neither need realtime operations nor scalabiliy to $O(n^2)$:



Sharding architectures are extremely well suited when both the input traffic and the data is **already partitioned**. For example, when several thousands or even millions of customers are operating on disjoint data sets, like in web hosting where each webspace is residing in its own home directory, or when each of millions of mySQL database instances has to be isolated from its neighbour. Masses of customers are also appearing at cloud storage applications like Cloud Filesystems (e.g. Dropbox or similar).

Even in cases when any customer may potentially access any of the data items residing in the whole storage pool (e.g. like in a search engine), sharding can be often applied. The trick is to create some relatively simple content-based dynamic switching or redirect mechanism in the input network traffic, similar to HTTP load balancers or redirectors.

Only when partitioning of input traffic plus data is not possible in a reasonable way, big cluster architectures as implemented for example in Ceph or Swift (and partly even possible with MARS when restricted to the block layer) have a very clear use case.

When sharding is possible, it is the preferred model due to reliability and cost and performance reasons.

1.4.1. Variants of Sharding

LocalSharding The simplest possible sharding architecture is simply putting both the storage and the compute CPU power onto the same iron.

Example: at 1&1 Shared Hosting Linux (ShaHoLin), we have dimensioned several variants of this. (a) we are using 1U pizza boxes with local hardware RAID controllers with fast hardware BBU cache and up 10 local disks for the majority of LXC container instances where the "small-sized" customers (up to ~100 GB webspace per customer) are residing. Since most customers have very small home directories with extremely many but small files, this is a very cost-efficient model. (b) less that 1 permille of all customers have > 250 GB (up to 2TB) per home directory. For these few customers we are using another dimensioning variant of the same architecture: 4U servers with 48 high-capacity spindles on 3 RAID sets, delivering a total PV capacity of ~300 TB, which are then cut down to ~10 LXC containers of ~30 TB each.

In order to operate this model at a bigger scale, you should consider the "container football" method as described in section 1.4.3 and in chapter 8 on page 120.

RemoteSharding This variant needs a (possibly dedicated) storage network, which is however only $O(n)$. Each storage server exports a block device over iSCSI (or over another trans-

port) to at most $O(k)$ dedicated compute nodes where k is some **constant**.
Hint 1: it is advisable to build this type of storage network with **local switches** and no routers inbetween, in order to avoid $O(n^2)$-style network architectures and traffic. This reduces error propagation upon network failures. Keep the storage and the compute nodes locally close to each other, e.g. in the same datacenter room, or even in the same rack.
Hint 2: additionally, you can provide some (low-dimensioned) backbone for **exceptional(!)** cross-traffic between the local storage switches. Don't plan to use any realtime cross-traffic *regularly*, but only in clear cases of emergency!

FlexibleSharding This is a dynamic combination of LocalSharding and RemoteSharding, dynamically re-configurable, as explained below.

BigClusterSharding The sharding model can also be placed **on top of** a BigCluster model, or possibly "internally" in such a model, leading to a similar effect. Whether this makes sense needs some discussion. It can be used to reduce the *logical* BigCluster size from $O(n)$ to some $O(k)$, such that it is no longer a "big cluster" but a "small cluster", and thus reducing the serious problems described in section 1.6 to some degree. This could make sense in the following use cases:

- When you **already have** invested into a big cluster, e.g. Ceph or Swift, which does not really scale and/or does not really deliver the expected reliability. Some possible reasons for this are explained in section 1.6.
- When you really need a *single* LV which is necessarily **bigger** than can be reasonably built on top of local LVM. This means, you are likely claiming that you really need **strict consistency** as provided by a block device on more than 1 PB with current technology (2018). Examples are very **big enterprise databases** like classical SAP (c.f. section 1.3), or if you really need **POSIX-compliance** on a single big filesystem instance. Be conscious when you think this is the only solution to your problem. Double-check or triple-check whether there is *really* no other solution than creating such a huge block device and/or such a huge filesystem instance. Such huge SPOFs are tending to create similar problems as described in section 1.6 for similar reasons.

When building a **new** storage system, be sure to check the following use cases. You should seriously consider a LocalSharding / RemoteSharding / FlexibleSharding model in favor of BigClusterSharding when ...

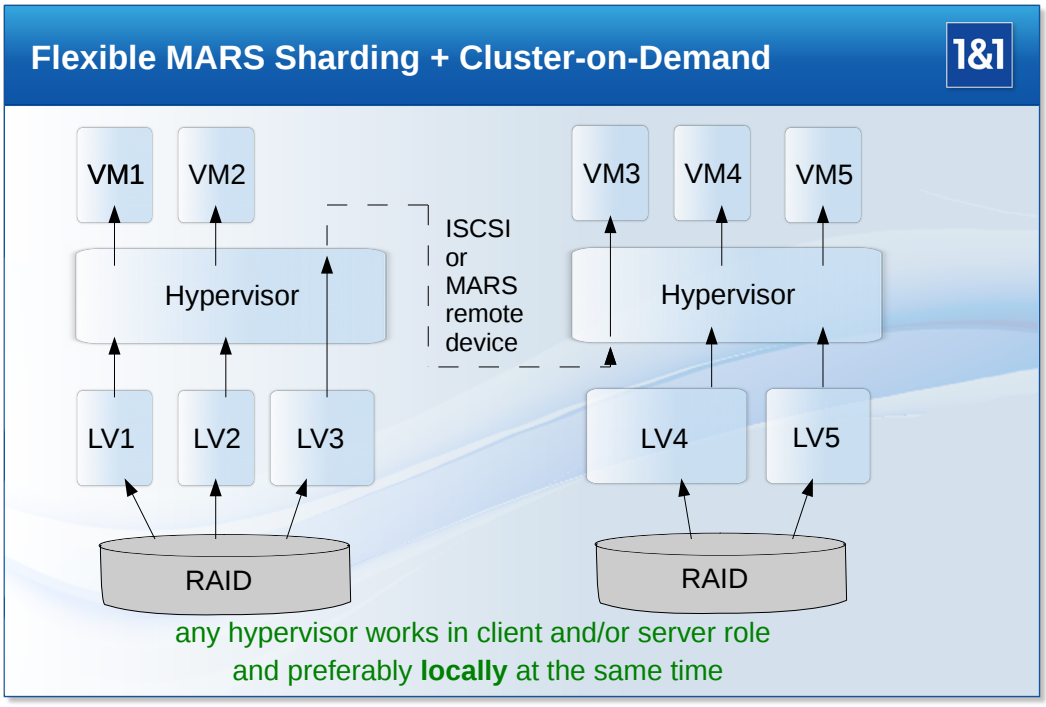
- ... when more than 1 LV instance is placed onto your "small cluster" shards. Then a **{Local,Remote,Flexible}Sharding** model could be likely used instead. Then the total overhead (**total cost of ownership**) introduced by a BigCluster *model* but actually stripped down to a "SmallCluster" *implementation / configuration* should be examined separately. Does it really pay off?

- ... when there are **legal requirements** that you can tell at any time where your data is. Typically, this is all else but easy on a BigCluster model, even when stripped down to SmallCluster size.

1.4.2. FlexibleSharding

Notice that MARS' new remote device feature from the 0.2 branch series (which is kind of replacement for iSCSI) *could* be used for implementing some sort of "big cluster" model at block layer.

Nevertheless, such models re-introducing some kind of "big dedicated storage network" into MARS operations are not the preferred model. Following is the a super-model which combines both the "big cluster" and sharding model at block layer in a very flexible way. The following example shows only two servers from a pool consisting of hundreds or thousands of servers:



The idea is to use iSCSI or the MARS remote device *only where necessary*. Preferably, local storage is divided into multiple Logical Volumes (LVs) via LVM, which are *directly* used *locally* by Virtual Machines (VMs), such as KVM or filesystem-based variants like LXC containers.

In the above example, the left machine has relatively less CPU power or RAM than storage capacity. Therefore, not *all* LVs could be instantiated locally at the same time without causing operational problems, but *some* of them can be run locally. The example solution is to *exceptionally(!)* export LV3 to the right server, which has some otherwise unused CPU and RAM capacity.

Notice that local operations of VMs doesn't produce any storage network traffic at all. Therefore, this is the preferred runtime configuration.

Only in cases of resource imbalance, such as (transient) CPU or RAM peaks (e.g. caused by DDOS attacks), *some* VMs or containers may be run somewhere else over the network. In a well-balanced and well-dimensioned system, this will be the **vast minority**, and should be only used for dealing with timely load peaks etc.

Running VMs directly on the same servers as their storage is a **major cost reducer.**

You simply don't need to buy and operate $n + m$ servers, but only about $\max(n, m) + m \cdot \epsilon$ servers, where ϵ corresponds to some relative small extra resources needed by MARS.

In addition to this and to reduced networking costs, there are further cost savings at power consumption, air conditioning, Height Units (HUs), number of HDDs, operating costs, etc as explained below in section 1.5.

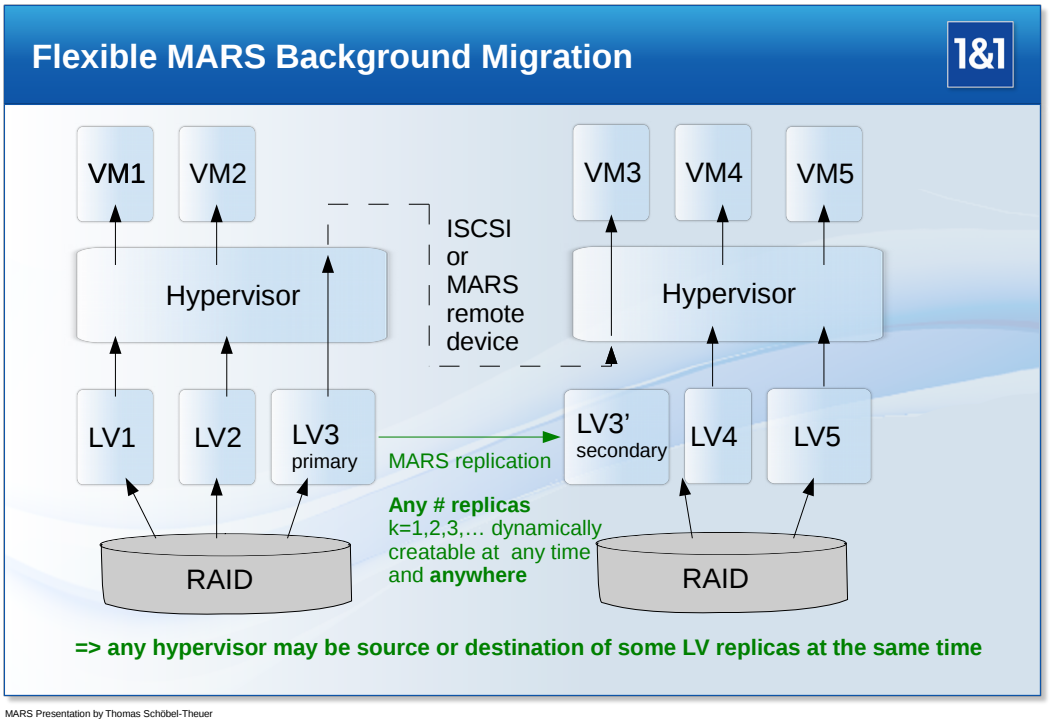
1.4.3. Principle of Background Migration

The sharding model needs a different approach to load balancing of storage space than the big cluster model. There are serveral possibilities at different layers, each addressing different **granularities**:

- Moving customer data at filesystem or database level via `rsync` or `mysqldump` or similar. Example: at 1&1 Shared Hosting Linux, we have about 9 millions of customer home directories. We also have a script `movespace.pl` using incremental `tar` for their moves. Now, if we would try to move around *all* of them this way, it could easily take years or even decades for millions of extremely small home directories, due to overhead like DNS

updates etc. However, there exist a small handful of large customer home directories in the terabyte range. For these, and only for these, it is a clever idea to use `movespace.pl` because thereby the size of a LV can be regulated more fine grained than at LV level.

- Dynamically growing the sizes of LVs during operations: `lvresize` followed by `marsadm resize` followed by `xfs_growfs` or similar operations.
- Moving whole LVs via MARS, as shown in the following example:



The idea is to dynamically create *additional* LV replicas for the sake of **background migration**. Examples:

- In case you had no redundancy at LV level before, you have $k = 1$ replicas during ordinary operation. If not yet done, you should transparently introduce MARS into your LVM-based stack by using the so-called "standalone mode" of MARS. When necessary, create the first MARS replica with `marsadm create-resource` on your already-existing LV data, which is retained unmodified, and restart your application again. Now, for the sake of migration, you just create an additional replica at another server via `marsadm join-resource` there and wait until the second mirror has been fully **synced** in background, while your application is running and while the contents of the LV is modified *in parallel* by your ordinary applications. Then you do a primary **handover** to your mirror. This is usually a matter of minutes, or even seconds. Once the application runs again at the new location, you can delete the old replica via `marsadm leave-resource` and `lvremove`. Finally, you may re-use the freed-up space for something else (e.g. `lvresize` of *another* LV followed by `marsadm resize` followed by `xfs_growfs` or similar). For the sake of some hardware lifecycle, you may run a different strategy: evacuate the original source server completely via the above MARS migration method, and eventually decommission it.
- In case you already have a redundant LV copy somewhere, you should run a similar procedure, but starting with $k = 2$ replicas, and temporarily increasing the number of replicas to either $k' = 3$ when moving each replica step-by-step, or you may even directly go up to $k' = 4$ when moving pairs at once.
  Example: see `football.sh` in the `football/` directory of MARS, which is a checkout of the Football sub-project (see chapter 8).

- When already starting with $k > 2$ LV replicas in the starting position, you can do the same analogously, or you may then use a lesser variant. For example, we have some mission-critical servers at 1&1 which are running $k = 4$ replicas all the time on relatively small but important LVs for extremely increased safety. Only in such a case, you may have the freedom to temporarily decrease from $k = 4$ to $k' = 3$ and then going up to $k'' = 4$ again. This has the advantage of requiring less temporary storage space for *swapping* some LVs.

1.5. Cost Arguments

A common pre-jugdement is that "big cluster" is the cheapest scaling storage technology when built on so-called "commodity hardware". While this is very often true for the "commodity hardware" part, it is often not true for the "big cluster" part. But let us first look at the "commodity" part.

1.5.1. Cost Arguments from Technology

Here are some rough market prices for basic storage as determined around end of 2016 / start of 2017:

Technology	Enterprise-Grade	Price in € / TB
Consumer SATA disks via on-board SATA controllers	no (small-scale)	< 30 possible
SAS disks via SAS HBAs (e.g. in external 14" shelfs)	halfways	< 80
SAS disks via hardware RAID + LVM (+DRBD/MARS)	yes	80 to 150
Commercial storage appliances via iSCSI	yes	around 1000
Cloud storage, S3 over 5 years lifetime	yes	3000 to 8000

You can see that any self-built and self-administered storage (whose price varies with slower high-capacity disks versus faster low-capacity disks) is much cheaper than any commercial offering by about a factor of 10 or even more. If you need to operate several petabytes of data, self-built storage is always cheaper than commercial one, even if additional manpower is needed for commissioning and operating. You don't have to pay the shareholders of the storage provider. Here we just assume that the storage is needed permanently for at least 5 years, as is the case in web hosting, databases, backup / archival systems, and many other application areas.

Commercial offerings of cloud storage are way too much hyped. Some people apparently don't know that the generic term "Cloud Storage" refers to a *storage class*, not to a particular *instance* like original Amazon S3, and that it is possible to build and operate almost any instance of any storage class yourself. From a commercial perspective, **outsourcing** of *huge masses* of enterprise-critical storage (to whatever class of storage) usually pays off **only when** your storage demands are either *relatively low*, or are *extremely* varying over time, and/or when you need some *extra* capacity only *temporarily* for a *very* short time.

1.5.2. Cost Arguments from Architecture

In addition to basic storage prices, many further factors come into play when roughly comparing big cluster architectures versus sharding. The following table bears the *unrealistic assumption* that BigCluster can be reliably operated with 2 replicas (the suffix ×2 means with additional geo-redundancy):

	BC	SHA	BC×2	SHA×2
# of Disks	>200%	<120%	>400%	<240%
# of Servers	≈ ×2	≈ ×1.1 possible	≈ ×4	≈ ×2.2
Power Consumption	≈ ×2	≈ ×1.1	≈ ×4	≈ ×2.2
HU Consumption	≈ ×2	≈ ×1.1	≈ ×4	≈ ×2.2

As shown in section 1.6, two replicas are typically not sufficient for BigCluster. Even addicts of BigCluster are typically recommending 3 replicas in some so-called "best practices", leading to the following more realistic table:

	BC	SHA	BC×2	SHA×2
# of Disks	>300%	<120%	>600%	<240%
# of Servers	≈ ×3	≈ ×1.1 possible	≈ ×6	≈ ×2.2
Power Consumption	≈ ×3	≈ ×1.1	≈ ×6	≈ ×2.2
HU Consumption	≈ ×3	≈ ×1.1	≈ ×6	≈ ×2.2

The crucial point is not only the number of extra servers needed for dedicated storage boxes, but also the total number of HDDs. While big cluster implementations like Ceph or Swift can *theoretically* use some erasure encoding for avoiding full object replicas, their *practice* as seen in internal 1&1 Ceph clusters is similar to RAID-10, but just on objects instead of block-based sectors.

Therefore a big cluster typically needs >300% disks to reach the same net capacity as a simple sharded cluster. The latter can typically take advantage of hardware RAID-60 with a significantly smaller disk overhead, while providing sufficient failure tolerance at disk level.

There is a surprising consequence from this: geo-redundancy is not as expensive as many people are believing. It just needs to be built with the proper architecture. A sharded geo-redundant pool based on hardware RAID-60 (last column "SHA×2") costs typically *less* than a non-georedundant big cluster with typically needed / recommended number of replicas (column "BC"). A geo-redundant sharded pool provides even better failure compensation (see section 1.6).

Notice that geo-redundancy implies by definition that an unforeseeable **full datacenter loss** (e.g. caused by **disasters** like a terrorist attack or an earthquake) must be compensated for **several days or weeks**. Therefore it is *not* sufficient to take a big cluster and just spread it to two different locations.

In any case, a MARS-based geo-redundant sharding pool is cheaper than using commercial storage appliances which are much more expensive by their nature.

1.6. Reliability Arguments from Architecture

A contemporary common belief is that big clusters and their random replication methods would provide better reliability than anything else. There are some practical observations at 1&1 and its daughter companies which cannot confirm this.

Similar experiences are part of a USENIX paper about copysets, see https://www.usenix.org/system/files/conference/atc13/atc13-cidon.pdf. Their proposed solution is different from the solution proposed here, but interestingly their *problem analysis* part contains not only similar observations, but also comes to similar conclusions about random replication. Citation from the abstract:

> However, random replication is **almost guaranteed** to lose data in the common scenario of simultaneous node failures due to cluster-wide power outages. [emphasis added by me]

Stimulated by our practical experiences even in truly less disastrous scenarios than mass power outage, theoretical explanations were sought. Surprisingly, they show that LocalSharding is superior to true big clusters under practically important preconditions. Here is an intutitive explanation. A detailed mathematical description of the model can be found in appendix G on page 144.

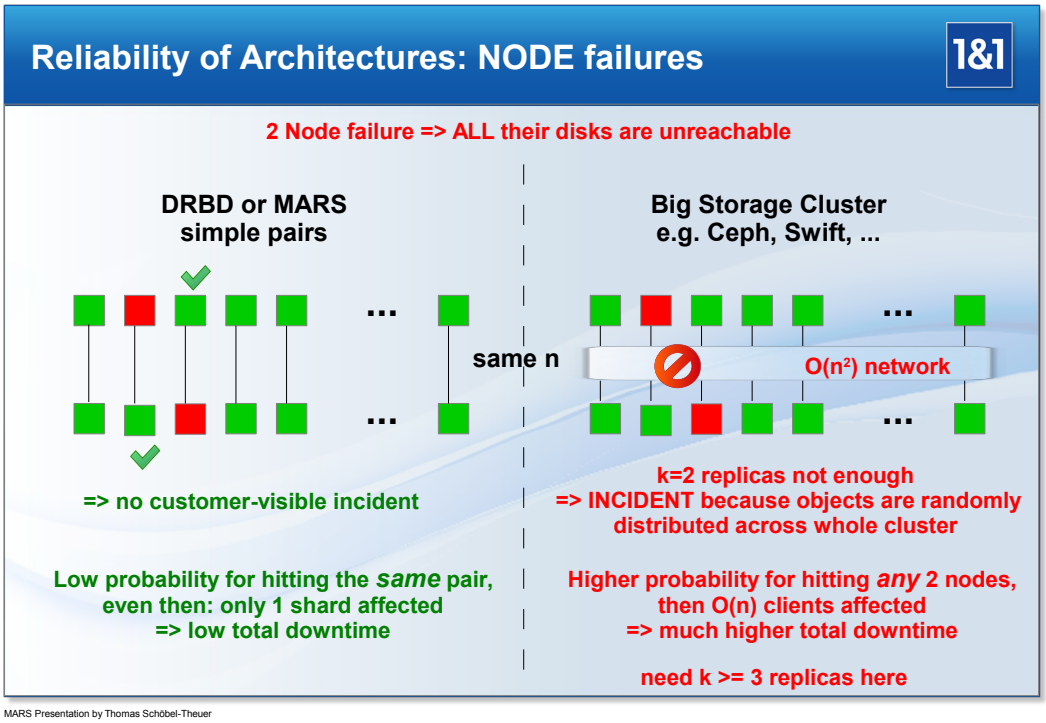
1.6.1. Storage Server Node Failures

1.6.1.1. Simple intuitive explanation

Block-level replication systems like DRBD are constructed for failover in local redundancy scenarios. Or, when using MARS, even for geo-redundant failover scenarios. They are traditionally dealing with **pairs** of servers, or with triples, etc. In order to get a storage incident with them, *both* sides of a DRBD or MARS small-cluster (also called **shard**) must have an incident at the same time.

In contrast, big clusters are spreading their objects over a huge number of nodes $O(n)$, with some redundancy degree k denoting the number of replicas. As a consequence, *any* k node failures out of $O(n)$ will produce an incident. For example, when $k = 2$ and n is equal for both

models, then *any* combination to two node failures occurring at the same time will lead to an incident:



Intuitively, it is easy to see that hitting both members of the same pair at the same time is less likely than hitting *any* two nodes of a big cluster.

If you are curious about some concrete numbers, read on.

1.6.1.2. Detailed explanation

For the sake of simplicity, the following more detailed explanation is based on the following assumptions:

- We are looking at **storage node** failures only.
- Disk failures are regarded as already solved (e.g. by local RAID-6 or by the well-known compensation mechanisms of big clusters). Only in case they don't work, they are mapped to node failures, and are already included in the probability of storage node failures.
- We restrict ourselves to temporary / **transient** failures, without regarding permanent data loss. Otherwise, the differences between local-storage sharding architectures and big clusters would become even worse. When loosing some physical storage nodes forever in a big cluster, it is typically all else but easy to determine which data of which application instances / customers have been affected, and which will need a restore from backup.
- Storage network failures (as a whole) are ignored. Otherwise a fair comparison between the architectures would become difficult. If they were taken into account, the advantages of LocalSharding would become even bigger.
- We assume that the storage network (when present) forms no bottleneck. Network implementations like TCP/IP versus Infiniband or similar are thus ignored.
- Software failures / bugs are also ignored. We only compare *architectures* here, not their various implementations.
- The x axis shows the number of basic storage units n, where one basic storage unit equals to the total disk space provided by one storage node.

- We assume that the number of application instances is linearly scaling with n. For simplicity, we assume that the number of applications running on the whole pool is exactly n.

- For the BigCluster architecture, we assume that all objects are always distributed to $O(n)$ nodes. For simiplicy of the model, we assume a distribution via a *uniform* hash function. When other hash functions were used (e.g. distributing only to a constant number of nodes), it would no longer be a big cluster architecture in our sense.
  In the following example, we assume a uniform object distribution to exactly n nodes. Notice that any other $n' = O(n)$ with $n' < n$ will produce similar results for $n' \to \infty$, but may be better in detail for smaller n'.

- For the LocalSharding (DRBDorMARS) architecture, we assume that only local storage is used. For higher replication degrees $k = 2, \ldots$, the only occurring communication is *among* the pairs / triples / and so on (shards), but no communication to other shards is necessary.

- For simplicity of the example, we assume that any single storage server node used in either architecture, including all of its local disks, has a reliability of 99.99% (four nines). This means, the probability of a storage node failure is uniformly assumed as $p = 0.0001$.

- This means, during an observation period of $T = 10,000$ operation hours, we will have a total downtime of 1 hour per server in statistical average. For simplicity, we assume that the failure probability of a single server does neither depend on previous failures nor on the operating conditions of any other server. It is known that this is not true in general, but otherwise our model would become extremely complex.

- More intuitively, our observation period of $T = 10,000$ operation hours corresponds to about 13 months, or slightly more than a year.

- Consequence: when operating a pool of 10,000 storage servers, then in statistical *average* there will be *almost always* one node which is failed. This is like a "permanent incident" which has to be solved by the competing storage architectures.

- Hint: the term "statistical average" is somewhat vague here, in order to not confuse readers[9]. A more elaborate statistical model can be found in appendix G on page 144.

Let us start the comparison with a simple corner case: plain old servers with no further redundancy, other than their local RAIDs. This naturally corresponds to $k = 1$ replicas when using the DRBDorMARS architecture.

Now we apply the corner case of $k = 1$ replicas to both architectures, i.e. also to BigCluster, in order to shed some spotlight at the fundamental properties of the architectures.

Under the precondition of $k = 1$ replicas, an incident of each one of the n servers has two possible ways to influence the downtime from an application's perspective:

1. Downtime of 1 storage node only influences 1 application unit depending on 1 basic storage unit. This is the case with the DRBDorMARS model, because there is no communication between shards, and we assumed that 1 storage server unit also carries exactly 1 application unit.

2. Downtime of 1 storage node will **tear down more** than 1 application unit, because any of the application units have spread their storage to more than 1 storage node via uniform hashing, as is the case at BigCluster.

For ease of understanding, let us zoom into the special case $n = 2$ and $k = 1$ for a moment. These are the smallest numbers where you already can see the effect. In the following table, we denote 4 possible status combinations out of 2 servers A and B, where the cells are showing the number of application units influenced:

[9] The problem is that sometimes more servers than average can be down, and sometimes less. Average values should not be used in the mathematical model, but exact ones. However, humans can often better imagine when provided with "average behaviour", so we use it here just for the sake of ease of understanding.

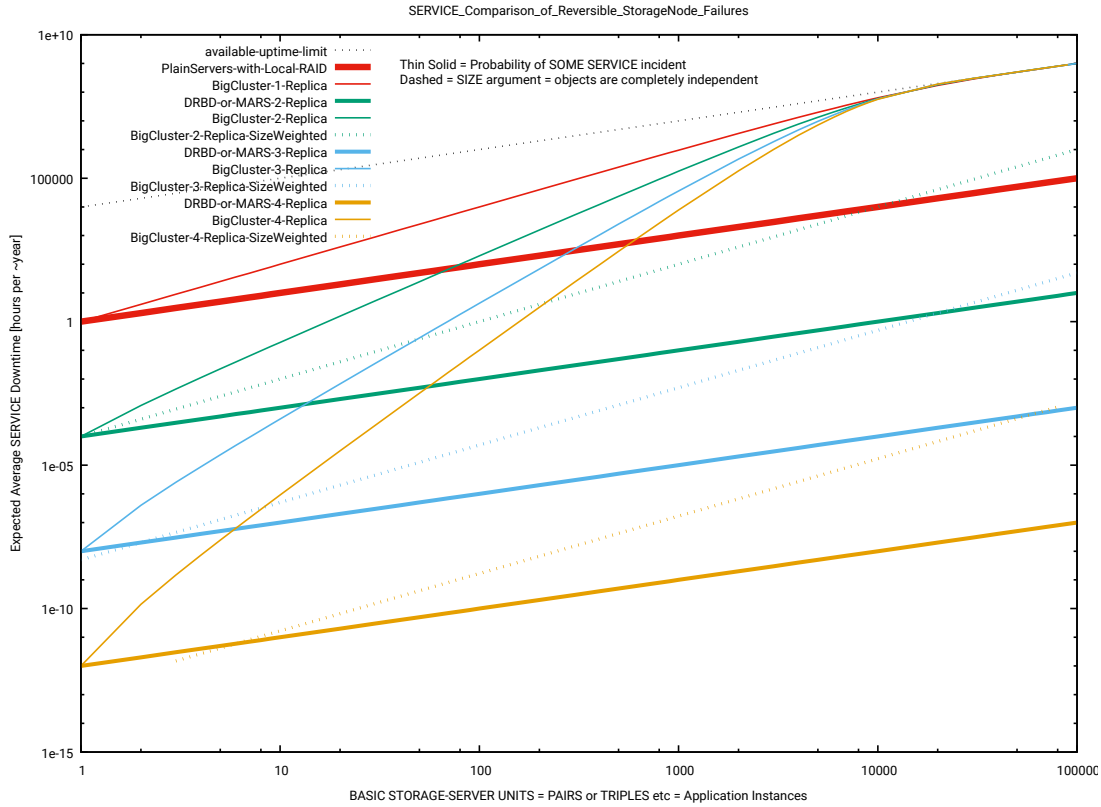
LocalSharding (DRBDorMARS)	A up	A down
B up	0	1
B down	1	2

BigCluster	A up	A down
B up	0	2
B down	2	2

What is the heart of the difference? While a node failure at LocalSharding (DRBDorMARS) will tear down only the local application, the teardown produced by BigCluster will spread to *all* of the $n = 2$ application units, because of the uniform hashing and because we have only $k = 1$ replica.

Would it help to increase both n and k to larger values?

In the following graphics, the thick red line shows the behaviour for $k = 1$ PlainServers (which is the same as $k = 1$ DRBDorMARS) with increasing number of storage units n, ranging from 1 to 10,000 storage units = number of servers for $k = 1$. Higher values of $k \in [1, 4]$ are also displayed. All lines corresponding to the same k are drawn in the same color. Notice that both the x and y axis are logscale:



When you look at the thin solid BigCluster lines for $k = 2, \ldots$ drawn in different colors, you may wonder why they are alltogether converging to the thin red BigCluster line, which corresponds to $k = 1$ BigCluster. And they also converge against the grey dotted topmost line indicating the total possible uptime of all applications (depending on x). It can be explained as follows:

The x axis shows the number of basic storage units. When you have to create 10,000 storage units with a replication degree of $k = 2$ replicas, then you will have to deploy $k{*}10,000 = 20,000$ servers in total. When operating a pool of 20,000 servers, in statistical average 2 servers of them will be down at any given point in time. However, 2 is the same number as the replication degree k. Because our BigCluster model as defined above will distribute *all* objects to *all* servers uniformly, there will almost always *exist* some objects for which no replica is available at any given point in time. This means, you will almost always have a **permanent incident** involving the same number of nodes as your replication degree k, and in turn *some* of your objects will not be accessible at all. This means, at $x = 10,000$ storage units you will loose almost any advantage from increasing the number of replicas. Adding more replicas will no longer help at $x \geq 10,000$ storage units.

Notice that the *solid* lines are showing the probability of *some* incident, disregarding the **size of the incident**.

What's about the *dashed* lines showing much better behaviour for BigCluster?

 Under some further preconditions, it would be possible to argue with the *size* of incidents. However, now a big fat warning. When you are **responsible** for operations of thousands of servers, you should be very conscious about these preconditions. Otherwise you could risk your career. In short:

- When your application, e.g. a smartphone app, consists of accessing only 1 object at all during a reasonably long timeframe, you can safely **assume that there is no interdependency** between all of your objects. In addition, you have to assume (and you should check) that your cluster operating software as a whole does not introduce any further **hidden / internal interdependencies**. Only in this case, and only then, you can take the dashed lines arguing with the number of inaccessible objects instead of with the number of basic storage units.

- Whenever your application uses **bigger structured logical objects**, such as filesystems or block devices or whole VMs / containers, then you likely will get **interdependent objects** at your big cluster storage layer.
  Practical example: experienced sysadmins will confirm that even a data loss rate of only 1/1,000,000 of blocks in a classical Linux filesystem like `xfs` or `ext4` will likely imply the need of an offline filesystem check (`fsck`), which is a major incident for the affected filesystem instances.
  Theoretical explanation: servers are running for a very long time, and filesystems are typically also mounted for a long time. Notice that the probability of hitting any vital filesystem data roughly equals the probability of hitting any other data. Sooner or later, any defective sector in the metadata structures or in freespace management etc will stop your whole filesystem, and in turn will stop your application instance(s) running on top of it.
  Similar arguments hold for transient failures: most classical filesystems are not constructed for compensation of hanging IO, typically leading to **system hangs**.

 Blindly taking the dashed lines will expose you to a high risk of error. Practical experience shows that there are often **hidden dependencies** in many applications, often also at application level. You cannot necessarily see them when inspecting their data structures! You will only notice some of them by analyzing their **runtime behaviour**, e.g. with tools like `strace`. Notice that in general the runtime behaviour of an arbitrary program is **undecidable**. Be cautious when drawing assumptions out of thin air!

1.6.2. Optimum Reliability from Architecture

Another argument could be: don't distribute the BigCluster objects to exactly n nodes, but to less nodes. Would the result be better than DRBDorMARS LocalSharding?

When distributing to $O(k')$ nodes with some constant k', we have no longer a BigCluster architecture, but a mixed BigClusterSharding form.

As can be generalized from the above tables, the reliability of **any** BigCluster on $k' > k$ nodes is **always** worse than of LocalSharding on exactly k nodes, where k is also the redundancy degree. In general:

> **The LocalSharding model is the optimum model for reliability of operation, compared to any other model truly distributing its data and operations over truly more nodes, like RemoteSharding or BigClusterSharding or BigCluster does.**

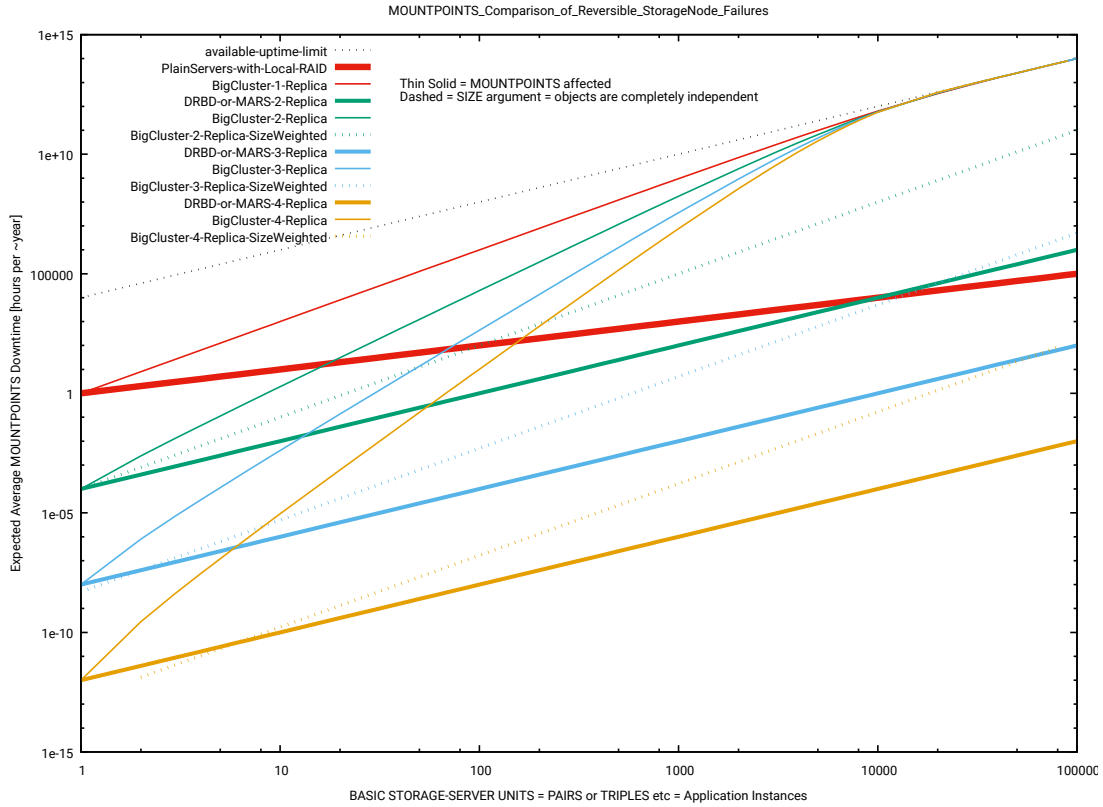
There exists no better model because shards consisting of exactly k nodes where k is the redundancy degree are already the *smallest possible shards* under the assumptions of section 1.6.1.2. Any other model truly involving $k' > k$ nodes for distribution of objects at any shard is **always** worse in the dimension of reliability. Thus the above sentence follows by induction.

The above sentence is formulating a **fundamental law of storage systems**.

1.6.3. Error Propagation to Client Mountpoints

The following is only applicable when filesystems (or their objectstore counterparts) are exported over a storage network, in order to be mounted in parallel at $O(n)$ mountpoints each.

In such a scenario, any problem / incident inside of your storage pool for the filesystem instances will be spread to $O(n)$ clients, leading to an increase of the incident size by a factor of $O(n)$ when measured in number of affected mountpoints:



As a results, we now have a total of $O(n^2)$ mountpoints = our new basic application units. Such $O(n^2)$ architectures are quickly becoming even worse than before. Thus a clear warning: don't try to build systems in such a way.

Notice: DRBD or MARS are traditionally used for running the application on the same box as the storage. Thus they are not vulnerable to these kinds of failure propagation over network. Even with traditional iSCSI exports over DRBD or MARS, you won't have suchalike problems. Your only chance to increase the error propagation are $O(n)$ NFS or `glusterfs` exports to $O(n)$ clients leading to a total number of $O(n^2)$ mountpoints, or similar setups.

Clear advice: don't do that. It's a bad idea.

1.6.4. Similarities and Differences to Copysets

This section is mostly of academic interest. You can skip it when looking for practical advice.

The USENIX paper about copysets (see https://www.usenix.org/system/files/conference/atc13/atc13-cidon.pdf) relates to the Sharding model in the following way:

Similarities The concept of Random Replication of the storage data to large number of machines will reduce reliability. When chosing too big sets of storage machines, then the storage system as a whole will become practically unusable. This is common sense between the USENIX paper and the Sharding Approach as propagated here.

Differences The USENIX paper and many other Cloud Storage approaches are *presuming* that there exists a storage network, allowing real-time distribution of replicas over this kind of network.

In contrast, the Sharding Approach to Cloud Storage tries to *avoid* real-time storage networks *as much as possible*. Notice that RemoteSharding and further variants (including future improvements) do *not* preclude it, but are trying to *avoid* real-time storage network traffic. Instead, the load-balancing problem is addressed via **background data migration**.

This changes the *timely granularity* of data access: many real-time accesses are *shifted over* to migration processes, which in turn are weakening the requirements to the network.

In detail, there are some more differences to the USENIX paper. Some examples:

- Terminology: the scatter width S is defined (see page 39 of the paper) as: each node's data is split *uniformly* across a group of S *other* nodes. In difference, we neither assume uniformity, nor do we require the data to be distributed to *other* nodes. By using the term "other", the USENIX paper (as well as many other BigCluster approaches) are probably presuming something like a distinction between "client" and "server" machines: while data processing is done on a "client", data storage is on a "server".

- We don't disallow this in variants like RemoteSharding or FlexibleSharding and so on, but we gave some arguments why we are trying to *avoid* this.

- It seems that some definitions in the USENIX paper may implicitly relate to "each chunk". In contrast, the Sharding Approach typically relates to LVs (logical volumes), which could however be viewed as a special case of "chunk", e.g. by minimizing the number of chunks in a system. However notice: there exists definitions of "chunk" where it is the basic transfer unit. An LV has the fundamental property that small-granularity **update in place** (at any offset inside the LV) can be executed.

- Notice: we do not preclude further fine-grained distribution of LV data, but this is something which should be *avoided* if not absolutely necessary. Preferred method in typical practical use cases: some storage servers may have some spare RAID slots to be populated later, by resizing the PVs = Physical Volumes before resizing LVs.

- Notice that a typical local RAID system *is also* a Distributed System, according to some reasonable definition. Typical RAID implementations just involve SAS cables instead of Ethernet cables or Infiniband cables. Notice that this also applies to many "Commodity Hardware" approaches, like Ceph storage nodes driving dozens of local HDDs connected over SAS or SATA. The main difference is just that instead of a hardware RAID controller, a hardware HBA = Host Bus Adapter is used instead. Instead of Ethernet switches, SAS multiplexers in backplanes are used. Anyway, this forms a locally distributed sub-system.

- Future variants of the Sharding Approach might extend this already present locally Distributed System to a somewhat wider one. For example, creation of a local LV (called "disk" in MARS terminology) could be implemented by a subordinate DRBD instance implementing a future RAID-10 mode over local Infiniband or crossover Ethernet cables, avoiding local switches. While DRBD would essentially create the "local" LV, the higher-level MARS instance would then be responsible for its wide-distance replication. See chapter 2 about use cases of MARS vs DRBD. Potential future use cases could be *extremely huge* LVs where external SAS disk shelves are no longer sufficient to get the desired capacity.

- The USENIX paper needs to treat the following parameters as more or less fixed (or only slowly changable) **constants**, given by the system designer: the replication degree R, and the scatter width S. In contrast, the replication degree k of our Sharding Approach is not necessarily firmly given by the system, but can be **dynamically changed** at runtime on a per-LV basis. For example, during background migration via MARS the command `marsadm join-resource` is used for creating additional per-LV replicas. However notice: this freedom is limited by the total number of deployed hardware nodes. If you want $k = 3$ replicas at the *whole* pool, then you will need to (dynamically) deploy at least about $k * x$ nodes in general.

- The USENIX paper defines its copysets on a per-chunk basis. Similarly to before, we can transfer this definition to a Sharding Approach by relating it to a per-LV basis. As a side effect, a copyset can then trivially become identical to S when the definition is S is also

changed to a per-LV basis, analogously. In the Sharding Approach, a distiction is not absolutely necessary, while the USENIX paper has to invest some effort into clarifying the relationship between S and copysets as defined on a BigCluster model.

- Neglecting the mentioned differences, we see our typical use case (LocalSharding) roughly equivalent to $S = R$ in the terminology of the USENIX paper, or to $S = k$ (our number of replicas) in our terminology.

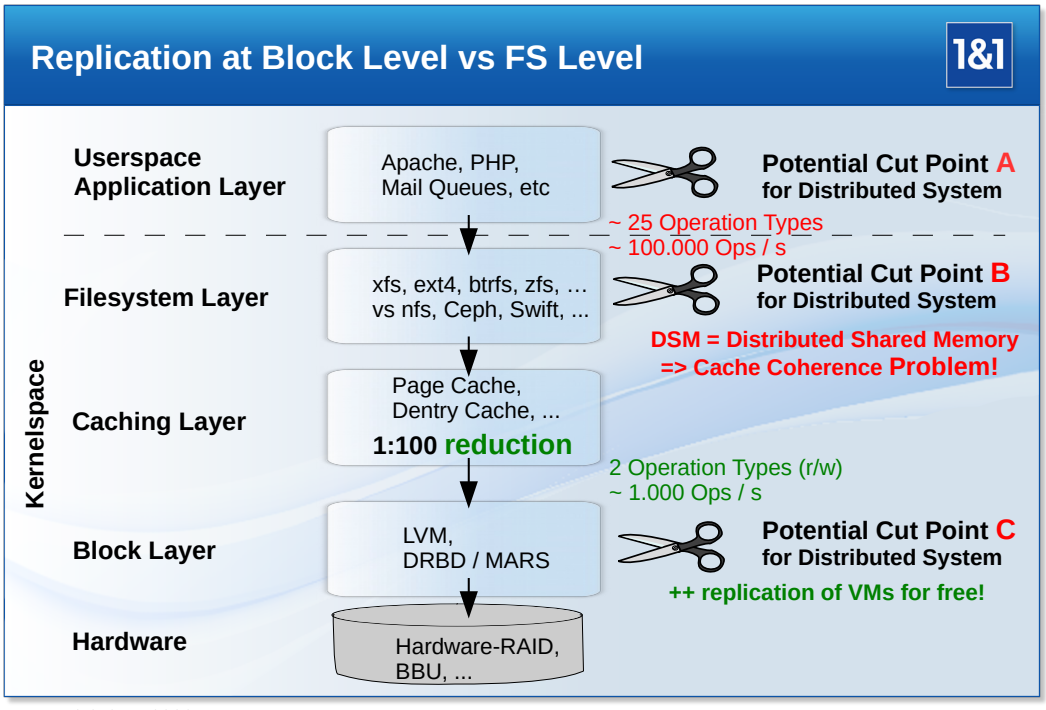
- This means: we try to minimize the *size* of S for any given per-LV k, which will lead to the best possible reliability (under the conditions described in section 1.6.1.2) as has been shown in section 1.6.2.

1.7. Performance Arguments from Architecture

Some people think that replication is easily done at filesystem layer. There exist lots of cluster filesystems and other filesystem-layer solutions which claim to be able to replicate your data, sometimes even over long distances.

Trying to replicate several petabytes of data, or some billions of inodes, is however a much bigger challenge than many people can imagine.

Choosing the wrong layer for **mass data replication** may get you into trouble. Here is an explanation why replication at the block layer is more easy and less error prone:



The picture shows the main components of a standalone Unix / Linux system. In the late 1970s / early 1980s, a so-called *Buffer Cache* had been introduced into the architecture of Unix. Today's Linux has refined the concept to various internal caches such as the **Page Cache** (for data) and the **Dentry Cache** (for metadata).

All these caches serve one main purpose[10]: they are reducing the load onto the storage by exploitation of fast RAM. A well-tuned cache can yield high cache hit ratios, typically 99%. In some cases (as observed in practice) even more than 99.9%.

Now start distributing the system over long distances. There are potential cut points A and B and C[11].

[10] Another important purpose is **providing shared memory**.

[11] In theory, there is another cut point D by implementing a generically distributed cache. There exists some academic research on this, but practically usable enterprise-grade systems are rare and not wide-spread.

Cut point A is application specific, and can have advantages because it has knowledge of the application. For example, replication of mail queues can be controlled much more fine-grained than at filesystem or block layer.

Cut points B and C are *generic*, supporting a wide variety of applicactions, without altering them. Cutting at B means replication at filesystem level. C means replication at block level.

When replicating at B, you will notice that the caches are *below* your cut point. Thus you will have to re-implement **distributed caches**, and you will have to **maintain cache coherence**.

When replicating at C, the Linux caches are *above* your cut point. Thus you will receive much less traffic, typically already reduced by a factor of 100, or even more. This is much more easy to cope with. You will also profit from **journalling filesystems** like `ext4` or `xfs`. In contrast, *truly distributed*[12] journalling is typically not available with distributed cluster filesystems.

A *potential* drawback of block layer replication is that you are typically limited to active-passive replication. An active-active operation is not impossible at block layer (see combinations of DRBD with `ocfs2`), but less common, and less safe to operate.

This limitation isn't necessarily caused by the choice of layer. It is simply caused by the **laws of physics**: communication is always limited by the speed of light. A distributed filesystem is nothing else but a logically **distributed shared memory** (DSM).

Some decades of research on DSM have shown that there exist applications / workloads where the DSM model is *inferior* to the direct communication paradigm. Even in short-distance / cluster scenarios. Long-distance DSM is extremely cumbersome.

Therefore: you simply shouldn't try to solve long-distance communication needs via communication over filesystems. Even simple producer-consumer scenarios (one-way communication) are less performant (e.g. when compared to plain TCP/IP) when it comes to distributed POSIX semantics. There is simply too much **synchronisation overhead at metadata level**.

If you have a need for mixed operations at different locations in parallel: just split your data set into disjoint filesystem instances (or database / VM instances, etc). All you need is careful thought about the *appropriate granularity* of your data sets (such as well-chosen *sets* of user homedirectory subtrees, or database sets logically belonging together, etc).

Replication at filesystem level is often at single-file granularity. If you have several millions or even billions of inodes, you may easily find yourself in a snakepit.

Conclusion: active-passive operation over long distances (such as between continents) is even an advantage. It keeps you from trying bad / almost impossible things.

1.8. Scalability Arguments from Architecture

Some people are talking about scalability by (1) looking at a relatively small example cluster *implementation* of their respective (pre-)chosen *architecture* having n machines or n network components or running n application instances, and then (2) extrapolating its behaviour to bigger n. They think if it runs with small n, it will also run for bigger n.

This way of thinking and acting is completely broken, and can endanger both companies and careers.

1.8.1. Example Failures of Scalability

The following description is a **must read** for sysadmins and system architects, and also for managers who are **responsible**. The numbers and some details are from my memory, thus it need not be 100% accurate in all places.

It is about an operation environment for a *new* product, which was a proprietary web page editor running under a complicated variant of a LAMP stack.

The setup started with a `BigCluster` *architecture*, but actually sized as a `"SmallCluster"` implementation.

Setup 1 (NFS) The first setup consisted of $n = 6$ storage servers, each replicated to another datacenter via DRBD. Each were exporting their filesystems via NFS to about the same number of client servers, where Apache/PHP was supposed to serve the HTTP requests from the

[12] In this context, "truly" means that the POSIX semantics would be always guaranteed cluster-wide, and even in case of partial failures. In practice, some distributed filesystems like NFS don't even obey the POSIX standard *locally* on 1 standalone client. We know of projects which have *failed* right because of this.

customers, which were entering the client cluster via a HTTP load balancer. The load balancer was supposed to spread the HTTP load to the client servers in a **round-robin** fashion.

At this point, eager readers may notice some similarity with the error propagation problem treated in section 1.6.3 on page 27. Notice that this is about *scalability* instead, but you should compare with that, to find some similarities.

After the complicated system was built up and was working well enough, the new product was launched via a marketing campaign with free trial accounts, limited to some time.

So the number of customers was ramping up from 0 to about 20,000 within a few weeks. When about 20,000 customers were running on the client machines, system hangs were noticed, and also from a customer's perspective. When too many customers were pressing the "save" button in parallel on reasonably large web page projects, a big number of small files, including a huge bunch of small image files, was generated over a short period of time. A few customers were pressing the "save" button several times a minute, each time re-creating all of these files again and again from the proprietary web page generator. Result: the system appeared to hang.

However, all of the servers, including the storage servers, were almost *idle* with respect to CPU consumption. RAM sizes were also no problem.

After investigating the problem for a while, it was noticed that the ***network*** was the bottleneck, but not in terms of throughput. The internal sockets were forming some **queues** which were *delaying* the NFS requests in some **ping-pong** like fashion, almost resulting in a "deadlock" from a customer's perspective (a better term would be **distributed livelock** or **distributed thrashing**).

Setup 2 (`ocfs2`) Due to some external investigations and recommendations, the system was converted from NFS to `ocfs2`. Now DRBD was operated in active-active mode. Only one system software component was replaced with another one, without altering the `BigCluster` architecture, and without changing the number of servers, which remained a stripped-down `SmallCluster` implementation.

Result: the problem with the "hangs" disappeared.

However, after the number of customers had exceeded the **next scalability limit** of about 30,000 customers, the "hang" problem appeared once again, in a similar way. The system showed systematical incidents again.

Setup 3 (`glusterfs` as a substitute for NFS) After investigating the network queueing behaviour and the lock contention problems of `ocfs2`, the next solution was `glusterfs`.

However, when the number of customers exceeded the ***next*** **scalability limit**, which was about 50,000 customers hammering the cluster with their "save" button, the "hangs" appeared again.

Setup 4 (`glusterfs` replication as a substitute for DRBD) After analyzing the problem once again, it was discovered by accident that `drbdadm disconnect` *appeared* to "solve" the problem.

Therefore DRBD was replaced with `glusterfs` replication. There exists a `glusterfs` feature allowing replication of files at filesystem level.

This attempt was *immediately* resulting in an almost fatal disaster, and thus was stopped immediately: the cluster completely broke down. Almost nothing was working anymore.

The problem was even worse: switching off the `glusterfs` replication and rollback to DRBD did not work. The system remained unusable.

As a temporary workaround, `drbdadm disconnect` was improving the situation enough for some humbling operation.

Retrospective explanation: some of the reasons can be found in section 2.1.1 on page 38. `glusterfs` replication does not scale at all because it stores its replication information at **per-inode granularity** in EAs (extended attributes), which must *necessarily* be worse than DRBD, because there were some hundreds of millions of them in total as reported by `df -i` (see the cut point discussion in section 1.7 on page 29). Overnight in some cron jobs, these EAs had to be deleted in reasonably sized batches in order to become more or less "operable" again.

Setup5 (Sharding) After the almost fatal incident had been resolved to a less critical one, the responsibility for setup was taken over by another person. After the $O(n^2)$ behaviour from section 1.4 on page 15 had been understood, and after it was clear that sharding is only $O(k)$ from a customer's perspective, it was the final solution. Now the problem was resolved at ***architectural level***, no longer by just replacing some components with some others.

The system was converted to a variant of a `RemoteSharding` model (see section 1.4.1 on page 17), and some `migrate` scripts were introduced for load balancing of customer homedirectories and databases between shards.

As a side effect, the load balancer became a new role: instead of spreading *all* of the HTTP requests to *all* of the client servers in a round-robin fashion, it now acted as a redirection mechanism at *shard granularity*, e.g. when one of the client servers was handed over to another one for maintenance.

This setup is working until today, scaling up to the current number of customers, which is more than an order of magnitude, in the range of about a million of customers. Of course, the number of shards had to be increased, but this is just what sharding is about.

1.8.2. Properties of Storage Scalability

1.8.2.1. Influence Factors at Scalability

In general, scalability of storage systems depends on the following factors (list may be incomplete):

1. The **application class**, in particular its principal **workingset behaviour** (in both dimensions: timely and locality). More explanations about workingsets can be found at http://blkreplay.org.

2. The **size** x of the application data and/or the **number of application instances** (possibly also denoted by x), and the amount of storage needed for it (could be also termed x). Besides the data itself, the corresponding **metadata** (inodes, indexes, etc) can form an important factor, or can even *dominate* the whole story. Typically, critical datacenter application data is tremendously differently sized from workstation data.

    Caution! Many people think erronously that scalability would be *linearly* depending on x. However, as is known at least since the 1960s (read some ancient papers from Saltzer and/or from Denning), scalability is **never linear**, but sometimes even ***disruptive***, in particular when RAM size is the bottleneck. IO queues and/or networking queues are often also reacting to overload in a disruptive fashion. This means: after exceeding the **scalability limit** of a particular system for its particular class of applications, the system will always **break down** from a customer's perspective, sometimes almost completely, and sometimes even ***fatally***.

    On the other hand, some other systems are reacting with **graceful degradation**. Whether a particular systems reacts to a particular type of (over)load, either with graceful degradation, or with fatal disruption, or with some intermediate behaviour, is some sort of "quality property" of the system and/or of the application.

    EVERY SYSTEM, even sharded systems, and even the internet as a whole, has *always* some scalability limit *somewhere*. There exists **no "inifinitely scaling" system** on earth!

3. The ***distribution*** of the application behaviour in both **timely** and **locality** dimensions. Depending on the application class, this is often an *exponential* distribution according to Zipf's law. By falsely *assuming* an equal distribution (or a Gaussian distribution) instead of actually measuring the distribution in both dimensions, you can easily induce zillions of costly problems for big x, or even fatal failure of the whole system / project.

4. The **transformation** of the application workingset behaviour at architectural level, sometimes caused by certain components resp their specific implementation or parameteriza-

tion. Examples are intermediate virtualization layers, e.g. vmware `*.vmdk` or KVM `*.qcow2` container formats which can completely change the game, not only in extreme cases. Another example is **random distribution** to object stores, which can turn some uncomplicated sequential workloads into highly problematic *random IO* workloads. Don't overlook such potential pitfalls!

5. The storage **architecture** to be chosen, such as `CentralStorage` vs `BigCluster` vs `*Sharding`. Choice of the wrong architecture can be fatal for big n and/or for certain timely / spatial application behaviour. Changing an architecture during operations on some petabytes of data and/or some billions of inodes can be almost impossible, and/or can consume a lot of time and money.

6. The **number** of storage **nodes** n. In some architectures, addition of more nodes can make the system *worse* instead of better, c.f. section 1.6 on page 22.

7. In case of architectures relying on a storage network: choice of **layer** for cut point, e.g. filesystem layer vs block layer, see section 1.7 on page 29, and/or introduction of an additional intermediate object storage layer (which can result in major degradation from an architectural view). Due to fundamental differences in distributed vs local **cache coherence**, suchalike can have a *tremendous* effect on scalability.

8. The **implementation** of the architecture. Be sure to understand the difference between an *architecture* and an *implementation* of that architecture.

9. The size and types / properties of various **caches** at various layers. You need to know the general properties of **inclusive** vs **exclusive** cache architecture. You absolutely need to know what **thrashing** is, and under which conditions it can occur.
   It is advantagous for system architects to know[13] pre-loading strategies, as well as replacement strategies. It is advantageous to know what `LRU` or `MFU` means, what their induced *overhead* is, and how they *really* work on *actual* data, not just on some artificial lab data. You also should know what an **anomaly** is, and how it can be produced not only by `FIFO` strategies, but also by certain types of ill-designed multi-layer caching. Beware: there are places where `FIFO`-like behaviour is almost impossible to avoid, such as networks. All of these is outside the scope of this MARS manual. You should *measure*, when possible, the **overhead** of cache implementations. I know of *examples* where caching is *counter-productive*. For example, certain types and implementations of SSD caches are over-hyped. Removing a certain cache will then *improve* the situation. Notice: caches are conceptually based on some type of **associative memory**, which is either very costly when directly implemented in hardware, or can suffer from tremendous performance penalties when implemented inappropriately in software.

10. **Hardware dimensioning** of the implementation: choice of storage hardware, for each storage node. This includes SSDs vs HDDs, their attachment (e.g. SAS multiplexing bottlenecks), RAID level, and controller limitations, etc.

11. Only for architectures relying on a storage network: network **throughput** and network **latencies**, and network **bottlenecks**, including the **queueing** behaviour / congestion control / **packet loss** behaviour upon overload. The latter is often neglected, leading to unexpected behaviour at load peaks, and/or leading to costly over-engineering (examples see section 1.8.1 on page 30).

12. ***Hidden*** **bottlenecks** of various types. A complete enumeration is almost impossible, because there are too many "opportunities". To reduce the latter, my general advice is to try to build bigger systems as *simple* as possible. This is why you should involve some *real* experts in storage systems, at least on critical enterprise data.

 *Any* of these factors can be dangerous when not carefully thought about and treated, depending on your use case.

[13] Reading a few Wikipedia articles does not count as "knowledge". You need to be able to *apply* your knowdedge to enterprise level systems (as opposed to workstation-sized systems), *sustainable* and *reproducible*. Therefore you need to have *actually worked* in the matter and gained some extraordinary experiences, on top of deep understanding of the matter.

1.8.2.2. Example Scalability Scenario

To get an impression what "enterprise critical data" can mean in a concrete example, here are some characteristic numbers on 1&1 ShaHoLin (Shared Hosting Linux) around spring 2018, which would be the *input parameters* for *any* potential solution architecture `CentralStorage` vs `BigCluster` vs `Sharding`:

- About 9 millions of customer homedirectories.
- About 10 billions of inodes, with daily incremental backup.
- More than 4 petabytes of *net* data (total `df` filling level) in spring 2018, with a growth rate of 21% per year.
- All of this permanently replicated into a second datacenter.
- Webhosting very close to 24/7/365. For maintenance, any resource must be switchable to the other datacenter at any time, indepently from other resources; while in catastrophic failure scenarios *all* resources must be switchable within a short time.

For simplicity of our sandbox game, we assume that all of this is in one campus. In reality, about 30% is residing in another continent. Introducing this as an additional input parameter would not fundamentally change the game. Many other factors, like dependencies from existing infrastructure, are also neglected.

Theoretical Solution: `CentralStorage` Let us assume somebody would try to operate this on classical `CentralStorage`, and let us assume that migration of this amount of data including billions of inodes would be no technical problem. What would be the outcome?

With current technology, finding a single `CentralStorage` appliance would be all else but easy. Dimensioning would be needed for the *lifetime* of such a solution, which is at least 5 years. In five years, the data would grow by a factor of about $1.21^5 = 2.6$, which is then about 10.5 petabytes. This is only the *net* capacity; at hardware layer much more is needed for spare space and for local redundancy. The single `CentralStorage` instance will need to scale up to at least this number, in one datacenter (under the simplified game assumptions).

The current number of client LXC containers is about 2600, independent from location. You will have to support growth in number of them. For maintenance, any of these need to be switchable to a different location at any time. The number of bare metal servers running them can vary with hardware architecture / hardware lifecycle, and with growth. You will need to dimension a dedicated storage network for all of this.

If you find a solution which can do this with current `CentralStorage` technology for the next 5 years, then you will have to ensure that restore from backup[14] can be done in less than 1 day in case of a fatal disaster, see also treatment of `CentralStorage` reliability in section 1.3.5 on page 13. Notice that the current self-built backup solution for a total of 15 billions of inodes is based on a sharding model; converting this to some more or less centralized solution turns out as another challenge.

Attention! Buying 10 or 50 or 100 CentralStorage instances does not count as a `CentralStorage` architecture. By definition, suchalike would be `RemoteSharding` instead. Notice that the current 1&1 solution is already a mixture of `LocalSharding` and `RemoteSharding`, so you would win *nothing* at architectural level.

In your business case, you would need to justify the price difference between the current component-based hardware solution (horizontally extensible by *scale-out*) and `CentralStorage`, which is about a factor of 10 per terabyte according to the table in section 1.5. Even if you manage to find a vendor who is willing to subsidize to a factor of only 3, this is not all you need. You need to add the costs for the dedicated storage network. On top of this, you need to account for the *migration costs* after the lifetime of 5 years has passed, where the full data set needs to be migrated to a successor storage system.

Notice that classical argumentations with ***manpower*** will not work. The current operating team is about 10 persons, with no dedicated storage admin. This relatively small team is

[14] Local snapshots, whether LVM or via some COW filesystem, do not count as backups! You need a *logical* copy, not a *physical* one, in case your production filesystem instance gets damaged.

not only operating a total of more than 6,000 shared boxes in all datacenters, but also some tenthousands of managed dedicated servers, running essentially the same software stack, with practically fully automated mass deployment. Most of their tasks are related to central software installation, which is then automatically distributed, and to operation / monitoring / troubleshooting of masses of client servers. Storage administration tasks in isolation are costing only a *fraction* of this. Typical claims that `CentralStorage` would require less manpower will not work here. Almost everything which is needed for *mass automation* is already automated.

Neglecting the tenthousands of managed dedicated servers would be a catastrophic ill-design. Their hardware is already given, by existing customer contracts, some of them decades old. You simply cannot fundamentally change the hardware of these customers including their *dedicated* local disks, which is their *main selling point*. You cannot simply convert them to a shared `CentralStorage`, even if it would be technically possible, and if it would deliver similar IOPS rates than tenthousands of local spindles (and if you could reach the bundled performance of local SSDs from newer contracts), and even if you would introduce some interesting **storage classes** for all of this. A dedicated server on top of a shared storage is no longer a dedicated one. You would have to migrate these customers to another product, with all of its consequences. Alone for these machines, *most*[15] of the current automation of `LocalStorage` is needed *anyway*, although they are not geo-redundant at current stage.

Conclusion: `CentralStorage` is simply *unrealistic*.

Theoretical Solution: `BigCluster` The main problem of `BigCluster` is **reliability**, as explained intuitively in section 1.6 on page 22 and mathematically in appendix G on page 144, and as observed in numerous installations not working as expected.

Let us assume that all of these massive technical problems were solved, somehow. Then the business case would have to deal with the following:

The total number of servers would need to be roughly *doubled*. Not only their CAPEX, but also the corresponding OPEX (electrical power, rackspace, manpower) would increase. Alone their current electrical power cost, including cooling, is more than the current sysadmin manpower cost. Datacenter operations would also increase. On top, a dedicated storage network and its administration would also be needed.

With respect to the tenthousands of managed dedicated servers and their customer contracts, a similar argument as above holds. You simply cannot convert them to `BigCluster`.

Conclusion: `BigCluster` is also *unrealistic*. There is nothing to win, but a lot to loose.

Current Solution: `LocalSharding`, sometimes `RemoteSharding` Short story: it works since decades, and is both cheap and robust since geo-redundancy had been added around 2010.

With the advent of Football (see chapter 8 on page 120), the `LocalSharding` architecture is raising up on par with the most important management abilities of `CentralStorage` and `BigCluster` / Software Defined Storage.

The story with the tenthousands of managed dedicated servers is arguing vice versa: without the traditional ShaHoLin sharding architecture and all of its automation, including the newest addition called Football, the product "managed dedicated servers" would not be possible in this scale.

Summay: the sharded "shared" product enables another "dedicated" product, which is sharded by definition, and it actually is known to scale up by at least another order of magnitude (in terms of number of servers).

1.8.3. Scalability of Filesystem Layer vs Block Layer

Following factors are responsible for better architectural scalability of the block layer vs the filesystem layer, at least in many cases, with a few exceptions (list may be incomplete):

1. **Granularity** of access: **metadata** is often smaller than the content data it refers to, but access to data is typically not possible without accessing corresponding metadata *first*. When masses of metadata are present (e.g. some billions of inodes as in section 1.8.2.2),

[15] Only a few out of >1000 self-built or customized Debian packages are dealing with MARS and/or with the clustermanager `cm3`.

and when it is accessed **more frequently** than the corresponding data (e.g. in stateless designs like Apache), it is likely to become the bottleneck.

 Neglecting metadata and its access patterns is a major source of ill-designs. I know of projects which have failed (in their original setup) because of this. Repair will typically involve some non-trivial architectural changes.

 By default, the block layer itself has almost no metadata at all (or only tiny ones, such as describing a whole block device). Therefore it has an *inherent advantage* over the filesystem layer in such use cases.

2. **Caching**: shared memory caches in kernelspace (page cache + dentry cache) vs distributed caches over network. See the picture in section 1.7 on page 29.

    There exist *examples* where shared distributed caches do not work at all. I know of *several* projects which have failed. Another project than mentioned in section 1.8.1 on page 30 has failed because of violations of POSIX filesystem semantics.

3. Only in distributed systems: the **cache coherence problem**, both on metadata and on data. Depending on load patterns, this can lead to tremendous performance degradation, see example in section 1.8.1.

4. Dimensioning of the **network**: throughput, latencies, queueing behaviour.

There exist a few known exceptions (list may be incomplete, please report further examples if you know some):

- Databases: these are typically operating on specific container formats, where no frequent *external* metadata access is necessary, and where no sharing of the *container as such* is necessary. Typically, there is no big difference between storing them in block devices vs local filesystems.

   Exception from the exception: MyISAM is an old design from the 1980s, originally based on DBASE data structures. Don't try to access them over NFS or similar. Or, better, try to avoid them at all if possible.

- VM images: these are logical BLOBS, so there is typically no big difference whether you have an intermediate *true* filesystem layer, or not.

   Filesystems on top of object stores are no true filesystems. They are violating Dijkstra's important layering rules, as stated in his famous articles on THE. A similar argument holds for block devices on top of object stores. Intermediate container formats like `*.vmdk` or `*.qcow2` can also act as game changers. This does not mean that you have to avoid them at all. However, be sure to **check their influence**, and don't forget their *workingset* and their *caching behaviour* (which can go both into positive and into negative direction), in order to really *know what you are doing!*

1.9. Recommendations for Designing and Operating Storage Systems

In order of precedence, do the following:

1. **Fix and/or limit and/or tune the *application*.**
   Some extreme examples:
   - When you encounter a classical Unix **fork bomb**, you have no chance against it. Even the "best and the most expensive hardware" is unable to successfully run a fork bomb. The only countermeasure is *limitation of resources*. Reason: unlimited resources do not exist on earth.

- If you think that this were only of academic interest: several types of internet **DDOS attacks** are acting like a fork bomb, and **Apache** is also acting similar to a fork bomb when not configured properly. This is not about academics, it is about *your survival* (in the sense of Darwin).
- If you think it cannot hurt you because you are running `fast-cgi` or another application scheme where forks are not part of the game (e.g. databases and many others): please notice that **network queues** are often acting as a replacement for processes. Overflow of queues can have a similar effect than fork bombs from the viewpoint of customers: they simply don't get the service they are expecting.
- Real-life example: some percentage of `WordPress` customers are typically and *systematically* **misconfiguring** their `wp-cron` cron jobs. They create backups of their website, which *include* their old backups. Result: in each generation of the backups, the needed disk space will roughly *double*. Even if you had "unlimited storage" on top of the "best and the most expensive storage system", and even if you would like to give "unlimited storage" to your customers, it simply cannot work at all. Exponential growth is exponential growth. After a few months of this kind of daily backup, you would need more storage than atoms exist in the whole universe. You *must* introduce some quota limits somewhere. And you *must* ensure that the `wp-cron` misconfiguration is fixed, whoever is responsible for fixing it.
- Another `WordPress` example: the `wp-cron` configuration syntax is not easily understandable by laymen. It is easy to **misconfigure** such that a backup is created *once per minute*. As long as the website is very small, this will not even be noticed by sysadmins. However, for bigger websites (and they are typically growing over time), the IO load may increase to a point until even asynchronous replication over 10Gig interfaces cannot catch up. Even worse: the next run of `wp-cron` may start before the old one has finished within a minute. Again, there is no chance except fixing the *root cause* at application level.

2. **Choose the right *overall* architecture** (not limited to storage).
   An impressive example for ill-design can be found in section 1.8.1. Important explanations are in section 1.8.2, in particular subsection 1.8.2.1 on page 32, and section 1.8.3 on page 35. A strategic example is in subsection 1.8.2.2. It is absolutely necessary to know the standard cache hierarchy of Unix (similarly also found in Windows) from section 1.7 on page 29. More explanations are in this manual at many places.

    In general, major ill-designs of overall architectures (end-to-end) cannot be fixed at component level. Even the "best tuning of the world" executed by the "best tuning expert" on top of the "best and most expensive storage *components* and the best storage *network* of the world" cannot compensate major ill-designs, such as $O(n^2)$ behaviour.

    Similarly for reliability: if you have problems with too many and/or too large incidents affecting too many customers, read sections 1.6 on page 22 and 1.3.5 on page 13.

3. **Choice and tuning of components**.
   No further explanations necessary, because most people already know this. In case you think this is the only way: no, it is typically the *worst* and typically only the *last resort* when compared to the previous enumeration items.
   Exception: choice of wrong components with insufficient properties for your particular application / use case. But this is an *architectural* problem in reality.

2. Use Cases for MARS vs DRBD

DRBD has a long history of successfully providing HA features to many users of Linux. With the advent of MARS, many people are wondering what the difference is. They ask for recommendations. In which use cases should DRBD be recommended, and in which other cases is MARS the better choice?

The following table is a short guide to the most important cases where the decision is rather clear:

Use Case	Recommendation
server pairs, each directly connected via **crossover cables**	DRBD
active-active / dual-primary, e.g. `gfs2`, `ocfs2`	DRBD
distance > **50km**	MARS
> **100 server pairs** over a short-distance **shared** line	MARS
all else / intermediate cases	read the following details

There exist some use cases where DRBD is clearly better than MARS. 1&1 has a long history of experiences with DRBD where it works very fine, in particular coupling Linux devices rack-to-rack via crossover cables. DRBD is just *constructed* for that use case (RAID-1 over network). In such a scenario, DRBD is better than MARS because it uses up less disk space resources. In addition, newer DRBD versions can run over high-speed but short-distance interconnects like Infiniband (via the SDP protocol). Another use case for DRBD is active-active / dual-primary mode, e.g. `ocfs2`[1] over short[2] distances.

On the other hand, there exist other use cases where DRBD did not work as expected, leading to incidents and other operational problems. We analyzed them for our specific use cases. The later author of MARS came to the conclusion that they could only be resolved by fundamental changes in the overall architecture of DRBD. The development of MARS started at the personal initiative of the author, first in form of a personal project during holidays, but later picked up by 1&1 as an official project.

MARS and DRBD simply have **different application areas**.

In the following, we will discuss the pros and cons of each system in particular situations and contexts, and we shed some light at their conceptual and operational differences.

2.1. Network Bottlenecks

2.1.1. Behaviour of DRBD

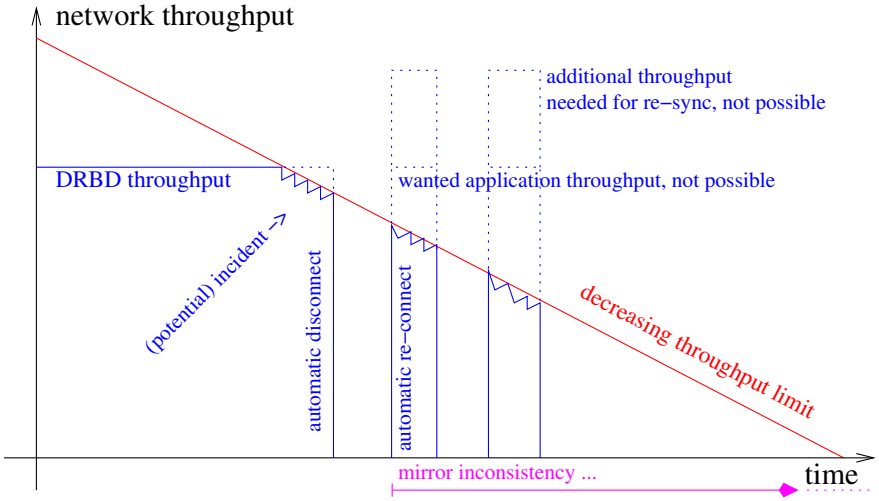
In order to describe the most important problem we found when DRBD was used to couple whole datacenters (each encompassing thousands of servers) over metro distances, we strip down that complicated real-life scenario to a simplified laboratory scenario in order to demonstrate the effect with minimal means.

[1] Notice that `ocfs2` is appearantly not constructed for long distances. 1&1 has some experiences on a specific short distance cluster where the `ocfs2` / `DRBD` combination scaled a little bit better than `NFS`, but worse than `glusterfs` (using 2 clients in both cases – notice that `glusterfs` showed extremely bad performance when trying to enable active-active `glusterfs` replication between 2 server instances, therefore we ended up using active-passive DRBD replication below a single `glusterfs` server). Conclusion: `NFS` < `ocfs2` < `glusterfs` < sharding. We found that `glusterfs` on top of active-passive DRBD scalability was about 2 times better than `NFS` on top of active-passive DRBD, while `ocfs2` on top of `DRBD` in active-active mode was somewhere inbetween. All cluster comparisons with an increasing workload over time (measured as number of customers which could be safely operated). Each system was replaced by the next one when the respective scalability was at its respective end, each time leading to operational problems. The ultimate solution was to replace all of these clustering concepts by the general concept of **sharding**.

[2] Active-active won't work over long distances at all because of high network latencies (cf chapter 1). Probably, for replication of whole clusters over long distances DRBD and MARS could be stacked: using DRBD on top for MARS for active-active clustering of `gfs2` or `ocfs2`, and a MARS instance *below* for failover of *one* of the DRBD replicas over long distances.

Notice that the following DRBD effect does not appear at crossover cables. The following scenario covers a non-standard case of DRBD. DRBD works fine when no network bottleneck appears!

The following picture illustrates an effect which has been observed in 1&1 datacenters when running masses of DBRD instances through a single network bottleneck. In addition, the effect is also reproducible by an elder version of the MARS test suite[3]:



The simplified scenario is the following:

1. DRBD is loaded with a low to medium, but constant rate of write operations for the sake of simplicity of the scenario.

2. The network has some throughput bottleneck, depicted as a red line. For the sake of simplicity, we just linearly decrease it over time, starting from full throughput, down to zero. The decrease is very slowly over time (some minutes, or even hours).

What will happen in this scenario?

As long as the actual DRBD write throughput is lower than the network bandwidth (left part of the horizontal blue line), DRBD works as expected.

Once the maximum network throughput (red line) starts to fall short of the required application throughput (first blue dotted line), we get into trouble. By its very nature, DRBD works **synchronously**. Therefore, it *must* transfer all your application writes through the bottleneck, but now it is impossible[4] due to the bottleneck. As a consequence, the application running on top of DRBD will see increasingly higher IO latencies and/or stalls / hangs. We found practical cases (at least with former versions of DRBD) where IO latencies exceeded practical monitoring limits such as 5 s by far, up to the range of *minutes*. As an experienced sysadmin, you know what happens next: your application will run into an incident, and your customers will be dissatisfied.

In order to deal with such situations, DRBD has lots of tuning parameters. In particular, the `timeout` parameter and/or the `ping-timeout` parameter will determine when DRBD will give up in such a situation and simply drop the network connection as an emergency measure. Dropping the network connection is roughly equivalent to an automatic `disconnect`, followed by an automatic re-connect attempt after `connect-int` seconds. During the dropped connection, the incident will appear as being resolved, but at some hidden cost[5].

[3] The effect has been demonstrated some years ago with DRBD version 8.3.13. By construction, is is independent from any of the DRBD series 8.3.x, 8.4.x, or 9.0.x.

[4] This is independent from the DRBD protocols A through C, because it just depends on an information-theoretic argument independently from any protocol. We have a fundamental conflict between network capabilities and application demands here, which cannot be circumvented due to the **synchronous** nature of DRBD.

[5] By appropriately tuning various DRBD parameters, such as `timeout` and/or `ping-timeout`, you can keep the impact of the incident below some viable limit. However, the automatic disconnect will then happen

What happens next in our scenario? During the `disconnect`, DRBD will record all positions of writes in its bitmap and/or in its activity log. As soon as the automatic re-connect succeeds after `connect-int` seconds, DRBD has to do a partial re-sync of those blocks which were marked dirty in the meantime. This leads to an *additional* bandwidth demand[6] as indicated by the upper dotted blue box.

Of course, there is *absolutely no chance* to get the increased amount of data through our bottleneck, since not even the ordinary application load (lower dotted lines) could be transferred.

Therefore, you run at a **very high risk** that the re-sync cannot finish before the next `timeout` / `ping-timeout` cycle will drop the network connection again.

What will be the final result when that risk becomes true? Simply, your secondary site will be *permanently* in state `inconsistent`. This means, you have lost your redundancy. In our scenario, there is no chance at all to become consistent again, because the network bottleneck declines more and more, slowly. It is simply *hopeless*, by construction.

In case you lose your primary site now, you are lost at all.

Some people may argue that the probability for a similar scenario were low. We don't agree on such an argumentation. Not only because it really happens in pratice, and it may even last some days until problems are fixed. In case of **rolling disasters**, the network is very likely to become flaky and/or overloaded shortly before the final damage. Even in other cases, you can easily end up with inconsistent secondaries. It occurs not only in the lab, but also in practice if you operate some hundreds or even thousands of DRBD instances.

The point is that you can produce an ill behaviour *systematically* just by overloading the network a bit for some sufficient duration.

 When coupling whole datacenters via some thousands of DRBD connections, any (short) network loss will almost certainly increase the re-sync network load each time the outage appears to be over. As a consequence, overload may be *provoked* by the re-sync repair attempts. This may easily lead to self-amplifying **throughput storms** in some resonance frequency (similar to self-destruction of a bridge when an army is marching over it in lockstep).

The only way for reliable prevention of loss of secondaries is to start any re-connect *only* in such situations where you can *predict in advance* that the re-sync is *guaranteed* to finish before any network bottleneck / loss will cause an automatic disconnect again. We don't know of any method which can reliably predict the future behaviour of a complex network.

 Conclusion: in the presence of network bottlenecks, you run a considerable risk that your DRBD mirrors get destroyed just in that moment when you desperately need them.

 Notice that crossover cables usually never show a behaviour like depicted by the red line. Crossover cables are *passive components* which normally[7] either work, or not. The binary connect / disconnect behaviour of DRBD has no problems to cope with that.

---

earlier and more often in practice. Flaky or overloaded networks may easily lead to an enormous number of automatic disconnects.

[6] DRBD parameters `sync-rate` resp `resync-rate` may be used to tune the height of the additional demand. In addition, the newer parameters `c-plan-ahead`, `c-fill-target`, `c-delay-target`, `c-min-rate`, `c-max-rate` and friends may be used to dynamically adapt to *some* situations where the application throughput *could* fit through the bottleneck. These newer parameters were developed in a cooperation between 1&1 and Linbit, the maker of DRBD.

Please note that lowering / dynamically adapting the resync rates may help in lowering the *probability* of occurrences of the above problems in practical scenarios where the bottleneck would recover to viable limits after some time. However, lowering the rates will also increase the *duration* of re-sync operations accordingly. The *total amount of re-sync data* simply does not decrease when lowering `resync-rate`; it even tends to increase over time when new requests arrive. Therefore, the *expectancy value* of problems caused by *strong* network bottlenecks (i.e. when not even the ordinary application rate is fitting through) is *not* improved by lowering or adapting `resync-rate`, but rather the expectancy value mostly depends on the *relation* between the amount of holdback data versus the amount of application write data, both measured for the duration of some given strong bottleneck.

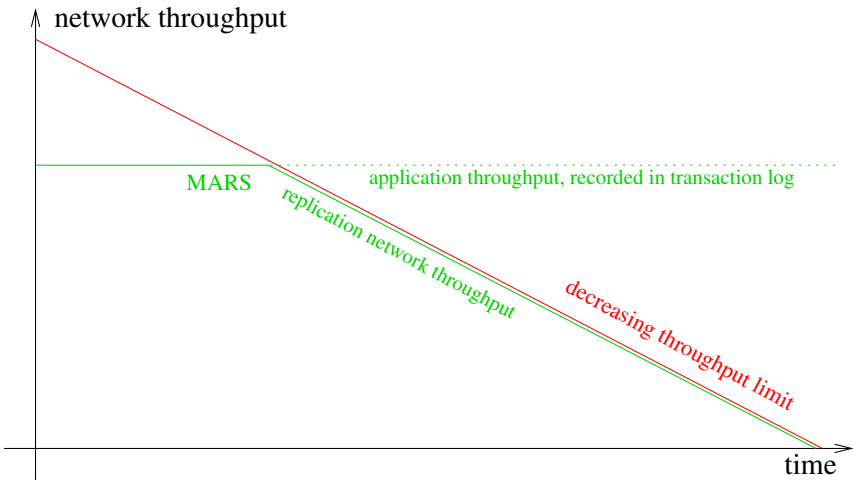
[7] Exceptions might be mechanical jiggling of plugs, or electro-magnetical interferences. We never noticed any of them.

 or  Linbit recommends a **workaround** for the inconsistencies during re-sync: LVM snapshots. We tried it, but found a *performance penalty* which made it prohibitive for our concrete application. A problem seems to be the cost of destroying snapshots. LVM uses by default a BOW strategy (Backup On Write, which is the counterpart of COW = Copy On Write). BOW increases IO latencies during ordinary operation. Retaining snapshots is cheap, but reverting them may be very costly, depending on workload. We didn't fully investigate that effect, and our experience is a few years old. You might come to a different conclusion for a different workload, for newer versions of system software, or for a different strategy if you carefully investigate the field.

 DRBD problems usually arise *only* when the network throughput shows some "awkward" analog behaviour, such as overload, or as occasionally produced by various switches / routers / transmitters, or other potential sources of packet loss.

2.1.2. Behaviour of MARS

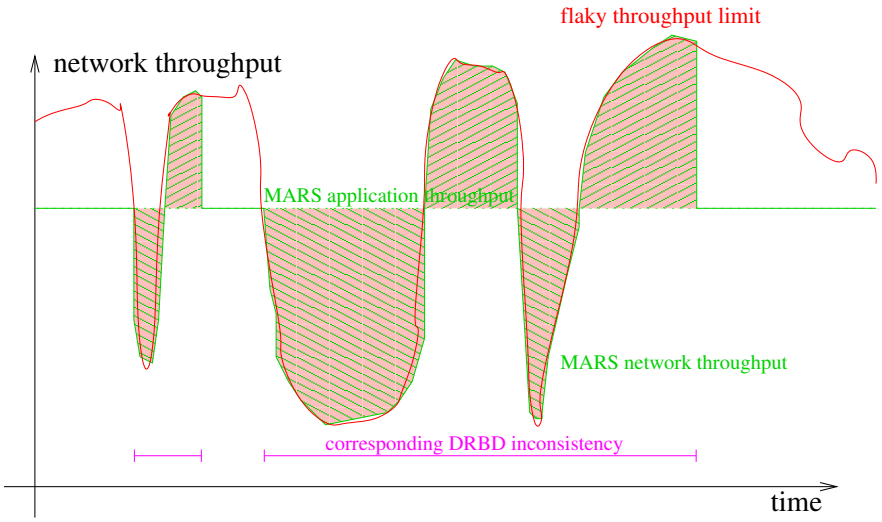
The behaviour of MARS in the above scenario:



When the network is restrained, an asynchronous system like MARS will continue to serve the user IO requests (dotted green line) without any impact / incident while the actual network throughput (solid green line) follows the red line. In the meantime, all changes to the block device are recorded at the transaction logfiles.

 Here is one point in favour of DRBD: MARS stores its transaction logs on the filesystem `/mars/`. When the network bottleneck is lasting very long (some days or even some weeks), the filesystem will eventually run out of space some day. Section 4.4 discusses countermeasures against that in detail. In contrast to MARS, DRBD allocates its bitmap *statically* at resource creation time. It uses up less space, and you don't have to monitor it for (potential) overflows. The space for transaction logs is the price you have to pay if you want or need anytime consistency, or asynchronous replication in general.

In order to really grasp the *heart* of the difference between synchronous and asynchronous replication, we look at the following modified scenario:



This time, the network throughput (red line) is varying[8] in some unpredictable way. As before, the application throughput served by MARS is assumed to be constant (dotted green line, often superseded by the solid green line). The actual replication network throughput is depicted by the solid green line.

As you can see, a network dropdown undershooting the application demand has no impact on the application throughput, but only on the replication network throughput. Whenever the network throughput is held back due to the flaky network, it simply catches up as soon as possible by overshooting the application throughput. The amount of lag-behind is visualized as shaded area: downward shading (below the application throughput) means an increase of the lag-behind, while the upwards shaded areas (beyond the application throughput) indicate a decrease of the lag-behind (catch-up). Once the lag-behind has been fully caught up, the network throughput suddenly jumps back to the application throughput (here visible in two cases).

 Note that the existence of lag-behind areas is roughly corresponding to DRBD disconnect states, and in turn to DRBD inconsistent states of the secondary as long as the lag-behind has not been fully cought up. The very rough[9] duration of the corresponding DRBD inconsistency phase is visualized as magenta line at the time scale.

 MARS utilizes the existing network bandwidth as best as possible in order to pipe through as much data as possible, provided that there exists some data requiring expedition. Conceptually, there exists no better way due to information theoretic limits (besides data compression).

 Note that *in average* during a longer period of time, the network must have emough capacity for transporting all of your data. MARS cannot magically break through information-theoretic limits. It cannot magically transport gigabytes of data over modem lines. Only *relatively short* network problems / packet loss can be compensated.

 In case of lag-behind, the version of the data replicated to the secondary site corresponds

[8] In real life, many long-distance lines or even some heavily used metro lines usually show fluctuations of their network bandwidth by an order of magnitude, or even higher. We have measured them. The overall behaviour can be characterized as "**chaotic**".

[9] Of course, this visualization is not exact. On one hand, the DRBD inconsistency phase may start later as depicted here, because it only starts *after* the first automatic disconnect, upon the first automatic re-connect. In addition, the amount of resync data may be smaller than the amount of corresponding MARS transaction logfile data, because the DRBD bitmap will coalesce multiple writes to the same block into one single transfer. On the other hand, DRBD will transfer no data at all during its disconnected state, while MARS continues its best. This leads to a prolongation of the DRBD inconsistent phase. Depending on properties of the workload and of the network, the real duration of the inconsistency phase may be both shorter or longer.

to some time in the past. Since the data is always transferred in the same order as originally submitted at the primary site, the secondary never gets inconsistent. Your mirror always remains usable. Your only potential problem could be the outdated state, corresponding to some state in the past. However, the "as-best-as-possible" approach to the network transfer ensures that your version is always *as up-to-date as possible* even under ill-behaving network bottlenecks. **There is simply no better way to do it.** In presence of temporary network bottlenecks such as network congestion, there exists no better method than prescribed by the information theoretic limit (red line, neglecting data compression).

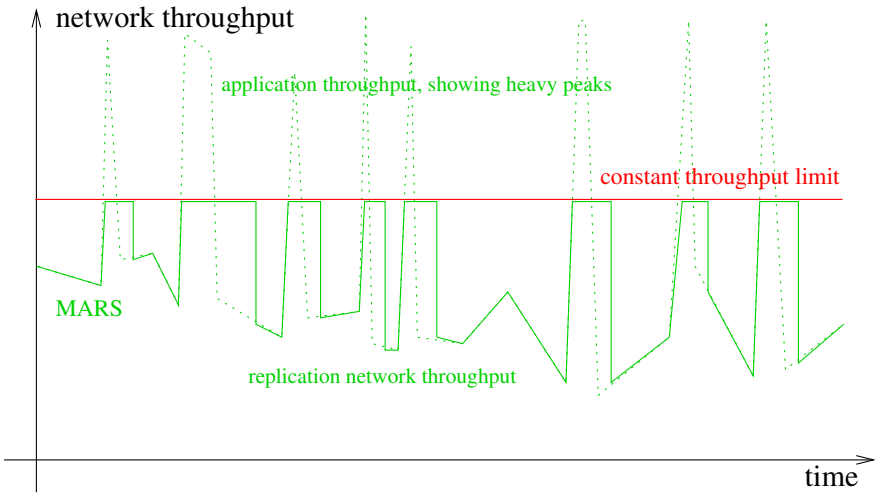

In order to get all of your data through the line, somewhen the network must be healthy again. Otherwise, data will be recorded until the capacity of the `/mars/` filesystem is exhausted, leading to an emergency mode (see section B.3).


MARS' property of never sacrificing local data consistency (at the possible cost of actuality, as long as you have enough capacity in `/mars/`) is called **Anytime Consistency**.


Even when the capacity of `/mars/` is exhausted and when emergency mode is entered, the replicas will not become inconsistent by themselves. However, when the emergency mode is later *cleaned up* for a replica, it will become temporarily inconsistent during the fast full sync. Details are in section B.3.


Conclusion: you can even use **traffic shaping** on MARS' TCP connections in order to globally balance your network throughput (of course at the cost of actuality, but without sacrificing local data consistency). If you would try to do the same with DRBD, you could easily provoke a disaster. MARS simply tolerates any network problems, provided that there is enough disk space for transaction logfiles. Even in case of completely filling up your disk with transaction logfiles after some days or weeks, you will not lose local consistency anywhere (see section 4.4).

Finally, here is yet another scenario where MARS can cope with the situation:



This time, the network throughput limit (solid red line) is assumed to be constant. However, the application workload (dotted green line) shows some heavy peaks. We know from our 1&1 datacenters that such an application behaviour is very common (e.g. in case of certain kinds of DDOS attacks etc).

When the peaks are exceeding the network capacities for some short time, the replication network throughput (solid green line) will be limited for a short time, stay a little bit longer at the limit, and finally drop down again to the normal workload. In other words, you get a flexible buffering behaviour, coping with the peaks.

Similar scenarios (where both the application workload has peaks and the network is flaky to some degree) are rather common. If you would use DRBD there, you were likely to run into regular application performance problems and/or frequent automatic disconnect cycles, depending on the height and on the duration of the peaks, and on network resources.

2.2. Long Distances / High Latencies

In general and in some theories, latencies are conceptually independent from throughput, at least to some degree. There exist all 4 possible combinations:

1. There exist communication lines with high latencies but also high throughput. Examples are raw fibre cables at the ground of the Atlantic.
2. High latencies on low-throughput lines is very easy to achieve. If you never saw it, you never ran interactive `vi` over `ssh` in parallel to downloads on your old-fashioned modem line.
3. Low latencies need not be incompatible with high throughput. See Myrinet, InfiniBand or high-speed point-to-point interconnects, such as modern RAM busses.
4. Low latency combined with low throughput is also possible: in an ATM system (or another pre-reservation system for bandwidth), just increase the multiplex factor on low-capacity but short lines, which is only possible at the cost of assigned bandwidth.

In the *internet* practice, however, it is very likely that high latencies will also lead to worse throughput, because of the *congestion control algorithms* running all over the world.

We have experimented with extremely large TCP send/receive buffers plus various window sizes and congestion control algorithms over long-distance lines between the USA and Europe. Yes, it is possible to improve the behaviour to some degree. But magic does not happen. Natural laws will always hold. You simply cannot travel faster than the speed of light.

Our experience leads to the following rule of thumb, not formally proven by anything, but just observed in practice:

> In general[10], synchronous data replication (not limited to applications of DRBD) works reliably only over distances < 50 km, or sometimes even less.

There may be some exceptions, e.g. when dealing with low-end workstation loads. But when you are responsible for a whole datacenter and/or some centralized storage units, don't waste your time by trying (almost) impossible things. We recommend to use MARS in such use cases.

2.3. Higher Consistency Guarantees vs Actuality

We already saw in section 2.1 that certain types of network bottlenecks can easily (and reproducibly) destroy the consistency of your DRBD secondary, while MARS will preserve local consistency at the cost of actuality (**anytime consistency**).

Some people, often located at database operations, are obtrusively arguing that actuality is such a high good that it must not be sacrificed under any circumstances.

Anyone arguing this way has at least the following choices (list may be incomplete):

1. None of the above use cases for MARS apply. For instance, short distance replication over crossover cables is sufficient (which occurs very often), or the network is reliable enough such that bottlenecks can never occur (e.g. because the total load is extremely low, or conversely the network is extremely overengineered / expensive), or the occurrence of bottlenecks can *provably* be taken into account. In such cases, DRBD is clearly the better solution than MARS, because it provides better actuality than the current version of MARS, and it uses up less disk resources.

[10] We have heard of cases where even less than 50 km were not working with DRBD. It depends on application workload, on properties of the line, and on congestion caused by other traffic. Some other people told us that according to *their* experience, much lesser distances should be considered operable, only in the range of a few single kilometers. However, they agree that DRBD is rock stable when used on crossover cables.

2. In the presence of network bottlenecks, people didn't notice and/or didn't understand and/or did under-estimate the risk of accidental invalidation of their DRBD secondaries. They should carefully check that risk. They should convince themselves that the risk is *really* bearable. Once they are hit by a systematic chain of events which *reproducibly* provoke the bad effect, it is too late[11].

3. In the presence of network bottlenecks, people found a solution such that DRBD does not automatically re-connect after the connection has been dropped due to network problems (c.f. `ko-count` parameter). So the risk of inconsistency *appears* to have vanished. In some cases, people did not notice that the risk has *not completely*[12] vanished, and/or they did not notice that now the actuality produced by DRBD is even drastically worse than that of MARS (in the same situation). It is true that DRBD provides better actuality in `connected` state, but for a full picture the actuality in `disconnected` state should not be neglected[13]. So they didn't notice that their argumentation on the importance of actuality may be fundamentally wrong. A possible way to overcome that may be re-reading section 2.1.2 and comparing its outcome with the corresponding outcome of DRBD in the same situation.

4. People are stuck in contradictive requirements because the current version of MARS does not yet support synchronous or pseudo-synchronous operation modes. This should be resolved some day.


A common misunderstanding is about the actuality guarantees provided by filesystems. The buffer cache / page cache uses by default a **writeback strategy** for performance reasons. Even modern journalling filesystems will (by default) provide only consistency guarantees, but no strong actuality guarantee. In case of power loss, some transactions may be even *rolled back* in order to restore consistency. According to POSIX[14] and other standards, the only *reliable* way to achieve actuality is usage of system calls like `sync()`, `fsync()`, `fdatasync()`, flags like `O_DIRECT`, or similar. For performance reasons, the *vast majority of applications* don't use them at all, or use them only sparingly!


It makes no sense to require strong actuality guarantees from any block layer replication (whether DRBD or future versions of MARS) while higher layers such as filesystems or even applications are already sacrificing them!


In summary, the **anytime consistency** provided by MARS is an argument you should consider, even if you need an extra hard disk for transaction logfiles.

[11] Some people seem to need a bad experience before they get the difference between risk caused by reproducible effects and inverted luck.

[12] Hint: what's the *conceptual* difference beween an automatic and a manual re-connect? Yes, you can try to *lower* the risk in some cases by transferring risks to human analysis and human decisions, but did you take into account the possibility of human errors?

[13] Hint: a potential hurdle may be the fact that the current format of `/proc/drbd` does neither display the timestamp of the first *relevant* network drop nor the total amount of lag-behind user data (which is *not* the same as the number of dirty bits in the bitmap), while `marsadm view` can display it. So it is difficult to judge the risks. Possibly a chance is inspection of DRBD messages in the syslog, but quantification could remain hard.

[14] The above argumentation also applies to Windows filesystems in analogous way.

3. Quick Start Guide

This chapter is for impatient but experienced sysadmins who already know DRBD. For more complete information, refer to chapter The Sysadmin Interface (`marsadm` and `/proc/sys/mars/`)
.

3.1. Preparation: What you Need

Typically, you will use MARS at servers in a datacenter for replication of big masses of data.

Typically, you will use MARS for replication *between* multiple datacenters, when the distances are greater than ≈ 50 km. Many other solutions, even from commercial storage vendors, will not work reliably over large distances when your network is not *extremely* reliable, or when you try to push huge masses of data from high-performance applications through a network bottleneck. If you ever encountered suchalike problems (or try to avoid them in advance), MARS is for you.

You can use MARS both at dedicated storage servers (e.g. for serving Windows clients), or at standalone Linux servers where CPU and storage are not separated.

In order to protect your data from low-level disk failures, you should use a hardware RAID controller with BBU. Software RAID is explicitly *not* recommended, because it generally provides worse performance due to the lack of a hardware BBU (for some benchmark comparisons with/out BBU, see https://github.com/schoebel/blkreplay/raw/master/doc/blkreplay.pdf).

Typically, you will need more than one RAID set[1] for big masses of data. Therefore, use of LVM is also recommended[2] for your data.

MARS' tolerance of networking problems comes with some cost. You will need some extra space for the transaction logfiles of MARS, residing at the `/mars/` filesystem.

The exact space requirements for `/mars/` depend on the *average write rate* of your application, not on the size of your data. We found that only few applications are writing more than 1 TB per day. Most are writing even less than 100 GB per day. Usually, you want to dimension `/mars/` such that you can survive a network loss lasting 3 days / about one weekend. This can be achieved with current technology rather easily: as a simple rule of thumb, just use one **dedicated disk** having a capacity of 4 TB or more. Typically, that will provide you with plenty of headroom even for bigger networking incidents.

Dedicated disks for `/mars/` have another advantage: their mechanical head movement is completely independent from your data head movements. For best performance, attach that dedicated disk to your hardware RAID controller with BBU, building a separate RAID set (even if it consists only of a single disk – notice that the **hardware BBU** is the crucial point).

If you are concerned about reliability, use two disks switched together as a relatively small RAID-1 set. For extremely high performance demands, you may consider (and check) RAID-10.

Since the transaction logfiles are highly sequential in their access pattern, a cheap but high-capacity SATA disk (or nearline-SAS disk) is usually sufficient. At the time of this writing, standard SATA SSDs have shown to be *not* (yet) preferable. Although they offer high random IOPS rate, their sequential throughput is worse, and their long-term stability is questioned by many people at the time of this writing. However, as technology evolves and becomes more mature, this could change in future.

[1] For low-cost storage, RAID-5 is no longer regarded safe for today's typical storage sizes, because the error rate is regarded too high. Therefore, use RAID-6. If you need more than 15 disks in total, create multiple RAID sets (each having at most 15 disks, better about 12 disks) and stripe them via LVM (or via your hardware RAID controller if it supports RAID-60).

[2] You may also combine MARS with commercial storage boxes connected via Fibrechannel or iSCSI, but we have not yet operational experiences at 1&1 with such setups.

Use `ext4` for `/mars/`. Avoid `ext3`, and don't use `xfs`[3] at all.

Notice that the filesystem `/mars/` has nothing to do with an ordinary filesystem. It is completely reserved for MARS internal purposes, namely as a **storage container** for MARS' persistent data. It does not obey any userspace rules like FHS (filesystem hierarchy standard), and it should not be accessed by any userspace tool execpt the official `marsadm` tool. Its internal data format should be a regarded as a **blackbox** by you. The internal data format may change in future, or the complete `/mars/` filesystem may be even replaced by a totally different container format, while the official `marsadm` interface is supposed to remain stable.

That said, you might look into its contents *by hand* for curiosity or for *debugging purposes*, and only as root. But don't program any tools / monitoring scripts / etc bypassing the official `marsadm` tool.

Like DRBD, the current version of MARS has **no security** built in. MARS assumes that it is running in a **trusted network**. Anyone who can connect to the MARS ports (default 7777 to 7779) can potentially breach in and become root! Therefore, you **must** protect your network by appropriate means, such as firewalling and/or encrypted VPN.

Currently, MARS provides no shared secret like DRBD, because a simple shared secret is way too weak to provide any real security (potentially misleading people about the real level of security). Future versions of MARS should provide at least 2-factor authorization, and encryption via dynamic session keys. Until that is implemented, use a secured VPN instead! And don't forget to *audit* it for security holes!

3.2. Setup Primary and Secondary Cluster Nodes

If you already use DRBD, you may migrate to MARS (or even back from MARS to DRBD) if you use *external*[4] DRBD metadata (which is not touched by MARS).

3.2.1. Kernel and MARS Module

The MARS kernel module should be available or can be built via one of the following methods:

1. As an external Debian or rpm kernel module, as provided by a package contributor (or hopefully by standard distros in the future).
2. As a separate kernel module, only for experienced[5] sysadmins: see file `Makefile.dist` (tested with some older versions of Debian; may need some extra work with other distros).
3. Build for senior sysadmins or developers, inplace in the kernel source tree: first apply `0001-mars-minimum-pre-patch-for-mars.patch` and `0001-mars-SPECIAL-for-in-tree-build.patch` or similar, then `cd block/ && git clone --recurse-submodules https://github.com/schoebel/mars` Then `cd ..` and build your kernel as usual. Config options for MARS should appear under "Enable the block layer". Just activate MARS as a **kernel module** via "m" (don't try a fixed compile-in), and leave all else MARS config options at the default (except you know what you are doing).

Further / more accurate / latest instructions can be found in `README` and in `INSTALL`. You must not only install the kernel and the `mars.ko` kernel module to all of your cluster nodes, but also the `marsadm` userspace tool.

[3] It seems that the late internal resource allocation strategy of `xfs` (or another currently unknown reason) could be the reason for some resource deadlocks which appear only with `xfs` and only under *extremely* high IO load in combination with high memory pressure.

[4] *Internal* DRBD metadata should also work as long as the filesystem inside your block device / disk already exists and is not re-created. The latter would destroy the DRBD metadata, but even that will not hurt you really: you can always switch back to DRBD using *external* metadata, as long as you have some small spare space somewhere.

[5] You should be familiar with the problems arising from orthogonal combination of different kernel versions with different MARS module versions and with different `marsadm` userspace tool versions at the package management level. Hint: `modinfo` is your friend.

Starting with `mars0.1stable38` and other branches having merged this feature, a prepatch for vanilla kernels 3.2 through 4.4 is no longer needed. However, **IO performance** is currently somewhat worse when the pre-patch is not applied. This will be addressed in a later release.

Therefore, application of the pre-patch to the kernel is *recommended* for large-scale production systems for now.

Kernel pre-patches can be found in the `pre-patches/` subdirectory of the MARS source tree. Following are the types of pre-patches:

- `0001-mars-minimum-pre-patch-for-mars.patch` or similar. Please prefer this one (when present for your kernel version) in front of `0001-mars-generic-pre-patch-for-mars.patch` or similar. The latter should not be used anymore, except for testing or as an emergency fallback.
- `0001-mars-SPECIAL-for-in-tree-build.patch` or similar. This is *only* needed when building the MARS kernel module together with all other kernel modules in a single `make` pass. For separate external module builds, this patch *must not* be applied (but the pre-patch *should* when possible). When using this patch, please apply the aforementioned pre-patch also, because your kernel is patched anyway.

Starting from version `mars0.1stable56` or `mars0.1abeta8`, **submodules** have been added to the github repo of MARS. If you have an old checkout, please say `git pull --recurse-submodules=yes` or similar. Otherwise you may be missing an important future part of the MARS release, without notice (depending on your local `git` version and its local configuration).

3.2.2. Setup your Cluster Nodes

For your cluster, you need at least two nodes. In the following, they will be called A and B. In the beginning, A will have the `primary` role, while B will be your initial `secondary`. The roles may change later.

1. You must be `root`.
2. On each of A and B, create the `/mars/` mountpoint.
3. On each node, create an `ext4` filesystem on your separate disk / RAID set via `mkfs.ext4` (for requirements on size etc see section Preparation: What you Need).
4. On each node, mount that filesystem to `/mars/`. It is advisable to add an entry to `/etc/fstab`.
5. For security reasons, execute `chmod 0700 /mars` everyhwere after `/mars/` has been mounted. If you forget this step, any following `marsadm` command will drop you a warning, but will fix the problem for you.
6. On node A, say `marsadm create-cluster`.
   This must be done *exactly once*, on exactly one node of your cluster. Never do this twice or on different nodes, because that would create two different clusters which would have nothing to do with each other. The `marsadm` tool protects you against accidentally joining / merging two different clusters. If you accidentally created two different clusters, just umount that `/mars/` partition and start over with step 3 at that node.
7. On node B, you must have a working `ssh` connection to node A (as `root`). Test it by saying `ssh A w` on node B. It should work without entering a password (otherwise, use `ssh-agent` to achieve that). In addition, `rsync` must be installed.
8. On node B, say `marsadm join-cluster A`
9. Only *after*[6] that, do `modprobe mars` on each node.

[6] In fact, you may already `modprobe mars` at node A after the `marsadm create-cluster`. Just don't do any of the `*-cluster` operations when the kernel module is loaded. All other operations should have no such restriction.

3.3. Creating and Maintaining Resources

In the following example session, a block device `/dev/lv-x/mydata` (shortly called *disk*) must already exist on both nodes A and B, respectively, having the same[7] size. For the sake of simplicity, the disk (underlying block device) as well as its later logical resource name as well as its later virtual device name will all be named uniformly by the same suffix `mydata`. In general, you might name each of them differently, but that is not recommended since it may easily lead to confusion in larger installations.

You may have already some data inside your disk `/dev/lv-x/mydata` at the initially primary side A. Before using it for MARS, it must be unused for any other purpose (such as being mounted, or used by DRBD, etc). MARS will require **exclusive access** to it.

1. On node A, say `marsadm create-resource mydata /dev/lv-x/mydata`.
   As a result, a directory `/mars/resource-mydata/` will be created on node A, containing some symlinks. Node A will automatically start in the primary role for this resource. Therefore, a new pseudo-device `/dev/mars/mydata` will also appear after a few seconds. Note that the initial contents of `/dev/mars/mydata` will be exactly the same as in your pre-existing disk `/dev/lv-x/mydata`.
   If you like, you may already use `/dev/mars/mydata` for mounting your already pre-existing data, or for creating a fresh filesystem, or for exporting via iSCSI, and so on. You may even do so before any other cluster node has joined the resource (so-called "standalone mode"). But you can also do so later after setup of (one ore many) secondaries.

2. Wait a few seconds until the directory `/mars/resource-mydata/` and its symlink contents also appears on cluster node B. The command `marsadm wait-cluster` may be helpful.

3. On node B, say `marsadm join-resource mydata /dev/lv-x/mydata`.
   As a result, the initial full-sync from node A to node B should start automatically.

    Of course, your old contents of your disk `/dev/lv-x/mydata` at side B (and *only* there!) is overwritten by the version from side A. Since you are an experienced sysadmin, you knew that, and it was just the effect you deliberately wanted to achieve. If you didn't check that your old contents didn't contain any valuable data (or if you accidentally provided a wrong disk device argument), it is too late now. The `marsadm` command checks that the disk device argument is really a block device, and that exclusive access to it is possible (as well as some further safety checks, e.g. matching sizes). However, MARS cannot know the *purpose* of your generic block device. MARS (as well as DRBD) is completely ignorant of the *contents* of a generic block device; it does not interpret it in any way. Therefore, you may use MARS (as well as DRBD) for mirroring Windows filesystems, or raw devices from databases, or virtual machines, or whatever.

    Hint: by default, MARS uses the so-called "fast fullsync" algorithm. It works similar to `rsync`, first reading the data on both sides and computing an md5 checksum for each block. Heavy-weight data is only transferred over the long-distance network upon checksum mismatch. This is extremely fast if your data is already (almost) identical on both sides. Conversely, if you know in advance that your initial data is completely different on both sides, you may choose to switch off the fast fullsync algorithm via `echo 0 > /proc/sys/mars/do_fast_fullsync` in order to save the additional IO overhead and network latencies introduced by the separate checksum comparison steps.

4. Optionally, only for experienced sysadmins who *really* know what they are doing: if you will create a *new* filesystem on `/dev/mars/mydata` *after(!)* having created the MARS resource as well as *after* having already joined it on every replica, you may abandon the fast fullsync phase *before* creating the fresh filesystem, because the old content of

[7] Actually, the disk at the initially secondary side may be larger than that at the initially primary side. This will waste space and is therefore not recommended.

`/dev/mars/mydata` will then be just garbage not used by the freshly created filesystem[8]. Then, and only then, you may say `marsadm fake-sync mydata` in order to abort the sync operation.

 Never do a `fake-sync` unless you are **absolutely sure** that you really don't need to sync the data! Otherwise, you are *guaranteed* to have produced harmful inconsistencies. If you accidentally issued `fake-sync`, you may startover the fast full sync at your secondary side by saying `marsadm invalidate mydata` (analogously to the corresponding DRBD command).

3.4. Keeping Resources Operational

3.4.1. Logfile Rotation / Deletion

As explained in section The Transaction Logger, all changes to your resource data are recorded in transaction logfiles residing on the `/mars/` filesystem. These files are always growing over time. In order to avoid filesystem overflow, the following must be done in regular time intervals:

1. `marsadm log-rotate all`
   This starts appending to a new logfile on all of your resources. The logfiles are automatically numbered by an increasing 9-digit logfile number. This will suffice for many centuries even if you would logrotate once a minute. Practical frequencies for logfile rotation are more like once an hour, or every 10 minutes when having highly-loaded storage servers.

2. `marsadm log-delete-all all`
   This determines all logfiles from all resources which are no longer needed (i.e. which are *fully* replayed, on *all* relevant secondaries). All superfluous logfiles are then deleted, including all copies on all secondaries.

    The current version of MARS deletes either *all* replicas of a logfile everywhere, or *none* of the replicas. This is a simple rule, but has the drawback that one node may hinder other nodes from freeing space in `/mars/`. In particular, the command `marsadm pause-replay $res` (as well as `marsadm disconnect $res`) will freeze the space reclamation in the whole cluster when the pause is lasting very long.

    During such space accumulation, also the number of so-called deletions will accumulate in /mars/todo-global/ and sibling directories. In very big installations consisting of thousands of nodes, it is a good idea to regularly monitor the number of deletions similarly to the following: `$(find /mars/ -name ‘‘delete-*’’ | wc -l)` should not exceed a limit of ~150 entries.

Please prefer the short form `marsadm cron` as an equivalent to scripting two separate commands `marsadm log-rotate all` and `marsadm log-delete-all all`. The short form is not only easier to remember, but also future-proof in case some new MARS features should be implemented in future.

 Best practice is to run `marsadm cron` in a `cron` job, such as `/etc/cron.d/mars`. An example cronjob can be found in the `userspace/cron.d/` subdirectory of the git repo.

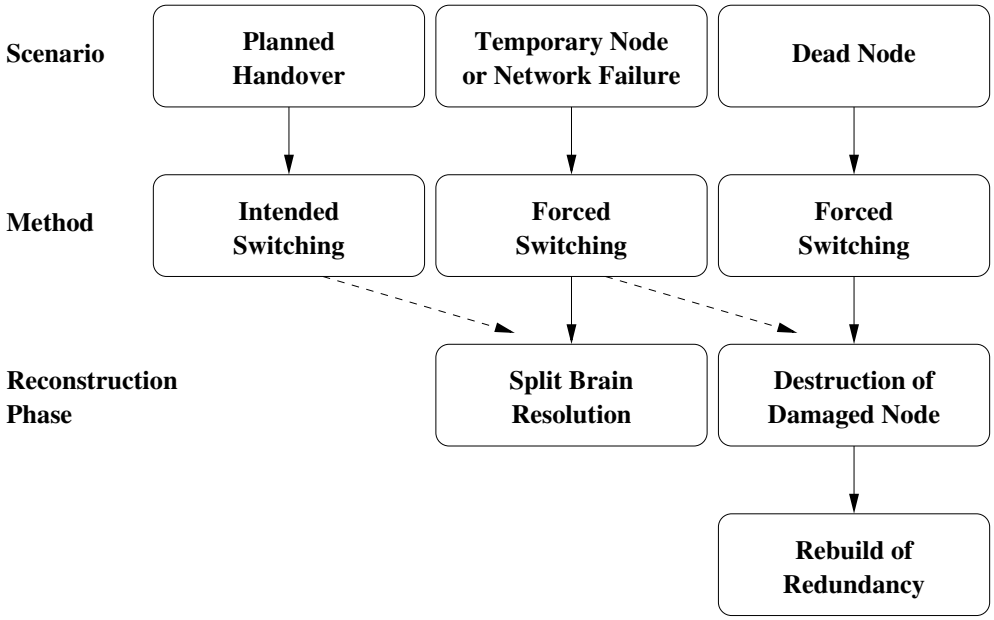
[8] It is *vital* that the transaction logfile contents created by `mkfs` is *fully* propagated to the secondaries and then replayed there.

Analogously, another exception is also possible, but at your own risk (be careful, really!): when migrating your data from DRBD to MARS, and you have ensured that (1) at the end of using DRBD both your replicas were really equal (you should have checked that), and (2) before and after setting up any side of MARS (`create-resource` as well as `join-resource`) nothing has been written at all to it (i.e. no usage, neither of `/dev/lv/mydata` nor of `/dev/mars/mydata` has occurred in any way), the first transaction logfile `/mars/resource-mydata/log-000000001-$primary` created by MARS will be empty. Check whether this is really true! Then, and only then, you may also issue a `fake-sync`.

In addition, you should establish some regular monitoring of the free space present in the `/mars/` filesystem.

More detailed information about about avoidance of `/mars/` overflow is in section 4.4.

3.4.2. Switch Primary / Secondary Roles



In contrast to DRBD, MARS distinguishes between *intended* and *forced* switching. This distinction is necessary due to differences in the communication architecture (asynchronous communication vs synchronous communication, see sections 4.2 and 4.3).

Asynchronous communication means that (in worst case) a message may take (almost) arbitrary time in a distorted network to propagate to another node. As a consequence, the risk for accidentally creating an (unintended) split brain is increased (compared to a synchronous system like DRBD).

In order to minimize this risk, MARS has invested a lot of effort into an internal handover protocol when you start an *intended* primary switch.

3.4.2.1. Intended Switching / Planned Handover

Before starting a planned handover from your old primary `A` to a new primary `B`, you should check the replication of the resource. As a human, use `marsadm view mydata`. For scripting, use the macros from section 5.1.2 (see also section 5.3; an example can be found in `contrib/example-scripts/check-mars-switchable.sh`). The network should be OK, and the amount of replication delay should be as low as possible. Otherwise, handover may take a very long time.

Best practice is to **prepare a planned handover** by the following steps:

1. Check the network and the replication lag. It should be low (a few hundred megabytes, or a low number of gigabytes - see also the rough time forecast shown by `marsadm view mydata` when there is a larger replication delay, or directly access the forecast by `marsadm view-replinfo`).

2. Only when the `systemd` method from section 7.2 is *not* used: stop your application, then umount `/dev/mars/mydata` on host `A`.

3. Optionally: when the `systemd` method from section 7.2 is *not* used, and when scripting something else, or when typing extremely fast by hand, or for better safety: say `marsadm wait-umount mydata` on host `B`. When your network is OK, the propagation of the device usage state[9] should take only a few seconds. Otherwise, check for any network problems or any other problems.

   
   This step is not really necessary, because `marsadm primary` will also wait for the `umount` before it will proceed. However, scripting this intermediate step gives you some more options: if the `umount` takes too long, you may program a different action, like re-starting at the old primary, or its contrary, some forced umount, or even continuing with a forceful failover instead (see section 3.4.2.2).

4. Optionally, and when the `systemd` method from section 7.2 is *not* used: on host `B`, wait until `marsadm view mydata` (or `view-diskstate`) shows `UpToDate`. It is possible to omit this step, but then you have no control on the duration of the handover, and in case of any transfer problems, disk space problems, etc you are potentially risking to produce a split brain (although `marsadm` will do its best to avoid it). Doing the wait by yourself, *before* starting `marsadm primary`, has a big advantage: you can abort the handover cycle at any time, just by re-mounting the device `/dev/mars/mydata` at the old primary `A` again, and by re-starting your application. Once you have started `marsadm primary` on host `B`, you might have to switch back, or possibly even via `primary --force` (see sections 3.4.2.2 and 3.4.3).

Switching the roles is very similar to DRBD: just issue the command

- `marsadm primary mydata`

on your formerly secondary node `B`. In combination with a properly set-up `systemd` method (see section 7.2), this will even automatically start your application at the new site.


The most important difference to DRBD: don't use an intermediate `marsadm secondary mydata` anywhere. Although it would be possible, it has some *disadvantages*. Always switch *directly*!


In contrast to DRBD, MARS remembers the designated primary, even when your system crashes and reboots. While in case of a crash you have to re-setup DRBD with commands like `drbdadm up ...; drbdadm primary ...`, MARS will automatically resume its former roles just by saying `modprobe mars`. In combination with a properly set-up `systemd` method (see section 7.2), this will even automatically re-start your application.


Another fundamental difference to DRBD: when the network is healthy, there can only exist *one* designated primary at a time (modulo some communication delays caused by the "eventually consistent" communication model, see section 4.2). By saying `marsadm primary mydata` on host `B`, **all other** hosts (including `A`) will **automatically go into secondary role** after a while!

[9] Notice that the usage check for `/dev/mars/mydata` on host `B` is based on the *open count* transferred from *another* node `A`. Since MARS is operating asynchronously (in contrast to DRBD), it may take some time until our node `B` knows that the device is no longer used at `A`. This can lead to a race condition if you automate an intended takeover with a script like `ssh root@A "umount /dev/mars/mydata"; ssh root@B "marsadm primary mydata"` because your second ssh command may be faster than the internal MARS symlink tree propagation (cf section 4.3). In order to prevent such races, you are strongly advised to use the command

- `marsadm wait-umount mydata`

on node `B` before trying to become primary. See also section 5.3.


You simply *don't need* an intermediate `marsadm secondary mydata` for planned handover!

Precondition for a plain `marsadm primary` (without `systemd`) is that you are up, that means in attached and connected state (cf. `marsadm up`), that you are no sync target anymore, and (only when `systemd` isn't configured to automatically stop the application at the old site) that any old primary (in this case `A`) does not use its `/dev/mars/mydata` device any longer, and that the network is healthy. If some (parts of) logfiles are not yet (fully) transferred to the new primary, you will need enough space on `/mars/` at the target side. If one of the preconditions described in section 6.2.2 is violated, `marsadm primary` may refuse to start.

These preconditions try to protect you from doing silly things, such as accidentally provoking a split brain error state. We try to avoid split brain as best as we can. Therefore, we distinguish between *intended* and *emergeny* switching. Intended switching will try to avoid split brain *as best as it can*.


Don't *rely* on split brain avoidance, in particular when scripting any higher-level applications such as cluster managers (cf. section 5.3). `marsadm` does its best, but at least in case of (unnoticed) network outages / partitions (or *extremely, really extremely* slow / overloaded networks), an attempt to become `UpToDate` may fail. If you want to *ensure* that no split brain can result from intended primary switching, please obey the the best practices from above, and please give the `primary` command only after your secondary is *known*[10] to be *really* `UpToDate` (see `marsadm wait-cluster` and `marsadm view` and other macros described in section 3.6).


A *very rough* estimation of the time to become `UpToDate` is displayed by `marsadm view mydata` or other macros (e.g. `view-replinfo`). However, on very flaky networks, the estimation may not only flicker much, but also be inaccurate.

3.4.2.2. Forced Switching

In case the connection to the old primary is lost for whatever reason, we just don't know anything about its *current* state (which may deviate from its *last known* state). The following command sequence will skip many checks (essentially you just need to be attached and you must not be a current sync target) and tell your node to become primary forcefully:

- `marsadm pause-fetch mydata`

  
  notice that this is similar to `drbdadm disconnect mydata` as you are probably used from DRBD. For better compatibility with DRBD, you may use the alternate syntax `marsadm disconnect mydata` instead. However, there is a subtle difference to DRBD: DRBD will drop *both* sides of its single bi-directional connection and no longer try to reconnect from any of both sides, while `pause-fetch` is equivalent to `pause-fetch-local`, which instructs only the *local* host to stop fetching logfiles. Other members of the cluster, including the former primary, are *not* instructed to do so. They may continue fetching logfiles over their own private TCP connections, potentially using many connections in parallel, and potentially even from any *other* member of the resource, if they think they can get the data from there. In order to instruct[11] *all* members of the resource to stop fetching logfiles, you may use `marsadm pause-fetch-global mydata` instead (cf section 6.2.2).

- `marsadm primary mydata --force`

[10] As noted in many places in this manual, checking this cannot be done by looking at the local state of a single cluster node. You have to check several nodes. `marsadm` can only check the *local* node reliably!

[11] Notice that not all such instructions may arrive at all sites when the network is interrupted (or extremely slow).

 this is the forceful failover. Depending on the current replication lag, you may loose some data. Use `--force` only if you know what you are doing!

 When `systemd` is configured properly (see section 7.2), your application will start automatically at the new primary site.

 when the network is interrupted, the old primary site cannot know this, and will continue running. Once the metadata exchange is working again (by default on port 7777), the old site will be automatically shut down by its local `systemd` configuration, when configured properly (see section 7.2). In difference to the *planned* handover from section 3.4.2.1, this may happen much later. In case of long-last network outages, even days or weeks!

 Running both sites in parallel for a long time may seriously damage your business. Ensure that any **customer traffic** cannot go to the old site! Be sure to configure your BGP in a proper way, such that *only*, and *only* the new site will receive any customer traffic from both inside and outside networks, like the internet.

- `marsadm resume-fetch mydata`

  As such, the new primary does not really need this, because primaries are producing their own logfiles without need for fetching. This is only to undo the previous `pause-fetch`, in order to avoid future surprises when the new primary will somewhen change to secondary mode again (in the far-distant future), and you have forgotten to remember the fact that fetching had been switched off.

When using `--force`, many precondition checks and other internal checks are skipped, and in particular the internal handover protocol for split brain avoidance.

Therefore, use of `--force` is *likely* to **provoke a split brain**.

 **Split brain** is always an **erroneous state** which should be never entered deliberately! Once you have entered it accidentally, you **must** resolve it ASAP (see section 3.4.3), otherwise you cannot operate your resource in the long term.

In order to impede you from giving an accidental `--force`, the precondition is different: `--force` works only in *locally disconnected* state. This is similar to DRBD.

Remember: `marsadm primary` without `--force` tries to prevent split brain as best as it can. Use of the `--force` option will almost *certainly* provoke a split brain, at least if the old primary continues to operate on its local `/dev/mars/mydata` device. Therefore, you are **strongly advised** to do this **only** after

1. `marsadm primary` without `--force` has failed *for no good reason*[12], and
2. You are sure you *really* want to switch, even when that eventually leads to a split brain. You also declare that you are willing to do *manual* split-brain resolution as described in section 3.4.3, or even destruction / reconstruction of a damaged node as described in section 3.4.4.

 Notice: in case of *connection loss* (e.g. networking problems / network partitions), you may not be able to reliably detect whether a split brain actually resulted, or not.

Some Background In contrast to DRBD, split brain situations are handled differently by MARS . When two primaries are accidentally active at the same time, each of them writes into different logfiles `/mars/resource-mydata/log-000000001-A` and `/mars/resource-mydata/log-000000001-B`

[12] Most reasons will be displayed by `marsadm` when it is rejecting the planned handhover.

where the *origin* host is always recorded in the filename. Therefore, both nodes *can theoretically* run in primary mode independently from each other, at least for some time. They *might* even `log-rotate` independently from each other. However, this is really no good idea. The replication to third nodes will likely get stuck, and your `/mars/` filesystem(s) will eventually run out of space. Any further secondary node (when having $k > 2$ replicas) will certainly get into serious problems: it simply does not know which split-brain version it should follow. Therefore, you will certainly loose the actuality of your redundancy.

 `marsadm secondary` is *strongly discouraged*. It tells the whole cluster that *nobody* is designated as primary any more. *All* nodes should go into secondary mode, globally. In the current version of MARS, the secondaries will no long fetch any logfiles, since they don't know which version is the "right" one. Syncing is also not possible. When the device `/dev/mars/mydata` is in use somewhere, it will remain in *actual* primary mode during that time. As soon as the local `/dev/mars/mydata` is released, the node will *actually* go into secondary mode if it is no longer designated as primary. You should avoid it in advance by always *directly* switching over from one primary to another one, without intermediate `secondary` command. This is different from DRBD.

 Split brain situations are detected *passively* by secondaries. Whenever a secondary detects that somewhere a split brain has happend, it refuses to replay any logfiles behind the split point (and also to fetch them when possible), or anywhere where something appears suspect or ambiguous. This tries to keep its local disk state always being consistent, but outdated with respect to any of the split brain versions. As a consequence, becoming primary may be impossible, because it cannot always know which logfiles are the correct ones to replay before `/dev/mars/mydata` can appear. The ambiguity must be resolved first.

 If you *really* need the local device `/dev/mars/mydata` to disappear *everywhere* in a split brain situation, you don't need a *strongly discouraged* `marsadm secondary` command for this. `marsadm detach` or `marsadm down` can do it also, without destroying knowledge about the former designated primary.

 `marsadm primary -force` is rejected in newer[13] marsadm versions if your replica is a current sync target. This is not a bug: it should prevent you from forcing an inconsistent replica into primary mode, which will *certainly* lead to inconsistent data. However, in extreme rare cases of severe damage of *all* of your replicas, you may be desperate. Only in such a rare case, and only then, you might decide to force any of your replicas (e.g. based on their last sync progress bar) into primary role although none of the re-syncs had finished before. In such a case, and only if you really know what you are doing, you may use `marsadm fake-sync` to first mark your inconsisten replica as UpToDate (which is a **lie**) and then force it to primary as explained above. Afterwards, you will certainly need an `fsck` or similar repair before you can restart your application. Good luck! And don't forget to check the size of `lost+found` afterwards. This is really your *very last* chance if nothing else had succeeded before.

3.4.3. Split Brain Resolution

Split brain can naturally occur during a long-lasting network outage (aka network partition) when you (forcefully) switch primaries inbetween, or due to final loss of your old primary node (fatal node crash) when not all logfile data had been transferred immediately before the final crash.

 Remember that split brain is an **erroneous state** which must be resolved as soon as

[13] Beware: older versions before `mars0.1stable52` did deliberately skip this check because a few years ago somebody at 1&1 did place a *requirement* on this. Fortunately, the requirement now has gone, so a more safe behaviour could be implemented. The new behaviour is for your safety, to prevent you from doing "silly" things in case you are under pressure during an incident (try to safeguard human error as best as possible).

possible!

Whenever split brain occurs for whatever reason, you have two choices for resolution: either destroy one of your versions, or retain it under a different resource name.

In any of both cases, do the following steps ASAP:

1. **Manually** check which (surviving) version is the "right" one. Any error is up to you: destroying the wrong version is *your* fault, not the fault of MARS.
2. If you did not already switch your primary to the final destination determined in the previous step, do it now (see description in section 3.4.2.2). Don't use an intermediate `marsadm secondary` command (as known from DRBD): *directly* switch to the new designated primary!
3. Unless `systemd` is configured properly (see section 7.2), do the following manually: on each non-right version (which you don't want to retain) which had been primary before, umount your `/dev/mars/mydata` or otherwise stop using it (e.g. stop iSCSI or other users of the device). Wait until each of them has actually left primary state and until their local logfile(s) have been fully written back to the underlying disk.
4. Wait until the network works again. All your (surviving) cluster nodes *must*[14] be able to communicate with each other. If that is not possible, or if it takes too long, you may fall back to the method described in section 3.4.4, but do this only as far as necessary.

The next steps are different for different use cases:

Destroying a Wrong Split Brain Version Continue with the following steps, each on those cluster node(s) where you do not want to retain its split-brain version. In preference, start with the old "wrong" primaries first (see advice at the end of this section):

5. `marsadm invalidate mydata`

When no split brain is reported anymore after that (via `marsadm view all`), you are done. You need to repeat this on other secondaries only when necessary.

In very rare cases when things are screwed up very heavily (e.g. a partly destroyed `/mars/` partition), you may try an alternate method described in appendix C.

Keeping a Split Brain Version On those cluster node(s) where you want to retain the version (e.g. for inspection purposes):

5. `marsadm leave-resource mydata`
6. After having done this on *all* those cluster nodes, check that the split brain is gone (e.g. by saying `marsadm view mydata`), as documented above. In very rare cases, you might also need a `log-purge-all` (see page 99).
7. Rename the underlying local disk `/dev/lv-x/mydata` is into something like `/dev/lv-x/mynewdata` (see `man lvrename`) This is *extremely* recommended to avoid confusion with the old resource name!
8. Check that each underlying local disk `/dev/lv-x/mynewdata` is really usable afterwards, e.g. by test-mounting it (or `fsck` if you can afford it). If all is OK, don't forget to umount it before proceeding with the next step.
9. Create a completely new MARS resource out of the underlying disk `/dev/lv-x/mynewdata` having a different name, best is `mynewdata` (see description in section 3.3 on page 49).

Generally: **best practice** is to always keep your LV names equal to your MARS resource names. This can avoid a *lot* of unnecessary confusion.

[14] If you are a MARS expert and you really know what you are doing (in particular, you can anticipate the effects of the Lamport clock and of the symlink update protocol including the "eventually consistent" behaviour including the not-yet-consistent intermediate states, see sections 4.2 and 4.3), you may deviate from this requirement.

Keeping a Good Version When you had a secondary which did not participate in the split brain, but just got confused and therefore stopped replaying logfiles immediately before the split-brain point, it may very well happen[15] that you don't need to do any action for it. When all wrong versions have disappeared from the cluster (by `invalidate` or `leave-resource` as described before), the confusion should be over, and the secondary should automatically resume tracking of the new unique version.

Please check that *all* of your secondaries are no longer stuck. You need to execute split brain resolution only for *stuck* nodes.

Hint / advice for $k > 2$ replicas: it is a good idea to start split brain resolution *first* with those (few) nodes which had been (accidentally) primary before, but are not the new designated primary. Usually, you had 2 primaries during split brain, so this will apply only to *one* of them. Leave the other one intact, by not umounting `/dev/mars/mydata` at all, and keeping your applications running. Even during emergency mode, see section 4.4.2. *First* resolve the problem of the "wrong" primary(s) via `invalidate` or `leave-resource`. Wait for a short while. Then check the rest of your secondaries, whether they now are already following the new (unique) primary, and finally check whether the split brain warning reported by `marsadm view all` is gone everywhere. This way, you can often skip unnecessary invalidations of replicas.

3.4.4. Final Destruction of a Damaged Node

When a node has eventually died, do the following steps ASAP:

1. *Physically* remove the dead node from your network. Unplug all network cables! Failing to do so might provoke a disaster in case it somehow resurrects in an uncontrolled manner, such as a partly-damaged `/mars/` filesystem, a half-defective kernel, RAM / kernel memory corruption, disk corruption, or whatever. Don't risk any such unpredictable behaviour!

2. **Manually** check which of the surviving versions will be the "right" one. Any error is up to you: resurrecting an unnecessarily old / outdated version and/or destroying the newest / best version is *your* fault, not the fault of MARS.

3. If you did not already switch your primary to the final destination determined in the previous step, do it now (see description in section 3.4.2.2).

4. On a surviving node, but preferably *not* the new designated primary, give the following commands:

   a) `marsadm --host=your-damaged-host down mydata`

   b) `marsadm --host=your-damaged-host leave-resource mydata`

   Check for misspellings, in particular the hostname of the dead node, and check the command syntax before typing return! Otherwise, you may forcefully destroy the wrong[16] node!

5. In case any of the previous commands should fail (which is rather likely), repeat it with an additional `--force` option. Don't use `--force` in the first place, alway try first without it!

6. Repeat the same with *all* resources which were formerly present at `your-damaged-host`.

[15] In general, such a "good" behaviour cannot be guaranteed for all secondaries. Race conditions in complex networks may asynchronously transfer "wrong" logfile data to a secondary much earlier than conflicting "good" logfile data which will be marked "good" only in the *future*. It is impossible to predict this in advance.

[16] That said, MARS is rather tolerant of human error. Once a sysadmin accidentally destroyed a cluster while it was continuously running as primary. Fortunately, the problem was detected early enough for a correction without causing any extraordinary customer downtime outside of accepted tolerances, and no data loss at all.

7. Finally, say `marsadm --host=your-damaged-host leave-cluster` (optionally augmented with `--force`).

Now your surviving nodes should *believe* that the old node `your-damaged-host` does no longer exist, and that it does no longer participate in any resource.


Even if your dead node comes to life again in some way: always ensure that the mars kernel module cannot run any more. *Never* do a `modprobe mars` on a node marked as dead this way!

Further instructions for complicated cases are in appendix D and E.

3.4.5. Online Resizing during Operation

You should have LVM or some other means of increasing the physical size of your disk (e.g. via firmware of some RAID controllers). The network must be healthy. Do the following steps:

1. Increase your local disks (usually `/dev/vg/mydata`) *everywhere* in the whole cluster. In order to avoid wasting space, increase them *uniformly* to the same size (when possible). The `lvresize` tool is documented elsewhere.
2. Check that all MARS switches are on. If not, say `marsadm up mydata` everywhere.
3. At the primary: `marsadm resize mydata`
4. If you have intermediate layers such as iSCSI, you may need some `iscsiadm` update or other command.
5. Now you may increase your filesystem. This is specific for the filesystem type and documented elsewhere.


Hint: the secondaries will start syncing the increased new part of the underlying primary disk. In many cases, this is not really needed, because the new junk data just does not care. If you are sure and if you know what you are doing, you may use `marsadm fake-sync mydata` to abort such unnecessary traffic.

3.5. The State of MARS

In general, MARS tries to *hide* any network failures from you as best as it can. After a network problem, any internal low-level socket connections are *transparently* tried to re-open ASAP, without need for sysadmin intervention. In difference to DRBD, network failures will *not* automatically alter the state of MARS, such as switching to `disconnected` after a `ko_timeout` or similar. From a high-level sysadmin viewpoint, communication may just take a very long time to succeed.

When the behaviour of MARS is different from DRBD, it is usually intended as a feature.

MARS is not only an **asynchronous** system at block IO level, but also **at control level**.

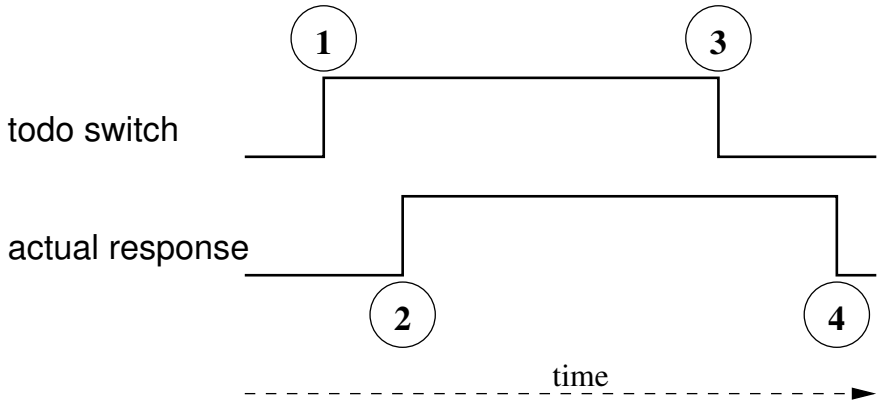
This is *necessary* because in a widely distributed long-distance system running on slow or even temporarily failing networks, actions may take a long time, and there may be many actions **started in parallel**.


Synchronous concepts are generally not sufficient for expressing that. Because of inherent asynchronicity and of dynamic creation / joining of resources, it is neither possible to comprehensively depict a complex distributed MARS system, nor a comprehensive standalone snippet of MARS, as a finite state transition diagram[17].

[17] Probably it could be possible to formally model MARS as a Petri net. However, complete Petri nets are tending to become very conplex, and to describe lots of low-level details. Expressing hierarchy, in a top-down fashion, is cumbersome. We find no clue in trying to do so.

Although MARS tries to *approximate / emulate* the synchronous control behaviour of DRBD at the interface level (`marsadm`) in many situations as best as it can, the *internal* control model is necessarily asynchronous. As an experiencend sysadmin, you will be curious how it works in principle. When you know something about it, you will no longer be surprised when some (detail) behaviour is different from DRBD.

The general principle is an asynchronous 2-edge handshake protocol, which is used almost everywhere in MARS:



We have a binary todo switch, which can be either in state "on" or "off". In addition, we have an actual response indicator, which is similar to an LED indicating the actual status. In our example, we imagine that both are used for controlling a big ventilator, having a huge inert mass. Imagine a big machine from a power plant, which is as tall as a human.

We start in a situation where the binary switch is off, and the ventilator is stopped. At point 1, we turn on the switch. At that moment, a big contactor will sound like "zonggg", and a big motor will start to hum. At first you won't hear anything else. It will take a while, say 1 minute, until the big wheel will have reached its final operating RPM, due to the huge inert mass. During that spin-up, the lights in your room will become slightly darker. When having reached the full RPM at point 2, your workplace will then be noisier, but in exchange your room lights will be back at ordinary strength, and the actual response LED will start to lit in order to indicate that the big fan is now operational.

Assume we want to turn the system off. When turning the todo switch to "off" at point 3, first nothing will seem to happen at all. The big wheel will keep spinning due to its heavy inert mass, and the RPM as well as the sound will go down only slowly. During spin-down, the actual response LED will stay illuminated, in order to warn you that you should not touch the wheel, otherwise you may get injuried[18]. The LED will only go off after, say, 2 minutes, when the wheel has actually stopped at point 4. After that, the cycle may potentially start over again.

As you can see, all four possible cartesian product combinations between two boolean values are occurring in the diagram.

The same handshake protocol is used in MARS for communication between userspace and kernelspace, as well as for communication in the widely distributed system.

3.6. Inspecting the State of MARS

The main command for viewing the current state of MARS is

- `marsadm view mydata`

or its more specialized variant

- `marsadm view-`*`$macroname`* `mydata`

[18] Notice that it is only safe to access the wheel when *both* the switch and the LED are off. Conversely, if at least one of them is on, something is going on inside the machine. Transferred to MARS: always look at *both* the todo switch and the correponding actual indicator in order to not miss something.

where *$macroname* is one of the macros described in chapter 5, or a macro which has been written by yourself.

As always, you may replace the resource name `mydata` with the special keyword `all` in order to get the state of all locally joined resources, as well as a list of all those resources.

When using the variant `marsadm view all`, additionally the global communication status will be displayed. This helps humans in diagnosing problems.

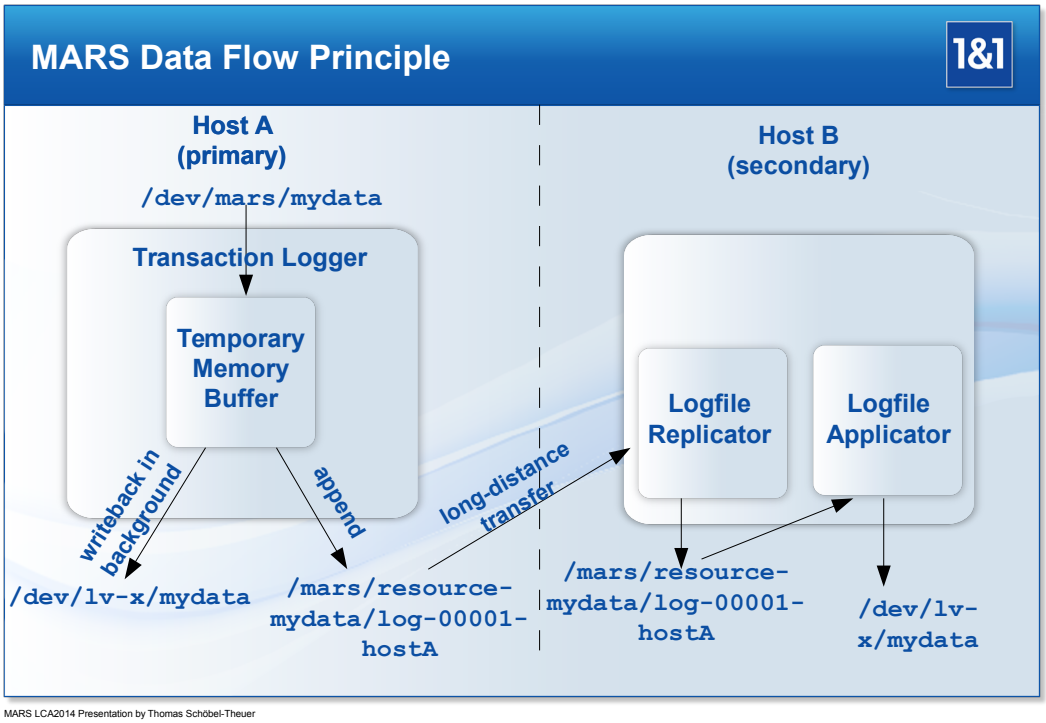
Hint: use the compound command `watch marsadm view all` for continuous display of the current state of MARS. When starting this side-by-side in `ssh` terminal windows for all your cluster nodes, you can easily watch what's going on in the whole cluster.

4. Basic Working Principle

Even if you are impatient, please read this chapter. At the *surface*, MARS appears to be very similar to DRBD. It looks like almost being a drop-in replacement for DRBD.

When taking this naïvely, you could easily step into some trivial pitfalls, because the internal working principle of MARS is totally different from DRBD. Please forget (almost) anything you already know about the internal working principles of DRBD, and look at the very different working principles of MARS.

4.1. The Transaction Logger



The basic idea of MARS is to record all changes made to your block device in a so-called **transaction logfile**. *Any* write reqeuest is treated like a transaction which changes the contents of your block device.

This is similar in concept to some database systems, but there exists no separate "commit" operation: *any* write request is acting like a commit.

The picture shows the flow of write requests. Let's start with the primary node.

Upon submission of a write request on `/dev/mars/mydata`, it is first buffered in a *temporary* memory buffer.

The temporary memory buffer serves multiple purposes:

- It keeps track of the order of write operations.
- Additionally, it keeps track of the positions in the underlying disk `/dev/lv-x/mydata`. In particular, it detects when the same block is overwritten multiple times.
- During pending write operation, any concurrent reads are served from the memory buffer.

After the write has been buffered in the temporary memory buffer, the main logger thread of the transaction logger creates a so-called *log entry* and starts an "append" operation on the transaction logfile. The log entry contains vital information such as the logical block number in the underlying disk, the length of the data, a timestamp, some header magic in order to detect corruption, the log entry sequence number, of course the data itself, and optional information like a checksum or compression information.

Once the log entry has been written through to the `/mars/` filesystem via fsync(), the application waiting for the write operation at `/dev/mars/mydata` is signalled that the write was successful.

This may happen even *before* the writeback to the underlying disk `/dev/lv-x/mydata` has started. Even when you power off the system right now, the information is not lost: it is present in the logfile, and can be reconstructed from there.

Notice that the order of log records present in the transaction log defines a total order among the write requests which is *compatible* to the partial order of write requests issued on `/dev/mars/mydata`.

Also notice that despite its sequential nature, the transaction logfile is typically *not* the performance bottleneck of the system: since appending to a logfile is almost purely sequential IO, it runs much faster than random IO on typical datacenter workloads.

In order to reclaim the temporary memory buffer, its content must be written back to the underlying disk `/dev/lv-x/mydata` somewhen. After writeback, the temporary space is freed. The writeback can do the following optimizations:

1. writeback may be in *any* order; in particular, it may be *sorted* according to ascending sector ´numbers. This will reduce the average seek distances of magnetic disks in general.

2. when the same sector is overwritten multiple times, only the "last" version need to be written back, skipping some intermediate versions.

In case the primary node crashes during writeback, it suffices to replay the log entries from some point in the past until the end of the transaction logfile. It does no harm if you accidentally replay some log entries twice or even more often: since the replay is in the original total order, any temporary inconsistency is *healed* by the logfile application.

In mathematics, the property that you can apply your logfile twice to your data (or even as often as you want), is called **idempotence**. This is a very desirable property: it ensures that nothing goes wrong when replaying "too much" / starting your replay "too early". Idempotence is even more beneficial: in case anything should go wrong with your data on your disk (e.g. IO errors), replaying your logfile once more often may[1] even **heal** some defects. Good news for desperate sysadmins forced to work with flaky hardware!

The basic idea of the asynchronous replication of MARS is rather simple: just transfer the logfiles to your secondary nodes, and replay them onto their copy of the disk data (also called *mirror*) in the same order as the total order defined by the primary.

Therefore, a mirror of your data on any secondary may be outdated, but it always corresponds to some version which was valid in the past. This property is called **anytime consistency**[2]**.**

As you can see in the picture, the process of transfering the logfiles is *independent* from the process which replays the logfiles onto the data at some secondary site. Both processes can be switched on / off separately (see commands `marsadm {dis,}connect` and `marsadm {pause,resume}-replay` in section 6.2.2). This may be *exploited*: for example, you may replicate your logfiles as soon as possible (to protect against catastrophic failures), but deliberately

[1] Miracles cannot be guaranteed, but *higher chances* and *improvements* can be expected (e.g. better chances for `fsck`).

[2] Your secondary nodes are always consistent in themselves. Notice that this kind of consistency is a *local* consistency model. There exists no global consistency in MARS. Global consistency would be practically impossible in long-distance replication where Einstein's law of the speed of light is limiting global consistency. The front-cover pictures showing the planets Earth and Mars tries to lead your imagination away from global consistency models as used in "DRBD Think(tm)", and try to prepare you mentally for local consistency as in "MARS Think(tm)".

wait one hour until it is replayed (under regular circumstances). If your data inside your filesystem `/mydata/` at the primary site is accidentally destroyed by `rm -rf /mydata/`, you have an old copy at the secondary site. This way, you can substitute *some parts*[3] of conventional backup functionality by MARS. In case you need the actual version, just replay in "fast-forward" mode (similar to old-fashioned video tapes).


Future versions of MARS Full are planned to also allow "fast-backward" rewinding, of course at some cost.

4.2. The Lamport Clock

MARS is always *asynchonously* communicating in the distributed system on *any* topics, even strategic decisions.

If there were a *strict* global consistency model, which would be roughly equivalent to a standalone model, we would need *locking* in order to serialize conflicting requests. It is known for many decades that *distributed locks* do not only suffer from performance problems, but they are also cumbersome to get them working reliably in scenarios where nodes or network links may fail at any time.

Therefore, MARS uses a very different consistency model: **Eventually Consistent**.


Notice that the network bottleneck problems described in section 2.1 are *demanding* an "eventually consistent" model. You have **no chance** against natural laws, like Einstein's laws. In order to cope with the problem area, you have to *invest some additional effort*. Unfortunately, asynchronous communication models are more tricky to program and to debug than simple strictly consistent models. In particular, you *have to cope with* additional **race conditions** *inherent to* the "eventually consistent" model. In the face of the laws of the universe, motivate yourself by looking at the graphics at the cover page: the planets are a *symbol* for what you have to do!


Example: the asynchronous communication protocol of MARS leads to a different behaviour from DRBD in case of **network partitions** (temporary interruption of communication between some cluster nodes), because MARS *remembers* the old state of remote nodes over long periods of time, while DRBD knows absolutely nothing about its peers in disconnected state. Sysadmins familiar with DRBD might find the following behaviour unusual:

Event	DRBD Behaviour	MARS Behaviour
1. the network partitions	automatic disconnect	nothing happens, but replication lags behind
2. on A: `umount $device`	works	works
3. on A: `{drbd,mars}adm secondary`	works	works
4. on B: `{drbd,mars}adm primary`	works, split brain happens	**refused** because B believes that A is primary
5. the network resumes	automatic connect attempt fails	communication automatically resumes

If you intentionally want to switch over (and to produce a split brain as a side effect), the following variant must be used with MARS:

Event	DRBD Behaviour	MARS Behaviour
1. the network partitions	automatic disconnect	nothing happens, but replication lags behind
2. on A: `umount $device`	works	works
3. on A: `{drbd,mars}adm secondary`	works	works (but *not remmonended!*)
4. on B: `{drbd,mars}adm primary`	split brain, but nobody knows	**refused** because B believes that A is primary
5. on B: `marsadm disconnect`	-	works, nothing happens
6. on B: `marsadm primary --force`	-	works, split brain happens on B, but A doesn't know
7. on B: `marsadm connect`	-	works, nothing happens
8. the network resumes	automatic connect attempt fails	communication resumes, A now detects the split brain

[3] Please note that MARS cannot *fully* substitute a backup system, because it can keep only *physical* copies, and does not create logical copies.

In order to implement the consistency model "eventually consistent", MARS uses a so-called Lamport[4] clock. MARS uses a special variant called "physical Lamport clock".

The physical Lamport clock is another almost-realtime clock which *can* run independently from the Linux kernel system clock. However, the Lamport clock tries to remain as near as possible to the system clock.

Both clocks can be queried at any time via `cat /proc/sys/mars/lamport_clock`. The result will show both clocks in parallel, in units of seconds since the Unix epoch, with nanosecond resolution.

When there are no network messages at all, both the system clock and the Lamport clock will show almost the same time (except some minor differences of a few nanoseconds resulting from the finite processor clock speed).

The physical Lamport clock works rather simple: *any* message on the network is augmented with a Lamport time stamp telling when the message was *sent* according to the local Lamport clock of the sender. Whenever that message is received by some receiver, it checks whether the time ordering relation would be violated: whenever the Lamport timestamp in the message would claim that the sender had sent it *after* it arrived at the receiver (according to drifts in their respective local clocks), something must be wrong. In this case, the local Lamport clock of the *receiver* is advanced shortly after the sender Lamport timestamp, such that the time ordering relation is no longer violated.

As a consequence, any local Lamport clock may precede the corresponding local system clock. In order to avoid accumulation of deltas between the Lamport and the system clock, the Lamport clock will run slower after that, possibly until it reaches the system clock again (if no other message arrives which sets it forward again). After having reached the system clock, the Lamport clock will continue with "normal" speed.

MARS uses the local Lamport clock for anything where other systems would use the local system clock: for example, timestamp generation in the `/mars/` filesystem. Even symlinks created there are timestamped according to the Lamport clock. Both the kernel module and the userspace tool `marsadm` are always operating in the timescale of the Lamport clock. Most importantly, all timestamp comparisons are always carried out with respect to Lamport time.

 Bigger differences between the Lamport and the system clock can be annoying from a human point of view: when typing `ls -l /mars/resource-mydata/` many timestamps may appear as if they were created in the "future", because the `ls` command compares the output formatting against the system clock (it does not even know of the existence of the MARS Lamport clock).

 Always use `ntp` (or another clock synchronization service) in order to pre-synchronize your system clocks as close as possible. Bigger differences are not only annoying, but may lead some people to wrong conclusions and therefore even lead to bad human decisions!

In a professional datacenter, you should use `ntp` anyway, and you should monitor its effectiveness anyway.

 Hint: many internal logfiles produced by the MARS kernel module contain Lamport timestamps written as numerical values. In order to convert them into human-readable form, use the command `marsadm cat /mars/5.total.status` or similar.

4.3. The Symlink Tree

 The symlink tree as described here will be replaced by another representation in future versions of MARS. Therefore, don't do any scripting by directly accessing symlinks! Use the primitive macros described in section 5.1.2.

The current `/mars/` filesystem container format contains not only transaction logfiles, but also acts as a generic storage for (persistent) state information. Both configuration information

[4] Published in the late 1970s by Leslie Lamport, also known as inventor of LaTeX.

and runtime state information are currently stored in symlinks. Symlinks are "misused[5]" in order to represent some `key -> value` pairs.

It is not yet clear / decided, but there is a *chance* that the *concept* of `key -> value` pairs will be retained in future versions of MARS. Instead of being represented by symlinks, another representation will be used, such that hopefully the `key` part will remain in the form of a pathname, even if there were no longer a physical representation in an actual filesystem.

A fundamentally different behaviour than DRBD: when your DRBD primary crashed some time ago, and now comes up again, you have to setup DRBD again by a sequence of commands like `modprobe drbd; drbdadm up all; drbdadm primary all` or similar. In contrast, MARS needs only `modprobe mars` (after `/mars/` has been mounted by `/etc/fstab`). The *persistence* of the symlinks residing in `/mars/` will automatically remember your previous state, even if some your resources were primary while others were secondary (mixed operations). You don't need to do any actions in order to "restore" a previous state, no matter how "complex" it was.

(Almost) all symlinks appearing in the `/mars/` directory tree are automatically replicated thoughout the whole cluster, provided that the cluster `uuid`s are equal[6] at all sites. Thus the `/mars/` directory forms some kind of *global namespace*.

In order to avoid name clashes, each pathname created at node A follows a convention: the node name A should be a suffix of the pathname. Typically, internal MARS names follow the scheme `/mars/`*`something`*`/myname-A`. When using the expert command `marsadm {get,set}-link` (which will likely be replaced by something else in future MARS releases), you should follow the best practice of systematically using pathnames like `/mars/userspace/myname-A` or similar. As a result, each node will automatically get informed about the state at any other node, like B when the corresponding information is recorded on node B under the name `/mars/userspace/myname-B` (context-dependent names).

Experts only: the symlink replication works generically. You might use the `/mars/userspace/` directory in order to place your own symlink there (for whatever purpose, which need not have to do with MARS). However, the symlinks are likely to disappear. Use `marsadm {get,set}-link` instead. There is a chance that these abstract commands (or variants thereof) will be retained, by acting on the new data representation in future, even if the old symlink format will vanish some day.

Important: the convention of placing the **creator host name** inside your pathnames should be used wherever possible. The name part is a kind of "ownership indicator". It is crucial that no other host writes any symlink not "belonging" to him. Other hosts may read foreign information as often as they want, but never modify them. This way, your cluster nodes are able to *communicate* with each other via symlink / information updates.

Although experts might create (and change) the current symlinks with userspace tools like `ln -s`, you should use the following marsadm commands instead:

- `marsadm set-link myvalue /mars/userspace/mykey-A`
- `marsadm delete-file /mars/userspace/mykey-A`

There are many reasons for this: first, the `marsadm set-link` command will automatically use the Lamport clock for symlink creation, and therefore will avoid any errors resulting from a "wrong" system clock (as in `ln -s`). Second, the `marsadm delete-file` (which also deletes

[5] This means, the symlink targets need not be other files or directories, but just any values like integers or strings.

[6] This is protection against accidental "merging" of two unrelated clusters which had been created at different times with different `uuids`.

symlinks) works on the *whole cluster*. And finally, there is a chance that this will work in future versions of MARS even after the symlinks have vanished.

What's the difference? If you would try to remove your symlink locally by hand via `rm -f`, you will be surprised: since the symlink has been replicated to the other cluster nodes, it will be re-transferred from there and will be resurrected locally after some short time. This way, you cannot delete any object reliably, because your whole cluster (which may consist of many nodes) remembers all your state information and will "correct" it whenever "necessary".

In order to solve the deletion problem, MARS uses some internal deletion protocol using auxiliary symlinks residing in `/mars/todo-global/.` The deletion protocol ensures that all replicas get deleted in the whole cluster, and only thereafter the auxiliary symlinks in `/mars/todo-global/` are also deleted eventually.

You may update your already existing symlink via `marsadm set-link some-other-value /mars/userspace/mykey-A` . The new value will be propagated throughout the cluster according to a **timestamp comparison protocol**: whenever node B notices that A has a *newer* version of some symlink (according to the Lamport timestamp), it will replace its elder version by the newer one. The opposite does *not* work: if B notices that A has an elder version, just nothing happens. This way, the timestamps of symlinks can only progress in forward direction, but never backwards in time.

As a consequence, symlink updates made "by hand" via `ln -sf` may get lost when the local system clock is much more earlier than the Lamport clock.

When your cluster is fully connected by the network, the last timestamp will finally win everywhere. Only in case of network outages leading to *network partitions*, some information may be *temporarily inconsistent*, but only for the duration of the network outage. The timestamp comparison protocol in combination with the Lamport clock and with the persistence of the `/mars/` filesystem will automatically heal any temporary inconsistencies as soon as possible, even in case of temporary node shutdown.

The meaning of some internal MARS symlinks residing in `/mars/` will be hopefully documented in section 9.4 some day.

4.4. Defending Overflow of `/mars/`

This section describes an important difference to DRBD. The metadata of DRBD is allocated *statically* at *creation time* of the resource. In contrast, the MARS transaction logfiles are allocated *dynamically* at *runtime*.

This leads to a potential risk from the perspective of a sysadmin: what happens if the `/mars/` filesystem runs out of space?

No risk, no fun. If you want a system which survives long-lasting network outages while keeping your replicas always consistent (anytime consistency), you *need* dynamic memory for that. It is *impossible* to solve that problem using static memory[7].

Therefore, DRBD and MARS have different application areas. If you just want a simple system for mirroring your data over short distances like a crossover cable, DRBD will be a suitable choice. However, if you need to replicate over longer distances, or if you need higher levels of reliability even when multiple failures may accumulate (such as network loss during a *re*sync of DRBD), the transaction logs of MARS can solve that, but at some *cost*.

4.4.1. Countermeasures

4.4.1.1. Dimensioning of `/mars/`

The first (and most important) measure against overflow of `/mars/` is simply to dimension it large enough to survive longer-lasting problems, at least one weekend.

Recommended size is at least one dedicated disk, residing at a hardware RAID controller with BBU (see section 3.1). During normal operation, that size is needed only for a small fraction, typically a few percent or even less than one percent. However, it is your **safety margin**. Keep it high enough!

[7] The bitmaps used by DRBD don't preserve the *order* of write operations. They cannot do that, because their space is $O(k)$ for some constant k. In contrast, MARS preserves the order. Preserving the order as such (even when only *facts* about the order were recorded without recording the actual data contents) requires $O(n)$ space where n is infinitely growing over time.

4.4.1.2. Monitoring

The next (equally important) measure is **monitoring in userspace**.

Following is a list of countermeasures both in userspace and in kernelspace, in the order of "defensive walling":

1. Regular userspace monitoring must throw an INFO if a certain freespace limit l_1 of `/mars/` is undershot. Typical values for l_1 are 30%. Typical actions are automated calls of `marsadm cron` (or `marsadm log-rotate all` followed by `marsadm log-delete-all all`). You have to implement that yourself in sysadmin space.

2. Regular userspace monitoring must throw a WARNING if a certain freespace limit l_2 of `/mars/` is undershot. Typical values for l_2 are 20%. Typical actions are (in addition to `log-rotate` and `log-delete-all`) alarming human supervisors via SMS and/or further stronger automated actions.

   
   Frequently large space is occupied by files stemming from debugging output, or from other programs or processes. A hot candidate is "forgotten" removal of debugging output to `/mars/`. Sometimes, an `rm -rf $(find /mars/ -name '*.log')` can work miracles.

   
   Another source of space hogging is a "forgotten" `pause-sync` or `disconnect`. Therefore, a simple `marsadm connect-global all` followed by `marsadm resume-replay-global all` may also work miracles (if you didn't want to freeze some mirror deliberately).

   
   If you just wanted to freeze a mirror at an outdated state for a very long time, you simply *cannot* do that without causing infinite growth of space consumption in `/mars/`. Therefore, a `marsadm leave-resource $res` at *exactly that(!)* secondary site where the mirror is frozen, can also work miracles. If you want to automate this in unserspace, be careful. It is easy to get unintended effects when choosing the wrong site for `leave-resource`.

   
   Hint: you can / should start some of these measures even earlier at the INFO level (see item 1), or even earlier.

3. Regular userspace monitoring must throw an ERROR if a certain freespace limit l_3 of `/mars/` is undershot. Typical values for l_3 are 10%. Typical actions are alarming the CEO via SMS and/or even stronger automated actions. For example, you may choose to automatically call `marsadm leave-resource $res` on some or all secondary nodes, such that the primary will be left alone and now has a chance to really delete its logfiles because no one else is any longer potentially needing it.

4. First-level kernelspace action, automatically executed when `/proc/sys/mars/required_free_space_4_gb` + `/proc/sys/mars/required_free_space_3_gb` + `/proc/sys/mars/required_free_space_2_gb` + `/proc/sys/mars/required_free_space_1_gb` is undershot:
   a warning will be issued.

5. Second-level kernelspace action, automatically executed when `/proc/sys/mars/required_free_space_3_gb` + `/proc/sys/mars/required_free_space_2_gb` + `/proc/sys/mars/required_free_space_1_gb` is undershot:
   all locally secondary resources will delete local copies of transaction logfiles which are no longer needed locally. This is a desperate action of the kernel module.

6. Third-level kernelspace action, automatically executed when `/proc/sys/mars/required_free_space_2_gb` + `/proc/sys/mars/required_free_space_1_gb` is undershot:

all locally secondary resources will stop fetching transaction logfiles. This is a more desperate action of the kernel module. You don't want to get there (except for testing).

7. Last desperate kernelspace action when all else has failed and /proc/sys/mars/required_free_space_1_gb is undershot:
   all locally primary resources will enter **emergency mode** (see description below in section 4.4.2). This is the most desperate action of the kernel module. You don't want to get there (except for testing).

In addition, the kernel module obeys a general global limit /proc/sys/mars/required_total_space_0_gb + the sum of all of the above limits. When the *total size* of /mars/ undershots that sum, the kernel module refuses to start at all, because it assumes that it is senseless to try to operate MARS on a system with such low memory resources.

The current level of emergency kernel actions may be viewed at any time via /proc/sys/mars/mars_emergency_mode.

4.4.1.3. Throttling

The last measure for defense of overflow is **throttling your performance pigs**.

Motivation: in rare cases, some users with ssh access can do *very* silly things. For example, some of them are creating their own backups via user-cron jobs, and they do it every 5 minutes. Some example guy created a zip archive (almost 1GB) by regularly copying his old zip archive into a new one, then appending deltas to the new one, and finally deleting the old archive. Every 5 minutes. Yes, every 5 minutes, although almost never any new files were added to the archive. Essentially, he copied over his archive, for nothing. This led to massive bulk write requests, for ridiculous reasons.

In general, your hard disks (or even RAID systems) allow much higher write IO rates than you can ever transport over a standard TCP network from your primary site to your secondary, at least over longer distances (see use cases for MARS in chapter 2). Therefore, it is easy to create a such a high write load that it will be *impossible* to replicate it over the network, *by construction*.

Therefore, we *need* some mechanism for throttling bulk writers whenever the network is weaker than your IO subsystem.

Notice that DRBD will *always* throttle your writes whenever the network forms a bottleneck, due to its synchronous operation mode. In contrast, MARS allows for buffering of performance peaks in the transaction logfiles. *Only when* your buffer in /mars/ runs short (cf subsection 4.4.1.1), MARS will start to throttle your application writes.

There are a lot of screws named /proc/sys/mars/write_throttle_* with the following meaning:

write_throttle_start_percent Whenever the used space in /mars/ is below this threshold, no throttling will occur at all. Only when this threshold is exceeded, throttling will start *slowly*. Typical values for this are 60%.

write_throttle_end_percent Maximum throttling will occur once this space threshold is reached, i.e. the throttling is now at its maximum effect. Typical values for this are 90%. When the actual space in /mars/ lies between write_throttle_start_percent and write_throttle_end_percent, the strength of throttling will be interpolated linearly between the extremes. In practice, this should lead to an equilibrum between new input flow into /mars/ and output flow over the network to secondaries.

write_throttle_size_threshold_kb (readonly) This parameter shows the internal strength calculation of the throttling. Only write[8] requests exceeding this size (in KB) are throttled at all. Typically, this will hurt the bulk performance pigs first, while leaving ordinary users (issuing small requests) unaffected.

[8] Read requests are never throttled at all.

`write_throttle_ratelimit_kb` Set the global IO rate in KB/s for those write requests which are throttled. In case of strongest[9] throttling, this parameters determines the input flow into `/mars/`. The default value is 5.000 KB/s. Please adjust this value to your application needs and to your environment.

`write_throttle_rate_kb` (readonly) Shows the current rate of exactly those requests which are actually throttled (in contrast to *all* requests).

`write_throttle_cumul_kb` (logically readonly) Same as before, but the cumulative sum of all throttled requests since startup / reset. This value can be reset from userspace in order to prevent integer overflow.

`write_throttle_count_ops` (logically readonly) Shows the cumulative number of throttled requests. This value can be reset from userspace in order to prevent integer overflow.

`write_throttle_maxdelay_ms` Each request is delayed at most for this timespan. Smaller values will improve the responsiveness of your userspace application, but at the cost of potentially retarding the requests not sufficiently.

`write_throttle_minwindow_ms` Set the minimum length of the measuring window. The measuring window is the timespan for which the average (throughput) rate is computed (see `write_throttle_rate_kb`). Lower values can increase the responsiveness of the controller algorithm, but at the cost of accuracy.

`write_throttle_maxwindow_ms` This parameter must be set sufficiently much greater than `write_throttle_minwindow_ms`. In case the flow of throttled operations pauses for some natural reason (e.g. switched off, low load, etc), this parameter determines when a completely new rate calculation should be started over[10].

4.4.2. Emergency Mode and its Resolution

When `/mars/` is almost full and there is really absolutely no chance of getting rid of any local transaction logfile (or free some space in any other way), there is only one exit strategy: stop creating new logfile data.

This means that the ability for replication gets lost.

When entering emergency mode, the kernel module will execute the following steps for all resources where the affected host is acting as a primary:

1. Do a kind of "logrotate", but create a *hole* in the sequence of transaction logfile numbers. The "new" logfile is left empty, i.e. no data ist written to it (for now). The hole in the numbering will prevent any secondaries from replaying any logfiles behind the hole (should they ever contain some data, e.g. because the emergency mode has been left again). This works because the secondaries are regularly checking the logfile numbers for contiguity, and they will refuse to replay anything which is not contiguous. As a result, the secondaries will be left in a consistent, but outdated state (at least if they already were consistent before that).

2. The kernel module writes back all data present in the temporary memory buffer (see figure in section 4.1). This may lead to a (short) delay of user write requests until that has finished (typically fractions of a second or a few seconds). The reason is that the temporary memory buffer must not be increased in parallel during this phase (race conditions).

3. After the temporary memory buffer is empty, all local IO requests (whether reads or writes) are directly going to the underlying disk. This has the same effect as if MARS would not be present anymore. Transaction logging does no longer take place.

[9] In case of lighter throttling, the input flow into `/mars/` may be higher because small requests are not throttled.

[10] Motivation: if requests would pause for one hour, the measuring window could become also an hour. Of course, that would lead to completely meaningless results. Two requests in one hour is "incorrect" from a human point of view: we just have to ensure that averages are computed with respect to a reasonable maximum time window in the magnitude of 10s.

4. Any sync from any secondary is stopped ASAP. In case they are resuming their sync somewhen later, they will start over from the beginning (position 0).

In order to leave emergency mode, the sysadmin should do the following steps:

1. Free enough space. For example, delete any foreign files on `/mars/` which have nothing to do with MARS, or resize the `/mars/` filesystem, or whatever.
2. If `/proc/sys/mars/mars_reset_emergency` is not set, now it is time to set it. Normally, it should be already set.
3. Notice: as long as not enough space has been freed, a message containing `''EMEGENCY MODE HYSTERESIS''` (or similar) will be displayed by `marsadm view all`. As a consequence, any sync will be automatically halted. This applies to freshly invoked syncs also, for example created by `invalidate` or `join-resource`.
4. On the secondaries, use `marsadm invalidate $res` in order to request updating your outdated mirrors.
5. On the primary: `marsadm log-delete-all all`
6. As soon as emough space has been freed everywhere to leave the `EMEGENCY MODE HYSTERESIS`, sync should really start. Until that it had been halted.

Alternatively, there is another method by roughly following the instructions from appendix C, but in a slightly different order. In this case, do `leave-resource` everywhere on *all* secondaries, but *don't* start the `join-resource` phase *for now*. Then cleanup all your secondaries via `log-purge-all`, and finally `log-delete-all all` at the primary, and wait until the emergency has vanished everywhere. Only after that, re-`join-resource` your secondaries.

Expert advice for $k = 2$ replicas: this means you had only 1 mirror per resource before the overflow happened. Provided that you have enough space on your LVMs and on `/mars/`, and provided that transaction logging has automatically restarted after `leave-resource` and `log-purge-all`, you can recover redundancy by creating a *new* replica via `marsadm join-resource $res` on a *third* node. Only after the initial full sync has finished there, run `join-resource` at your original mirror. This way, you will always retain at least one **consistent mirror** somewhere. After all is up-to-date, you can delete the superfluous mirror by `marsadm leave-resource $res` and reclaim the disk space from its underlying LVM disk.

If you already have $k > 2$ replicas in total, it may be a wise idea to prefer the `leave-resource ; log-purge-all ; join-resource` method in front of `invalidate` because it does not invalidate *all* your replicas at the same time (when handled properly in the right order).

5. The Macro Processor

`marsadm` comes with a customizable macro processor. It can be used for high-level complex display of the state of MARS (so-called *complex macros*), as well as for low-level display of lots of individual state values (so-called *primitive macros*).

From the commandline, any macro can be called via `marsadm view-`*`$macroname`* `mydata`. The short form `marsadm view mydata` is equivalent to `marsadm view-default mydata`.

In general, the command `marsadm view-`*`$macroname`* `all` will first call the macro *`$macroname`* in a loop for *all* resources we are a *member locally*. Finally, a trailing macro *`$macroname`*`-global` will be called with an empty `%{res}` argument, provided that such a macro is defined. This way, you can produce per-resource output followed by global output which does not depend on a particular resource.

5.1. Predefined Macros

The macro processor is a very flexible and versatile tool for **customizing**. You can create your own macros, but probably the rich set of predefined macros is already sufficient for your needs.

5.1.1. Predefined Complex and High-Level Macros

The following predefined complex macros try to address the information needs of humans. Use them only in scripts when you are prepared about the fact that the output format may change during development of MARS.

Notice: the definitions of predefined complex macros may be updated in the course of the MARS project. However, the primitive macros recursively called by the complex ones will be hopefully rather stable in future (with the exception of bugfixes). If you want to retain an old / outdated version of a complex macro, just check it out from git, follow the instructions in section 5.2, and preferably give it a different name in order to avoid confusion with the newer version. In general, it should be possible to use old macros with newer versions of `marsadm`[1].

`default` This is equivalent to `marsadm view mydata` without *`-maroname`* suffix. It shows a one-line status summary for each resource, optionally followed by informational lines such as progress bars whenever a sync or a fetch of logfiles is currently running. The status line has the following fields:

`%{res}` resource name.

`[`*`this_count`*`/`*`total_count`*`]` total number of replicas of this resource, out of total number of cluster members.

`%include{diskstate}` see `diskstate` macro below.

`%include{replstate}` see `replstate` macro below.

`%include{flags}` see `flags` macro below.

`%include{role}` see `role` macro below.

`%include{primarynode}` see `primarynode` macro below.

`%include{commstate}` see `commstate` macro below.

After that, optional lines such as progress bars are appearing only when something unusual is happening. These lines are subject to future changes. For examples, wasted disk space due to missing `resize` is reported when `%{threshold}` is exceeded.

[1] You might need to check out also old versions of further macros and adapt their names, whenever complex macros call each other.

`1and1` or `default-1and1` A variant of `default` for internal use by 1&1 Internet AG. You may call this complex macro by saying `marsadm view-1and1 all`.

Note: the `marsadm view-1and1` command has been intensely tested in Spring 2014 to produce exactly the same output than the 1&1 internal[2] tool `marsview`[3]

Customization via your own macros (see section 5.2) is explicitly encouraged by the developer. It would be nice if a vibrant user community would emerge, helping each other by exchange of macros.

Hint: in order to produce your own customized inspection / monitoring tools, you may ask the author for an official reservation of a macro sub-namespace such as `*-yourcompanyname`. You will be fully responsible for your own reserved namespace and can do with it whatever you want. The official MARS release will guarantee that *no name clashes* with your reserved sub-namespace will occur in future.

`default-global` Currently, this just calls `comminfo` (see below). May be extended in future.

`diskstate` Shows the status of the underlying disk device, in the following order of precedence[4]:

`NotJoined` (cf `%get-disk{}`) No underlying disk device is configured.

`NotPresent` (cf `%disk-present{}`) The underlying disk device (as configured, see `marsadm view-get-disk`) does not exist or the device node is not accessible. Therefore MARS cannot work. Check that LVM or other software is properly configured and running.

`Detached` (cf `InConsistent`, `NeedsReplay`, `%todo-attach{}`, `%is-attach{}`) The underlying disk is willingly switched off (see `marsadm detach`), and it actually is no longer opened by MARS.

`Detaching` (cf `%todo-attach{}` and `%is-attach{}`) Access to the underlying disk is switched off, but actually not yet `close()`d by MARS. This can happen for a long time on a primary when other secondaries are accessing the disk remotely for syncing.

`DefectiveLog[`*`description-text`*`]` (cf `%replay-code{}`) Typically this indicates an `md5` checksum error in a transaction logfile, or another (hardware / filesystem) defect. This occurs extremely rarely in practice, but has been observed more frequently during a massive failure of air conditioning in a datacenter, when disk temperatures raised to more than 80° Celsius. Notice that a secondary **refuses** to apply any knowingly defective logfile data to the disk. Although this message is *not directly* referring to the underlying disk, it is mentioned here because of its superior **relevance** for the diskstate. A damaged transaction logfile will always affect the *actuality* of the disk, but not its *integrity* (by itself). What to do in such a case?

[2] In addition to allow for customization, the macro processor is also meant as an exit strategy for removing dependencies from non-free software. **Please put your future macros also under GPL!**

[3] There are some subtle differences: numbers are displayed in a different precision, some bug fixes in the macro version (which might have occurred *in the meantime*) may lead to different output as a side effect from bug fixes in *predefined* macros, because the original `marsview` command is currently not actively maintained. Documentation of `marsview` can be found in the corresponding manpage, see `man marsview`. By construction, this is also the (unmaintained) documentation of `marsadm view-1and1` and other `-1and1` macros. Notice that all `*-1and1` macros are not officially supported by the developer of MARS, and they may disappear in a future major release. However, they could be useful for your own customization macros.

[4] When an earlier list item is displayed, no combinations with following items are possible. This kind of "hiding effect" can lead to an *information loss*. In order to get a non-lossy picture from the state of your system, please look at the `flags` which are able to display cartesian combinations of more detailed internal states.

1. When the damage is only at one of your secondaries, you should first ensure that the primary has a good logfile after a `marsadm log-rotate`, then try `marsadm invalidate` at the damaged secondary. It is crucial that the primary has a fresh correct logfile behind the error position, and that it is continuing to operate correctly.
2. When *all* of your secondaries are reporting `DefectiveLog`, the primary could have *produced* a damaged logfile (e.g. in RAM, in a DMA channel, etc) while continuing to operate, and all of your secondaries got that defective logfile. After `marsadm log-delete-all all`, you can check this by comparing the `md5sum` of the first primary logfile (having the lowest serial number) with the versions on your replicas. The problem is that you don't know whether the primary side has a silent corruption on any of its disks, or not. You will need to take an operational decision whether to switchover to a secondary via `primary --force`, or whether to continue operation at the primary and `invalidate` your secondaries.
3. When the original primary is affected in a very bad way, such that it crashed badly and afterwards even recovery of the *primary* is impossible[5] due to this error (which typically occurs extremely rarely, observed two times during 7 millions of operating hours on defective hardware), you need to take an operational decision between the following alternatives:
   a) switchover to a former secondary via `primary --force`, producing a split brain, and producing some (typically small) data loss. However, integrity is more important than actuality in such an extreme case.
   b) deconstruction of the resource at *all* replicas via `leave-resource --force`, running `fsck` or similar tools by hand at the underlying disks, selecting the best replica out of them, and finally reconstructing the resource again.
   c) restore your backup.

`Orphan` The secondary cannot replay data anymore, because it has been kicked out for avoidance of emergency mode. The data is not recent anymore. Typically, `marsadm invalidate` needs to be done.

`NoAttach` (cf `%is-attach{}`) The underlying disk is currently not opened by MARS. Reasons may be that the kernel module is not loaded, or an exclusive `open()` is currently not possible because somebody else has already opened it.

`InConsistent` (cf `%is-consistent{}`) A logfile replay and/or sync is known to be needed / or to complete (e.g. after `invalidate` has started) in order to restore local consistency (for details, look at `flags`).

Hint: in the current implementation of MARS, this will never happen on secondaries during ordinary replay (but only when either sync has not yet finished, or when the *initial* logfile replay after the sync has not yet finished), because the ordinary logfile replay always maintains anytime consistency once a consistent state had been reached.

Only in case of a primary node crash, and *only* after attempts have failed to become primary again (e.g. IO errors, etc), this *can* (but need

[5] In such a rare case, the *original primary* (but not any other host) **refuses** to come up during recovery with *his own* logfile originally produced by *himself*. This is not a bug, but saves you from incorrectly assuming that your original primary disk were consistent - it is *known* to be inconsistent, but recovery is impossible due to the damaged logfile. Thus *this one* replica is trapped by defective hardware. The other replicas shouldn't.

not) mean that something went wrong. Even in such an extremely unlikely event, chances are high that `fsck` can fix any remaining problems (and, of course, you can also switchover to a former secondary).

When this message appears, simply start MARS again (e.g. `modprobe mars; marsadm up all`), in whatever role you are intending. This will *automatically* try to replay any necessary transaction logfile(s) in order to fix the inconsistency. Only if the automatic fix fails and this message persists for a long time without progress, you *might* have a problem. Typically, as observed at a large installation at 1&1, this happens extremely rarely, and then typically indicates that your hardware is likely to be defective.

`OutDated[FR]` (cf `%work-reached{}`) Only at secondaries. Tells whether it is *currently known* that the disk has any lag-behind when compared to the *currently known* state of the current designated primary (if there exists one). Only meaningful if a current designated primary exists. Notice that this kind of status display is subject to *natural races*, for example when new logfile data has been produced in parallel, or network propagation is very slow. Additional information is in brackets:

- `[F]` Fetch is known to be needed.
- `[R]` Replay is known to be needed.
- `[FR]` Both are known to be needed.

`WriteBack` (cf `%is-primary{}`) Appears only at actual primaries (whether designated or not), when the writeback from the RAM buffer is active (see section 4.1)

`Recovery` (cf `%todo-primary{}`) Appears only at the designated primary before it actually has become primary. Similar to database recovery, this indicates the recovery phase after a crash[6].

`EmergencyMode` (cf `%is-emergency{}`) A current designated primary exists, and it is known that this host has entered emergency mode. See section 4.4.2.

`UpToDate` Displayed when none of the above has been detected.

`diskstate-1and1` A variant for internal use by 1&1 Internet AG. See above note.

`replstate` Shows the status of the replication in the following order of precedence:

`ModuleNotLoaded` (cf `%is-module-loaded{}`) No kernel module is loaded, and as a consequence no `/proc/sys/mars/` does exist.

`UnResponsive` (cf `%is-alive{%{host}}`) The main thread `mars_light` did not do any noticable work for more than `%{window}` (default 60) seconds. Notice that this may happen when deleting *extremely* large logfiles (up to hundreds of gigabytes or terabytes). If this happens for a *very* long time, you should check whether you might need a reboot in order to fix the hang. The time window may be changed by `--window=$seconds`.

`NotJoined` (cf `%get-disk{}`) No underlying disk device is configured for this resource.

`NotStarted` (cf `%todo-attach{}`) Replication has not been started.

- When the current host is designated as a primary, the rest of the precedence list looks as follows:

  `EmergencyMode` (cf. `%is-emergency{}`) See section 4.4.2.

  `Replicating` (cf. `%is-primary{}`) Primary mode has been entered.

[6] In some cases, `primary --force` may also trigger this message.

`NotYetPrimary` (catchall) This means the current host *should* act as a primary (see `marsadm primary` or `marsadm primary --force`), but currently doesn't (yet). This happens during logfile replay, before primary mode is actually entered. Notice that replay of very big logfiles may take a long time.

- When the current host is *not* designated as a primary:

  `PausedSync` (cf. `%sync-rest{}` and `%todo-sync{}`) Some data needs to be synced, but sync is currently switched off. See `marsadm {pause,resume}-sync`.

  `Syncing` (cf. `%is-sync{}`) Sync is currently running.

  `PausedFetch` (cf. `%todo{fetch}`) Fetch is currently switched off. See `marsadm {pause,resume}-fetch`.

  `PausedReplay` (cf. `%todo{replay}`) Replay is currently switched off. See `marsadm {pause,resume}-replay`.

  `NoPrimaryDesignated` (cf. `%get-primary{}`) A `secondary` command has been given somewhere in the cluster. Thus no designated primary exists. All resource members are in state `Secondary` or try to approach it. Sync and other operations are not possible. This state is therefore not recommended.

  `PrimaryUnreachable` (cf. `%is-alive{}`) A current designated primary has been set, but this host has not been remotely updated for more than 60 seconds (see also `--window=$seconds`).

  `Orphan` The secondary cannot replay data anymore, because it has been kicked out for avoidance of emergency mode. The data is not recent anymore. Typically, `marsadm invalidate` needs to be done.

  `Replaying` (catchall) None of the previous conditions have triggered.

`replstate-1and1` A variant for internal use by 1&1 Internet AG. See above note.

`flags` For each of disk, consistency, attach, sync, fetch, and replay, show exactly one character. Each character is either a capital one, or the corresponding lowercase one, or a dash. The meaning is as follows:

disk/device: `D` = the device `/dev/mars/mydata` is present, `d` = only the underlying disk `/dev/lv-x/mydata` is present, `-` = none present / configured.

consistency: this relates to the *underlying disk*, not to `/dev/mars/mydata`! `C` = locally consistent, `c` = maybe inconsistent (no guarantee), `-` = cannot determine. Notice: this does not tell anything about *actuality*. Notice: like the other flags, this flag is subject to races and therefore should be relied on only in *detached* state! See also description of macro `is-consistent` below.

attach: `A` = attached, `a` = currently trying to attach/detach but not yet ready (intermediate state), `-` = attach is switched off.

sync: `S` = sync finished, `s` = currently syncing, `-` = sync is switched off.

fetch: `F` = according to knowlege, fetched logfiles are up-to-date, `f` = currently fetching (some parts of) a logfile, `-` = fetch is switched off.

replay: `R` = all fetched logfiles are replayed, `r` = currently replaying, `-` = replay is switched off.

`flags-1and1` A variant for internal use by 1&1 Internet AG.

`todo-role` Shows the *designated* state: `None`, `Primary` or `Secondary`.

`role` Shows the *actual* state: `None`, `NotYetPrimary`, `Primary`, `RemainsPrimary`, or `Secondary`. Any differences to the designated state are indicated by a prefix to the keyword `Primary`: `NotYet` means that it *should* become primary, but actually hasn't. Vice versa, `Remains` means that it *should* leave primary state in order to become secondary,

but actually cannot do that because the `/dev/mars/mydata` device is currently in use
.

	`%todo-primary{} == 0`	`%todo-primary{} == 1`
`%is-primary{} == 0`	`None` / `Secondary`	`NotYetPrimary`
`%is-primary{} == 1`	`RemainsPrimary`	`Primary`

`role-1and1` A variant for internal use by 1&1 Internet AG.

`primarynode` Display `(none)` or the hostname of the designated primary.

`primarynode-1and1` A variant for internal use by 1&1 Internet AG.

`commstate` When the last metadata communication to the designated primary is longer ago than `${window}` (see also `--window=`*`seconds`* option), display that age in human readable form. See also primitive macro `%alive-age{}`.

`syncinfo` Shows an informational progress bar when sync is running. Intended for humans. Scripts should not rely on any details from this. Scripts may use this only as an *approximate* means for detecting progress (when comparing the *full* output text to a prior version and finding *any* difference, they may conclude that some progress has happened, how small whatsoever).

`syncinfo-1and1` A variant for internal use by 1&1 Internet AG.

`replinfo` Shows an informational progress bar when fetch is running. This should not be used for scripting at all, because it contains realtime information in human-readable form.

`replinfo-1and1` A variant for internal use by 1&1 Internet AG.

`fetch-line` Additional details, called by `replinfo`. Shows the amount of data to be fetched, as well as the current transfer rate and a very rough estimation of the future duration. When primitive macros `%fetch-age{}` or `%fetch-lag{}` exceed `${window}`, their values are also displayed for human informational purposes. See description of these primitive macros.

`replay-line` Additional details, called by `replinfo`. Shows the amount of data to be replayed, as well as the current replay rate and a very rough estimation of the future duration. When primitive macro `%replay-age{}` exceeds `${window}`, it is also displayed for human informational purposes.

`comminfo` When the network communication is in an unusual condition, display it. Otherwise, don't produce any output.

5.1.2. Predefined Primitive Macros

5.1.2.1. Intended for Humans

In the following, shell glob notation `{a,b}` is used to document similar variants of similar macros in a single place. When you actually call the macro, you must choose one of the possible variants (excluding the braces).

`the-err-msg` Show reported errors for a resource. When the resource argument is missing or empty, show global error information.

`all-err-msg` Like before, but show all information including those which are `OK`. This way, you get a list[7] of *all* potential error information present in the system.

`{all,the}-wrn-msg` Show all / reported warnings in the system.

`{all,the}-inf-msg` Show all / reported informational messages in the system.

`{all,the}-msg` Show all / reported messages regardless of its classification.

[7] The list may be extended in future versions of MARS.

`{all,the}-global-msg` Show global messages not associated with any resource (the resource argument of the `marsadm` command is ignored in this case).

`{all,the}-global-{inf,wrn,err}-msg` Dito, but more specific.

`{all,the}-pretty-{global-,}{inf-,wrn-,err-,}msg` Dito, but show numerical timestamps in a human readable form.

`{all,the}-{global-,}{inf-,wrn-,err-,}count` Instead of showing the messages, show their count (number of lines).

`errno-text` This macro takes 1 argument, which must represent a Linux `errno` number, and converts it to human readable form (similar to the C `strerror()` function).

`todo-{attach,sync,fetch,replay,primary}` Shows a boolean value (0 or 1) indicating the current state of the corresponding todo switch (whether on or off). The meaning of todo switches is illustrated in section 3.5.

`get-resource-{fat,err,wrn}` Access to the internal error status files. This is not an official interface and may thus change at any time without notice. Use this only for human inspection, not for scripting!

 These macros, as well as the error status files, are likely to disappear in future versions of MARS. They should be used for debugging only. At least when merging into the upstream Linux kernel, only the `*-msg` macros will likely survive.

`get-resource-{fat,err,wrn}-count` Dito, but get the number of lines instead of the text.

`replay-code` Indicate the current state of logfile replay / recovery:

(empty)	Unknown.
0	No replay is currently running.
1	Replay is currently running.
2	Replay has successfully stopped.
<0	See Linux `errno` code. Typically this indicates a damaged logfile, or another filesystem error at `/mars`.

`is-{attach,sync,fetch,replay,primary,module-loaded}` Shows a boolean value (0 or 1) indicating the *actual* state, whether the corresponding action has been actually carried out, or not (yet). Notice that the values indicated by `is-*` may differ from the `todo-*` values when something is not (yet) working. More explanations can be found in section 3.5.

`is-split-brain` Shows whether split brain (see section 3.4.3) has been detected, or not.

`is-consistent` Shows whether the *underlying disk* is in a locally consistent state, i.e. whether it *could* be (potentially) detached and then used for read-only test-mounting[8]. Don't confuse this with the consistency of `/dev/mars/mydata`, which is by construction *always* locally consistent once it has appeared[9]. By construction of MARS, the disk of secondaries will *always* remain in a locally consistent state once the initial sync has finished as well as the initial logfile replay. Notice that local consistency does not necessarily imply actuality (see high-level explanation in section 2.1.2).

[8] Notice that the *writeback* at the primary side is out-of-order by default, for performance reasons. Therefore, the underlying disk is only guaranteed to be consistent when there is no data left to be written back. Notice that this condition is racy by construction. When your primary node crashes during writeback and then comes up again, you must do a `modprobe mars` first in order to automatically replay the transaction logfiles, which will automatically heal such temporary inconsistencies.

[9] Exceptions are possible when using `marsadm fake-sync`. Even in split brain situations, `marsadm primary --force` tries to prevent any further potential exception as best as it can, by not letting `/dev/mars/mydata` to appear and by insisting on split brain resolution first. In future implementations, this might change if more pressure is put on the developer to sacrifice consistency in preference to not waiting for a full logfile replay.

`is-emergency` Shows whether emergency mode (see section 4.4.2) has been entered for the named resource, or not.

`rest-space` (global, no resource argument necessary) Shows the *logically* available space in `/mars/`, which may deviate from the physically available space as indicated by the `df` command.

`get-{disk,device}` Show the name of the underlying disk, or of the `/dev/mars/mydata` device (if it is available).

`{disk,device}-present` Show (as a boolean value) whether the underlying disk, or the `/dev/mars/mydata` device, is available.

`device-opened` Show (as a number) how often `/dev/mars/mydata` has been actually openend, e.g. by `mount` or by some processes like `dd`, or by iSCSI, etc.

5.1.2.2. Intended for Scripting

While complex macros may output a whole bunch of information, the following primitive macros are outputting exactly one value. They are intended for script use (cf. section 5.3). Of course, curious humans may also try them :)

In the following, shell glob notation `{a,b}` is used to document similar variants of similar macros in a single place. When you actually call the macro, you must choose one of the possible variants (excluding the braces).

Name Querying

`cluster-members` Show a newline-separated list of all host names participating in the cluster.

`resource-members` Show a newline-separated list of all host names participating in the particular resource `%{res}`. Notice that this may be a subset of `%cluster-members{}`.

`{my,all}-resources` Show a newline-separated list of either all resource names existing in the cluster, or only those where the current host `%{host}` is member. Optionally, you may specify the hostname as a parameter, e.g. `%my-resources{`*`otherhost`*`}`.

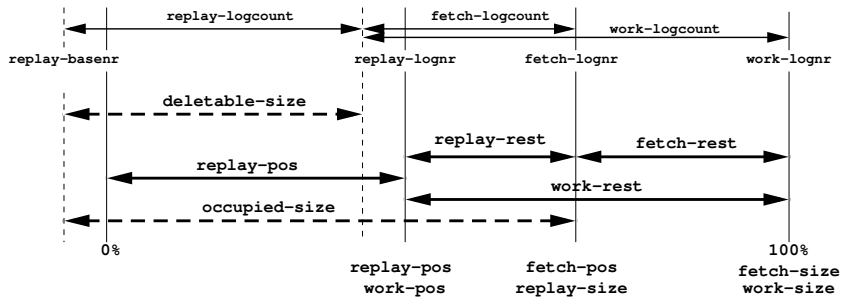


Figure 5.1.: overview on amounts / cursors

Amounts of Data Inquiry The following macros are meaningful for both primary and secondary nodes:

`deletable-size` Show the total amount of *locally present* logfile data which *could* be deleted by `marsadm log-delete-all mydata`. This differs almost always from both `replay-pos` and `occupied-size` due to granularity reasons (only whole logfiles can be deleted). Units are *bytes*, not kilobytes.

`occupied-size` Show the total amount of *locally present* logfile data (sum of all file sizes). This is often roughly approximate to `fetch-pos`, but it may differ vastly (in both directions) when logfiles are not completely transferred, when some are damaged, during split brain, after a `join-resource` / `invalidate`, or when the resource is in emergency mode (see section 4.4.2).

`disk-size` Show the size of the underlying local disk in bytes.

`resource-size` Show the logical size of the resource in bytes. When this value is lower than `disk-size`, you are wasting space.

`device-size` At a primary node, this may differ from `resource-size` only for a very short time during the `resize` operation. At secondaries, there will be no difference.

The following macros are only meaningful for secondary nodes. By information theoretic limits, they can only tell what is *locally known*. They **cannot** reflect the "true (global) state[10]" of a cluster, in particular during network partitions.

`{sync,fetch,replay,work}-size` Show the total amount of data which is / was to be processed by either sync, fetch, or replay. `work-size` is equivalent to `fetch-size`. `replay-size` is equivalent to `fetch-pos` (see below). Units are *bytes*, not kilobytes.

`{sync,fetch,replay,work}-pos` Show the total amount of data which is already processed (current "cursor" position). `work-pos` is equivalent to `replay-pos`.

The 0% point is the *locally contiguous* amount of data since the last `create-resource`, `join-resource`, or `invalidate`, or since the last emergency mode, but possibly shortened by `log-deletes`. Notice that the 0% point may be different on different cluster nodes, because their resource history may be different or non-contiguous during split brain, or after a `join-resource`, or after `invalidate`, or during / after emergency mode.

`{sync,fetch,replay,work}-rest` Shows the difference between `*-size` and `*-pos` (amount of work to do). `work-rest` is therefore the difference between `fetch-size` and `replay-pos`, which is the *total* amount of work to do (regardless whether to be fetched and/or to be replayed).

`{sync,fetch,replay,work}-reached` Boolean value indicating whether `*-rest` dropped down to zero[11].

`{fetch,replay,work}-threshold-reached` Boolean value indicating whether `*-rest` dropped down to `%{threshold}`, which is pre-settable by the `--threshold=`*`size`* command line option (default is 10 MiB). In asynchronous use cases of MARS, this should be preferred over `*-reached` for *human display*, because it produces less flickering by the inevitable replication delay.

`{fetch,replay,work}-almost-reached` Boolean value indicating whether `*-rest` *almost / approximately* dropped down to zero. The default is that at lease 990 permille are reached. In asynchronous use cases of MARS, this can be preferred over `*-reached` for *human display* only, because it produces less flickering by the inevitable replication delay. However, don't base any decisions on this!

[10] Notice that according to Einstein's law, and according to observations by Lamport, the concept of "true state" does not exist at all in a distributed system. Anything you can know in a distributed system is always local knowlege, which races with other (remote) knowlege, and may be outdated at *any* time.

[11] Recall from chapter 2 that MARS (in its current stage of development) does only guarantee local consistency, but cannot guarantee actuality in all imaginable situations. Notice that a general notion of "actuality" is *undefinable* in a widely distributed system at all, according to Einstein's laws.

Let's look at an example. In case of a node crash, and after the node is up again, a `modprobe mars` has to occur, in order to replay the transaction logs of MARS again. However, at the recovery phase before, the journalling `ext4` filesystem `/mars/` *may* have rolled back some internal symlink updates which have occurred immediately before the crash. MARS is relying on the fact that journalling filesystems like `ext4` should do their recovery in a consistent way, possibly by sacrifycing actuality a little bit. Therefore, the above macros cannot guarantee to deliver true information about what is persisted at the moment.

Notice that there are further potential caveats.

In case of `{sync,fetch}-reached`, MARS uses `bio` callbacks resp. `fdatasync()` by default, thus the underlying storage layer has *told* us that it *believes* it has commited the data in a reboot-safe way. Whether this is *really* true does not depend on MARS, but on the lower layers of the storage hierarchy. There exists hardware where this claim is known to be wrong under certain circumstances, such as certain hard disk drives in certain modes of operation. Please check the hardware for any violations of storage semantics under certain circumstances such as power loss, and check information sources like magazines about the problem area. Please notice that such a problem, if it exists at all, is independent from MARS. It would also exist if you wouldn't use MARS on the same system.

`{sync,fetch,replay,work}-percent` The cursor position `*-pos` as a percentage of `*-size`.

`{sync,fetch,replay,work}-permille` The cursor position `*-pos` as permille of `*-size`.

`{sync,fetch,replay,work}-rate` Show the current throughput in bytes[12] per second. `work-rate` is the *maximum* of `fetch-rate` and `replay-rate`.

`{sync,fetch,replay,work}-remain` Show the *estimated* remaining time for completion of the respective operation. This is just a very raw guess. Units are seconds.

`summary-vector` Show the colon-separated CSV value `%replay-pos{}:%fetch-pos{}:%fetch-size{}`.

`replay-basenr` Get currently first reachable logfile number (see figure 5.1 on page 78). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{pos,size}` macros instead.

`{replay,fetch,work}-lognr` Get current logfile number of replay or fetch position, or of the currently known last reachable number (see figure 5.1 on page 78). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{pos,size}` macros instead.

`{replay,fetch,work}-logcount` Get current number of logfiles which are already replayed, or are already fetched, or are to be applied in total (see figure 5.1 on page 78). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{rest}` macros instead.

`alive-timestamp` Tell the Lamport Unix timestamp (seconds since 1970) of the last metadata communication to the designated primary (or to any other host given by the first argument). Returns −1 if no such host exists.

`{fetch,replay,work}-timestamp` Tell the Lamport Unix timestamp (seconds since 1970) when the last progress has been made. When no such action exists, −1 is returned. `%work-timestamp{`*`hostname`*`}` is the maximum of `%fetch-timestamp{`*`hostname`*`}` and `%replay-timestamp{`*`hostname`*`}`. When the parameter *`hostname`* is empty, the local host will be reported (default). Example usage: `marsadm view all --macro='%replay-timestamp{%to` shows the timestamp of the last reported[13] writeback action at the designated primary.

`{alive,fetch,replay,work}-age` Tell the number of seconds since the last respective action, or −1 if none exists.

`{alive,fetch,replay,work}-lag` Report the time difference (in seconds) between the last *known* action at the local host and at the designated primary (or between any other hosts when 2 parameters are given). Returns −1 if no such action exists at any of the two hosts. Attention! This need not reflect the *actual* state in case of networking problems. Don't draw wrong conclusions from a high `{fetch,replay}-lag` value: it could also mean that simply no write operation at all has occurred at the primary side for a long time. Conversely, a low lag value does not imply that the replication is recent: it may refer to *different* write operations at each of the hosts; therefore it only tells that *some* progress has been made, but says nothing about the amount of the progress.

Misc Informational Status

`get-primary` Return the name of the current designated primary node as locally known.

[12] Notice that the internal granularity reported by the kernel may be coarser, such as KiB. This interfaces abstracts away from kernel internals and thus presents everything in byte units.

[13] Updates of this information are occurring with lower frequency than actual writebacks, for performance reasons. The metadata network update protocol will add further delays. Therefore, the accuracy is only in the range of minutes.

`actual-primary` (deprecated) try to determine the name of the node which *appears* to be the actual primary. This only a ***guess***, because it is not generally unique in split brain situations! Don't use this macro. Instead, use `is-primary` on those nodes you are interested in. The explanations from section 3.5 also apply to `get-primary` versus `actual-primary` analogously.

`is-alive` Boolean value indicating whether all other nodes participating in `mydata` are reachable / healthy.

`uuid` (global) Show the unique identifier created by `create-cluster` or by `create-uuid`. Hint: this is immutable, and it is firmly bound to the `/mars/` filesystem. It can only be destroyed by deleting the whole filesystem (see section 6.2).

`tree` (global) Indicate symlink tree version (see section 4.3).

Experts Only The following is for hackers who know what they are doing. The following is not officially supported.

`wait-{is,todo}-{attach,sync,fetch,replay,primary}-{on,off}` This may be used to program some useful waiting conditions in advanced macro scripts. Use at your own risk!

5.2. Creating your own Macros

In order to create your own macros, you could start writing them from scratch with your favorite ASCII text editor. However, it is much easier to take an existing macro and to customize it to your needs. In addition, you can learn something about macro programming by looking at the existing macro code.

Go to a new empty directory and say

- `marsadm dump-macros`

in order to get the most interesting complex macros, or say

- `marsadm dump-all-macros`

in order to additionally get some primitive macros which could be customized if needed. This will write lots of files `*.tpl` into your current working directory.

Any modfied or new macro file should be placed either into the current working directory `./` , or into `$HOME/.marsadm/` , or into `/etc/marsadm/` . They will be searched in this order, and the first match will win. When no macro file is found, the built-in version will be used if it exists. This way, you may override builtin macros.

Example: if you have a file `./mymacro.tpl` you just need to say `marsadm view-mymacro mydata` in order to invoke it in the resource context `mydata`.

5.2.1. General Macro Syntax

Macros are simple ASCII text, enriched with calls to other macros.

ASCII text outside of comments are copied to the output verbatim. Comments are skipped. Comments may have one of the following well-known forms:

- `# skipped text until / including next newline character`
- `// skipped text until / including next newline character`
- `/* skipped text including any newline characters */`
- denoted as Perl regex: `\\\n\s*` (single backslash directly followed by a newline character, and eating up any whitespace characters at the beginning of the next line) Hint: this may be fruitfully used to structure macros in a more readable form / indentation.

Special characters are always initiated by a backslash. The following pre-defined special character sequences are recognized:

- `\n` newline
- `\r` return (useful for DOS compatibility)
- `\t` tab
- `\f` formfeed
- `\b` backspace
- `\a` alarm (bell)
- `\e` escape (e.g. for generating ANSI escape sequences)
- \ followed by anything else: assure that the next character is taken verbatim. Although possible, please don't use this for escaping letters, because further escape sequences might be pre-defined in future. Best practice is to use this only for escaping the backslash itself, or for escaping the percent sign when you don't want to call a macro (protect against evaluation), or to escape a brace directly after a macro call (verbatim brace not to be interpreted as a macro parameter).
- All other characters stand for their own. If you like, you should be able to produce XML, HTML, JSON and other ASCII-based output formats this way.

Macro calls have the following syntax:

- `%`*`macroname`*`{`*`arg1`*`}{`*`arg2`*`}{`*`argn`*`}`
- Of course, arguments may be empty, denoted as `{}`
- It is possible to supply more arguments than required. These are simply ignored.
- There must be always at least 1 argument, even for parameterless macros. In such a case, it is good style to leave it empty (even if it is actually ignored). Just write `%parameterlessmacro{}` in such a case.
- `%{`*`varname`*`}` syntax: As a special case, the macro name may be empty, but then the first argument must denote a previously defined variable (such as assigned via `%let{varname}{myvalue}`, or a pre-defined standard variable like `%{res}` for the current resource name, see later paragraph 5.2.3).
- Of course, parameter calls may be (almost) arbitrarily nested.
- Of course, the *correctness* of nesting of braces must be generally obeyed, as usual in any other macro processor language. General rule: for each opening brace, there must be exactly one closing brace somewhere afterwards.

These rules are hopefully simple and intuitive. There are currently no exceptions. In particular, there is no special infix operator syntax for arithmetic expressions, and therefore no operator precedence rules are necessary. You have to write nested arithmetic expressions always in the above prefix syntax, like `%*{7}{%+{2}{3}}` (similar to non-inverse polish notation).

When deeply nesting macros and their braces, you may easily find yourself in a feeling like in the good old days of Lisp. Use the above backslash-newline syntax to indent your macros in a readable and structured way. Fortunately, modern text editors like (x)emacs or vim have modes for dealing with the correctness of nested braces.

5.2.2. Calling Builtin / Primitive Macros

Primitive macros can be called in two alternate forms:

- `%primitive-`*`macroname`*`{`*`something`*`}`
- `%`*`macroname`*`{`*`something`*`}`

When using the `%primitive-*{}` form, you *explicitly disallow* interception of the call by a `*.tpl` file. Otherwise, you may override the standard definition even of primitive macros by your own template files.

 Notice that `%call{}` conventions are used in such a case. The parameters are passed via `%{0}` ...`%{n}` variables (see description below).

Standard MARS State Inspection Macros These are already described in section 5.1.2. When calling one of them, the call will simply expand to the corresponding value.

Example: `%get-primary{}` will expand to the hostname of the current designated primary node.

Further MARS State Inspection Macros

Variable Access Macros

- `%let{`*`varname`*`}{`*`expression`*`}` Evaluates both *`varname`* and the *`expression`*. The *`expression`* is then assigned to `varname`.
- `%let{`*`varname`*`}{`*`expression`*`}` Evaluates both *`varname`* and the *`expression`*. The *`expression`* is then appended to `varname` (concatenation).
- `%{`*`varname`*`}` Evaluates *`varname`*, and outputs the value of the corresponding variable. When the variable does not exist, the empty string is returned.
- `%{++}{`*`varname`*`}` or `%{`*`varname`*`}{++}` Has the obvious well-known side effect e.g. from C or Java. You may also use `--` instead of `++`. This is handy for programming loops (see below).
- `%dump-vars{}` Writes all currently defined variables (from the currently active scope) to `stderr`. This is handy for debugging.

CSV Array Macros

- `%{`*`varname`*`}{`*`delimiter`*`}{`*`index`*`}` Evaluates all arguments. The contents of *`varname`* is interpreted as a comma-separated list, delimited by *`delimiter`*. The *`index`*'th list element is returned.
- `%set{`*`varname`*`}{`*`delimiter`*`}{`*`index`*`}{`*`expression`*`}` Evaluates all arguments. The contents of the old *`varname`* is interpreted as a comma-separated list, delimited by *`delimiter`*. The *`index`*'th list element is the assigend to, or substituted by, *`expression`*.

Arithmetic Expression Macros The following macros can also take more than two arguments, carrying out the corresponding arithmetic operation in sequence (it depends on the operator whether this accords to the associative law).

- `%+{`*`arg1`*`}{`*`arg2`*`}` Evaluates the arguments, inteprets them as numbers, and adds them together.
- `%-{`*`arg1`*`}{`*`arg2`*`}` Subtraction.
- `%*{`*`arg1`*`}{`*`arg2`*`}` Multiplication.
- `%/{`*`arg1`*`}{`*`arg2`*`}` Division.

- %%{*arg1*}{*arg2*} Modulus.
- %&{*arg1*}{*arg2*} Bitwise Binary And.
- %|{*arg1*}{*arg2*} Bitwise Binary Or.
- %^{*arg1*}{*arg2*} Bitwise Binary Exclusive Or.
- %<<{*arg1*}{*arg2*} Binary Shift Left.
- %>>{*arg1*}{*arg2*} Binary Shift Right.
- %min{*arg1*}{*arg2*} Compute the arithmetic minimum of the arguments.
- %max{*arg1*}{*arg2*} Compute the arithmetic maximum of the arguments.

Boolean Condition Macros

- %=={*arg1*}{*arg2*} Numeral Equality.
- %!={*arg1*}{*arg2*} Numeral Inequality.
- %<{*arg1*}{*arg2*} Numeral Less Then.
- %<={*arg1*}{*arg2*} Numeral Less or Equal.
- %>{*arg1*}{*arg2*} Numeral Greater Then.
- %>={*arg1*}{*arg2*} Numeral Greater or Equal.
- %eq{*arg1*}{*arg2*} String Equality.
- %ne{*arg1*}{*arg2*} String Inequality.
- %lt{*arg1*}{*arg2*} String Less Then.
- %le{*arg1*}{*arg2*} String Less or Equal.
- %gt{*arg1*}{*arg2*} String Greater Then.
- %ge{*arg1*}{*arg2*} String Greater or Equal.
- %=~{*string*}{*regex*}{*opts*} or %match{*string*}{*regex*}{*opts*} Checks whether *string* matches the Perl regular expression *regex*. Modifiers can be given via *opts*.

Shortcut Evaluation Operators The following operators evaluate their arguments only when needed (like in C).

- %&&{*arg1*}{*arg2*} Logical And.
- %and{*arg1*}{*arg2*} Alias for %&&{}.
- %||{*arg1*}{*arg2*} Logical Or.
- %or{*arg1*}{*arg2*} Alias for %||{}.

Unary Operators

- %!{*arg*} Logical Not.
- %not{*arg*} Alias for %!{}.
- %~{*arg*} Bitwise Ńegation.

String Functions

- %length{*string*} Return the number of ASCII characters present in *string*.
- %toupper{*string*} Return all ASCII characters converted to uppercase.
- %tolower{*string*} Return all ASCII characters converted to lowercase.
- %append{*varname*}{*string*} Equivalent to %let{*varname*}{%{*varname*}*string*}.
- %subst{*string*}{*regex*}{*subst*}{*opts*} Perl regex substitution.
- %sprintf{*fmt*}{*arg1*}{*arg2*}{*argn*} Perl sprintf() operator. Details see Perl manual.
- %human-number{*unit*}{*delim*}{*unit-sep*}{*number*1}{*number*2}... Convert a number or a list of numbers into human-readable B, KiB, MiB, GiB, TiB, as given by *unit*. When *unit* is empty, a reasonable unit will be guessed automatically from the maximum of all given numbers. A single result string is produced, where multiple numbers are separated by *delim* when necessary. When *delim* is empty, the slash symbol / is used by default (the most obvious use case is result strings like "17/32 KiB"). The final unit text is separated from the previous number(s) by *unit-sep*. When *unit-sep* is empty, a single blank is used by default.
- %human-seconds{*number*} Convert the given number of seconds into hh:mm:ss format.

Complex Helper Macros

- %progress{20} Return a string containing a progress bar showing the values from %summary-vector{}. The default width is 20 characters plus two braces.
- %progress{20}{*minvalue*}{*midvalue*}{*maxvalue*} Instead of taking the values from %summary-vector{}, use the supplied values. minvalue and midvalue indicate two different intermediate points, while maxvalue will determine the 100% point.

Control Flow Macros

- %if{*expression*}{*then-part*} or %if{*expression*}{*then-part*}{*else-part*} Like in any other macro or programming language, this evaluates the expression once, not copying its outcome to the output. If the result is non-empty and is not a string denoting the number 0, the *then-part* is evaluated and copied to the output. Otherwise, the else-part is evaluated and copied, provided that one exists.
- %unless{*expression*}{*then-part*} or %unless{*expression*}{*then-part*}{*else-part*} Like %if{}, but the expression is logically negated. Essentially, this is a shorthand for %if{%not{expression}}{...} or similar.
- %elsif{*expr1*}{*then1*}{*expr2*}{*then2*}... or %elsif{*expr1*}{*then1*}{*expr2*}{*then2*}...{*odd-else-p*
  This is for simplification of boring if-else-if chains. The classical if-syntax (as shown above) has the drawback that inner if-parts need to be nested into outer else-parts, so rather deep nestings may occur when you are programming longer chains. This is an alternate syntax for avoidance of deep nesting. When giving an odd number of arguments, the last argument is taken as final else-part.
- %elsunless... Like %elsif, but *all* conditions are negated.
- %while{*expression*}{*body*} Evaluates the *expression* in a while loop, like in any other macro or programming language. The *body* is evaluated exactly as many times as the *expression* holds. Notice that endless loops can be only avoided by a calling a non-pure macro inspecting external state information, or by creating (and checking) another side effect somewhere, like assigning to a variable somewhere.
- %until{*expression*}{*body*} Like %while{*expression*}{*body*}, but negate the expression.

- `%for{`*`expr1`*`}{`*`expr2`*`}{`*`expr3`*`}{`*`body`*`}` As you will expect from the corresponding C, Perl, Java, or (add your favorite language) construct. Only the syntactic sugar is a little bit different.

- `%foreach{`*`varname`*`}{`*`CSV-delimited-string`*`}{`*`delimiter`*`}{`*`body`*`}` As you can expect from similar `foreach` constructs in other languages like Perl. Currently, the macro processor has no arrays, but can use comma-separated strings as a substitute.

- `%eval{`*`count`*`}{`*`body`*`}` Evaluates the *`body`* exactly as many times as indicated by the numeric argument *`count`*. This may be used to re-evaluate the output of other macros once again.

- `%protect{`*`body`*`}` Equivalent to `%eval{0}{`*`body`*`}`, which means that the body is not evaluated at all, but copied to the output verbatim[14].

- `%eval-down{`*`body`*`}` Evaluates the *`body`* in a loop until the result does not change any more[15].

- `%tmp{`*`body`*`}` Evaluates the *`body`* once in a temporary scope which is thrown away afterwards.

- `%call{`*`macroname`*`}{`*`arg1`*`}{`*`arg2`*`}{`*`argn`*`}` Like in many other macro languages, this evaluates the named macro in the a new scope. This means that any side effects produced by the called macro, such as variable assignments, will be reverted after the call, and therefore not influence the old scope. However notice that the arguments *`arg1`* to *`argn`* are evaluted in the *old* scope before the call actually happens (possibly producing side effects if they contain some), and their result is respectively assigned to `%{1}` until `%{`*`n`*`}` in the new scope, analogously to the Shell or to Perl. In addition, the new `%{0}` gets the *`macroname`*. Notice that the argument evaluation happens non-lazily in the old scope and therefore differs from other macro processors like TEX.

- `%include{`*`macroname`*`}{`*`arg1`*`}{`*`arg2`*`}{`*`argn`*`}` Like `%call{}`, but evaluates the named macro in the *current* scope (similar to the `source` command of the bourne shell). This means that any side effects produced by the called macro, such as variable assignments, will *not* be reverted after the call. Even the `%{0}` until `%{`*`n`*`}` variables will continue to exist (and may lead to confusion if you aren't aware of that).

- `%callstack{}` Useful for debugging: show the current chain of macro invocations.

Time Handling Macros

- `%time{}` Return the current Lamport timestamp (see section 4.2), in units of seconds since the Unix epoch.

- `%real-time{}` Return the current system clock timestamp, in units of seconds since the Unix epoch.

- `%sleep{`*`seconds`*`}` Pause the given number of seconds.

- `%timeout{`*`seconds`*`}` Like `%sleep{`*`seconds`*`}`, but abort the `marsadm` command after the total waiting time has exceeded the timeout given by the `--timeout=` parameter.

Misc Macros

- `%warn{`*`text`*`}` Show a WARNING:

- `%die{`*`text`*`}` Abort execution with an error message.

[14] TEX or LATEX fans usually know what this is good for ;)
[15] Mathematicians knowing Banach's fixedpoint theorem will know what this is good for ;)

Experts Only - Risky The following macros are unstable and may change at any time without notice.

- `%get-msg{`*name*`}` Low-level access to system messages. You should not use this, since this is not extensible (you must know the name in advance).
- `%readlink{`*path*`}` Low-level access to symlinks. Don't misuse this for circumvention of the abstraction macros from the symlink tree!
- `%setlink{`*value*`}{`*path*`}` Low-level creation of symlinks. Don't misuse this for circumvention of the abstraction macros for the symlink tree!
- `%fetch-info{}` etc. Low-level access to internal symlink formats. Don't use this in scripts! Only for curious humans.
- `%is-almost-consistent{}` Whatever you guess what this could mean, don't use it, at least never in place of `%is-consistent{}` - it is risky to base decisions on this. Mostly for historical reasons.
- `%does{`*name*`}` Equivalent to `%is-`*name*`{}` (just more handy for computing the macro name). Use with care!

5.2.3. Predefined Variables

- `%{cmd}` The command argument of the invoked `marsadm` command.
- `%{res}` The resource name given to the `marsadm` command as a command line parameter (or, possibly expanded from `all`).
- `%{resdir}` The corresponding resource directory. The current version of MARS uses `/mars/resource-%{res}/`, but this may change in future. Normally, you should not need this, since anything should be already abstracted for you. In case you *really* need low-level access to something, please prefer this variable over `%{mars}/resource-%{res}` because it is a bit more abstracted.
- `%{mars}` Currently the fixed string `/mars`. This may change in future, probably with the advent of MARS Full.
- `%{host}` The hostname of the local node.
- `%{ip}` The IP address of the local node.
- `%{timeout}` The value given by the `--timeout=` option, or the corresonding default value.
- `%{threshold}` The value given by the `--threshold=` option, or the corresonding default value.
- `%{window}` The value given by the `--window=` option, or the corresonding default value (60s).
- `%{force}` The number of times the `--force` option has been given.
- `%{dry-run}` The number of times the `--dry-run` option has been given.
- `%{verbose}` The number of times the `--verbose` option has been given.
- `%{callstack}` Same as the `%callstack{}` macro. The latter gives you an opportunity for overriding, while the former is firmly built in.

5.3. Scripting HOWTO

Both the **asynchronous communication model** of MARS (cf section 4.2) including the Lamport clock, and the **state model** (cf section 3.5) is something you *definitely* should have in mind when you want to do some scripting. Here is some further concrete advice:

- Don't access anything on `/mars/` directly, except for debugging purposes. Use `marsadm`.
- Avoid running scripts in parallel, other than for inspection / monitoring purposes. When you give two `marsadm` commands in parallel (whether on the same host, or on different hosts belonging to the same cluster), it is very likely to produce a mess. `marsadm` has no internal locking. There is no cluster-wide locking at all. Unfortunately, some systems like Pacemaker are violating this in many cases (depending on their configuration). Best is if you have a dedicated / more or less centralized **control machine** which controls masses of your georedundant working servers. This reduces the risk of running interfering actions in parallel. Of course, you need backup machines for your control machines, and in different locations. Not obeying this advice can easily lead to problems such as complex races which are very difficult to solve in long-distance distributed systems, even in general (not limited to MARS).
- `marsadm wait-cluster` is your friend. Whenever your (near-)central script has to switch between different hosts `A` and `B` (of the same cluster), use it in the following way:
  `ssh A ''marsadm action1''; ssh B ''marsadm wait-cluster; marsadm action2''`
   Don't ignore this advice! Interference is almost *sure*! As a rule of thumb, precede almost any action command with some appropriate waiting command!
- Further friends are any `marsadm wait-*` commands, such as `wait-umount`.
- In some places, busy-wait loops might be needed, e.g. for waiting until a specific resource is `UpToDate` or matches some other condition. Examples of waiting conditions can be found under `github.com/schoebel/test-suite` in subdirectory `mars/modules/`, specifically `02_predicates.sh` or similar.
- In case of network problems, some command may hang (forever), if you don't set the `--timeout=` option. Don't forget the check the return state of any failed / timeouted commands, and to take appropriate measures!
- Test your scripts in failure scenarios!

6. The Sysadmin Interface (`marsadm` and `/proc/sys/mars/`)

In general, the term "after a while" means that other cluster nodes will take notice of your actions according to the "eventually consistent" propagation protocol described in sections 4.2 and 4.3. Please be aware that this "while" may last very long in case of network outages or bad firewall rules.

In the following tables, column "Cmp" means compatibility with DRBD. Please note that 100% exact compatibility is not possible, because of the asynchronous communication paradigm.

The following table documents common options which work with (almost) any command:

Option	Cmp	Description
`--dry-run`	no	Run the command without actually creating symlinks or touching files or executing rsync. This option *should* be used first at any dangerous command, in order to check what would happen.  Don't use in scripts! Only use by hand! This option does not change the waiting logic. Many commands are waiting until the desired effect has taken place. However, with `--dry-run` the desired effect will never happen, so the command may wait forever (or abort with a timeout). In addition, this option can lead to additional aborts of the commands due to unmet conditions, which cannot be met because the symlinks are not actually created / altered. Thus this option can give only a **rough estimate** of what would happen later!
`--force`	almost	Some preconditions are skipped, i.e. the command will / should work although some (more or less) vital preconditions are violated. Instead of giving `--force`, you may alternatively prefix your command with `force-`  THIS OPTION IS DANGEROUS! Use it only when you are absolutely sure that you know what you are doing! Use it only as a last resort if the same command without `--force` has failed *for no good reason*!
`--verbose`	no	Some (few) commands will become more speaky.
`--timeout=$seconds`	no	Some commands require response from either the local kernel module, or from other cluster nodes. In order to prevent infinite waiting in case of network outages or other problems, the command will fail after the given timeout has been reached. When $seconds is -1, the command will wait forever. When $seconds is 0, the command will not wait in case any precondition is not met, und abort without performing an action.. The default timeout is 5s.
`--window=$seconds`	no	The time window for checking the aliveness of other nodes in the network. When no symlink updates have occurred during the last window, the node is considered dead. Default is 60s.
`--threshold=$size`	no	The macros containing the substring `-threshold-` or `-almost-` are using this as a default value for approximation whether something has been approximately reached. Default is 10MiB. The $size argument may be a number optionally followed by one the lowercase characters k m g t p for indicating kilo mega giga tera or peta bytes as multiples of 1000. When using the corresponding uppercase character, multiples of 1024 are formed instead.
`--host=$host`	no	The command acts as if the command were executed on another host $host. This option should not be used regularly, because the local information in the symlink tree may be outdated or even wrong. Additionally, some local information like remote sizes of physical devices (e.g. remote disks) is not present in the symlink tree at all, or is wrong (reflecting only the *local* state).  THIS OPTION IS DANGEROUS! Use it only for final destruction of dead cluster nodes, see section 3.4.4.
Option	Cmp	Description

Option	Cmp	Description
--ip=$ip	no	By default, marsadm always uses the IP for $host as stored in the symlink tree (directory /mars/ips/). When such an IP entry does not (yet) exist (e.g. create-cluster or join-cluster), all local network interfaces are automatically scanned for IPv4 adresses, and the first one is taken. This may lead to wrong decisions if you have multiple network interfaces. In order to override the automatic IP detection and.to explicitly tell the IP address of your storage network, use this option.  Usually you will need this only at {create,join}-cluster.
--verbose	no	Some (few) commands will become more speaky.
Option	Cmp	Description

6.1. Cluster Operations

Command / Params	Cmp	Description
create-cluster	no	Precondition: the /mars/ filesystem must be mounted and it must be empty (mkfs.ext4, see instructions in section 3.2.2). The kernel module must *not* be loaded. Postcondition: the initial symlink tree is created in /mars/. Additionally, the /mars/uuid symlink is created for later distribution in the cluster. It uniquely indentifies the cluster in the world. This must be called exactly once at the initial primary. Hint: use the --ip= option if you have multiple interfaces.
join-cluster $host	no	Precondition: the /mars/ filesystem must be mounted and it must be empty (mkfs.ext4, see instructions in section 3.2.2). The kernel module must *not* be loaded. The cluster must have been already created at another node $host. A working ssh connecttion to $host as root must exist (without password). rsync must be installed at all cluster nodes. Postcondition: the initial symlink tree /mars/ is replicated from the remote host $host, and the local host has been added as another cluster member. This must be called exactly once at every initial secondary node. Hint: use the --ip= option if you have multiple interfaces.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`leave-cluster`	no	Precondition: the `/mars/` filesystem must be mounted and it must contain a valid MARS symlink tree produced by the other `marsadm` commands. The local node must no longer be member of any resource (see `marsadm leave-resource`). The kernel module should be loaded and the network should be operating in order to also propogate the effect to the other nodes. Postcondition: the local node is removed from the replicated symlink tree `/mars/` such that other nodes will cease to communicate with it after a while. The converse it not true: the local node may continue[a] passivley fetching the symlink tree. In order to really stop all communication, the kernel module should be unloaded afterwards. The local `/mars/` filesystem may be manually destroyed after that (at least if you need to reuse it). In case of an eventual node loss (e.g. fire, water, ...) this command should be used on another node $helper in order to finally remove $damaged from the cluster via the command `marsadm leave-cluster --host=$damaged --force`.  In case you cannot use `leave-resource` for any reason, you may do the following: just destroy the `/mars/` filesystem on the host `$deadhost` you want to remove (e.g. by `mkfs`), or take other measures to *ensure* that it cannot be accidentally re-used in any way (e.g. physical destruction of the underlying RAID, `lvremove`, etc). On all other hosts, do `rmmod mars`, then delete the symlink `/mars/ips/ip-$deadhost` everywhere by hand, and finally `modprobe mars` again.  Notice that the last `leave-resource` operation does not delete the cluster as such. It just creates an *empty* cluster which has no longer any members. In particular, the cluster ID `/mars/uuid` is *not* removed, deliberately[b].  Before you can re-use *any* left-over `/mars/` filesystem for creating / joining a new / different cluster, you *must* obey the instructions in section 3.2.2 and use `mkfs.ext4` accordingly. [a] Reason: `leave-cluster` removes only its *own* IP address from `/mars/ips/`, but does not destroy the usual symmetry of the symlink tree by leaving the other IPs intact. Therefore, the local node will continue fetching updates from all nodes present in `/mars/ips/`. As an effect, the local node will *passively* mirror the symlinks of other cluster members, but not vice versa. There is no communication from the local node to the other ones, turning the local node into a **whitness** according to some terminology from Distributed Systems. This is a feature, not a bug. It could be used for porst-mortem analysis, or for monitoring purposes. However, *deletions* of symlinks are not guaranteed to take place, so your whitness may *accumulate* thousands of old symlinks over a long time. If you want to eventually stop all communication to the local node, just run `rmmod`. [b] This is a feature, not a bug. The `uuid` is created once, but never altered anywhere. The only way to get rid of it is *external* deletion (not by `marsadm`) *together(!)* with all other contents of `/mars/`. This prevents you from accidentally merging half-dead remains which could have survived a disaster for any reason, such as snapshotting filesystems / VMs or whatever.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
merge-cluster $host	no	Precondition: the set of resources at the local cluster (transitively) and at the cluster of $host (transitively) must be disjoint. Create the union of both clusters, consisting of the union of all particpating machines (transitively). Resource memberships are unaffected. This is useful for creating a "virtual LVM cluster" where resources can be migrated later via join-resource / leave-resource operations.  Attention! The mars branch 0.1.y does not scale well in number of cluter members, because it evolved from a lab prototype with $O(n^2)$ behaviour at metadata exchange. Never exceed the maximum cluster members as described in appendix A on page 132. For safety, you should better stay at 1/2 of the numbers mentioned there. Use split-cluster for going back to smaller clusters again after your background data migration has completed.  Future versions of MARS, starting with branch 0.1b.y will be constructed for very big clusters in the range of thousands of nodes. Development has not yet stabilized there, and operational experiences are missing at the moment. Be careful until official announcements are appearing in the ChangeLog, reporting of operational experiences from the 1&1 big cluster at metadata level.
merge-cluster-check $host	no	Check in advance whether the set of resources at the local cluster and at the other cluster $host are disjoint.
split-cluster	no	This is almost the inverse operation of merge-cluster: it determines the minimum sub-cluster groups participating in some common resources. Then it splits the cluster memberships such that unnecessary connections between non-related nodes are interrupted. Use this for avoidance of too big clusters.
wait-cluster	no	See section 6.3.3.
create-uuid	no	Deprecated. Only for compatibility with old version light0.1beta05 or earlier. Precondition: the /mars/ filesystem must be mounted. A uuid (such as automatically created by recent versions of marsadm create-cluster) must not already exist; i.e. you have a very old and outdated symlink tree. Postcondition: the /mars/uuid symlink is created for later distribution in the cluster. It uniquely indentifies the cluster in the world. This must be called at most once at the current primary.
Command / Params	Cmp	Description

6.2. Resource Operations

Common precondition for all resource operations is that the /mars/ filesystem is mounted, that it contains a valid MARS symlink tree produced by other marsadm commands (including a unique uuid), that your current node is a valid member of the cluster, and that the kernel module is loaded. When communication is impossible due to network outages or bad firewall rules, most commands will succeed, but other cluster nodes may take a long time to notice your changes.

Instead of executing marsadm commands serveral times for each resource argument, you may give the special resource argument all. This work even when combined with --force, but be cautious when giving dangerous command combinations like marsadm delete-resource --force all.

 Beware when combining this with --host=somebody. In some very rare cases, like final destruction of a whole datacenter after an earthquake, you might need a combination like marsadm --host=defective delete-resource --force all. Don't use such combinations if you don't need them *really*! You can easily shoot yourself in your head if you are not carefully operating such commands!

6.2.1. Resource Creation / Deletion / Modification

Command / Params	Cmp	Description
Command / Params	Cmp	Description

Command / Params	Cmp	Description

`create-resource`
`$res`
`$disk_dev`
`[$mars_name]`
`[$size]`
	no	Precondition: the resource argument `$res` must not denote an already existing resource name in the cluster. The argument `$disk_dev` must denote an absolute path to a usable local block device, its size must be greater zero. When the optional `$mars_name` is given, that name must not already exist on the local node; when not given, `$mars_name` defaults to `$res`. When the optional `$size` argument is given, it must be a number, optionally followed by a lowercase suffix `k`, `m`, `g`, `t`, or `p` (denoting size factors as multiples of 1000), or an uppercase suffix `K`, `M`, `G`, `T` or `P` (denoting size factors as multiples of 1024). The given size must not exceed the actual size of `$disk_dev`. It will specify the future resource size as shown by `marsadm view-resource-size $res`.
		Postcondition: the resource `$res` is created, the inital role of the current node is primary. The corresponding symlink tree information is asynchonously distributed in the cluster (in the background). The device `/dev/mars/$mars_name` should appear after a while.
		Notice: when `$size` is strictly smaller than the size of `$disk_dev`, you will unnecessarily waste some space..
		This must be called exactly once for any new resource.

`join-resource`
`$res`
`$disk_dev`
`[$mars_name]`
	no	Precondition: the resource argument `$res` must denote an already existing resource in the cluster (i.e. its symlink tree information must have been received). The resource must have a designated primary, and it must no be in emergency mode. There must not exist a split brain in the cluster. The local node must not be already member of that resource. The argument `$disk_dev` must denote an absolute path to a usable (but currently unused) local block device, its size must be greater or equal to the logical size of the resource. When the optional `$mars_name` is given, that name must not already exist on the local node; when not given, `$mars_name` defaults to `$res`.
		Postcondition: the current node becomes a member of resource `$res`, the inital role is secondary. The initial full sync should start after a while.
		Notice: when the size of $disk_dev is strictly greater than the size of the resource, you will unnecessarily waste some space..

`leave-resource` `$res`	no	Precondition: the local node must be a member of the resource `$res`; its current role must be secondary. Sync, fetch and replay must be paused (see commands `pause-{sync,fetch,replay}` or their abbreviation `down`). The disk must be detatched (see commands `detach` or `down`). The kernel module should be loaded and the network should be operating in order to also propogate the effect to the other nodes. Postcondition: the local node is no longer a member of `$res`. Notice: as a side effect for other nodes, their `log-delete` may now become possible, since the current node does no longer count as a candidate for logfile application. In addition, a split brain situation may be (partly) resolved by this.  Please notice that this command *may* lead to (but does not guarantee) split-brain resolution.  The contents of the disk is not changed by this command. Before issuing this command, check whether the disk appears to be locally consistent (see `view-is-consistent`)! After giving this command, any internal information indicating the consistency state will be gone, and you will no longer be able to guess consistency properties.  When you are *sure*.that the disk was consistent before (or is now by manually checking it), you may re-create a new resource out of it via `create-resource`. In case of an eventual node loss (e.g. fire, water, ...) this command may be used on another node $helper in order to finally remove all the resources $damaged from the cluster via the command `marsadm leave-resource $res --host=$damaged --force`.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
delete-resource $res	no	Precondition: the resource must be empty (i.e. all members must have left via leave-resource). This precondition is overridable by --force, increasing the danger to maximum! It is even possible to combine --force with an invalid resource argument and an invalid --host=somebodyelse argument in order to desperately try to destroy remains of incomplete or pysically damaged hardware. Postcondition: all cluster members will somewhen be forcefully removed from $res. In case of network interruptions, the forced removal may take place far in the future.  THIS COMMAND IS *VERY* DANGEROUS! Use this only in desperate situations, and only manually. Don't call this from scripts. You are forcefully using a sledgehammer, even without --force! The danger is that the *true* state of other cluster nodes need not be known in case of network problems .Even when it were known, it could be compromised by **byzantine failures**. It is strongly advised to try this command with --dry-run first. When combined with --force, this command will definitely **murder** other cluster nodes, possibly after a long while, and even when they are operating in primary mode / having split brains / etc. However, there is no guarantee that other cluster nodes will be *really* dead – it is (theoretically) possible that they remain only *half dead*. For example, a half dead node may continue to write data to /mars/ and thus lead to overflow somewhen.  This command implies a forceful detach, possibly destroying consistency. It is similar in spirit to a **STONITH**. In particular, when a cluster node was operating in primary mode (/dev/mars/mydata being continuously in use), the forceful detach cannot be carried out until the device is completely unused. In the meantime, the current transaction logfile will be appended to, but the file *might* be already unlinked (orphan file filling up the disk). After the forceful detach, the underlying disk need not be consistent (although MARS does its best). Since this command deletes any symlinks which normally would indicate the consistency state, no guarantees about consistency can be given after this *in general*! Always check consistency by hand! When possible / as soon as possible, check the local state on the other nodes in order to *really* shutdown the resource everywhere (e.g. to *really* unuse the /dev/mars/mydata device, etc). After this command, you *should* rebuild the resource under a different name, in order to avoid any clashes caused by unexpected resurrection of "dead" or "half-dead" nodes (beware of shapshot / restores on virtual machines!!). MARS does its best to avoid problems even in case the new resource name should equal the old one, but there can be *no guarantee* in all possible failure scenarios / usage scenarios.  When possible, prefer leave-resource over this!
wait-resource $res {is-,}{attach, primary, device}{-off,}	no	See section 6.3.3.
Command / Params	Cmp	Description

6.2.2. Operation of the Resource

Common preconditions are the preconditions from section 6.2, plus the respective resource $res must exist, and the local node must be a member of it. With the single exception of attach itself, all other operations must be started in attached state.

When $res has the special reserved value all, the following operations will work on all resources where the current node is a member (analogously to DRBD).

Command / Params	Cmp	Description
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`attach` `$res`	yes	Precondition: the local disk belonging to $res is not in use by anyone else. Its contents has not been altered in the meantime since the last `detach`.  Mounting *read-only* is allowed during the detached phase.  However, be careful! If you *accidentally* forget to give the right readonly-mount flags, if you use `fsck` in repair mode inbetween, or alter the disk content in any other way (beware of LVM snapshots / restores etc), you will almost certainly produce an **unnoticed inconsistency** (not reported by `view-is-consistent`)! MARS has *no chance* to notice suchalike! Postcondition: MARS uses the local disk and is able to work with it (e.g. replay logfiles on it). Note: the local disk is opened in exclusive read-write mode. This should protect against most common misuse, such as opening the disk in parallel to MARS.  However, this does not necessarily protect against non-exclusive openers.
`detach` `$res`	yes	Precondition: the local `/dev/mars/mydata` device (when present) is no longer opened by anybody. Postcondition: the local disk belonging to $res is no longer in use.  In contrast to DRBD, you need not explicitly pause syncing, fetching, or replaying *to* (as apposed to *from*) the local disk. These processes are automatically paused. As another contrast to DRBD, the respective processes will usually *automatically* resume after re-attach, as far as possible in the respective new situation. This will usually work even over `rmmod` or reboot cycles, since the internal symlink tree will automatically persist all todo switches for you (c.f. section 3.5).  Notice: only *local* transfer operations *to* the local disk are paused by a detach. When another node is remotely running a sync *from* your local disk, it will likely remain in use for remote reading. The reason is that the server part of MARS is operating purely passively, in order serve all remote requests as best as possible (similar to the original Unix philosophy). In order to really stop all accesses, do a `pause-sync` on all other resource member where a sync is currently running. You may also try `pause-sync-global`.  WARNING! After this, and ather having paused any remote data access, you might use the underlying disk for your own purposes, such as test-mounting it in *readonly* mode. **Don't modifiy** its contents in any way! Not even by an `fsck`[a]! Otherwise, you will have inconsistencies *guaranteed*. MARS has no way for knowing of any modifications to your disk when bypassing `/dev/mars/*`.  In case you accidentally modified the underlying disk at the *primary* side, you may choose to resolve the inconsistencies by `marsadm invalide $res` on *each* secondary. [a] Some (but not all) `fsck` tools for some filesystems have options to start only a test repair / verify mode / dry run, without doing actual modifications to the data. Of course, these modes *can* be used. But be really sure! Double-check for the right options!
`pause-sync` `$res`	partly	Equivalent to `pause-sync-local`.
`pause-sync-local` `$res`	partly	Precondition: none additionally. Postcondition: any sync operation targeting the local disk (when not yet completed) is paused after a while (cf section 3.5). When successfully completed, this operation will remember the switch state forever and automatically become relevant if a sync is needed again (e.g. `invalidate` or `resize`).
Command / Params	Cmp	Description

Command / Params	Cmp	Description
pause-sync-global $res	partly	Like *-local, but operates on all members of the resource.
resume-sync $res	partly	Equivalent to resume-sync-local.
resume-sync-local $res	partly	Precondition: additionally, a primary must be designated, and it must not be in emergency mode. Postcondition: any sync operation targeting the local disk (when not yet completed) is resumed after a while. When completed, this operation will remember the switch state forever and become relevant if a sync is needed again (e.g. invalidate or resize).
resume-sync-global $res	partly	Like *-local, but operates on all members of the resource.
pause-fetch $res	partly	Equivalent to pause-fetch-local.
pause-fetch-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any transfer of (parts of) transaction logfiles which are present at another primary host to the local /mars/ storage are paused at their current stage.  This switch works independently from {pause,resume}-replay.
pause-fetch-global $res	partly	Like *-local, but operates on all members of the resource.
resume-fetch $res	partly	Equivalent to resume-fetch-local.
resume-fetch-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any (parts of) transaction logfiles which are present at another primary host shouldl be transferred to the local /mars/ storage as far as not yet locally present.  This works independently from {pause,resume}-replay.
resume-fetch-global $res	partly	Like *-local, but operates on all members of the resource.
pause-replay $res	partly	Equivalent to pause-replay-local.
pause-replay-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any local replay operations of transaction logfiles to the local disk are paused at their current stage.  This works independently from {pause,resume}-fetch resp. {dis,}connect.
pause-replay-global $res	partly	Like *-local, but operates on all members of the resource.
resume-replay $res	partly	Equivalent to pause-replay-local.
resume-replay-local $res	partly	Precondition: must be in secondary role. Postcondition: any (parts of) locally existing transaction logfiles (whether replicated from other hosts or produced locally) are started for replay to the local disk, as far as they have not yet been applied.
resume-replay-global $res	partly	Like *-local, but operates on all members of the resource.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`connect` `$res`	partly	Equivalent to `connect-local` and to `resume-fetch-local`.  Note: although this sounds similar to DRBD's `drbdadm connect`, there are subtle differences. DRBD has exactly one connection per resource, which is associated with *pairs* of nodes. In contrast, MARS may create multiple connections per resource at runtime, and these are associated with the *target* host (not with *pairs* of hosts). As a consequence, the fetch may *potentially* occur from any other other source host which happens to be reachable (although the current implementation prefers the current designated primary, but this may change in future). In addition, `marsadm disconnect` does not stop *all* communication. It only stops fetching logfiles. The symlink update running in background is *not* stopped, in order to always propagate as much metadata as possible in the cluster. In case of a later incident, chances are higher for a better knowledge of the *real* state of the cluster.
`connect-local` `$res`	partly	Equivalent to `resume-fetch-local`.
`connect-global` `$res`	partly	Equivalent to `resume-fetch-global`.
`disconnect` `$res`	partly	Equivalent to `disconnect-local` and to `pause-fetch-local`.  See above note at `connect`.
`disconnect-local` `$res`	partly	Equivalent to `pause-fetch-local`.
`disconnect-global` `$res`	partly	Equivalent to `pause-fetch-global`.
`up` `$res`	yes	Equivalent to `attach` followed by `resume-fetch` followed by `resume-replay` followed by `resume-sync`.
`down` `$res`	yes	Equivalent to `pause-sync` followed by `pause-fetch` followed by `pause-replay` followed by `detach`.  Hint: consider to prefer plain `detach` over this, because `detach` will remember the last state of all switches, while `down` will *not*.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
primary $res	almost	Precondition: sync must have finished at any resource member. All relevant transaction logfiles must be either already locally present, or be fetchable (see resume-fetch and resume-replay). When some logfile data is locally missing, there must be enough space on /mars/ to fetch it. Any replay must not have been interrupted by a replay error (see macro %replay-code{} or diskstate DefectiveLog). The current designated primary must be reachable over network. When there is no designated primary (i.e. marsadm secondary had been executed before, which is explicitly *not recommended*), *all* other members of the resource must be reachable (since we have no memory who was the old primary before), and then they must also match the same preconditions. When another host is currently primary (whether designated or not), it must match the preconditions of marsadm secondary (that means, its local /dev/mars/mydata device must not be in use any more). A split brain must not already exist. Postcondition: /dev/mars/$dev_name appears locally and is usable; the current host is in primary role. Switches the **designated primary**. There are two variants: 1) **Handover** when *not* giving --force: when another host is currently primary, it is first asked to leave its primary role, and it is waited until it actually has become secondary. After that, the local host is asked to become primary. Before actually becoming primary, all relevant logfiles are transferred over the network and replayed, in order to avoid accidental creation of split brain as best as possible[a]. Only after that, /dev/mars/$dev_name will appear. When network transfers of the symlink tree are very slow (or currently impossible), this command may take a very long time. In case a split brain is already detected at the initial situation, the local host will refuse to switch the designated primary without --force.  In case of $k > 2$ replicas: if you want to handover between host A and B while a sync is currently running at host C, you have the following options: 1. wait until the sync has finished (see macro sync-rest, or marsadm view in general). 2. do a leave-resouce on host C, and later join-resource after the handover completed successfully. 2) **Forced switching**: by giving –force while pause-fetch is active (but not pause-replay), most preconditions are ignored, and MARS does its best to actually become primary even if some logfiles are missing or incomplete or even defective.  primary --force is a potentially harmful variant, because it will provoke a split brain in most cases, and therefore in turn will lead to **data loss** because one of your split brain versions must be discarded later in order to resolve the split brain (see section 3.4.3).  **Never** call primary --force when primary without --force is sufficient! If primary without --force complains that the device is in use at the former primary side, take it seriously! Don't override with --force, but rather umount[b] the device at the other side!  Only use primary --force when something is *already broken*, such as a network outage, or a node crash, etc. During ordinary operations (network OK, nodes OK), you should never need primary --force!  If you umount /dev/mars/mydata on the old primary A, and then wait until marsadm view (or another suitable macro) on the target host B shows that everything is UpToDate, you can prevent a split brain by yourself even when giving primary --force afterwards. However, checking / assuring this is *your* responsibility!  primary --force switches the *designated* primary. In some extremely rare cases, when *multiple* faults have accumulated in a *weird* situation, it *might* be impossible becoming the / an actual primary. Typically you may be *already* in a split brain situation. This has not been observed for a long operations time on recent versions of MARS, but in general becoming primary via --force cannot be guaranteed always, although MARS does its best. In split brain situations, or if you ever encounter such a problem, you *must* resolve the split brain immediately after giving this command (see section 3.4.3).  Hint in case of $k > 2$ replicas: marsadm invalidate cannot always resolve a split brain at other secondaries (which are neither the old

Command / Params	Cmp	Description
`secondary` `$res`	almost	Precondition: the local `/dev/mars/$dev_name` is no longer in use (e.g. umounted). Postcondition: There exists no designated primary any more. During split brain and when the network is OK (again), all actual primaries (including the local host) will leave primary ASAP (i.e. when their `/dev/mars/mydata` is no longer in use). Any secondary will start following (old) logfiles (even from backlogs) by replaying transaction logs if it is *uniquely* possible (which is often violated during split brain). On any secondary, `/dev/mars/$dev_name` will have disappeared.  Notice: in difference to DRBD, you **don't need** this command during normal operation, including handover. Any resource member which is *not* designated as primary will *automatically* go into secondary role. For example, if you have $k = 4$ replicas, only *one of them* can be designated as a primary. When the network is OK, all other 3 nodes will know this fact, and they will *automatically* go into secondary mode, following the transaction logs from the (new) primary.  Hint: avoid this command. It turns off *any* primary, **globally**[a]. You cannot start a sync after that (e.g. `invalidate` or `join-resource` or `resume-sync`), because it is *not unique* wherefrom the data shall be fetched. In split brain situations (when the network is OK again), this may have further drawbacks. It is much better / easier to ***directly* switch the designated primary** from one node to another via the `primary` command. See also section 3.4.2.2.  There is only one valid use case where you *really* need this command: before finally destroying a resouce via the *last* `leave-resource` (or the dangerous `delete-resource`), you will need this before you can do that. [a] A serious **misconception** among some people is when they believe that they can switch "a certain node to secondary". It is not possible to switch individual nodes to secondary, without affecting other nodes! The concept of "designated primary" is **global** throughout a resource!
`wait-umount` `$res`	no	See section 6.3.3.
`log-purge-all` `$res`	no	Precondition: none additionally. Postcondition: all locally known logfiles and version links are removed, whenever they are not / no longer reachable by any split brain version. Rationale: remove hindering split-brain / `leave-resource` leftovers. Use this only when split brain does not go away by means of `leave-resource` (which *could* happen in very weird scenarios such as MARS running on virtual machines doing a restore of their snapshots, or otherwise unexpected resurrection of dead or half-dead nodes).  THIS IS POTENTIALLY DANGEROUS! This command *might* destroy some valuable logfiles / other information in case the local information is outdated or otherwise incorrect. MARS does its best for checking anything, but there is no guarantee. Hint: use `--dry-run` beforehand for checking!
`resize` `$res` `[$size]`	almost	Precondition: The local host must be primary. All disks in the cluster participating in `$res` must be physically larger than the logical resource size (e.g, by use of `lvm`; can be checked by macros `%disk-size{}` and `%resource-size{}`). When the optional `$size` argument is present, it must be smaller than the minimum of all physical sizes, but larger than the current logical size of the resource. Postcondition: the logical size of `/dev/mars/$dev_name` will reflect the new size after a while.
Command / Params	Cmp	Description

6.2.3. Logfile Operations

Command / Params	Cmp	Description
`cron`	no	Do all necessary housekeeping tasks. See `log-rotate` and `log-delete-all` for details. This should be regularly called by an external cron job or similar.
`log-rotate` `$res`	no	Precondition: the local node `$host` must be primary at `$res`. Postcondition: after a while, a new transaction logfile `/mars/resource-$res/log-$new_nr-$host` will be used instead of `/mars/resource-$res/log-$old_nr-$host` where `$new_nr` = `$old_nr` + 1. Without `--force`, this will only carry out actions at the primary side since it makes no sense on secondaries. With `--force`, secondaries are *trying* to *remotely* trigger a log-rotate, but without any guarantee (likely even a split-brain may result instead, so use this only if you are *really* desperate).
`log-delete` `$res`	no	Precondition: the local node must be a member of `$res`. Postcondition: when there exists some old transaction logfiles `/mars/resource-$res/log-*-$some_host` which are no longer referenced by any of the symlinks `/mars/resource-$res/replay-*` , those logfiles are marked for deletion in the whole cluster. When no such logfiles exist, nothing will happen.
`log-delete-one` `$res`	no	Precondition: the local node must be a member of `$res`. Postcondition: when there exists an old transaction logfile `/mars/resource-$res/log-$old_nr-$some_host` where `$old_nr` is the minimum existing number and that logfile is no longer referenced by any of the symlinks `/mars/resource-$res/replay-*` , that logfile is marked for deletion in the whole cluster. When no such logfile exists, nothing will happen.
`log-delete-all` `$res`	no	Alias for `log-delete`.
Command / Params	Cmp	Description

6.2.4. Consistency Operations

Command / Params	Cmp	Description
`invalidate` `$res`	no	Precondition: the local node must be in secondary role at `$res`. A *designated* primary must exist. When having $k > 2$ replicas, no split brain must exist (otherwise, or when `invalidate` does not work in case of $k = 2$, use the `leave-resource` ; `join-resource` method described in section 3.4.3). Postcondition: the local disk is marked as inconsistent, and a fast fullsync from the designated primary will start after a while. Notice that `marsadm {pause,resume}-sync` will influence whether the sync really starts. When the fullsync has finished successfully, the local node will be consistent again.
`fake-sync` `$res`	no	Precondition: the local node must be in secondary role at `$res`. Postcondition: when a fullsync is running, it will stop after a while, and the local node will be *marked* as consistent as if it were consistent again.  ONLY USE THIS IF YOU REALLY KNOW WHAT YOU ARE DOING! See the WARNING in section 3.3 Use this only *before* creating a fresh filesystem inside `/dev/mars/$res`.
`set-replay`	no	 ONLY FOR ADVANCED HACKERS WHO KNOW WHAT THEY ARE DOING! This command is deliberately not documented. You need the competence level RTFS (“read the fucking sources”).
Command / Params	Cmp	Description

6.3. Further Operations

6.3.1. Inspection Commands

Command / Params	Cmp	Description
`view-`*`macroname`* `$res`	no	Display the output of a macro evaluation. See section 3.6 for a thorough description.
`view` `$res`	no	Equivalent to `view-default`.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`role` `$res`	no	Deprecated. Use `view-role` instead.
`state` `$res`	no	Deprecated. Use `view-state` instead.
`cstate` `$res`	no	Deprecated. Use `view-cstate` instead.
`dstate` `$res`	no	Deprecated. Use `view-dstate` instead.
`status` `$res`	no	Deprecated. Use `view-status` instead.
`show-state` `$res`	no	Deprecated. Don't use it. Use `view-state` instead, or other macros.
`show-info` `$res`	no	Deprecated. Don't use it. Use `view-info` instead, or other macros.
`show` `$res`	no	Deprecated. Don't use it. Use or implement some macros instead.
`show-errors` `$res`	no	Deprecated. Use `view-the-err-msg` or `view-resource-err` similar macros.
`cat` `$file`	no	Write the file content to stdout, but replace all occurences of numeric timestamps converted to a human-readable format. Thus is most useful for inspection of status and log files, e.g. `marsadm cat /mars/5.total.log`
Command / Params	Cmp	Description

6.3.2. Setting Parameters

6.3.2.1. Per-Resource Parameters

Command / Params	Cmp	Description
`set-emergency-limit` `$res` *n*	no	The argument *n* must be percentage between 0 and 100 %. When the remaining store space in `/mars/` undershoots the given percentage, the resource will go *earlier* into emergency mode than by the global computation described in section 4.4. 0 means unlimited.
`get-emergency-limit` `$res`	no	Inquiry of the preceding value.
Command / Params	Cmp	Description

6.3.2.2. Global Parameters

Command / Params	Cmp	Description
`set-sync-limit-value` *n*	no	Limit the concurrency of sync operations to some maximum number. 0 means unlimited.
`get-sync-limit-value`	no	Inquiry of the preceding value.
`set-sync-pref-list` `res1,res2,resn`	no	Set the order of preferences for syning. The argument must be comma-separated list of resource names.
`get-sync-pref-list`	no	Inquiry of the preceding value.
`set-connect-pref-list` `host1,host2,hostn`	no	Set the order of preferences for connections when there are more than 2 hosts participating in a cluster. The argument must be comma-separated list of node names.
`get-connect-pref-list`	no	Inquiry of the preceding value.
Command / Params	Cmp	Description

6.3.3. Waiting

Command / Params	Cmp	Description
wait-cluster	no	Precondition: the /mars/ filesystem must be mounted and it must contain a valid MARS symlink tree produced by the other marsadm commands. The kernel module must be loaded. Postcondition: none. Wait until *all* nodes in the cluster have sent a message, or until timeout. The default timeout is 30 s (exceptionally) and  Be may be changed by --timeout=$seconds
wait-resource $res {is-,}{attach, primary, device}{-off,}	no	Precondition: the local node must be a member of the resource $res. Postcondition: none. Wait until the local node reaches a specified condition on $res, or until timeout. The default timeout of 60 s may be changed by --timeout=$seconds. The last argument denotes the condition. The condition is inverted if suffixed by -off. When preceded by is- (which is the most useful case), it is checked whether the condition is actually reached. When the is- prefix is left off, the check is whether another marsadm command has been already given which *tries* to achieves the intended result (typically, you may use this after the is- variant has failed).
wait-connect $res	almost	This is an alias for wait-cluster waiting until only those nodes are reachable which belong to $res (instead of waiting for the *full* cluster).
wait-umount $res	no	Precondition: none additionally. Postcondition: the local /dev/mars/$dev_name is no longer in use (e.g. umounted).
Command / Params	Cmp	Description

6.3.4. Low-Level Expert Commands

These commands are for experts and advanced sysadmins only. The interface is not stable, i.e. the meaning may change at any time. Use at your own risk!

Command / Params	Cmp	Description
set-link	no	RTFS.
get-link	no	RTFS.
delete-file	no	RTFS.
Command / Params	Cmp	Description

The following commands are for manual setup / repair of cluster membership. Only to be used by experts who know what they are doing! In general, cluster-wide operations on IP addresses may need to be repeated at all hosts in the cluster iff the communication is not (yet) possible and/or not (yet) actually working (e.g. firewalling problems etc).

Command / Params	Cmp	Description
lowlevel-ls-host-ips	no	List all configured cluster members together with their currently configured IP addresses, as known *locally*.
lowlevel-set-host-ip $hostname $ip	no	Change the assignment of IP addresses *locally*. May be used when hosts are moved to different network locations, or when different network interfaces are to be used for replication (e.g. dedicated replication IPs). Notice that the names of hosts must not change at all, only their IP addresses may be changed. Check active connections with netstat & friends. Updates may need some time to proceed (socket timeouts etc).
lowlevel-delete-host $hostname	no	Remove a host from the cluster membership *locally*, together with its IP address assignment. This does not remove any further information. In particular, resource memberships are untouched.
Command / Params	Cmp	Description

6.3.5. Senseless Commands (from DRBD)

Command / Params	Cmp	Description
syncer	no	
new-current-uuid	no	
create-md	no	
dump-md	no	
dump	no	
get-gi	no	
show-gi	no	
outdate	no	
adjust	yes	Implemented as NOP (not necessary with MARS).
hidden-commands	no	
Command / Params	Cmp	Description

6.3.6. Forbidden Commands (from DRBD)

These commands are not implemented because they would be dangerous in MARS context:

Command / Params	Cmp	Description
invalidate-remote	no	This would be too dangerous in case you have multiple secondaries. A similar effect can be achieved with the --host= option.
verify	no	This would cause unintended side effects due to races between log-file transfer / application and block-wise comparison of the underlying disks. However, marsadm join-resource or invalidate will do the same as DRBD verify followed by DRBD resync, i.e. this will automatically correct any found errors;. Note that the fast-fullsync algorithm of MARS will minimize network traffic.
Command / Params	Cmp	Description

6.4. The /proc/sys/mars/ and other Expert Tweaks

In general, you shouldn't need to deal with any tweaks in /proc/sys/mars/ because everything should already default to reasonable predefined values. This interface allows access to some internal kernel variables of the mars.ko kernel module at runtime. Thus it is *not* a stable interface. It is not only specific for MARS, but may also change between releases without notice.

This section describes only those tweaks intended for sysadmins, not those for developers / very deep internals.

6.4.1. Syslogging

All internal messages produced by the kernel module belong to one of the following classes:

0 debug messages

1 info messages

2 warnings

3 error messages

4 fatal error messages

5 any message (summary of 0 to 4)

6.4.1.1. Logging to Files

These classes are used to produce status files $class.*.status in the /mars/ and/or in the /mars/resource-*mydata*/ directory / directories.

When you create a file `$class.*.log` in parallel to any `$class.*.status`, the `*.log` file will be appended forever with the same messages as in `*.status`. The difference is that *.status is regenerated anew from an empty starting point, while *.log can (potentially) increase indefinitely unless you remove it, or rename it to something else.


Beware, any permamently present `*.log` file can easily fill up your `/mars/` partition until the problems described in section 4.4 will appear. Use `*.log` only for a **limited time**, and **only for debugging!**

6.4.1.2. Logging to Syslog

The classes also play a role in the following `/proc/sys/mars/` tweaks:

`syslog_min_class` (rw) The *mimimum* class number for *permanent* syslogging. By default, this is set to -1 in order to switch off perment logging completely. Permament logging can easily flood your syslog with such huge amounts of messages (in particular when class=0), that your system as a whole may become unusable (because vital kernel threads may be blocked too long or too often by the userspace syslog daemon). Instead, please use the flood-protected syslogging described below!

`syslog_max_class` (rw) The *maximum* class number for *permanent* syslogging. Please use the flood-protected version instead.

`syslog_flood_class` (rw) The mimimum class of flood-protected syslogging. The maximum class is always 4.

`syslog_flood_limit` (rw) The maxmimum number of messages after which the flood protection will start. This is a hard limit for the the number of messages written to the syslog.

`syslog_flood_recovery_s` (rw) The number of seconds after which the internal flood counter is reset (after flood protection state has been reached). When no new messages appear after this time, the flood protection will start over at count 0.


The rationale behind flood protected syslogging: sysadmins are usually only interested in the point in time where some problems / incidents / etc have *started*. They are usually not interested in capturing *each* and *every* single error message (in particular when they are flooding the system logs).


If you *really* need complete error information, use the `*.log` files described above, compress them and save them to somewhere else *regularly* by a cron job. This bears much less overhead than filtering via the syslog daemon, or even remote syslogging in real time which will almost surely screw up your system in case of network problems co-inciding with flood messages, such as caused in turn by those problems. Don't rely on real-time concepts, just do it the old-fashioned batch job way.

6.4.1.3. Tuning Verbosity of Logging

`show_debug_messages` Boolean switch, 0 or 1. Mostly useful only for developers. This can easily flood your logs if our are not careful.

`show_log_messages` Boolean switch, 0 or 1.

`show_connections` Boolean switch, 0 or 1. Show detailed internal statistics on sockets.

`show_statistics_local / show_statistics_global` Only useful for kernel developers. Shows some internal information on internal brick instances, memory usage, etc.

6.4.2. Tuning the Sync

`sync_flip_interval_sec` (rw) The sync process must not run in parallel to logfile replay, in order to easily guarantee consistency of your disk. If logfile replay would be paused for the full duration of very large or long-lasting syncs (which could take some days over very slow networks), your `/mars/` filesystem could overflow because no replay would be possible in the meantime. Therefore, MARS regulary flips between actually syncing and actually replaying, if both is enabled. You can set the time interval for flipping here.

`sync_limit` (rw) When > 0, this limits the maximum number of sync processes actually running parallel. This is useful if you have a large number of resources, and you don't want to overload the network with sync processes.

`sync_nr` (ro) Passive indicator for the number of sync processes currently running.

`sync_want` (ro) Passive indicator for the number of sync processes which *demand* running.

7. Tips and Tricks

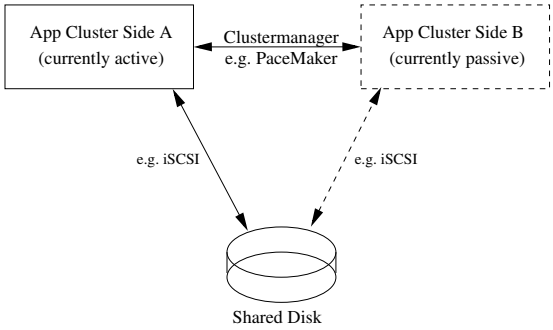
7.1. Avoiding Inappropriate Clustermanager Types for Medium and Long-Distance Replication

This section addresses some wide-spread misconceptions. Its main target audience is developers, but sysadmins will profit from **detailed explanations of problems and pitfalls**. When the problems described in this section are solved somewhen in future, this section will be shortened and some relevant parts moved to the appendix.

Doing **High Availability (HA)** wrong at *concept level* may easily get you into trouble, and may cost you several millions of € or $ in larger installations, or even knock you out of business when disasters are badly dealt with at higher levels such as clustermanagers.

7.1.1. General Cluster Models

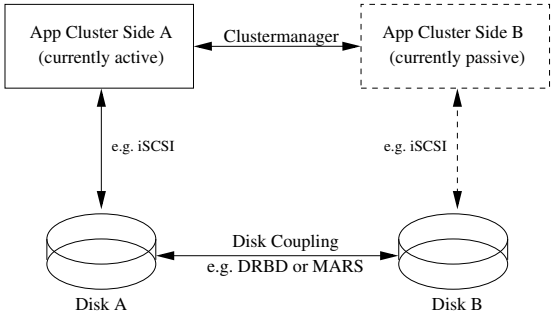
The most commonly known cluster model is called **shared-disk**, and typically controlled by clustermanagers like `PaceMaker`:



The most important property of shared-disk is that there exists only a single disk instance. Nowadays, this disk often has some *internal* redundancy such as RAID. At *system* architecure layer / network level, there exists no redundant disk at all. Only the application cluster is built redundant.

It should be immediately clear that shared-disk clusters are only suitable for short-distance operations in the same datacenter. Although running one of the data access lines over short distances between very near-by datacenters (e.g. 1 km) would be theoretically possible, there would be no sufficient protection against failure of a whole datacenter.

Both DRBD and MARS belong to a different architectural model called **shared-nothing**:



The characteristic feature of a shared-nothing model is (additional) **redundancy at network level**.

Shared-nothing "clusters[1]" could theoretically be built for *any* distances, from short to medium to long distances. However, concrete technologies of disk coupling such as synchronous operation may pose practical limits on the distances (see chapter 2).

In general, clustermanagers must fit to the model. Some clustermanager can be configured to fit to multiple models. If so, this must be done properly, or you may get into serious trouble.

Some people don't know, or they don't believe, that different architectural models like shared-disk or shared-nothing will *require* an *appropriate* type of clustermanager and/or a different configuration. Failing to do so, by selection of an inappropriate clustermanager type and/or an inappropriate configuration may be hazardous.

Selection of the right model alone is not sufficient. Some, if not many, clustermanagers have not been designed for long distances. As explained in section 7.1.5, long distances have further **hard requirements**. Disregarding them may be also hazardous!

7.1.2. Handover / Failover Reasons and Scenarios

From a sysadmin perspective, there exist a number of different **reasons** why the application workload must be switched from the currently active side A to the currently passive side B:

1. Some **defect** has occurred at cluster side A or at some corresponding part of the network.
2. Some **maintenance** has to be done at side A which would cause a longer downtime (e.g. security kernel update or replacement of core network equipment or maintainance of UPS or of the BBU cache etc - hardware isn't 24/7/365 in practice, although some vendors *claim* it - it is either not really true, or it becomes *extremely* expensive).

Both reasons are valid and must be automatically handled in larger installations. In order to deal with all of these reasons, the following basic mechanisms can be used in either model:

1. **Failover** (triggered either manually or automatically)
2. **Handover** (triggered manually[2])

It is important to not confuse handover with failover at concept level. Not only the reasons / preconditions are very different, but also the *requirements*. Example: precondition for handover is that *both* cluster sides are healthy, while precondition for failover is that *some relevant(!)* failure has been *detected* somewhere (whether this is *really* true is another matter). Typically, failover must be able to run in masses, while planned handover often has lower scaling requirements.

Not all existing clustermanagers are dealing with all of these cases (or their variants) equally well, and some are not even dealing with some of these cases / variants *at all*.

Some clustermanagers cannot easily express the concept of "automatic triggering" versus "manual triggering" of an action. There exists simply no cluster-global switch which selects either "manual mode" or "automatic mode" (except when you start to hack the code and/or write new plugins; then you might notice that there is almost no architectural layering / sufficient separation between mechanism and strategy). Being forced to permanently use an automatic mode for several hundreds or even thousands of clusters is not only boring, but bears a considerable risk when automatics do a wrong decision at hundreds of instances in parallel.

7.1.3. Granularity and Layering Hierarchy for Long Distances

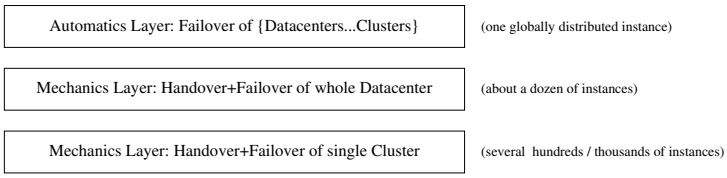
Many existing clustermanager solutions are dealing with a single cluster instance, as the term "*cluster*manager" suggests. However, when running several hundreds or thousands of cluster instances, you likely will not want to manage each of them individually. In addition, failover

[1] Notice that the term "cluster computing" usually refers to short-distance only. Long-distance coupling should be called "grid computing" in preference. As known from the scientific literature, grid computing requires different concepts and methods in general. Only for the sake of simplicity, we use "cluster" and "grid" interchangeably.

[2] Automatic triggering could be feasible for prophylactic treatments.

should *not only* be *triggered* (not to be confused with *executed*) individually at cluster level, but likely *also* at a higher granularity such as a room, or a whole datacenter. Otherwise, some chaos is likely to happen.

Here is what you probably will **need**, possibly in difference to what you may find on the market (whether OpenSource or not). For simplicity, the following diagram shows only two levels of granularity, but can be easily extended to multiple layers of granularity, or to some concept of various *subsets of clusters*:



Notice that many existing clustermanager solutions are not addressing the datacenter granularity at all. Typically, they use concepts like **quorums** for determining failures *at cluster level* solely, and then immediately executing failover of the cluster, sometimes without clean architectural distinction between trigger and execution (similar to the "separation of concerns" between **mechanism** and **strategy** in Operating Systems). Sometimes there is even no internal software layering / modularization according to this separation of concerns at all.

 When there is no distinction between different levels of granularity, you are hopelessly bound to a non-extensible and thus non-adaptable system when you need to operate masses of clusters.

 A lacking distinction between automatic mode and manual mode, and/or lack of corresponding **architectural software layers** is not only a blatant ignoration of well-established best practices of **software engineering**, but will bind you even more firmly to an inflexible system.

 Terminology: for practical reasons, we use the general term "clustermanager" also for speaking about layers dealing with higher granularity, such as datacenter layers, and also for long-distance replication scenarios, although some terminology from grid computing would be more appropriate in a scientific background.

Please consider the following: when it comes to long-distance HA, the above layering architecture is also motivated by vastly different numbers of instances for each layer. Ideally, the topmost automatics layer should be able to overview several datacenters in parallel, in order to cope with (almost) global network problems such as network partitions. Additionally, it should also detect single cluster failures, or intermediate problems like "rack failure" or "room failure", as well as various types of (partial / intermediate) (replication) network failures. Incompatible decisions at each of the different granularities would be a no-go in practice. Somewhere and somehow, you need one single[3] top-most *logical* problem detection / ranking instance, which should be *internally distributed* of course, typically using some **distributed consensus protocol**; but in difference to many published distributed consensus algorithms it should be able to work with multiple granularities at the same time.

7.1.4. Methods and their Appropriateness

7.1.4.1. Failover Methods

Failover methods are only needed in case of an incident. They should not be used for regular handover.

[3] If you have *logical pairs of datacenters* which are firmly bound together, you could also have several topmost automatics instances, e.g. for each *pair* of datacenters. However, that would be very **inflexible**, because then you cannot easily mix locations or migrate your servers between datacenters. Using $k > 2$ replicas with MARS would also become a nightmare. In your own interest, please don't create any concepts where masses of hardware are firmly bound to fixed constants at some software layers.

STONITH-like Methods STONITH = Shoot The Other Node In The Head

These methods are widely known, although they have several serious drawbacks. Some people even believe that *any* clustermanager must *always* have some STONITH-like functionality. This is wrong. There *exist* alternatives, as shown in the next paragraph.

The most obvious drawback is that STONITH will always create a **damage**, by definition.

Example: a typical contemporary STONITH implementation uses IPMI for automatically powering off your servers, or at least pushes the (virtual) reset button. This will *always* create a certain type of damage: the affected systems will definitely not be available, at least for some time until they have (manually) rebooted.

This is a conceptual contradiction: the reason for starting failover is that you want to restore availability as soon as possible, but in order to do so you will first *destroy* the availability of a particular *component*. This may be counter-productive.

Example: when your hot standby node B does not work as expected, or if it works even *worse* than A before, you will loose some time until you *can* become operational again at the old side A.

Here is an example method for handling a failure scenario. The old active side A is assumed to be no longer healthy anymore. The method uses a sequential state transition chain with a STONITH-like step:

Phase1 Check whether the hot standby B is currently usable. If this is violated (which may happen during certain types of disasters), abort the failover for any affected resources.

Phase2 *Try* to shutdown the damaged side A (in the *hope* that there is no *serious* damage).

Phase3 In case phase2 did not work during a grace period / after a timeout, assume that A is badly damaged and therefore STONITH it.

Phase4 Start the application at the hot standby B.

Notice: any cleanup actions, such as **repair** of defective hard- or software etc, are outside the scope of failover processes. Typically, they are executed much later when restoring redundancy.

Also notice: this method is a *heavily* distributed one, in the sense that sequential actions are alternated multiple times on different hosts. This is known to be cumbersome in distributed systems, in particular in presence of network problems.

Phase4 in more detail for DRBD, augmented with some pseudo code for application control:

1. at side B: `drbdadm disconnect all`
2. at side B: `drbdadm primary --force all`
3. at side B: `applicationmanager start all`

The same phase4 using MARS:

1. at side B: `marsadm pause-fetch all`
2. at side B: `marsadm primary --force all`
3. at side B: `applicationmanager start all`

This sequential 4-phase method is far from optimal, for the following reasons:

- The method tries to handle both failover and handover scenarios with one single sequential receipe. In case of a true failover scenario where it is *already known for sure* that side A is badly damaged, this method will unnecessarily waste time for phase 2. This could be fixed by introduction of a conceptual distinction between handover and failover, but it would not fix the following problems.
- Before phase4 is started (which will re-establish the service from a user's perspective), a lot of time is wasted by *both* phases 2 *and* 3. Even if phase 2 would be skipped, phase 3 would unnecessarily cost some time. In the next paragraph, an alternative method is explained which eliminates any unnecessary waiting time at all.
- The above method is adapted to the shared-disk model. It does not take advantage of the shared-nothing model, where further possibilities for better solutions exist.

- In case of long-distance network partitions and/or sysadmin / system management subnetwork outages, you may not even be able to (remotely) start STONITH at at. Thus the above method misses an important failure scenario.

Some people seem to have a *binary* view at the healthiness of a system: in their view, a system is either operational, or it is damaged. This kind of view is ignoring the fact that some systems may be half-alive, showing only *minor* problems, or occurring only from time to time.

It is obvious that damaging a healthy system is a bad idea by itself. Even *generally* damaging a half-alive system in order to "fix" problems is not generally a good idea, because it may increase the damage when you don't know the *real* reason[4].

Even worse: in a distributed system[5] you sometimes *cannot(!)* know whether a system is healthy, or to what degree it is healthy. Typical STONITH methods as used in some contemporary clustermanagers are **assuming a worst case**, even if that worst case is currently not for real.

Therefore, avoid the following **fundamental flaws** in failover concepts and healthiness models, which apply to implementors / configurators of clustermanagers:

- Don't mix up knowledge with conclusions about a (sub)system, and also don't mix this up with the real state of that (sub)system. In reality, you don't have any knowledge about a complex distributed system. You only may have *some* knowledge about *some* parts of the system, but you cannot "see" a complex distributed system as a whole. What you think is your knowledge, isn't knowledge in reality: in many cases, it is *conclusion*, not knowledge. Don't mix this up!
- Some systems are more complex than your model of it. Don't neglect important parts (such as networks, routers, switches, cables, plugs) which may lead you to wrong conclusions!
- Don't restrict your mind to boolean models of healthyness. Doing so can easily create unnecessary damage by construction, and even at concept level. You should know from software engineering that defects in concepts or models are much more serious than simple bugs in implementations. Choosing the wrong model cannot be fixed as easily as a typical bug or a typo.
- Try to deduce the state of a system as **reliably** as possible. If you don't know something for sure, don't generally assume that it has gone wrong. Don't confuse missing knowledge with the conclusion that something is bad. Boolean algebra restricts your mind to either "good" or "bad". Use at least **tri-state algebra** which has a means for expressing **"unknown"**. Even better: attach a probability to anything you (believe to) know. Errare humanum est: nothing is absolutely sure.
- Oversimplification: don't report an "unknown" or even a "broken" state for a complex system whenever a smaller subsystem exists for which you have some knowledge (or you can conclude something about it with reasonable evidence). Otherwise, your users / sysadmins may draw wrong conclusions, and assume that the whole system is broken, while in reality only some minor part has some minor problem. Users could then likely make wrong decisions, which may then easily lead to bigger damages.
- Murphy's law: **never assume that something can't go wrong!** Doing so is a blatant misconception at topmost level: the *purpose* of a clustermanager is creating High Availablity (HA) out of more or less "unreliable" components. It is the damn duty of both a clustermanager and its configurator to try to compensate *any* failures, *regardless of their probability*[6], as best as possible.

[4] Example, occurring in masses: an incorrectly installed bootloader, or a wrong BIOS boot priority order which unexpectedly lead to hangs or infinite reboot cycles once the DHCP or BOOTP servers are not longer available / reachable.

[5] Notice: the STONITH concept is more or less associated with short-distance scenarios where **crossover cables** or similare equipment are used. The assumption is that crossover cables can't go defective, or at least it would be an extremely unlikely scenario. For long-distance replication, this assumption is simply not true.

[6] Never claim that something has only low probability (and therefore it were not relevant). In the HA area, you simply **cannot know** that, because you typically have *sporadic* incidents. In extreme cases, the *purpose* of your HA solution is protection against 1 failure per 10 years. You simply don't have the time to wait for creating an incident statistics about that!

- Never confuse **probability** with **expectancy value!** If you don't know the mathematical term "expectancy value", or if you don't know what this means *in practice*, don't take responsibility for millions of € or $.

- When operating masses of hard- and software: never assume that a particular failure can occur only at a low number of instances. There are ***unknown(!)*** **systematic errors** which may pop up at the wrong time and in huge masses when you don't expect them.

- Multiple layers of fallback: *any* action can fail. Be prepared to have a plan B, and even a plan C, and even better a plan D, wherever possible.

- Never increase any damage anywhere, unnecessarily! Always try to *miminize* any damage! It can be mathematically proven that in deterministic probabilistic systems having finite state, increases of a damage level *at the wrong place* will *introduce* an *additional risk* of getting into an **endless loop**. This is also true for nondeterministic systems, as known from formal language theory[7].

- Use the **best effort principle**. You should be aware of the following fact: in general, it is impossible to create an *absolutely reliable system* out of unreliable components. You can *lower* the risk of failures to any $\epsilon > 0$ by investing a lot of resources and of money, but whatever you do: $\epsilon = 0$ is impossible. Therefore, be careful with boolean algebra. Prefer approximation methods / optimizing methods instead. Always do *your* best, instead of trying to reach a *global* optimum which likely does not exist at all (because the ϵ can only *converge* to an optimum, but will never actually reach it). The best effort principle means the following: if you discover a method for improving your operating state by reduction of a (potential) damage in a reasonable time and with reasonable effort, then **simply do it**. Don't argue that a particular step is no 100% solution for all of your problems. *Any improvement* is valuable. **Don't miss any valuable step** having reasonable costs with respect to your budget. Missing valuable measures which have low costs are certainly a violation of the best effort principle, because you are not doing *your* best. Keep that in mind.
  If you have *understood* this (e.g. deeply think at least one day about it), you will no longer advocate STONITH methods *in general*, when there are alternatives. STONITH methods are only valuable when you *know in advance* that the final outcome (after reboot) will most likely be better, and that waiting for reboot will most likely *pay off*. In general, this condition is *not true* if you have a healthy hot standby system. This should be easy to see. But there exist well-known clustermanager solutions / configurations blatantly ignoring[8] this. Only when the former standby system does not work as expected (this means that *all* of your redundant systems are not healthy enough for your application), *only then*[9] STONITH is unevitable as a *last resort* option.
  In short: blindly using STONITH without true need during failover is a violation of the best effort principle. You are simply not doing your best.

- When your budget is limited, carefully select those improvements which make your system **as reliable as possible**, given your fixed budget.

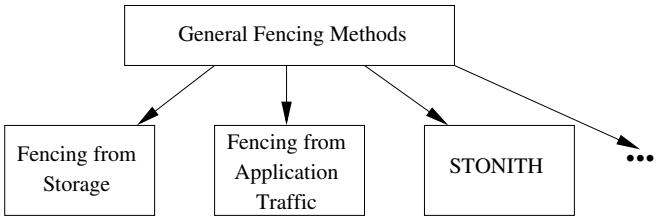
[7] Finite automatons are known to be transformable to deterministic ones, usually by an exponential increase in the number of states.

[8] For some *special(!)* cases of the shared-disk model, there exist some justifications for doing STONITH *before* starting the application at the hot standby. Under certain circumstances, it can happen that system A running amok could destroy the data on your single shared disk (example: a filesystem doubly mounted *in parallel*, which will certainly destroy your data, except you are using `ocfs2` or suchalike). This argument is only valid for *passive* disks which are *directly* attached to *both* systems A and B, such that there is no *external* means for fencing the disk. In case of iSCSI running over ordinary network equipment such as routers or switches, the argument "fencing the disk is otherwise not possible" does not apply. You can interrupt iSCSI connection at the network gear, or you can often do it at cluster A or at the iSCSI target. Even commercial storage appliances speaking iSCSI can be remotely controlled for forcefully aborting iSCSI sessions. In modern times, the STONITH method has no longer such a justification. The justification stems from ancient times when a disk was a purely passive mechanical device, and its disk controller was part of the server system.

[9] Notice that STONITH may be needed for (manual or partially automatic) *repair* in some cases, e.g. when you know that a system has a kernel crash. Don't mix up the repair phase with failover or handover phases. Typically, they are executed at different times. The repair phase is outside the scope of this section.

- Create statistics on the duration of your actions. Based on this, try to get a *balanced* optimum between time and costs.
- Whatever actions you can **start in parallel** for saving time, do it. Otherwise you are disregarding the best effort principle, and your solution will be sub-optimal. You will require deep knowledge of parallel systems, as well as experience with dealing with problems like (distributed) races. Notice that *any* distributed system is *inherently parallel*. Don't believe that sequential methods can deliver an optimum solution in such a difficult area.
- If you don't have the **necessary skills** for (a) recognizing already existing parallelism, (b) dealing with parallelism at concept level, (c) programming and/or configuring parallelism race-free and deadlock-free (or if you even don't know what a race condition is and where it may occur in practice), then don't take responsibility for millions of € or $.
- Avoid hard timeouts wherever possible. Use **adaptive timeouts** instead. Reason: depending on hardware or workload, the same action A may take a very short time on cluster 1, but take a very long time on cluster 2. If you need to guard action A from hanging (which is almost always the case because of Murphy's law), don't configure any fixed timeout for it. When having several hundreds of clusters, you would need to use the *worst case value*, which is the longest time occurring somewhere at the very slow clusters / slow parts of the network. This wastes a lot of time in case one of the fast clusters is hanging. Adaptive timeouts work differently: they use a kind of "progress bar" to monitor the *progress* of an action. They will abort only if there is *no progress* for a certain amount of time. Hint: among others, `marsadm view-*-rest` commands or macros are your friend.

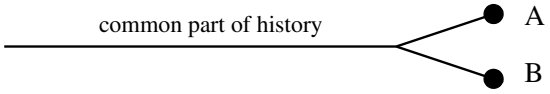
ITON = Ignore The Other Node This means **fencing from application traffic**, and can be used as an alternative to STONITH when done properly.



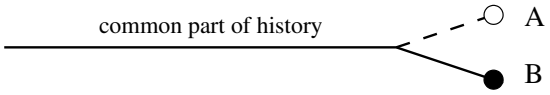
Fencing from application traffic is best suited for the shared-nothing model, but can also be adapted to the shared-disk model with some quirks.

The idea is simple: always route your application network traffic to the current (logically) active side, whether it is currently A or B. Just don't route any application requests to the current (logically) passive side at all.

For failover (and *only* for that), you *should not care about* any split brain occurring at the low-level generic block device:



Although having a split brain at the generic low-level block device, you now define the "logically active" and "logically passive" side by yourself by *logically ignoring* the "wrong" side as defined by yourself:



This is possible because the generic block devices provided by DRBD or MARS are completely **agnostic** of the "meaning" of either version A or B. Higher levels such as clustermanagers (or humans like sysadmins) can assign them a meaning like "relevant" or "not relevant", or "logically active" or "logically passive".

As a result of fencing from application traffic, the "logically passive" side will *logically* cease any actions such as updating user data, even if it is "physically active" during split-brain (when two primaries exist in DRBD or MARS sense[10]).

If you already have some load balancing, or BGP, or another *mechanism* for dynamic routing, you already have an important part for the ITON method. Additionally, ensure by an appropriate *strategy* that your balancer status / BGP announcement etc does always coincide with the "logically active" side (recall that even during split-brain *you* must define "logically active" **uniquely**[11] by yourself).

Example:

Phase1 Check whether the hot standby B is currently usable. If this is violated (which may happen during certain types of disasters), abort the failover for any affected resources.

Phase2 Do the following *in parallel*[12]*:*

- Start all affected applications at the hot standby B. This can be done with the same DRBD or MARS procedure as described on page 109.
- Fence A by fixedly routing all affected application traffic to B.

That's all which has to be done for a shared-nothing model. Of course, this will likely produce a split-brain (even when using DRBD in place of MARS), but that will not matter from a user's perspective, because the users will no longer "see" the "logically passive" side A through their network. Only during the relatively small time period where application traffic was going to the old side A while not replicated to B due to the incident, a very small number of updates *could* have gone lost. In fields like webhosting, this is taken into account. Users will usually not complain when some (smaller amount of) data is lost due to split-brain. They will complain when the service is unavailable.

This method is the fastest for restoring availability, because it doesn't try to execute any (remote) action at side A. Only from a sysadmin's perspective, there remain some cleanup tasks to be done during the following repair phase, such as split-brain resolution, which are outside the scope of this treatment.

By running the application fencing step *sequentially* (including wait for its partial successfulness such that the old side A can no longer be reached by any users) in front of the failover step, you may minimize the amount of lost data, but at the cost of total duration. Your service will take longer to be available again, while the amount of lost data is typically somewhat smaller.

A few people might clamour when some data is lost. In long-distance replication scenarios with high update traffic, there is *simply no way at all* for guaranteeing that no data can be lost ever. According to the laws of Einstein and the laws of Distributed Systems like the famous CAP theorem, this isn't the fault of DRBD+proxy or MARS, but simply the *consequence* of having long distances. If you want to protect against data loss as best as possible, then don't use $k = 2$ replicas. Use $k \geq 4$, and spread them over different distances, such as mixed small + medium + long distances. Future versions of MARS will support adaptive pseudo-synchronous modes, which will allow individual adaptation to network latencies / distances.

[10] Hint: some clustermanagers and/or some people seem to define the term "split-brain" differently from DRBD or MARS. In the context of generic block devices, split brain means that the *history* of both versions has been split to a Y-like **fork** (for whatever reason), such that re-joining them *incrementally* by ordinary write operations is no longer guaranteed to be possible. As a slightly simplified definition, you might alternatively use the definition "two incompatible primaries are existing in parallel", which means almost the same in practice. Details of formal semantics are not the scope of this treatment.

[11] A possible strategy is to use a Lamport clock for route changes: the change with the most recent Lamport timestamp will always win over previous changes.

[12] For database applications where no transactions should get lost, you should slightly modify the order of operations: first fence the old side A, then start the application at standby side B. However, be warned that even this cannot guarantee that no transaction is lost. When the network between A and B is interrupted *before* the incident happens, DRBD will automatically disconnect, and MARS will show a lagbehind. In order to fully eliminate this possibility, you can either use DRBD and configure it to hang forever during network outages (such that users will be unable to commit any transactions at all), or you can use the shared-disk model instead. But in the latter case, you are introducing a SPOF at the single shared disk. The former case is logically almost equivalent to shared-disk, but avoiding some parts of the physical SPOF. In a truly distributed system, the famous CAP theorem is limiting your possibilities. Therefore, no general solution exists fulfilling all requirements at the same time.

The ITON method can be adapted to shared-disk by additionally fencing the common disk from the (presumably) failed cluster node A.

7.1.4.2. Handover Methods

Planned handover is conceptually simpler, because both sides must be (almost) healthy as a *precondition*. There are simply no pre-existing failures to deal with.

Here is an example using DRBD, some application commands denoted as pseudo code:

1. at side A: `applicationmanager stop all`
2. at side A: `drbdadm secondary all`
3. at side B: `drbdadm primary all`
4. at side B: `applicationmanager start all`

MARS already has a conceptual distinction between handover and failover. With MARS, it becomes even simpler, because a generic handover procedure is already built in:

1. at side A: `applicationmanager stop all`
2. at side B: `marsadm primary all`
3. at side B: `applicationmanager start all`

7.1.4.3. Hybrid Methods

In general, a planned handover may fail at any stage. Notice that such a failure is also a failure, but (partially) caused by the planned handover. You have the following alternatives for automatically dealing with such cases:

1. In case of a failure, switch back to the old side A.
2. Instead, forcefully switch to the new side A, similar to the methods described in section 7.1.4.1.

Similar options exist for a failed failover (at least in theory), but chances are lower for actually recovering if you have only $k = 2$ replicas in total.

Whatever you decide to do in what case in whatever priority order, whether you decide it in advance or during the course of a failing action: it simply means that according to the best effort principle, you should **never leave your system in a broken state** when there exists a chance to recover availability with any method.

Therefore, you should *implement* neither handover nor failover in their pure forms. Always implement hybrid forms following the best effort principle.

7.1.5. Special Requirements for Long Distances

Most contemporary clustermanagers have been constructed for short distance shared-nothing clusters, or even for *local* shared-nothing clusters (c.f. DRBD over crossover cables), or even for shared-disk clusters (*originally*, when their *concepts* were developed). Blindly using them for long-distance replication without modification / adaptation bears some additional risks.

- Notice that long-distance replication always *requires* a **shared-nothing** model.
- As a consequence, **split brain** can appear *regularly* during failover. There is no way for preventing it! This is an *inherent property* of distributed systems, not limited to MARS (e.g. also ocurring with DRBD if you try to use it over long distances). Therefore, you *must* deal with occurences of split-brain as a *requirement*.
- The probability of **network partitions** is much higher: although you should have been required by Murphy's law to deal with network partitions already in short-distance scenarios, it now becomes *mandatory*.
- Be prepared that in case of certain types of (more or less global) internet partitions, you may not be able to trigger STONITH actions *at all*. Therefore, **fencing of application traffic** is *mandatory*.

7.2. systemd Templates

Starting with mars0.1stable57 (resp. mars0.1abeta9), you may use systemd as a cluster manager at the Mechanics Layer as explained in section 7.1.3 on page 107. MARS will replicate some systemd-relevant state information across the (big) cluster, so there is some limited remote operation support. In particular, automated handover via marsadm primary $resource is supported. More features will be added to future releases.

7.2.1. Why systemd?

All major Linux distributions are now systemd based. It is the new quasi standard. Although there have been some discussions in the community about its merits and shortcomings, it appears to be accepted now in large parts of the Linux world.

Systemd has a few advantages:

1. It is running as init process under the reserved pid=1. If it would ever die, then your system would die. There is no need for adding a new MARS clustermanager daemon or similar, which could fail independently from other parts of the system.

2. Although systemd has been criticised as being "monolithic" (referring to its internal software architecture), its *usage* by sysadmins is easily decomposable into many plugins called **units**.

3. Local LXC containers, local VMs, iSCSI exports, nfs exports and many other parts of the system are often already controlled by systemd. Together with udev and other parts, it already controls devices, LVM, mountpoints, etc. Since MARS is only a particular *component* in a bigger complicated stack, it is an advantage to use the same (more or less standardized and well-integrated) tools for managing the whole stack.

Systemd has also a few disadvantages:

1. It is not accepted everywhere. Therefore the systemd template extensions of marsadm are not mandatory for MARS operations. You can implement your own alternatives when necessary.

2. It can be messy to deal with. In particular, it can sometimes *believe* that the system *were* in a particular state, although in reality it isn't. Compensation is hairy.

3. Usablility / reporting: it is less usable for getting an overview over a bigger local system, and is practically unusable (out-of-the-box) for managing a bigger cluster at cluster level. Monitoring needs to be done separately.

7.2.2. Working Principle of the systemd Template Engine

Systemd already has some very basic templating capabilities. It is possible to create unit names containing the @ symbol, which can then be expanded under certain circumstances, e.g. to tty names etc. However, automatic expansion is only done when somebody knows the instance name already *in advance*. The author has not found any way for creating instance names out of "thin air", such as from dynamically created MARS resource names. Essentially, an *inference machine* for systemd templates does not yet exist.

This lacking functionality is completed with the following macro processing capabilities of marsadm:

Some ordinary or templated systemd unit files (see man systemd.unit) can be installed into one of the following directories: ./systemd-templates, $HOME/.marsadm/systemd-templates/, /etc/marsadm/systemd-templates/, /usr/lib/marsadm/systemd-templates/, /usr/local/lib/marsadm/systemd-templates/. Futher places can be defined by overriding the $MARS_PATH environment variable.

From these directories, ordinary systemd unit files will be just copied into /run/systemd/system/ (configurable via $SYSTEMD_TARGET_DIR) and then picked up by systemd as ordinary unit files.

Template unit files are nothing but unit files containing `@{varname}` parts or other macro definitions in their filename, and possibly also in their bodies, at arbitrary places. These `@{...}` parts are substituted by a `marsadm` macro processing engine.

The following macro capabilities are currently defined:

`@{varname}` Expands to the value of the variable. Predefined are the following variables:

- `@{res}` The MARS resource name.
- `@{resdir}` The MARS resource directory `/mars/resource-$res/`.
- `@{host}` The local host name as determined by `marsadm`, or as overridden by the `--host=` parameter.
- `@{cmd}` The `marsadm` command as given on the command line (only reasonable for debugging or for error messages).
- `@{varname}` Further variables as defined by the macro processor, see section 5.2.3 on page 87, and as definable by `%let{varname}{...}` statements, see also sections 5.1.1 on page 71 and 5.1.2 on page 76.

`@eval{text}` Calls the MARS macro processor as explained in chapter 5 on page 71, and substitutes its output.

`@esc{text}` Calls the `systemd-escape` tool for conversion of pathnames following the `systemd` naming conventions (see `man systemd-escape`). For example, a dash is converted to `\x2d`.


Omitting this can lead to problems when your resource names are containing special characters like dashes or other special symbols (in the sense of `systemd`). Bugs of this kind are hard to find and to debug. Either forbid special characters in your installation, or don't forget to test everything with some crude resource names!


Example snippet from a `.path` unit. Please notice where escaping is needed and where it must not be used (also notice that a dash is sometimes a legal part of the `.mount` unit name, but except from the resource name part):

```
[Path]
PathExists=/dev/mars/@{res}
Unit=vol-@escvar{res}.mount
```


Another source of crude bugs is the backslash character in the `systemd-escape` substitution, such as from `\x2d`. When passed to a shell, such as in certain `ExecStart=` statements, the backslash will be removed. Therefore, don't forget to either replace any single backslash with two backslashes, or to put the whole pathname in single quotes, or similar. Always check the result of your substitutions! It depends on the *target* (such as `bash`, as opposed to `systemd`) whether further escaping of the escapes is needed, or whether it *must not* be applied.


Become a master of the escaping hell by inserting debug code into your scripts (reporting to `/dev/stderr` or to log files) and do thorough testing like a devil.

`@escvar{varname}` Equivalent to `@esc{@{varname}}`.


When creating a new resource via `marsadm create-resource`, or when adding a new replica via `marsadm join-resource` or similar, the template system will automatically create new instances for the new resource or its replicas. Conversely, `marsadm leave-resource` and its friends like `delete-resource` etc will automatically remove the corresponding template instances from `/run/systemd/system/`.

7.2.3. Example systemd Templates

These can be found in the MARS repo in the `systemd/` subdirectory. At the moment, the following are available (subject to further extension and improvements):

`mars.path` This ensures that the mountpoint `/mars/` is already mounted before `mars.service` is started.

`mars.service` This starts and stops the MARS kernel module, provided that `/mars` is (somehow) mounted. The latter can be ensured by classical `/etc/fstab` methods, or by `.mount` units like your own hand-crafted `mars.mount` unit.

`mars-trigger.path` This is used for remote triggering of the marsadm template engine from another MARS cluster member, e.g. when initiating a handover. Local triggering is also possible via `touch /mars/userspace/systemd-trigger`. When triggered, the command `marsadm systemd-trigger` is executed. In turn, this will re-compute all `systemd` templates and start those units where the local host is in primary role.

`dev-mars-@{res}.path` This is used for generic triggering of any `systemd` unit as set by `marsadm set-systemd-unit $res $unit` (see below in section 7.2.4).

`vol-@{res}.mount` This is one of the possible sub-ordinate targets which depend on `dev-mars-@{res}.path`. For fully automatic activation of this target, use something like `marsadm set-systemd-unit mydata vol-mydata.mount` or similar.

7.2.4. Handover involving systemd

First, you need to install your systemd templates into one of the template directories mentioned in section 7.2.2. In case you have never used the template engine before, you can create the first instantiation via `marsadm systemd-trigger`. Afterwards, inspect `/run/systemd/system/` for newly created template instances and check them.

For each resource `$res`, you should set (potentially different) systemd targets via `marsadm set-systemd-unit $res ''$start_unit'' ''$stop_unit''`. Notice that `$start_unit` and `$stop_unit` are typically denoting different targets (with few exceptions) for the following reason:

Example: assume your stack consists of `vol-@{res}.mount` and `nfs-export-@{res}.service`. Before the filesystem can be exported via `nfs`, it *first* needs to be mounted. At startup, `systemd` can do this easily for you: just add a `Requires=` dependency between both targets, or similar. However, the situation can become tricky upon shutdown. Theoretically, `systemctl stop nfs-export-@{res}.service` *could* work in some cases, but in general it is not reliable. Reason: there might be other *sister* units which *also* depend on the mount. In some cases, you need not necessarily notice that sisters, because systemd can add further (internal) targets *automatically*. The problem is easily solvable by `systemctl stop vol-@{res}.mount`, which will automatically tear down all dependencies in reverse order.

For maximum safety, `$start_unit` should always point at the *tip* of your stack, while `$stop_unit` should point at the *bottom* (but one level higher than `/dev/mars/$res`).

Removing any systemd targets is also possible via `marsadm set-systemd-unit $res ''`.

When everything is set up properly, the following should work:

1. Issue `marsadm primary $res` on another node which is currently in secondary role.
2. As a consequence, `systemctl stop ''$stop_unit''` should be automatically executed at the old primary side.
3. After a while, the MARS kernel module will notice that `/dev/mars/$res` is no longer opened. You can check this manually via `marsadm view-device-opened $res` which will tell you a boolean result.

In case the device is not closed, ordinary handover cannot proceed, because somebody could (at least potentially) write some data into it, even after the handover, which

would lead to a split brain. Therefore MARS *must* insist that the device is closed before ordinary handover will proceed. In case it is not closed, you can (a) use `primary --force` which will likely provoke a split brain, or (b) check your `systemd` configuration or other sources of error why the device is not closed. Possible reasons could be hanging processes or hanging sessions which might need a `kill` or a `kill -9` or similar. Notice that `lsof` does not catch *all* possible sources like (recursive or bind-) mounts.

4. Once `/dev/mars/$res` has disappeared, the ordinary MARS handover from the old primary to the new site should proceed as usual.

5. After `/dev/mars/$res` has appeared at the new site, `systemctl start ''$start_unit''` should be automatically executed.

The rest depends on your `systemd` and its configuration. For example, you can configure systemd targets for activation of VMs, or for `LXC` containers, or for `iSCSI` exports, or for `nfs` exports, or for `glusterfs` exports, or for whatever you need. For true geo-redundancy, you will likely have to include some `quagga` or `bird` or other BGP configurations into your stack.

7.3. Creating Backups via Pseudo Snapshots

When all your secondaries are all homogenously located in a standby datacenter, they will be almost idle all the time. This is a waste of computing resources.

Since MARS is no substitute for a full-fledged backup system, and since backups may put high system load onto your active side, you may want to utilize your passive hardware resources in a better way.

MARS supports this thanks to its ability to switch the `pause-replay` *independently* from `pause-fetch`.

The basic idea is simple: just use `pause-replay` at your secondary site, but leave the replication of transaction logfiles intact by deliberately *not* saying `pause-fetch`. This way, your secondary replica (block device) will stay frozen for a limited time, without loosing your redundancy: since the transaction logs will continue to replicate in the meantime, you can start `resume-replay` at any time, in particular when a primary-side incident should happen unexpectedly. The former secondary will just catch up by replaying the outstanding parts of the transaction logs in order to become recent.

However, some *details* have to be obeyed. In particular, the current version of MARS needs an additional `detach` operation, in order to release exclusive access to the underlying disk `/dev/lv/$res`. Future versions of MARS are planned to support this more directly, without need for an intermediate `detach` operation.


Beware: `mount -o ro /dev/vg/$res` can lead to **unnoticed write operations** if you are not careful! Some journalling filesystems like `xfs` or `ext4` may replay their journals onto the disk, leading to *binary* differences and thus **destroying your consistency** later when you re-enable `resume-replay`!


Therefore, you may use small LVM snapshots (only in such cases). Typically, `xfs` journal replay will require only a few megabytes. Therefore you typically don't need much temporary space for this. Here is a more detailed description of steps:

1. `marsadm pause-replay $res`
2. `marsadm detach $res`
3. `lvcreate --size 100m --snapshot --name ro-$res /dev/vg/$res`
4. `mount -o ro /dev/vg/ro-$res /mnt/tmp`
5. Now draw your backup from `/mnt/tmp/`
6. `umount /mnt/tmp`

7. `lvremove -f /dev/vg/ro-$res`
8. `marsadm up $res`

Hint: during the backup, the transaction logs will accumulate on `/mars/`. In order to avoid overflow of `/mars/` (c.f. section 4.4), don't unnecessarily prolong the backup duration.

8. LV Football / VM Football / Container Football

The Football scripts can be obtained in two different ways:

1. `git clone --recurse-submodules https://github.com/schoebel/mars`
   then `cd mars/football/`

2. `git clone https://github.com/schoebel/football`

The `--recurse-submodule` method is the preferred way for non-developers because the main repo contains a link to the right version of Football.

Recommended MARS branch for playing Football is `mars0.1a.y`. Although the old stable branch `mars0.1.y` has been updated for the most important `marsadm` features `merge-cluster` and `split-cluster`, it does not scale well for Football and can cause operational problems when merging too many hosts together, showing some $O(n^2)$ metadata update behaviour where n is the number of machines in a MARS cluster. The future branch `mars0.1b.y` will contain more scalability improvements; in particular the `split-cluster` operation should no longer be needed at all because it is planned to scale with $O(k)$ where k is the number of resources at a *single* host. This should allow creation of a *virtual(!)* `BigCluster` pool at *metadata* level (where metadata transfer rates are typically measured in KiB/s), consisting of thousands of machines, while at the same time creating a `LocalSharding` or `FlexibleSharding` model at the realtime IO paths (where some petabytes are pumped through thick pipelines). Please check the other branches regularly at the github repo whether some newer branches will be marked "stable", or at least "beta". At the moment (spring 2018), `mars0.1a.y` is marked "beta" although it is in production at several thousands of machines for several months.

Low-level documentation is available by calling any of the scripts with `--help` parameter (see also appendix H.2 ff):

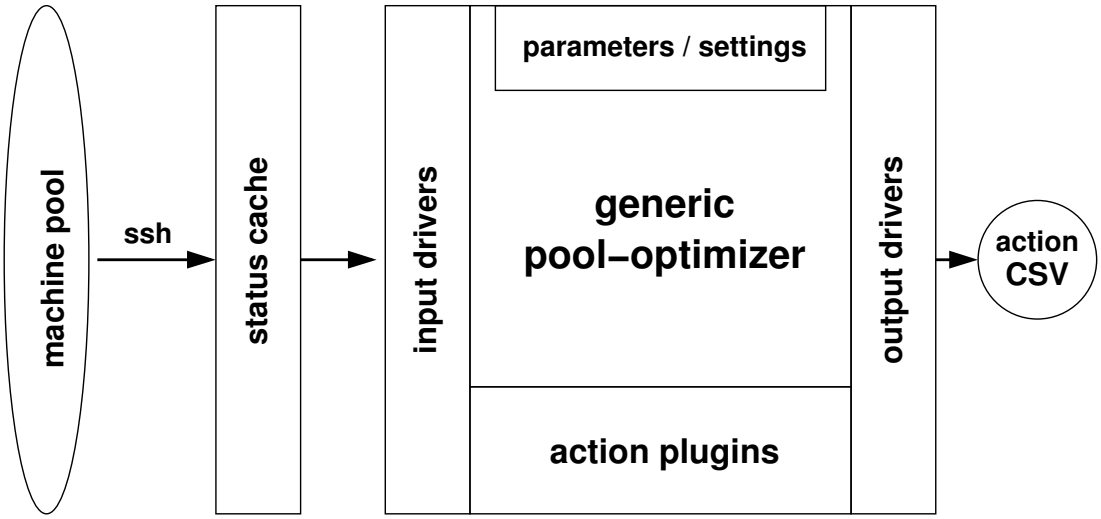
- `./football.sh --help`

- `./screener.sh --help`

By adding `--verbose`, you can get a list of parameters for configuring and tweaking.

8.1. Football Overview

Topmost architectural level:



The heart of the Football system is the generic pool optimizer, which aims to provide a similar functionality than Kubernetes, but working on a sharding architecture. Instead of controlling *stateless* Docker containers, its designated goal is to control masses of LVs on thousands of machines, creating a "Virtually Distributed LVM pool" (petabytes of total storage), and doing similar things than Software Defined Storage (SDS) on the virtual pool.

In addition to load balancing of storage space (and its special cases like hardware lifecycle), there are designated plugins for dealing with CPU and RAM dimensions. Further dimensions and a variety of goal functions could be added via future plugins. The optimizer itself aims to be as generic as possible, while functionality and interfaces can be added via plugins and/or drivers. Future versions might even support DRBD in addition to MARS. The current version uses a simple greedy algorithm for solving the underlying $\mathcal{NP}$-complete problem, but could be augmented with more sophisticated problem solvers in future.

The automatic operations generated by pool-optimizer are not only customizable by dozens of parameters, but also extendable by action plugins. At the moment, the following `football.sh` actions are implemented:

`migrate` This will move an LV (together with its VM / LXC container / etc) to a different machine in the machine pool. This is the classical Football "kick" operation.

`shrink` This decreases the occupied LV space of a filesystem (currently only `xfs` implemented, but easily extendable) via creation of a smaller temporary LV at the hypervisor, then transferring all data during operations via local `rsync`, then shutting down the VM for a short period, doing a final incremental `rsync`, renaming the copied temporary LV to its original name, restarting the VM on the new version (which contains the same data as before but wastes less space), and finally re-establishing the MARS replicas (but of course with smaller LV size).

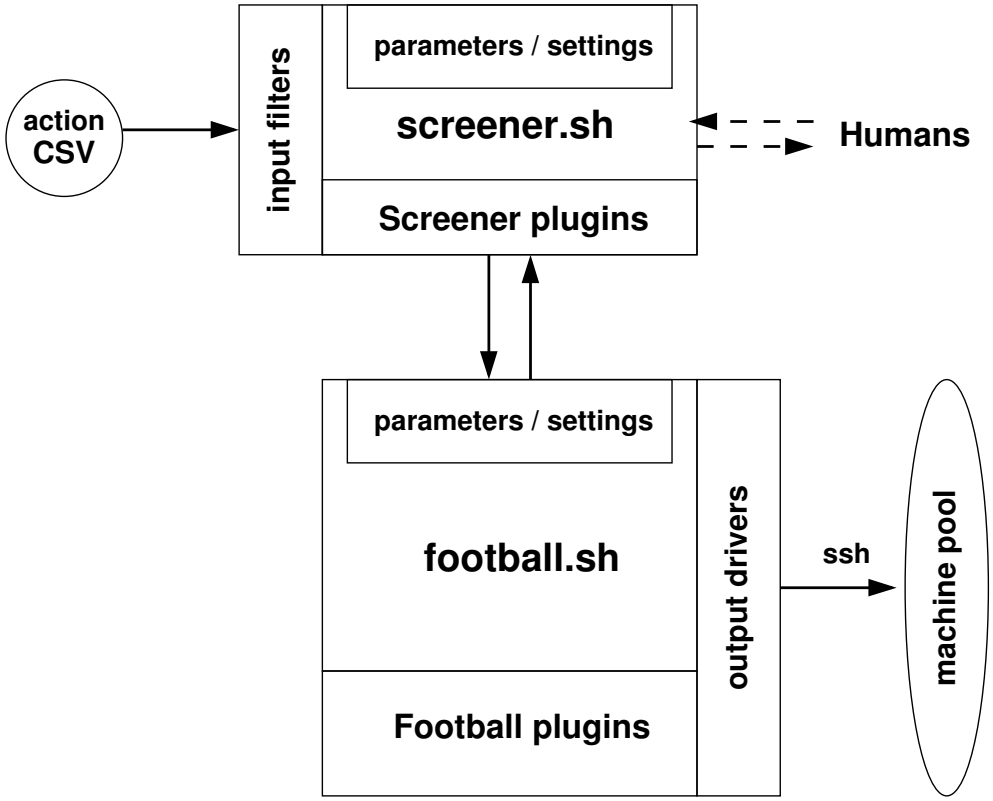
`extend` This is much easier than shrinking: it first increases the underlying LV size dynamically on all replicas, then `marsadm resize`, and finally calls `xfs_growfs` while the filesystem remains mounted and while the VM / container is running.

`migrate+shrink` Similar to `migrate` immediately followed by `shrink`, but produces less network traffic and runs faster.

`migrate+shrink+back` Use this when there is not enough local temporary space for shrinking. The LV is first migrated to a temporary host, then shrunk, and finally migrated back to its original position.

By running the overall system in an endless loop, a control loop for permanent optimization can be established. Typical periods are each few days, or once a week. In addition, manual triggering is also possible.

The result of an (incremental) pool-optimizer run is a CSV file, which may be automatically forwarded to the execution engine `football.sh` for *manual* execution, or to `screener.sh` for mass execution on a common control machine. Alternatively, intermediate steps like manual checking, filtering etc may be inserted into the processing pipeline.



The so-called Screener is simply a generic program allowing mass execution of arbitrary scripts in background `screen` sessions. This allows masses (several hundreds, possibly thousands) of long-lasting processes (hours or days) to run *unattended* in background, while allowing a (larger) group of sysadmins to attach / detach to `screen` sessions at any time for corrective by-hand actions, e.g. in case of failures or other problems, or for supervision, etc.

When Screener is combined with the Football execution engine `football.sh`, more specialized functionality is available (via a variety of plugins):

- Optional waiting for sysadmin confirmation before some customer downtime is initiated.
- Automatic generation of `motd` status reporting to other sysadmins.
- Automatic sending of email alerts or status reports, e.g. on errors or critical errors, etc. By sending email to SMS gateways, real-time alerting can be configured (e.g. over the weekend).
- Generic interfacing to external scripts with configurable parameters, e.g. for triggering monitoring systems, feeding external databases, etc.

Screener can detect and will automatically manage the following states (in this example, all state lists are empty):

```
$common_user> ./screener.sh list
List of waiting:
List of delayed:
List of running:
List of critical:
List of serious:
List of failed:
List of done:
```

Screener can discriminate the *seriosity* of errors as follows:

`failed` An error occurred *outside* of critical sections, e.g. during preparation of LV space etc. During ordinary operations, VMs / containers are usually running continuously, and there is no customer impact to be expected. Typically, `./screener.sh restart $resource`

should fix the problem if it is only a temporary problem. However, for maximum safety, manual inspection via `./screener.sh attach $resource` or inspection of the logfile via `./screener.sh show $resource` is recommended before trying an automatic restart.

`serious` An error occured while a VM / container was temporarily stopped, which **would** normally lead to customer downtime, but Football was able to *compensate* the problem *for now* by *automatically* restarting the VM. Thus no long-lasting customer impact has likely occurred. However, manual inspection and repair by sysadmins is likely necessary.

`critical` An *uncompensated* error occured during customer downtime. The VM / container is likely down. This will need manual sysadmin actions ASAP, such as hardware replacement, networking fixes, etc.

Ordinary Screener states during execution:

`running` This means that a (background) process is currently running. You can attach to the screen session either manually via `screen -x $pid.$resource`, or more comfortably via `./screener.sh attach $resource`. Then you can use `screen` as documented in `man screen`. The most important operation is detaching via keystrokes `Ctrl-a d`.


Notice: don't press `Ctrl-c` unless you know what you are doing. In most cases, this will terminate the running process, and in consequence lead to `failed` or even `critical` state (depending on the moment of keypress). Depending on parameter `drop_shell`, the Screener session will also terminate, or you will get an interactive shell for manual repair.

`waiting` When the plugins `football-waiting` and `screener-waiting` are configured properly (which is *not* the default), the script execution will pause immediately before a customer downtime action would be started. Now any sysadmin from the larger group has a chance to `./screener attach $resource` and to press RETURN to continue the waiting script and to personally watch the course of the critical section. There are some more comfortable variants like `./screener continue $resource` for background continuation of a single session, or `./screener continue 100` which can be used for continuing masses of waiting sessions. There are further variants which are automatically attaching to sessions, see Appendix H.4.

`delayed` This state is only entered before `lvremove $resource` is executed (which will destroy your old internal backup copy), and when configured appropriately. Typically, you also need to configure the `$wait_before_cleanup` variable in order to avoid endless waiting. Notice that old LV data gets soon outdated after a while, so please don't unnecessarily prolong the running time of your scripts by choosing too long `$wait_before_cleanup` values.

`done` This means that the script reported successful execution by exit status `0`. The background screen session terminated automatically. You can inspect the logfile manually via `./screener.sh show $resource`, or by looking into the directory `$screener_logdir/done/`.


Logfiles of other states can also be inspected (or monitored by standard tools like `grep`) by looking into sister directories, such as `$screener_logdir/running/`.


When running Screener for several months or years, old logfiles will accumulate in these directories over time. Call `./screener.sh purge` or `./screener.sh cron` regularly via a cron job, or archieve your old logfiles from time to time via another method.

8.2. HOWTO instantiate / customize Football

In order to install and operate Football, the recommended *deployment* strategy is bottom-up, layer by layer.

Top-down strategies should be used *only*, and *only*, for planning. An Egyptian pyramid can never be built, even if you had some billions of workers, by starting at the tip and by creating the foundations as the very last step. Suchalike attempt would end up in a disaster.

Testing of each layer **separately** is very important. Before proceeding to the next higher layer, first ensure that any lower layer is working *correctly*. Otherwise debugging can become tricky.

8.2.1. Block Device Layer

Step-by-step instructions can be found in chapter 3 on page 46.

Please ensure that your hardware (including RAID controllers and LVM and so on), and your operating system, and your network / setup, and MARS is working correctly before proceeding to the next layer.

8.2.2. Mechanics Layer of Cluster Operations

In the following example, it is assumed that `systemd` is used, as explained in section 7.2 on page 115, and now applied to `vm4711` supposed to run on hypervisors `hyper1234a` (primary role) and `hyper1234b` (secondary role), which is assumed to be controllable via the following `systemd` start and stop units:

- `marsadm set-systemd-unit vm4711 lxc-vm4711.target vol-vm4711.mount`

Test the cluster mechanics layer like in the following example:

- On host `hyper1234b`, the following must work: `marsadm primary vm4711`

This must result in an automatic handover of `vm4711` from the current primary site `hyper1234a` to the new primary `hyper1234b`, as explained in section 7.2 on page 115. Please check that `vm4711` is running correctly at the new location. It must be reachable via network. In case you are using BGP because `hyper1234a` and `hyper1234b` are located in different datacenters, ensure that BGP is also controlled by your `systemd` unit dependencies, and test it.

8.2.3. Mechanics Layer of Football Operations

At the moment, there are two alternative plugins already implemented in the Football subproject (see subdirectory `football/plugins/`). Of course, you can implement some further plugins. Please put them under GPL, and share them. Please contact the author of MARS for inclusion into the official MARS release.

`football-cm3.sh` This plugin can be only used at Shared Hosting Linux (ShaHoLin) at 1&1, since it is bound to a specific *proprietary* instance. However, the *sourcecode* of the *plugin* itself (not the code called by the plugin, e.g. over REST interfaces) is under GPL. You can (and *should*) *inspect* the plugin code, and **learn** how a real-world system (which has grown over some decades and bears a lot of history) is actually working at certain points. This plugin is automatically activated when called via the symlink `tetris.sh` instead of directly calling `football.sh`. This has historic reasons.

`football-basic.sh` This plugin uses the new `systemd` interface of `marsadm` for controlling the mechanics. See section 7.2 on page 115. You should be familiar with commands like `marsadm set-systemd-unit`. Manual handover via `marsadm primary $resource` must be already working (with high reliability $\rightsquigarrow$ check that any `umount` works everywhere without hangups) before you can start using this plugin for `football.sh`.
This plugin is automatically activated when calling football.sh. It can be deactivated by overriding variable `enable_basic`=0.

8.2.3.1. Configuring and Overriding Variables

A detailed list of all available customization options can be obtained via `./football.sh --help --verbose`. Each option is documented by some help text, and you can always see the default settings. See also section H.3 on page 159.

If you create any new plugin for Football, or if you modify an existing one, please follow these standards. Try to describe any option as concisely as possible.

Configuring is possible in the following ways, in order of precedence:

- at the command line via `./football.sh --$variable_name=$value $arguments`.
- via environment variables, e.g. globally via `export $variable_name=$value && ./football.sh $arguments`, or locally via `$variable_name=$value ./football.sh $arguments`.
- by adding some small `football-*.conf` files into one of the directories `/usr/lib/mars/plugins` `/etc/mars/plugins` `$script_dir/plugins` `$HOME/.mars/plugins` `./plugins`, in this order of precedence. This list of directories can be modifed externally over the environment variable `football_includes` (but not during already running inclusions of `football-*.conf` files).

8.2.3.2. `football-basic.sh` Customization

Here is a brief summary of the most important configuration tasks and options:

`initial_hostname_file` Somehow, the `football-basic.sh` plugin must know the hostnames of your pool. Once Football is working, the hostname will be *automatically* maintained whenever `marsadm join-cluster` or `marsadm merge-cluster` is executed somewhere.

For your hardware deployment strategy, this means the following: just deploy any new hardware, or remove your old one (after Football has emptied all of your former LV resources). It does not matter how you are doing this, e.g. via OpenStack, or via the proprietary `Schlunix` methods used at ShaHoLin, or whatever. Then you have the following options for adding the new machines to the Football hostname cache (see variable `hostname_cache`):

1. Write the pure hostname(s) into the file as configured with `initial_hostname_file` (by default: `./hostnames.input`). Each hostname must be on its own ASCII line. Not only these new hosts will be picked up automatically, but also...
2. ...any further hosts reported anywhere (at the already known hosts) by `marsadm view-cluster-members`, **transitively**.

   Consequence: if you are running the new `mars0.1b.y` (or newer) branch of MARS, you don't need `marsadm split-cluster` anymore. Then you can operate several thousands of machines as a big **virtual** cluster, even if their storage is local (see `LocalSharding` model described in section 1.4.1 on page 17).

   Previous versions of MARS, like `mars0.1.y` and `mars0.1a.y`, are not yet scalable at their **metadata** exchange level. Trying to `join-cluster` or `merge-cluster` several tens or even hundreds of machines with those versions will surely lead to a disaster. Always use `marsadm split-cluster` at those versions, regularly. First upgrade to the future `mars0.1b.y` (or later versions) before creating big clusters at *metadata* level!
3. Use `./football.sh basic_add_host $hostname` for adding a single new host manually. Afterwards, the transitive closure of all reachable hosts is computed as usual. This may also be used for the very first initialization of a fresh Football installation, provided you already have a big cluster at metadata level.

Test the Football mechanics like one of the following example command sequences, where it is assumed that `hyper4321a` and `hyper4321b` are already *newly* deployed hypervisors having enough local LVM storage, and have been already added to the MARS cluster via `marsadm join-cluster`, or have been at least added to `hostname_cache` as explained above:

- `ssh-add; ./football.sh migrate vm4711 hyper4321a hyper4321b`
- `ssh-add; ./football.sh migrate vm4711 hyper4321a hyper4321b --screener; ./screener.sh attach vm4711`

Check the automatically produced logfile (via `./screener.sh show vm4711`) that Football has automatically determined the old hypervisor where `vm4711` was running before, that it has automatically executed `marsadm merge-cluster` when necessary, and has created the LV replicas at the new hypervisors, and has executed some `marsadm join-resource` commands, has automatically waited for MARS fast fullsync to finish, then successfully executed an automatic handover to the new primary hypervisor, and finally has destructed the old MARS replicas including their old LVs. Check that `vm4711` is running correctly at the new hypervisor pair, and that handover between the new hypervisor sites `*a` and `*b` is working correctly.

A larger group of sysadmins can co-work over a central common control machine via ssh agent forwarding (which must be enabled in `/etc/ssh/sshd_config`) in the following way:

- At the workstation: `ssh-add; ssh -A football@common-control.mycompany.org`
  Then `cd $script_dir` and run your `./football.sh` or `./screener.sh` commands as usual. The automatically generated logfiles will be tagged with the *real* usernames from your original workstation login, as reported by `ssh-add -l`, even transitively when using ssh agent forwarding. Thus you may use a common username like `football` on the common[1] control machine.

Hint: use `./screener.sh list` (or one of its more specific variants like `./screener.sh list-running`) for determining what's currently going on in a larger group of sysadmins.

[1] Of course, it is also possible to maintain individual accounts for the same Unix group, and set `umask` and common directory permissions accordingly, such that the classical group-wise working concept from the 1970s will do the rest. This is much more work, but can establish more fine-grained access control. Even more sophisticated methods could involve ACLs, but suchalike is probably only necessary at extremely high-sensitive installations.

9. MARS for Developers

This chapter is organized strictly top-down.

If you are a sysadmin and want to inform yourself about internals (useful for debugging), the relevant information is at the beginning, and you don't need to dive into all technical details at the end.

If you are a kernel developer and want to contribute code to the emerging MARS community, please read it (almost) all. Due to the top-down organization, sometimes you will need to follow some forward references in order to understand details. Therefore I recommend reading this chapter twice in two different reading modes: in the first reading pass, you just get a raw network of principles and structures in your brain (you don't want to grasp details, therefore don't strive for a full understanding). In the second pass, you will exploit your knowlegde from the first pass for a deeper understanding of the details.

Alternatively, you may first read the sections about general architecture, and then start a bottom-up scan by first reading the last section about generic objects and aspects, and working in reverse *section* order (but read *sub*sections in-order) until you finally reach the kernel interfaces / symlink trees.

9.1. Motivation / Politics

MARS is not yet upstream in the Linux kernel. This section tries to clear up some potential doubts. Some people have asked why MARS uses its own internal framework instead of *directly*[1] being based on some already existing Linux kernel infrastructures like the device mapper. Here is a list of technical reasons:

1. The existing device mapper infrastructure is based on `struct bio`. In contrast, the new XIO personality of the generic brick infrastructure is based on the concept of AIO (Asynchronous IO), which is a **true superset** of block IO.

2. In particular, `struct bio` is firmly referencing to `struct page` (via intermediate `struct bio_vec`), using types like `sector_t` in the field `bi_sector`. Basic transfer units are blocks, or sectors, or pages, or the like. In contrast, `struct aio_object` used by the XIO personality can address **arbitrary granularity** memory with byte resolution even at odd[2] positions in (virtual) files / devices, similar to classical Unix file IO, but *asynchronously*. Practical experience shows that even non-functional properties like performance of many datacenter workloads are profiting from that[3]. The AIO/XIO abstraction contains no fixed link to kernel abstractions and should be **easily portable** to other environments. In summary, the new personality provides a uniform abstraction which abstracts away from multiple different kernel interfaces; it is designed to be useful even in userspace.

3. Kernel infrastructures for the concept of *direct IO* are different from those for *buffered IO*. The XIO personality used by MARS subsumes both concepts as use case *variants*.

[1] Notice that *indirect* use of pre-existing Linux infrastructure is not only possible, but actually implemented, by usinig it *internally* in brick *implementations* (black-box principle). However, such bricks are not portable to other environments like userspace.

[2] Some brick *implementations* (as opposed to the capabilities of the *interface*) may be (and, in fact, *are*) restricted to `PAGE_SIZE` operations or the like. This is no general problem, because IOP can automatically insert some translator bricks extending the capabilities to universal granularity (of course at some performance costs).

[3] The current transaction logger uses variable-sized headers at "odd" addresses. Although this increases `memcpy()` load due to "misalignment", the *overall performance* was provably better than in variants where sector / page alignment was strictly obeyed, but space was wasted for alignments. Such functionality is only possible if the XIO infrastructure *allows for* (but doesn't force) "mis-aligned" IO operations. In future, many different transaction logfile formats showing different runtime behaviour (e.g. optimized for high-throughput SSD loads) may co-exist in parallel. Note that properly aligned XIO operations bear no noticeable overhead compared to classical block IO, at least in typical datacenter RAID scenarios.

Buffering is an optional internal property of XIO bricks (almost non-functional property with support for consistency guarantees).

4. The AIO/XIO personality is generically designed for remote operations over networks, at arbitrary places in the IO stack, with (almost[4]) no semantic differences to local operations (built-in **network transparency**). There are universal provisions for mixed operation of different versions (**rolling software updates** in clusters / grids).

5. The generic brick infrastructure (as well as its personalities like XIO or any other future personality) supports **dynamic re-wiring / re-configuration** *during* operation (even while parallel IO requests are flying, some of them taking different paths in the IO stack in parallel). This is absolutely needed for MARS logfile rotation. In the long term, this would be useful for many advanced new features and products, not limited to multipathing.

6. The generic brick infrastructure (and in turn all personalities) provide **additional comfort** to the programmer while enabling **increased functionality**: by use of a generalization of **aspect orientation**[5], the programmer need no longer worry about dynamic memory allocations for *local state* in a brick instance. MARS is **automating local state** even when dynamically instantiating new bricks (possibly having the same brick type) at runtime. Specifically, XIO is automating **request stacking** at the completion path this way, even while dynamically reconfiguring the IO stack[6]. A similar automation[7] does not exist in the rest of the Linux kernel.

7. The generic brick infrastructure, together with personalities like XIO, enables **new long-term functional and non-functional opportunities** by use of concepts from instance-oriented programming (IOP[8]). The application area is **not limited to device drivers**. For example, a new personality for *stackable filesystems* could be developed in future.

In summary, anyone who would insist that MARS should be *directly*[9] based on pre-existing kernel structures / frameworks instead of contributing a new framework would cause a *massive regression of functionality*.

- On one hand, all code contributed by the MARS project is **non-intrusive** into the rest of the Linux kernel. From the viewpoint of other parts of the kernel, the whole addition *behaves like* a driver (although its infrastructure is much more than a driver).

- On the other hand, if people are interested, the contributed infrastructure *may* be used to *add* to the power of the Linux kernel. It is designed to be **open for contributions**.

[4] By default, automatic network connection re-establishment and infinite network retries are already implemented in the `xio_client` and `xio_server` bricks to provide fully transparent semantics. However, this may be undesirable in case of fatal crashes. Therefore, abort operations are also configurable, as well as network timeouts which are then mapped to classical IO errors.

[5] Similar to AOP, insertion of IOP bricks for checking / debugging etc is one of the key advantages of the generic brick infrastructure. In contrast to AOP where debugging is usually {en,dis}abled statically at compile time, IOP allows for *dynamic* (re-)configuration of debugging bricks, automatic repair, and many more features promoted by *organic computing*.

[6] The generic aspect orientation approach leads to better **separation of concerns**: local state needed by brick implementations is not visible from outside by default. In other words, local state is also **private state**. Accidental hampering of internal operations is impeded.

Example from the kernel: in `include/linux/blkdev.h` the definition of `struct request` contains the following comment: `/* the following two fields are internal, NEVER access directly */`. It appears that `struct request` contains not only fields relevant for the caller, but also **internal fields** needed only in *some specific* callees. For example, `rb_node` is documented to be used only in IO schedulers.

XIO goes one step further: there need not exist exactly one IO scheduler instance in the IO stack for a single device. Future `xio_scheduler_{deadline,cfq,...}` brick types could be each instantiated many times, and in arbitrary places, even for the same (logical) device. The equivalent of `rb_node` would then be automatically instantiated multiple times for the same IO request, by automatically instantiating the right local aspect instances.

[7] DM can achieve stacking and dynamic routing by a workaround called *request cloning*, potentially leading to mass creation of temporary / intermediate object instances.

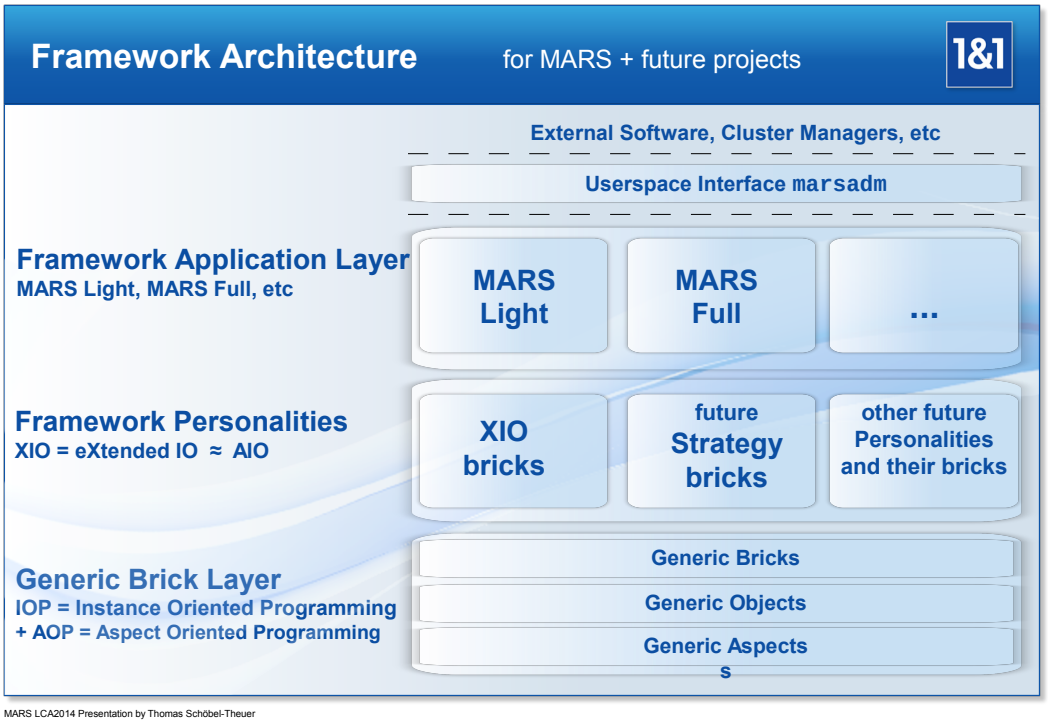
[8] See http://athomux.net/papers/paper_inst2.pdf

[9] Notice that kernel-specific structures like `struct bio` are of course used by MARS, but only *inside* the blackbox implementation of bricks like `mars_bio` or `mars_if` which act as **adaptors** to/from that structure. It is possible to write further adaptors, e.g. for direct interfacing to the device mapper infrastructure.

- A *possible* (but not the only possible) way to do this is giving the generic brick framework / the XIO personality as well as future personalities / the MARS application the status of a *subsystem* inside the kernel (in the long term), similar to the SCSI subsystem or the network subsystem. Noone is forced to use it, but anybody may use it if he/she likes.

- Politically, the author is a FOSS advocate willing to collaborate and to support anyone interested in contributions. The author's personal interest is long-term and is open for both in-tree and out-of-tree extensions of both the framework and MARS by any other party obeying the GPL and not hazarding FOSS by patents (instead supporting organizations like the Open Invention Network). The author is open to closer relationships with the Linux Foundation and other parts of the Linux ecosystem.

9.2. Architecture Overview

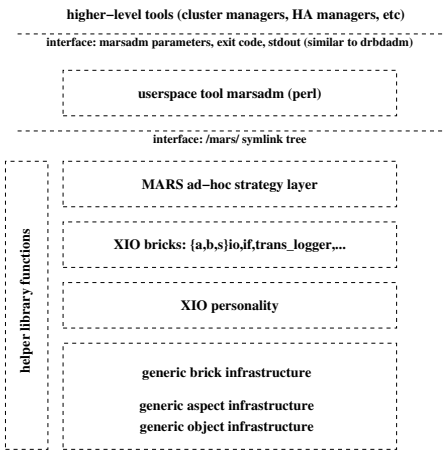


9.3. Some Architectural Details

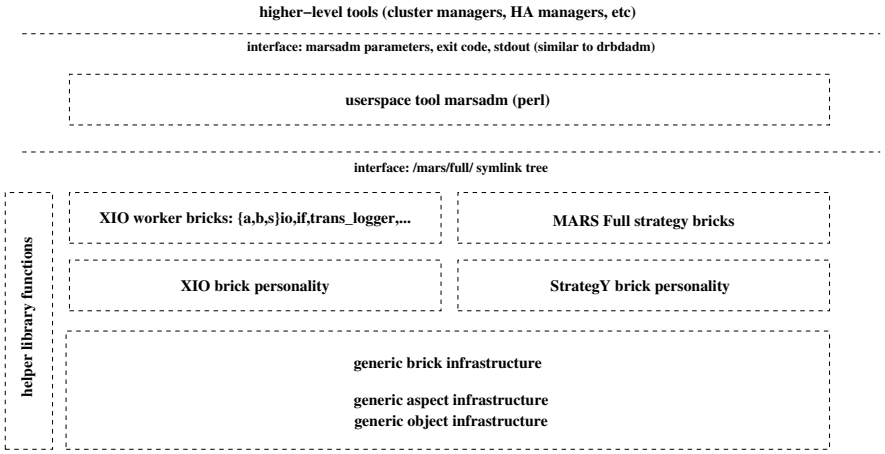
The following pictures show some "zones of responsibility", not necessarily a strict hierarchy (although Dijkstra's famous layering rules from THE are tried to be respected as much as possible). The construction principle follows the concept of **Instance Oriented Programming** (IOP) described in http://athomux.net/papers/paper_inst2.pdf. Please note that MARS is only instance-*based*[10], while MARS Full is planned to be fully instance-*oriented*.

9.3.1. MARS Architecture

[10] Similar to OOP, where "object-based" means a weaker form of "object-oriented", the term "instance-based" means that the *strategy* brick layer need not be fully modularized according to the IOP principles, but the *worker* brick layer already is.



9.3.2. MARS Full Architecture (planned)



9.4. Documentation of the Symlink Trees

The `/mars/` symlink tree is serving the following purposes, all at the same time:

1. For **communication** between cluster nodes, see sections 4.2 and 4.3. This communication is even the *only* communication between cluster nodes (apart from the *contents* of transaction logfiles and sync data).

2. ***Internal* interface** between the kernel module and the userspace tool `marsadm`.

3. ***Internal* persistent repository** which keeps state information between reboots (also in case of node crashes). It is even the *only* place where state information is kept. There is no other place like `/etc/drbd.conf`.

 Because of its internal character, its representation and semantics may change at any time without notice (e.g. via an *internal* upgrade procedure between major releases). It is *not* an external interface to the outer world. Don't build anything on it.

However, knowledge of the symlink tree is useful for advanced sysadmins, for **human inspection** and for **debugging**. And, of course, for developers.

As an "official" interface from outside, only the `marsadm` command should be used.

9.4.1. Documentation of the MARS Symlink Tree

9.5. XIO Worker Bricks

9.6. StrategY Worker Bricks

NYI

9.7. The XIO Brick Personality

9.8. The Generic Brick Infrastructure Layer

9.9. The Generic Object and Aspect Infrastructure

A. Technical Data MARS

MARS has some built-in limitations which should be overcome[1] by the future MARS Full. Please don't exceed the following limits:

- maximum 10 nodes per cluster
- maximum 10 resources per cluster
- maximum 100 logfiles per resource

[1] Some internal algorithms are quadratic. The reason is that MARS evolved from a lab prototype which wasn't originally intended for enterprise grade usage, but should have been succeeded by the fully instance-oriented MARS Full much earlier.

B. Handout for Midnight Problem Solving

Here are generic instructions for the generic `marsadm` and commandline level. Other levels (e.g. different types of cluster managers, PaceMaker, control scripts / `rc` scripts / `upstart` scripts, etc should be described elsewhere.

B.1. Inspecting the State of MARS

For manual inspection, please prefer the new `marsadm view all` over the old `marsadm view-1and1 all`. It shows more appropriate / detailed information.

Hint: this might change in future when somebody will program better marcros for the `view-1and1` variant, or create even better other macros.

```
# watch marsadm view all
```

Checking the low-level network connections at runtime:

```
# watch "netstat --tcp | grep 777"
```

Meaning of the port numbers (as currently configured into the kernel module, may change in future):

- 7777 = metadata / symlink propagation
- 7778 = transfer of transaction logfiles
- 7779 = transfer of sync traffic

7777 must be always active on a healthy cluster. 7778 and 7779 will appear only on demand, when some data is transferred.

Hint: when one of the columns Send-Q or Recv-Q are constantly at high values, you might have a network bottleneck.

B.2. Replication is Stuck

Indications for a stuck:

- One of the flags shown by `marsadm view all` or `marsadm view-flags all` contain a symbol `"-"` (dash). This means that some switch is currently switched off (deliberately). Please check whether there is a valid reason why somebody else switched it off. If the switch-off is just by accident, use the following command to fix the stuck:

  ```
  # marsadm up all
  ```

  (or replace `all` by a particular resource name if you want to start only a specific one). Note: `up` is equivalent to the sequence `attach; resume-fetch; resume-replay; resume-sync`. Instead of switching each individual knob, use `up` as a shortcut for switching on anything which is currently off.

- `netstat --tcp | grep 7777` does not show anything. Please check the following:

  - Is the kernel module loaded? Check `lsmod | grep mars`. When necessary, run `modprobe mars`.
  - Is the network interface down? Check `ifconfig`, and/or `ethtool` and friends, and fix it when necessary.
  - Is a `ping <partner-host>` possible? If not, fix the network / routing / firewall / etc. When fixed, the MARS connections should automatically appear after about 1 minute.
  - When `ping` is possible, but a MARS connection to port 7777 does not appear after a few minutes, try to connect to remote port 7777 by hand via `telnet`. But don't type anything, just abort the connection immediately when it works! Typing anything will almost certainly throw a harsh error message at the other server, which could unnecessarily alarm other people.

- Check whether `marsadm view all` shows some progress bars somewhere. Example:

```
istore-test-bap1:~# marsadm view all
---------- resource lv-0
 lv-0 OutDated[F] PausedReplay dCAS-R Secondary istore-test-bs1
    replaying: [>..................]  1.21%  (12/1020)MiB
logs: [2..3]
       > fetch: 1008.198 MiB rate: 0 B/sec
remaining: --:--:-- hrs
       > replay: 0 B rate: 0 B/sec  remaining: 00:00:00 hrs
```

  At least one of the `rate:` values should be greater than 0. When none of the `rate:` values indicate any progress for a longer time, try `marsadm up all` again. If it doesn't help, check and repair the network. If even this does not help, check the hardware for any IO hangups, or kernel hangups. First, check the RAID controllers. Often (but not certainly), a stuck kernel can be recognized when many processes are *permanently* in state "D", for a long time: `ps ax | grep " D" | grep -v grep` or similar. Please check whether there is just an overload, or *really* a true kernel problem. Discrimination is not easy, and requires experience (as with any other system; not limited to MARS). A truly stuck kernel can only be resurrected by rebooting. The same holds for any hardware problems.

- Check whether `marsadm view all` reports any lines like `WARNING: SPLIT BRAIN at '' detected`. In such a case, check that there is *really* a split brain, before obeying the instructions in section B.4. Notice that network outages or missing `marsadm log-delete-all all` or `cron` may continue to report an old split brain which has gone in the meantime.

- Check whether `/mars/` is too full. For a rough impression, `df /mars/` may be used. For getting authoritative values as internally used by the MARS emergency-mode computations, use `marsadm view-rest-space` (the unit is GiB). In practice, the differences are only marginal, at least on bigger `/mars/` partitions. When there is only few rest space (or none at all), please obey the instructions in section B.3.

B.3. Resolution of Emergency Mode

Emergency mode occurs when `/mars/` runs out of space, such that no new logfile data can be written anymore.

In emergency mode, the primary will write any write requests *directly* to the underlying disk, as if MARS were not present at all. Thus, your application will continue to run. Only the *replication* as such is stopped.

Notice: emergency mode means that your secondary nodes are usually in a *consistent*, but *outdated* state (exception: when a sync was running in parallel to the emergency mode, then the sync will be automatically started over again). You can check consistency via `marsadm view-flags all`. Only when a local disk shows a lower-case letter "d" instead of an uppercase "D", it is known to be inconsistent (e.g. during a sync). When there is a dash instead, it

usually means that the disk is detatched or misconfigured or the kernel module is not started. Please fix these problems first before believing that your local disk is unusable. Even if it is really inconsistent (which is very unlikely, typically occurring only as a consequence of hardware failures, or of the above-mentioned exception), you have a big chance to recover most of the data via `fsck` and friends.

A currently existing Emergency mode can be detected by

```
primary:~# marsadm view-is-emergency all
secondary:~# marsadm view-is-emergency all
```

Notice: this delivers the current state, telling nothing about the past.

Currently, emergency mode will also show something like `WARNING: SPLIT BRAIN at '' detected`. This ambiguity will be resolved in a future MARS release. It is however not crucial: the resolution methods for both cases are very similar. If in doubt, start emergency resolution first, and only proceed to split brain resoultion if it did not help.

Preconditions:

- Only current version of MARS: the space at the primary side should have been already released, and the emergency mode should have been already left. Otherwise, you might need the split-brain resolution method from section B.4.
- The network **must** be working. Check that the following gives an entry for each secondary:

  ```
  primary:~# netstat --tcp | grep 7777
  ```

  When necessary, fix the network first (see instructions above).

Emergency mode should now be resolved via the following instructions:

```
primary:~# marsadm view-is-emergency all
primary:~# du -s /mars/resource-* | sort -n
```

Remember the affected resources. Best practice is to do the following, starting with the *biggest* resource as shown by the `du | sort` output in reverse order, but *starting* the following only with the *affected* resources in the first place:

```
secondary1:~# marsadm invalidate <res1>
secondary1:~# marsadm log-delete-all all
... dito with all resources showing emergency mode
... dito on all other secondaries
primary:~# marsadm log-delete-all all
```

Hint: during the resolution process, some other resources might have gone into emergency mode concurrently. In addition, it is possible that some secondaries are stuck at particular resources while the corresponding primary has *not yet* entered emergency mode. Please repeat the steps in such a case, and look for emergency modes at secondaries additionally. When necessary, extend your list of *affected* resources.

Hint: be patient. Deleting large bulks of logfile data may take a long time, at least on highly loaded systems. You should give the cleanup processes at least 5 minutes before concluding that an `invalidate` followed by `log-delete-all` had no effect! Don't forget to give the `log-delete-all` at all cluster nodes, even when seemingly unaffected.

In very complex scenarios, when the primary roles of different resources are spread over diffent hosts (aka mixed operation), you may need to repeat the whole cycle iteratively for a few cycles until the jam is resolved.

If it does not go away, you have another chance by the following split-brain resolution process, which will also cleanup emergency mode as a side effect.

B.4. Resolution of Split Brain and of Emergency Mode

Hint: in many cases (but not guaranteed), the previous receipe for resolution of emergency mode will also cleanup split brain. Good chances are in case of $k = 2$ total replicas. Please collect your own experiences which method works better for you!

Precondition: the network must be working. Check that the following gives an entry for each secondary:

```
primary:~# netstat --tcp | grep 7777
```

When necessary, fix the network first (see instructions above).

Inspect the split brain situation:

```
primary:~# marsadm view all
primary:~# du -s /mars/resource-* | sort -n
```

Remember those resources where a message like `WARNING: SPLIT BRAIN at '' detected` appears. Do the following only for *affected* resources, starting with the biggest one (before proceeding to the next one).

Do the following with only *one* resource at a time (before proceeding to the next one), and repeat the actions on that resource at every secondary (if there are multiple secondaries):

```
secondary1:~# marsadm leave-resource $res1
secondary1:~# marsadm log-delete-all all
```

Check whether the split brain has vanished everywhere. Startover with other resources at their secondaries when necessary.

Finally, when no split brain is reported at any (former) secondary, do the following on the primary:

```
primary:~# marsadm log-delete-all all
primary:~# sleep 30
primary:~# marsadm view all
```

Now, the split brain should be gone even at the primary. If not, repeat this step.

In case even this should fail on some `$res` (which is very unlikely), read the PDF manual before using `marsadm log-purge-all $res`.

Finally, when the split brain is gone everywhere, rebuild the redundancy at every secondary via

```
secondary1:~# marsadm join-resource $res1 /dev/<lv-x>/$res1
```

If even this method does not help, setup the whole cluster afresh by `rmmod mars` everywhere, and creating a fresh `/mars/` filesystem everywhere, followed by the same procedure as installing MARS for the first time (which is outside the scope of this handout).

B.5. Handover of Primary Role

When there exists a method for primary handover in higher layers such as cluster managers, please prefer that method (e.g. `cm3` or other tools).

If suchalike doesn't work, or if you need to handover some resource `$res1` by hand, do the following:

- Stop the load / application corresponding to `$res1` on the old primary side.
- `umount /dev/mars/$res1`, or otherwise close any openers such as iSCSI.
- At the new primary: `marsadm primary $res1`
- Restart the application at the new site (in reverse order to above). In case you want to switch *all* resources which are not yet at the new side, you may use `marsadm primary all`.

B.6. Emergency Switching of Primary Role

Emergency switching is necessary when your primary is no longer reachable over the network for a *longer* time, or when the hardware is defective.

Emergency switching will very often lead to a split brain, which requires lots of manual actions to resolve (see above). Therefore, try to avoid emergency switching when possible!

Hint: MARS can automatically recover after a primary crash / reboot, as well as after secondary crashes, just by executing `modprobe mars` after `/mars/` had been mounted. Please consider to wait until your system comes up again, instead of risking a split brain.

The decision between emergency switching and continuing operation at the same primary side is an operational one. MARS can support your decision by the following information at the potentially new primary side (which was in secondary mode before):

```
istore-test-bap1:~# marsadm view all
---------- resource lv-0
lv-0 InConsistent Syncing dcAsFr Secondary istore-test-bs1
syncing: [====>.............] 27.84% (567/2048)MiB rate: 72583.00 KiB/sec remainin
> sync: 567.293/2048 MiB rate: 72583 KiB/sec remaining: 00:00:20 hrs
replaying: [>:::::::::::::::::::] 0.00% (0/12902)KiB logs: [1..1]
> fetch: 0 B rate: 38 KiB/s remaining: 00:00:00
> replay: 12902.047 KiB rate: 0 B/s remaining: --:--:--
```

When your target is syncing (like in this example), you cannot switch to it (same as with DRBD). When you had an emergency mode before, you should first resolve that (whenever possible). When a split brain is reported, try to resolve it first (same as with DRBD). Only in case you *know* that the primary is really damaged, or it is really impossible to the run the application there for some reason, emergency switching is desirable.

Hint: in case the secondary is inconsistent for some reason, e.g. because of an incremental fast full-sync, you have a last chance to recover most data after forceful switching by using a filesystem check or suchalike. This might be even faster than restoring data from the backup. But use it only if you are *really* desperate!

The amount of data which is *known* to be missing at your secondary is shown after the `> fetch:` in human-readable form. However, in cases of networking problems this information may be outdated. You *always* need to consider further facts which cannot be known by MARS.

When there exists a method for emergency switching of the primary in higher layers such as cluster managers, please prefer that method in front of the following one.

If suchalike doesn't work, or when a handover attempt has failed several times, or if you *really need* forceful switching of some resource `$res1` by hand, you can do the following:

- When possible, stop the load / application corresponding to `$res1` on the old primary side.
- When possible, `umount /dev/mars/$res1`, or otherwise close any openers such as iSCSI.
- When possible (if you have some time), wait until as much data has been propagated to the new primary as possible (watch the `fetch:` indicator).
- At the new primary: `marsadm disconnect $res1; marsadm primary --force $res1`
- Restart the application at the new site (in reverse order to above).
- After the application is known to run reliably, check for split brains and cleanup them when necessary.

C. Alternative Methods for Split Brain Resolution

Instead of `marsadm invalidate`, the following steps may be used. In preference, start with the old "wrong" primaries first:

1. `marsadm leave-resource mydata`
2. After having done this on one cluster node, check whether the split brain is already gone (e.g. by saying `marsadm view mydata`). There are chances that you don't need this on all of your nodes. Only in very rare[1] cases, it might happen that the preceding `leave-resource` operations were not able to clean up all logfiles produced in parallel by the split brain situation.
3. Read the documentation about `log-purge-all` (see page 99) and use it.
4. If you want to restore redundancy, you can follow-up a `join-resource` phase to the old resource name (using the correct device name, double-check it!) This will restore your redundancy by overwriting your bad split brain version with the correct one.

It is important to resolve the split brain *before* you can start the `join-resource` reconstruction phase! In order to keep as many "good" versions as possible (e.g. for emergency cases), don't re-join them all in parallel, but rather start with the oldest / most outdated / worst / inconsistent version first. It is recommended to start the next one only when the previous one has sucessfully finished.

[1] When your network had partitioned in a very awkward way for a long time, and when your partitioned primaries did several `log-rotate` operations indendently from each other, there is a small chance that `leave-resource` does not clean up *all* remains of such an awkward situation. Only in such a case, try `log-purge-all`.

D. Alternative De- and Reconstruction of a Damaged Resource

In case `leave-resource --host=` does not work, you may use the following fallback. On the surviving new designated primary, give the following commands:

1. `marsadm disconnect-all mydata`
2. `marsadm down mydata`
3. Check by hand whether your local disk is consistent, e.g. by test-mounting it readonly, `fsck`, etc.
4. `marsadm delete-resource mydata`
5. Check whether the other vital cluster nodes don't report the dead resource any more, e.g. `marsadm view all` at *each* of them. In case the resource has not disappeared anywhere (which may happen during network problems), do the `down ; delete-resource` steps also there (optionally again with `--force`).
6. Be sure that the resource has disappeared *everywhere*. When necessary, repeat the `delete-resource` with `--force`.
7. `marsadm create-resource newmydata ...` at the *correct* node using the *correct* disk device containing the *correct* version, and further steps to setup your resource from scratch, preferably under a different name to minimize any risk.

In any case, **manually check** whether a split brain is reported for any resource on any of your *surviving* cluster nodes. If you find one there (and only then), please (re-)execute the split brain resolution steps on the affected node(s).

E. Cleanup in case of Complicated Cascading Failures

MARS does its best to recover even from multiple failures (e.g. **rolling disasters**). Chances are high that the instructions from sections 3.4.3 3.4.4 or appendix C D will work even in case of multiple failures, such as a network failure plus local node failure at only 1 node (even if that node is the former primary node).

However, in general (e.g. when more than 1 node is damaged and/or when the filesystem `/mars/` is badly damaged) there is no general guarantee that recovery will *always* succeed under *any* (weird) circumstances. That said, your chances for recovery are *very* high when some disk remains usable at least at one of your surviving secondaries.

It should be very hard to finally trash a secondary, because the transaction logfiles are containing `md5` checksums for all data records. Any attempt to replay currupted logfiles is refused by MARS. In addition, the sequence numbers of `log-rotate`d logfiles are checked for contiguity. Finally, the *sequence path* of logfile applications (consisting of logfile names plus their respective length) is additionally secured by a `git`-like incremental checksum over the whole path history (so-called "version links"). This should detect split brains even if logfiles are appended / modified *after* a (forceful) switchover has already taken place.

That said, your risk of final data loss is very high if you remove the **BBU** from your hardware RAID controller before all hot data has been flushed to the physical disks. Therefore, never try to "repair" a seemingly dead node before your replication is up again somewhere else! Only unplug the network cables when advised, but never try to repair the hardware instantly!

In case of desperate situations where none of the previous instructions have succeeded, your last chance is rebuilding all your resources from intact disks as follows:

1. Do `rmmod mars` on all your cluster nodes and/or reboot them. Note: if you are less desperate, chances are high that the following will also work when the kernel module remains active and everywhere a `marsadm down` is given instead, but for an *ultimate* instruction you should eliminate *potential* kernel problems by `rmmod` / `reboot`, at least if you can afford the downtime on concurrently operating resources.

2. For safety, physically remove the storage network cables on *all* your cluster nodes. Note: the same disclaimer holds. MARS really does its best, even when `delete-resource` is given while the network is fully active and multiple split-brain primaries are actively using their local device in parallel (approved by some testcases from the automatic test suite, but note that it is impossible to catch all possible failure scenarios). Don't challenge your fate if you are desperate! Don't *rely* on this! Nothing is absolutely fail-safe!

3. **Manually** check which surviving disk is usable, and which is the "best" one for your purpose.

4. Do `modprobe mars` *only* on that node. If that fails, `rmmod` and/or reboot again, and start over with a completely fresh `/mars/` partition (`mkfs.ext4 /mars/` or similar) *everywhere* on *all* cluster nodes, and continue with step 7.

5. If your old `/mars/` works, and you did not already (forcefully) switch your designated primary to the final destination, do it now (see description in section 3.4.2.2). Wait until any old logfile data has been replayed.

6. Say `marsadm delete-resource mydata --force`. This will cleanup all internal symlink tree information for the resource, but will leave your disk data intact.

7. Locally build up the new resource(s) as usual, out of the underlying disks.
8. Check whether the new resource(s) work in standalone mode.
9. When necessary, repeat these steps with other resources.

Now you can choose how the rebuild your cluster. If you rebuilt `/mars/` anywhere, you *must* rebuild it on *all* new cluster nodes and start over with a fresh `join-cluster` on each of them, from scratch. It is not possible to mix the old cluster with the new one.

10. Finally, do all the necessary `join-resource`s on the respective cluster nodes, according to your new redundancy scenario after the failures (e.g. after activating spare nodes, etc). If you have $k > 2$ replicas, start `join-resource` on the worst / most damaged version first, and start the next preferably only after the previous sync has completed successfully. This way, you will be permanently retaining some (old and outdated, but hopefully potentially usable) replicas while a sync is running. Don't start too many syncs in parallel.

 Never use `delete-resource` twice on the same resource name, after you have already a working standalone primary[1]. You might accidentally destroy your again-working copy! You *can* issue `delete-resource` multiple times on different nodes, e.g. when the network has problems, but doing so *after* re-establishment of the initial primary bears some risk. Therefore, the safest way is first deleting the resources everywhere, and then starting over afresh.

Before re-connecting any network cable on any non-primary (new secondaries), ensure that all `/dev/mars/mydata` devices are no longer in use (e.g. from an old primary role before the incident happened), and that each local disk is detached. Only after that, you should be able to safely re-connect the network. The `delete-resource` given at the new primary should propagate now to each of your secondaries, and your local disk should be usable for a re-`join-resource`.

 When you did not rebuild your cluster from scratch with fresh `/mars/` filesystems, and one of the old cluster nodes is supposed to be removed permanently, use `leave-resource` (optionally with `--host=` and/or `--force`) and finally `leave-cluster`.

[1] Of course, when you don't have created the *same* resource anew, you may repeat `delete-resource` on other cluster nodes in order to get rid of local files / symlinks which had not been propagated to other nodes before.

F. Experts only: Special Trick Switching and Rebuild

The following is a further alternative for **experts** who really know what they are doing. The method is very simple and therefore well-suited for coping with mass failures, e.g. **power blackout of whole datacenters**.

In case a primary datacenter fails as a whole for whatever reason and you have a backup datacenter, do the following steps in the backup datacenter:

1. Fencing step: by means of firewalling, **ensure** that the (virtually) damaged datacenter nodes **cannot** be reached over the network. For example, you may place REJECT rules into all of your local iptables firewalls at the backup datacenter. Alternatively / additionally, you may block the routes at the appropriate central router(s) in your network.

2. Run the sequence `marsadm disconnect all; marsadm primary --force all` on all nodes in the backup datacenter.

3. Restart your services in the backup datacenter (as far as necessary). Depending on your network setup, further steps like switching BGP routes etc may be necessary.

4. Check that *all* your services are *really* up and running, before you try to repair anything! Failing to do so may result in data loss when you execute the following restore method for *experts*.

Now your backup datacenter should continue servicing your clients. The final reconstruction of the originally primary datacenter works as follows:

1. At the damaged primary datacenter, ensure that nowhere the MARS kernel module is running. In case of a power blackout, you shouldn't have executed an automatic `modprobe mars` anywhere during reboot, so you should be already done when all your nodes are up again. In case some nodes had no reboot, execute `rmmod mars` everywhere. If `rmmod` refuses to run, you may need to umount the `/dev/mars/mydata` device first. When nothing else helps, you may just mass reboot your hanging nodes.

2. At the failed side, do `rm -rf /mars/resource-$mydata/` for all those resources which had been primary before the blackout. Do this *only* for those cases, otherwise you will need unnecessary `leave-resource`s or `invalidate`s later (e.g. when half of your nodes were already running at the surving side). In order to avoid unnecessary traffic, please do this only as far as really necessary. Don't remove any other directories. In particular, `/mars/ips/` *must* remain intact. In case you accidentally deleted them, or you had to re-create `/mars/` from scratch, try `rsync` with the correct options.

    Caution! before doing this, check that the corresponding directory exists at the backup datacenter, and that it is *really* healthy!

3. Un-Fencing: restore your network firewall / routes and check that they work (`ping` etc).

4. Do `modprobe mars` everywhere. All missing directories and their missing symlinks should be automatically fetched from the backup datacenter.

5. Run `marsadm join-resource $res`, but only at those places where the directory was removed previously, while using the same disk devices as before. This will minimize actual traffic thanks to the fast full sync algorithm.

It is **crucial** that the fencing step **must** be executed *before* any `primary --force`! This way, no split brain will be *visible* at the backup datacenter side, because there is simply no chance for transferring different versions over the network. It is also crucial to remove any (potentially diverging) resource directories *before* the `modprobe`! This way, the backup datacenter never runs into split brain. This saves you a lot of detail work for split brain resolution when you have to restore bulks of nodes in a short time.

In case the repair of a full datacenter should take so extremely long that some `/mars/` partitions are about to run out of space at the surviving side, you may use the `leave-resource --host=failed-node` trick described earlier, followed by `log-delete-all`. Best if you have prepared a fully automatic script long before the incident, which executes suchalike only as far as necessary in each individual case.

Even better: train such scenarios in advance, and prepare scripts for mass automation. Look into section 5.3.

G. Mathematical Model of Architectural Reliability

The assumptions used in the model are explained in detail in section 1.6.1.2 on page 23. Here is a quick recap of the main parameters:

- n is the number of basic storage units. It is also used for the number of application units, assumed to be the same.
- k is the replication degree, or number of replicas. In general, you will have to deploy $N = k * n$ storage servers for getting n basic storage units. This applies to any of the competing architectures.
- s is the architecture-dependent spread exponent: it tells whether a storage incident will spread to the application units. Examples: $s = 0$ means that there is no spread between storage unit failures and application unit failures, other than a local 1:1 one. $s = 1$ means that an uncompensated storage node incident will cause n application incidents.
- p is the probability of a storage server incident. In the examples at section 1.6 on page 22, a fixed $p = 0.0001$ was used for easy understanding, but the following formulae should also hold for any other $p \in (0, 1)$.
- T is the observational period, introduced for convenience of understanding. The following can also be computed independently from any T, as long as the probability p does not change over time, which is assumed. Because T is only here for convenience of understanding, we set it to $T = 1/p$. In the examples from section 1.6.1.2 on page 23, a fixed $T = 10,000$ hours was used.

G.1. Formula for DRBD / MARS

We need not discrimiate between a storage failure probability S and an application failure probability A because applications are run locally at the storage servers 1:1. The probability for failure of a single shard consisting of k nodes is

$$A_p(k) = p^k$$

because all k shard members have to be down all at the same time. In section 1.6.1.2 on page 23 we assumed that there is no cross-communication between shards. Therefore they are completely independent from each other, and the total downtime of n shards during the observational period T is

$$A_{p,T}(k, n) = T * n * p^k$$

When introducing the spread exponent s, the formula turns into

$$A_{s,p,T}(k, n) = T * n^{s+1} * p^k$$

G.2. Formula for Unweighted BigCluster

This is based on the Bernoulli formula. The probability that exactly $\bar{k}$ storage nodes out of $N = k * n$ total storage nodes are down is

$$\bar{S}_p(\bar{k}, N) = \binom{N}{\bar{k}} * p^{\bar{k}} * (1 - p)^{N-\bar{k}}$$

Similarly, the probability for getting k or more storage node failures (up to N) at the same time is

$$S_p(k,N) = \sum_{\bar{k}=k}^{N} \bar{S}_p(\bar{k},N) = \sum_{\bar{k}=k}^{N} \binom{N}{\bar{k}} * p^{\bar{k}} * (1-p)^{N-\bar{k}}$$

By replacing N with $k*n$ (for conversion of the x axis into basic storage units) and by introducing T we get

$$S_{p,T}(k,n) = T * \sum_{\bar{k}=k}^{k*n} \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

For comparability with DRBDorMARS, we have to compute the application downtime A instead of the storage downtime S, which depends on the spread exponent s as follows:

$$A_{s,p,T}(k,n) = n^{s+1} * S_{p,T}(k,n) = n^{s+1} * T * \sum_{\bar{k}=k}^{k*n} \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

Notice that at $s = 0$ we have introduced a factor of n, which corresponds to the hashing effect (teardown of n application instances by a single uncompensated storage incident) as described in section 1.6.1.2 on page 23.

G.3. Formula for SizeWeighted BigCluster

In difference to above, we need to introduce a correction factor by the fraction of affected objects, relative to basic storage units. Otherwise the y axis would not stay comparable due to different units.

For the special case of $k = 1$, there is no difference to above.

For the special case of $k = 2$ replica, the correction factor is $1/(N-1)$, because we assume that all the replica of the affected first node are uniformly spread to all other nodes, which is $N-1$. The probability for hitting the intersection of the first node with the second node is thus $1/(N-1)$.

For higher values of k, and with a similar argument (never put another replica of the same object onto the same storage node) we get the correction factor as

$$C(k,N) = \prod_{l=1}^{k-1} \frac{1}{N-l}$$

Hint: there are maximum k physical replicas on the disks. For higher values of $\bar{k} \geq k$, there are $\binom{\bar{k}}{k}$ combinations of object intersections (when assuming that the number of objects on a node is very large such and no further object repetition can occur execpt for the k-fold replica placement). Thus the generalization to $\bar{k} \geq k$ is

$$C(k,\bar{k},N) = \binom{\bar{k}}{k} \prod_{l=1}^{k-1} \frac{1}{N-l}$$

By inserting this into the above fomula, we get

$$A_{s,p,T}(k,n) = n^{s+1} * T * \sum_{\bar{k}=k}^{k*n} C(k,\bar{k},k*n) * \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

H. Command Documentation for Userspace Tools

H.1. marsadm --help

```
Thorough documentation is in mars-manual.pdf. Please use the PDF manual
as authoritative reference! Here is only a short summary of the most
important sub-commands / options:

marsadm [<global_options>] <command> [<resource_name> | all | <args> ]
marsadm [<global_options>] view[-<macroname>] [<resource_name> | all ]

<global_option> =
  --force
    Skip safety checks.
    Use this only when you really know what you are doing!
    Warning! This is dangerous! First try --dry-run.
    Not combinable with 'all'.
  --dry-run
    Don't modify the symlink tree, but tell what would be done.
    Use this before starting potentially harmful actions such as
    'delete-resource'.
  --verbose
    Increase speakyness of some commands.
  --logger=/path/to/usr/bin/logger
    Use an alternative syslog messenger.
    When empty, disable syslogging.
  --max-deletions=<number>
    When your network or your firewall rules are defective over a
    longer time, too many deletion links may accumulate at
    /mars/todo-global/delete-* and sibling locations.
    This limit is preventing overflow of the filesystem as well
    as overloading the worker threads.
  --thresh-logfiles=<number>
  --thresh-logsize=<number>
    Prevention of too many small logfiles when secondaries are not
    catching up. When more than thresh-logfiles are already present,
    the next one is only created when the last one has at least
    size thresh-logsize (in units of GB).
  --timeout=<seconds>
    Abort safety checks after timeout with an error.
    When giving 'all' as resource agument, this works for each
    resource independently.
  --window=<seconds>
    Treat other cluster nodes as healthy when some communcation has
    occured during the given time window.
  --threshold=<bytes>
    Some macros like 'fetch-threshold-reached' use this for determining
    their sloppyness.
  --host=<hostname>
    Act as if the command was running on cluster node <hostname>.
```

```
    Warning! This is dangerous! First try --dry-run
  --backup-dir=</absolute_path>
    Only for experts.
    Used by several special commands like merge-cluster, split-cluster
    etc for creating backups of important data.
  --ip=<ip>
    Override the IP address stored in the symlink tree, as well as
    the default IP determined from the list of network interfaces.
    Usually you will need this only at 'create-cluster' or
    'join-cluster' for resolving ambiguities.
  --ssh-port=<port_nr>
    Override the default ssh port (22) for ssh and rsync.
    Useful for running {join,merge}-cluster on non-standard ssh ports.
  --ssh-opts="<ssh_commandline_options>"
    Override the default ssh commandline options. Also used for rsync.
  --macro=<text>
    Handy for testing short macro evaluations at the command line.

<command> =
  attach
    usage: attach <resource_name>
    Attaches the local disk (backing block device) to the resource.
    The disk must have been previously configured at
    {create,join}-resource.
    When designated as a primary, /dev/mars/$res will also appear.
    This does not change the state of {fetch,replay}.
    For a complete local startup of the resource, use 'marsadm up'.

  cat
    usage: cat <path>
    Print internal debug output in human readable form.
    Numerical timestamps and numerical error codes are replaced
    by more readable means.
    Example: marsadm cat /mars/5.total.status

  connect
    usage: connect <resource_name>
    See resume-fetch-local.

  connect-global
    usage: connect-global <resource_name>
    Like resume-fetch-local, but affects all resource members
    in the cluster (remotely).

  connect-local
    usage: connect-local <resource_name>
    See resume-fetch-local.

  create-cluster
    usage: create-cluster (no parameters)
    This must be called exactly once when creating a new cluster.
    Don't call this again! Use join-cluster on the secondary nodes.
    Please read the PDF manual for details.

  create-resource
    usage: create-resource <resource_name> </dev/lv/mydata>
    (further syntax variants are described in the PDF manual).
    Create a new resource out of a pre-existing disk (backing
```

```
    block device) /dev/lv/mydata (or similar).
    The current node will start in primary role, thus
    /dev/mars/<resource_name> will appear after a short time, initially
    showing the same contents as the underlying disk /dev/lv/mydata.
    It is good practice to name the resource <resource_name> and the
    disk name identical.

  cron
    usage: cron (no parameters)
    Do all necessary regular housekeeping tasks.
    This is equivalent to log-rotate all; sleep 5; log-delete-all all.

  delete-resource
    usage: delete-resource <resource_name>
    CAUTION! This is dangerous when the network is somehow
    interrupted, or when damaged nodes are later re-surrected
    in any way.

    Precondition: the resource must no longer have any members
    (see leave-resource).
    This is only needed when you _insist_ on re-using a damaged
    resource for re-creating a new one with exactly the same
    old <resource_name>.
    HINT: best practice is to not use this, but just create a _new_
    resource with a new <resource_name> out of your local disks.
    Please read the PDF manual on potential consequences.

  detach
    usage: detach <resource_name>
    Detaches the local disk (backing block device) from the
    MARS resource.
    Caution! you may read data from the local disk afterwards,
    but ensure that no data is written to it!
    Otherwise, you are likely to produce harmful inconsistencies.
    When running in primary role, /dev/mars/$res will also disappear.
    This does not change the state of {fetch,replay}.
    For a complete local shutdown of the resource, use 'marsadm down'.

  disconnect
    usage: disconnect <resource_name>
    See pause-fetch-local.

  disconnect-global
    usage: disconnect-global <resource_name>
    Like pause-fetch-local, but affects all resource members
    in the cluster (remotely).

  disconnect-local
    usage: disconnect-local <resource_name>
    See pause-fetch-local.

  down
    usage: down <resource_name>
    Shortcut for detach + pause-sync + pause-fetch + pause-replay.

  get-emergency-limit
    usage: get-emergency-limit <resource_name>
    Counterpart of set-emergency-limit (per-resource emergency limit)
```

```
get-sync-limit-value
  usage: get-sync-limit-value (no parameters)
  For retrieval of the value set by set-sync-limit-value.

get-systemd-unit
  usage: get-systemd-unit <resource_name>
  Show the system units (for start and stop), or empty when unset.

invalidate
  usage: invalidate <resource_name>
  Only useful on a secondary node.
  Forces MARS to consider the local replica disk as being
  inconsistent, and therefore starting a fast full-sync from
  the currently designated primary node (which must exist;
  therefore avoid the 'secondary' command).
  This is usually needed for resolving emergency mode.
  When having k=2 replicas, this can be also used for
  quick-and-simple split-brain resolution.
  In other cases, or when the split-brain is not resolved by
  this command, please use the 'leave-resource' / 'join-resource'
  method as described in the PDF manual (in the right order as
  described there).

join-cluster
  usage: join-cluster <hostname_of_primary>
  Establishes a new cluster membership.
  This must be called once on any new cluster member.
  This is a prerequisite for join-resource.

join-resource
  usage: join-resource <resource_name> </dev/lv/mydata>
  (further syntax variants are described in the PDF manual).
  The resource <resource_name> must have been already created on
  another cluster node, and the network must be healthy.
  The contents of the local replica disk /dev/lv/mydata will be
  overwritten by the initial fast full sync from the currently
  designated primary node.
  After the initial full sync has finished, the current host will
  act in secondary role.
  For details on size constraints etc, refer to the PDF manual.

leave-cluster
  usage: leave-cluster (no parameters)
  This can be used for final deconstruction of a cluster member.
  Prior to this, all resources must have been left
  via leave-resource.
  Notice: this will never destroy the cluster UID on the /mars/
  filesystem.
  Please read the PDF manual for details.

leave-resource
  usage: leave-resource <resource_name>
  Precondition: the local host must be in secondary role.
  Stop being a member of the resource, and thus stop all
  replication activities. The status of the underlying disk
  will remain in its current state (whatever it is).
```

```
log-delete
  usage: log-delete <resource_name>
  When possible, globally delete all old transaction logfiles which
  are known to be superflous, i.e. all secondaries no longer need
  to replay them.
  This must be regularly called by a cron job or similar, in order
  to prevent overflow of the /mars/ directory.
  For regular maintainance cron jobs, please prefer 'marsadm cron'.
  For details and best practices, please refer to the PDF manual.

log-delete-all
  usage: log-delete-all <resource_name>
  Alias for log-delete

log-delete-one
  usage: log-delete-one <resource_name>
  When possible, globally delete at most one old transaction logfile
  which is known to be superfluous, i.e. all secondaries no longer
  need to replay it.
  Hint: use this only for testing and manual inspection.
  For regular maintainance cron jobs, please prefer cron
  or log-delete-all.

log-purge-all
  usage: log-purge-all <resource_name>
  This is potentially dangerous.
  Use this only if you are really desperate in trying to resolve a
  split brain. Use this only after reading the PDF manual!

log-rotate
  usage: log-rotate <resource_name>
  Only useful at the primary side.
  Start writing transaction logs into a new transaction logfile.
  This should be regularly called by a cron job or similar.
  For regular maintainance cron jobs, please prefer 'marsadm cron'.
  For details and best practices, please refer to the PDF manual.

lowlevel-delete-host
  usage: lowlevel-delete-host <resource_name>
  Delete cluster member.

lowlevel-ls-host-ips
  usage: lowlevel-ls-host-ips <resource_name>
  List cluster member names and IP addresses.

lowlevel-set-host-ip
  usage: lowlevel-set-host-ip <resource_name>
  Set IP for host.

merge-cluster
  usage: merge-cluster <hostname_of_other_cluster>
  Precondition: the resource names of both clusters must be disjoint.
  Create the union of two clusters, consisting of the
  union of all machines, and the union of all resources.
  The members of each resource are _not_ changed by this.
  This is useful for creating a big "virtual LVM cluster" where
  resources can be almost arbitrarily migrated between machines via
  later join-resource / leave-resource operations.
```

```
merge-cluster-check
  usage: merge-cluster-check <hostname_of_other_cluster>
  Check whether the resources of both clusters are disjoint.
  Useful for checking in advance whether merge-cluster would be
  possible.

merge-cluster-list
  usage: merge-cluster-list
  Determine the local list of resources.
  Useful for checking or analysis of merge-cluster disjointness by hand.

pause-fetch
  usage: pause-fetch <resource_name>
  See pause-fetch-local.

pause-fetch-global
  usage: pause-fetch-global <resource_name>
  Like pause-fetch-local, but affects all resource members
  in the cluster (remotely).

pause-fetch-local
  usage: pause-fetch-local <resource_name>
  Stop fetching transaction logfiles from the current
  designated primary.
  This is independent from any {pause,resume}-replay operations.
  Only useful on a secondary node.

pause-replay
  usage: pause-replay <resource_name>
  See pause-replay-local.

pause-replay-global
  usage: pause-replay-global <resource_name>
  Like pause-replay-local, but affects all resource members
  in the cluster (remotely).

pause-replay-local
  usage: pause-replay-local <resource_name>
  Stop replaying transaction logfiles for now.
  This is independent from any {pause,resume}-fetch operations.
  This may be used for freezing the state of your replica for some
  time, if you have enough space on /mars/.
  Only useful on a secondary node.

pause-sync
  usage: pause-sync <resource_name>
  See pause-sync-local.

pause-sync-global
  usage: pause-sync-global <resource_name>
  Like pause-sync-local, but affects all resource members
  in the cluster (remotely).

pause-sync-local
  usage: pause-sync-local <resource_name>
  Pause the initial data sync at current stage.
  This has only an effect if a sync is actually running (i.e.
```

```
    there is something to be actually synced).
    Don't pause too long, because the local replica will remain
    inconsistent during the pause.
    Use this only for limited reduction of system load.
    Only useful on a secondary node.

  primary
    usage: primary <resource_name>
    Promote the resource into primary role.
    This is necessary for /dev/mars/$res to appear on the local host.
    Notice: by concept there can be only _one_ designated primary
    in a cluster at the same time.
    The role change is automatically distributed to the other nodes
    in the cluster, provided that the network is healthy.
    The old primary node will _automatically_ go
    into secondary role first. This is different from DRBD!
    With MARS, you don't need an intermediate 'secondary' command
    for switching roles.
    It is usually better to _directly_ switch the primary roles
    between both hosts.
    When --force is not given, a planned handover is started:
    the local host will only become actually primary _after_ the
    old primary is gone, and all old transaction logs have been
    fetched and replayed at the new designated priamry.
    When --force is given, no handover is attempted. A a consequence,
    a split brain situation is likely to emerge.
    Thus, use --force only after an ordinary handover attempt has
    failed, and when you don't care about the split brain.
    For more details, please refer to the PDF manual.

  resize
    usage: resize <resource_name>
    Prerequisite: all underlying disks (usually /dev/vg/$res) must
    have been already increased, e.g. at the LVM layer (cf. lvresize).
    Causes MARS to re-examine all sizing constraints on all members of
    the resource, and increase the global logical size of the resource
    accordingly.
    Shrinking is currently not yet implemented.
    When successful, /dev/mars/$res at the primary will be increased
    in size. In addition, all secondaries will start an incremental
    fast full-sync to get the enlarged parts from the primary.

  resume-fetch
    usage: resume-fetch <resource_name>
    See resume-fetch-local.

  resume-fetch-global
    usage: resume-fetch-global <resource_name>
    Like resume-fetch-local, but affects all resource members
    in the cluster (remotely).

  resume-fetch-local
    usage: resume-fetch-local <resource_name>
    Start fetching transaction logfiles from the current
    designated primary node, if there is one.
    This is independent from any {pause,resume}-replay operations.
    Only useful on a secondary node.
```

```
resume-replay
  usage: resume-replay <resource_name>
  See resume-replay-local.

resume-replay-global
  usage: resume-replay-global <resource_name>
  Like resume-replay-local, but affects all resource members
  in the cluster (remotely).

resume-replay-local
  usage: resume-replay-local <resource_name>
  Restart replaying transaction logfiles, when there is some
  data left.
  This is independent from any {pause,resume}-fetch operations.
  This should be used for unfreezing the state of your local replica.
  Only useful on a secondary node.

resume-sync
  usage: resume-sync <resource_name>
  See resume-sync-local.

resume-sync-global
  usage: resume-sync-global <resource_name>
  Like resume-sync-local, but affects all resource members
  in the cluster (remotely).

resume-sync-local
  usage: resume-sync-local <resource_name>
  Resume any initial / incremental data sync at the stage where it
  had been interrupted by pause-sync.
  Only useful on a secondary node.

secondary
  usage: secondary <resource_name>
  Promote all cluster members into secondary role, globally.
  In contrast to DRBD, this is not needed as an intermediate step
  for planned handover between an old and a new primary node.
  The only reasonable usage is before the last leave-resource of the
  last cluster member, immediately before leave-cluster is executed
  for final deconstruction of the cluster.
  In all other cases, please prefer 'primary' for direct handover
  between cluster nodes.
  Notice: 'secondary' sets the global designated primary node
  to '(none)' which in turn prevents the execution of 'invalidate'
  or 'join-resource' or 'resize' anywhere in the cluster.
  Therefore, don't unnecessarily give 'secondary'!

set-emergency-limit
  usage: set-emergency-limit <resource_name> <value>
  Set a per-resource emergency limit for disk space in /mars.
  See PDF manual for details.

set-sync-limit-value
  usage: set-sync-limit-value <new_value>
  Set the maximum number of resources which should by syncing
  concurrently.

set-systemd-unit
```

```
    usage: set-systemd-unit <resource_name> <start_unit_name> [<stop_unit_name>]
    This activates the systemd template engine of marsadm.
    Please read mars-manual.pdf on this.
    When <stop_unit_name> is omitted, it will be treated equal to
    <start_unit_name>.

  split-cluster
    usage: split-cluster (no parameters)
    NOT OFFICIALLY SUPPORTED - ONLY FOR EXPERTS.
    RTFS = Read The Fucking Sourcecode.
    Use this only if you know what you are doing.

  up
    usage: up <resource_name>
    Shortcut for attach + resume-sync + resume-fetch + resume-replay.

  wait-cluster
    usage: wait-resource [<resource_name>]
    Waits until a ping-pong communication has succeeded in the
    whole cluster (or only the members of <resource_name>).
    NOTICE: this is extremely useful for avoiding races when scripting
    in a cluster.

  wait-connect
    usage: wait-connect [<resource_name>]
    See wait-cluster.

  wait-resource
    usage: wait-resource <resource_name>
                         [[attach|fetch|replay|sync][-on|-off]]
    Wait until the given condition is met on the resource, locally.

  wait-umount
    usage: wait-umount <resource_name>
    Wait until /dev/mars/<resource_name> has disappeared in the
    cluster (even remotely).
    Useful on both primary and secondary nodes.

<resource_name> = name of resource or "all" for all resources


<macroname> = <complex_macroname> | <primitive_macroname>

<complex_macroname> =
  1and1
  comminfo
  commstate
  cstate
  default
  default-global
  diskstate
  diskstate-1and1
  dstate
  fetch-line
  fetch-line-1and1
  flags
  flags-1and1
  outdated-flags
```

```
  outdated-flags-1and1
  primarynode
  primarynode-1and1
  replay-line
  replay-line-1and1
  replinfo
  replinfo-1and1
  replstate
  replstate-1and1
  resource-errors
  resource-errors-1and1
  role
  role-1and1
  state
  status
  sync-line
  sync-line-1and1
  syncinfo
  syncinfo-1and1
  todo-role


<primitive_macroname> =
  deletable-size
  device-opened
  errno-text
    Convert errno numbers (positive or negative) into human readable text.
  get-log-status
  get-resource-{fat,err,wrn}{,-count}
  get-{disk,device}
  is-{alive}
  is-{split-brain,consistent,emergency,orphan}
  occupied-size
  present-{disk,device}
    (deprecated, use *-present instead)
  replay-basenr
  replay-code
    When negative, this indidates that a replay/recovery error has occurred.
  rest-space
  summary-vector
  systemd-unit
  tree
  uuid
  wait-{is,todo}-{attach,sync,fetch,replay,primary}-{on,off}
  {alive,fetch,replay,work}-{timestamp,age,lag}
  {all,the}-{pretty-,}{global-,}{{err,wrn,inf}-,}msg
  {cluster,resource}-members
  {disk,device}-present
  {disk,resource,device}-size
  {fetch,replay,work}-{lognr,logcount}
  {get,actual}-primary
  {is,todo}-{attach,sync,fetch,replay,primary}
  {my,all}-resources
  {sync,fetch,replay,work,syncpos}-{size,pos}
  {sync,fetch,replay,work}-{rest,{almost-,threshold-,}reached,percent,permille,vector}
  {sync,fetch,replay}-{rate,remain}
  {time,real-time}
```

H.2. football.sh --help

```
Usage:
  ./football.sh --help [--verbose]
     Show help
  ./football.sh --variable=<value>
     Override any shell variable

Actions for resource migration:

  ./football.sh migrate         <resource> <target_primary> [<target_secondary>]
     Run the sequence
     migrate_prepare ; migrate_wait ; migrate_finish; migrate_cleanup.

  ./football.sh migrate_prepare <resource> <target_primary> [<target_secondary>]
     Allocate LVM space at the targets and start MARS replication.

  ./football.sh migrate_wait    <resource> <target_primary> [<target_secondary>]
     Wait until MARS replication reports UpToDate.

  ./football.sh migrate_finish  <resource> <target_primary> [<target_secondary>]
     Call hooks for handover to the targets.

  ./football.sh migrate_cleanup <resource>
     Remove old / currently unused LV replicas from MARS and deallocate
     from LVM.

Actions for inplace FS shrinking:

  ./football.sh shrink          <resource> <percent>
     Run the sequence shrink_prepare ; shrink_finish ; shrink_cleanup.

  ./football.sh shrink_prepare  <resource> [<percent>]
     Allocate temporary LVM space (when possible) and create initial
     raw FS copy.
     Default percent value(when left out) is 85.

  ./football.sh shrink_finish   <resource>
     Incrementally update the FS copy, swap old <=> new copy with
     small downtime.

  ./football.sh shrink_cleanup  <resource>
     Remove old FS copy from LVM.

Actions for inplace FS extension:

  ./football.sh extend          <resource> <percent>

Combined actions:

  ./football.sh migrate+shrink <resource> <target_primary> [<target_secondary>] [<percent>]
     Similar to migrate ; shrink but produces less network traffic.
     Default percent value (when left out) is 85.

  ./football.sh migrate+shrink+back <resource> <tmp_primary> [<percent>]
     Migrate temporarily to <tmp_primary>, then shrink there,
     finally migrate back to old primary and secondaries.
     Default percent value (when left out) is 85.
```

```
Actions for (manual) repair in emergency situations:

  ./football.sh manual_migrate_config  <resource> <target_primary> [<target_secondary>]
     Transfer only the cluster config, without changing the MARS replicas.
     This does no resource stopping / restarting.
     Useful for reverting a failed migration.

  ./football.sh manual_config_update <hostname>
     Only update the cluster config, without changing anything else.
     Useful for manual repair of failed migration.

  ./football.sh manual_merge_cluster <hostname1> <hostname2>
     Run "marsadm merge-cluster" for the given hosts.
     Hostnames must be from different (former) clusters.

  ./football.sh manual_split_cluster <hostname_list>
     Run "marsadm split-cluster" at the given hosts.
     Useful for fixing failed / asymmetric splits.
     Hint: provide _all_ hostnames which have formerly participated
     in the cluster.

  ./football.sh repair_vm <resource> <primary_candidate_list>
     Try to restart the VM <resource> on one of the given machines.
     Useful during unexpected customer downtime.

  ./football.sh repair_mars <resource> <primary_candidate_list>
     Before restarting the VM like in repair_vm, try to find a local
     LV where a stand-alone MARS resource can be found and built up.
     Use this only when the MARS resources are gone, and when you are
     desperate. Problem: this will likely create a MARS setup which is
     not usable for production, and therefore must be corrected later
     by hand. Use this only during an emergency situation in order to
     get the customers online again, while buying the downsides of this
     command.

  ./football.sh manual_lock   <item> <host_list>
  ./football.sh manual_unlock <item> <host_list>
     Manually lock or unlock an item at all of the given hosts, in
     an atomic fashion. In most cases, use "ALL" for the item.

Global maintenance:

  ./football.sh lv_cleanup      <resource>

General features:

  - Instead of <percent>, an absolute amount of storage with suffix
    'k' or 'm' or 'g' can be given.

  - When <resource> is currently stopped, login to the container is
    not possible, and in turn the hypervisor node and primary storage node
    cannot be automatically determined. In such a case, the missing
    nodes can be specified via the syntax
        <resource>:<hypervisor>:<primary_storage>

  - The following LV suffixes are used (naming convention):
    -tmp = currently emerging version for shrinking
```

```
    -preshrink = old version before shrinking took place

  - By adding the option --screener, you can handover football execution
    to ./screener.sh .
    When some --enable_*_waiting is also added, then the critical
    sections involving customer downtime are temporarily halted until
    some sysadmins says "screener.sh continue $resource" or
    attaches to the sessions and presses the RETURN key.


PLUGIN football-1and1config

   1&1 specfic plugin for dealing with the cm3 clusters
   and its concrete configuration .

PLUGIN football-cm3

   1&1 specfic plugin for dealing with the cm3 cluster manager
   and its concrete operating enviroment (singleton instance).

   Current maximum cluster size limit:

   Maximum #syncs running before migration can start:

   Following marsadm --version must be installed:

   Following mars kernel modules must be loaded:


PLUGIN football-basic

   Generic driver for systemd-controlled MARS pools.
   The current version supports only a flat model:
   (1) There is a single "big cluster" at metadata level.
       All cluster members are joined via merge-cluster.
       All occurring names need to be globally unique.
   (2) The network uses BGP or other means, thus any hypervisor
       can (potentially) start any VM at any time.
   (3) iSCSI or remote devices are not supported for now
       (LocalSharding model). This may be extended in a future
       release.
   This plugin is exclusive-or with cm3.

Plugin specific actions:

   ./football.sh basic_add_host <hostname>
      Manually add another host to the hostname cache.


PLUGIN football-motd

  Generic plugin for motd. Communicate that Football is running
  at login via motd.


PLUGIN football-report

  Generic plugin for communication of reports.
```

```
PLUGIN football-waiting

  Generic plugig, interfacing with screener: when this is used
  by your script and enabled, then you will be able to wait for
  "screener.sh continue" operations at certain points in your
  script.
```

H.3. football.sh --help --verbose

```
verbose=1
Usage:
  ./football.sh --help [--verbose]
     Show help
  ./football.sh --variable=<value>
     Override any shell variable

Actions for resource migration:

  ./football.sh migrate         <resource> <target_primary> [<target_secondary>]
     Run the sequence
     migrate_prepare ; migrate_wait ; migrate_finish; migrate_cleanup.

  ./football.sh migrate_prepare <resource> <target_primary> [<target_secondary>]
     Allocate LVM space at the targets and start MARS replication.

  ./football.sh migrate_wait    <resource> <target_primary> [<target_secondary>]
     Wait until MARS replication reports UpToDate.

  ./football.sh migrate_finish  <resource> <target_primary> [<target_secondary>]
     Call hooks for handover to the targets.

  ./football.sh migrate_cleanup <resource>
     Remove old / currently unused LV replicas from MARS and deallocate
     from LVM.

Actions for inplace FS shrinking:

  ./football.sh shrink          <resource> <percent>
     Run the sequence shrink_prepare ; shrink_finish ; shrink_cleanup.

  ./football.sh shrink_prepare  <resource> [<percent>]
     Allocate temporary LVM space (when possible) and create initial
     raw FS copy.
     Default percent value(when left out) is 85.

  ./football.sh shrink_finish   <resource>
     Incrementally update the FS copy, swap old <=> new copy with
     small downtime.

  ./football.sh shrink_cleanup  <resource>
     Remove old FS copy from LVM.

Actions for inplace FS extension:

  ./football.sh extend          <resource> <percent>
```

```
Combined actions:

  ./football.sh migrate+shrink <resource> <target_primary> [<target_secondary>] [<percent>]
     Similar to migrate ; shrink but produces less network traffic.
     Default percent value (when left out) is 85.

  ./football.sh migrate+shrink+back <resource> <tmp_primary> [<percent>]
     Migrate temporarily to <tmp_primary>, then shrink there,
     finally migrate back to old primary and secondaries.
     Default percent value (when left out) is 85.

Actions for (manual) repair in emergency situations:

  ./football.sh manual_migrate_config  <resource> <target_primary> [<target_secondary>]
     Transfer only the cluster config, without changing the MARS replicas.
     This does no resource stopping / restarting.
     Useful for reverting a failed migration.

  ./football.sh manual_config_update <hostname>
     Only update the cluster config, without changing anything else.
     Useful for manual repair of failed migration.

  ./football.sh manual_merge_cluster <hostname1> <hostname2>
     Run "marsadm merge-cluster" for the given hosts.
     Hostnames must be from different (former) clusters.

  ./football.sh manual_split_cluster <hostname_list>
     Run "marsadm split-cluster" at the given hosts.
     Useful for fixing failed / asymmetric splits.
     Hint: provide _all_ hostnames which have formerly participated
     in the cluster.

  ./football.sh repair_vm <resource> <primary_candidate_list>
     Try to restart the VM <resource> on one of the given machines.
     Useful during unexpected customer downtime.

  ./football.sh repair_mars <resource> <primary_candidate_list>
     Before restarting the VM like in repair_vm, try to find a local
     LV where a stand-alone MARS resource can be found and built up.
     Use this only when the MARS resources are gone, and when you are
     desperate. Problem: this will likely create a MARS setup which is
     not usable for production, and therefore must be corrected later
     by hand. Use this only during an emergency situation in order to
     get the customers online again, while buying the downsides of this
     command.

  ./football.sh manual_lock   <item> <host_list>
  ./football.sh manual_unlock <item> <host_list>
     Manually lock or unlock an item at all of the given hosts, in
     an atomic fashion. In most cases, use "ALL" for the item.

Global maintenance:

  ./football.sh lv_cleanup      <resource>

General features:
```

```
- Instead of <percent>, an absolute amount of storage with suffix
  'k' or 'm' or 'g' can be given.

- When <resource> is currently stopped, login to the container is
  not possible, and in turn the hypervisor node and primary storage node
  cannot be automatically determined. In such a case, the missing
  nodes can be specified via the syntax
      <resource>:<hypervisor>:<primary_storage>

- The following LV suffixes are used (naming convention):
  -tmp = currently emerging version for shrinking
  -preshrink = old version before shrinking took place

- By adding the option --screener, you can handover football execution
  to ./screener.sh .
  When some --enable_*_waiting is also added, then the critical
  sections involving customer downtime are temporarily halted until
  some sysadmins says "screener.sh continue $resource" or
  attaches to the sessions and presses the RETURN key.

## football_includes
# List of directories where football-*.sh and football-*.conf
# files can be found.
football_includes="${football_includes:-/usr/lib/mars/plugins /etc/mars/plugins $script_dir/

## football_confs
# Another list of directories where football-*.conf files can be found.
# These are sourced in a second pass after $football_includes.
# Thus you can change this during the first pass.
football_confs="${football_confs:-/usr/lib/mars/confs /etc/mars/confs $script_dir/confs $HOM

## football_creds
# List of directories where various credential files can be found.
football_creds="${football_creds:-/usr/lib/mars/creds /etc/mars/creds $script_dir/creds $scr

## dry_run
# When set, actions are only simulated.
dry_run=${dry_run:-0}

## verbose
# increase speakiness.
verbose=${verbose:-0}

## confirm
# Only for debugging: manually started operations can be
# manually checked and confirmed before actually starting opersions.
confirm=${confirm:-1}

## force
# Normally, shrinking and extending will only be started if there
# is something to do.
# Enable this for debugging and testing: the check is then skipped.
force=${force:-0}

## debug_injection_point
# RTFS don't set this unless you are a developer knowing what you are doing.
debug_injection_point="${debug_injection_point:-0}"
```

```
## football_logdir
# Where the logfiles should be created.
# HINT: after playing Football in masses for a whiile, your $logdir will
# be easily populated with hundreds or thousands of logfiles.
# Set this to your convenience.
football_logdir="${football_logdir:-${logdir:-$HOME/football-logs}}"

## screener
# When enabled, handover execution to the screener.
# Very useful for running Football in masses.
screener="${screener:-0}"

## min_space
# When testing / debugging with extremely small LVs, it may happen
# that mkfs refuses to create extemely small filesystems.
# Use this to ensure a minimum size.
min_space="${min_space:-20000000}"

## cache_repeat_lapse
# When using the waiting capabilities of screener, and when waits
# are lasting very long, your dentry cache may become cold.
# Use this for repeated refreshes of the dentry cache after some time.
cache_repeat_lapse="${cache_repeat_lapse:-120}" # Minutes

## ssh_opt
# Useful for customization to your ssh environment.
ssh_opt="${ssh_opt:--4 -A -o StrictHostKeyChecking=no -o ForwardX11=no -o KbdInteractiveAuthentic

## rsync_opt
# The rsync options in general.
# IMPORTANT: some intermediate progress report is absolutely needed,
# because otherwise a false-positive TIMEOUT may be assumed when
# no output is generated for several hours.
rsync_opt="${rsync_opt:- -aSH --info=progress2,STATS}"

## rsync_opt_prepare
# Additional rsync options for preparation and updating
# of the temporary shrink mirror filesystem.
rsync_opt_prepare="${rsync_opt_prepare:---exclude='.filemon2' --delete}"

## rsync_nice
# Typically, the preparation steps are run with background priority.
rsync_nice="${rsync_nice:-nice -19}"

## rsync_repeat_prepare and rsync_repeat_hot
# Tuning: increases the reliability of rsync and ensures that the dentry cache
# remains hot.
rsync_repeat_prepare="${rsync_repeat_prepare:-5}"
rsync_repeat_hot="${rsync_repeat_hot:-3}"

## rsync_skip_lines
# Number of rsync lines to skip in output (avoid overflow of logfiles).
rsync_skip_lines="${rsync_skip_lines:-1000}"

## wait_timeout
# Avoid infinite loops upon waiting.
wait_timeout="${wait_timeout:-$(( 24 * 60 ))}" # Minutes
```

```
## lvremove_opt
# Some LVM versions are requiring this for unattended batch operations.
lvremove_opt="${lvremove_opt:--f}"

## critical_status
# This is the "magic" exit code indicating _criticality_
# of a failed command.
critical_status="${critical_status:-199}"

## serious_status
# This is the "magic" exit code indicating _seriosity_
# of a failed command.
serious_status="${serious_status:-198}"

## pre_hand or --pre-hand=
# Set this to do an ordinary handover to a new start position
# (in the source cluster) before doing anything else.
# This may be used for handover to a different datacenter,
# in order to minimize cross traffic between datacenters.
pre_hand="${pre_hand:-}"

## post_hand or --post-hand=
# Set this to do an ordinary handover to a final position
# (in the target cluster) after everything has successfully finished.
# This may be used to establish a uniform default running location.
post_hand="${post_hand:-}"

## lock_break_timeout
# When remote ssh commands are failing, remote locks may sustain forever.
# Avoid deadlocks by breaking remote locks after this timeout has elapsed.
# NOTICE: these type of locks are only intended for short-term locking.
lock_break_timeout="${lock_break_timeout:-3600}" # seconds

## startup_when_locked
# When == 0:
#  Don't abort and don't wait when a lock is detected at startup.
# When == 1 and when enable_startup_waiting=1:
#  Wait until the lock is gone.
# When == 2:
#  Abort start of script execution when a lock is detected.
#  Later, when a locks are set _during_ execution, they will
#  be obeyed when enable_*_waiting is set (instead), and will
#  lead to waits instead of aborts.
startup_when_locked="${startup_when_locked:-1}"

## user_name
# Normally automatically derived from ssh agent or from $LOGNAME.
# Please override this only when really necessary.
export user_name="${user_name:-$(get_real_ssh_user)}"
export user_name="${user_name:-$LOGNAME}"

## replace_ssh_id_file
# When set, replace current ssh user with this one.
# The new user should hot have a passphrase.
# Useful for logging out the original user (interrupting the original
# ssh agent chain).
replace_ssh_id_file="${replace_ssh_id_file:-}"
```

```
PLUGIN football-1and1config

   1&1 specfic plugin for dealing with the cm3 clusters
   and its concrete configuration .
```

H.4. screener.sh --help

```
./screener.sh: Run _unattended_ processes in screen sessions.
    Useful for MASS automation, running hundreds of unattended
    commands in parallel.
    HINT: for running more than ~500 sessions in parallel, you might need
    some system tuning (e.g. rlimits, kernel patches etc) for creating
    a huge number of file descritor / sockets / etc.
    ADVANTAGE: You may attach to individual screens, kill them, or continue
    some waiting commands.

Synopsis:
  ./screener.sh --help [--verbose]
  ./screener.sh list-running
  ./screener.sh list-waiting
  ./screener.sh list-failed
  ./screener.sh list-critical
  ./screener.sh list-serious
  ./screener.sh list-done
  ./screener.sh list
  ./screener.sh list-screens
  ./screener.sh run <file.csv> [<condition_list>]
  ./screener.sh start <screen_id> <cmd> <args...>
  ./screener.sh [<options>] <operation> <screen_id>

Inquiry operations:

  ./screener.sh list-screens
    Equivalent to screen -ls

  ./screener.sh list-<type>
    Show a list of currently running, waiting (for continuation), failed,
    and done/completed screen sessions.

  ./screener.sh list
    First show a list of currently running screens, then
    for each <type> a list of (old) failed / completed / sessions
    (and so on).

  ./screener.sh status <screen_id>
    Like list-*, but filter <sceen_id> and dont report timestamps.

  ./screener.sh show <screen_id>
    Show the last logfile of <screen_id> at standard output.

  ./screener.sh less <screen_id>
    Show the last logfile of <screen_id> using "less -r".

MASS starting of screen sessions:

  ./screener.sh run <file.csv> <condition_list>
    Commands are launched in screen sessions via "./screener.sh start" commands,
```

```
    unless the same <screen_id> is already running,
    or is in some error state, or is already done (see below).
    The commands are given by a column with CSV header name
    containing "command", or by the first column.
    The <screen_id> needs to be given by a column with CSV header
    name matching "screen_id|resource".
    The number and type of commands to launch can be reduced via
    any combination of the following filter conditions:

      --max=<number>
        Limit the number of _new_ sessions additionally started this time.

      --<column_name>==<value>
        Only select lines where an arbitrary CSV column (given by its
        CSV header name in C identifier syntax) has the given value.

      --<column_name>!=<value>
        Only select lines where the colum has _not_ the given value.

      --<column_name>=~<bash_regex>
        Only select lines where the bash regular expression matches
        at the given column.

      --max-per=<number>
        Limit the number per _distinct_ value of the column denoted by
        the _next_ filter condition.
        Example: ./screener.sh run test.csv --dry-run --max-per=2 --dst_network=~.
        would launch only 2 Football processes per destination network.

    Hint: filter conditions can be easily checked by giving --dry-run.

Start / restart / kill / continue screen sessions:

  ./screener.sh start <screen_id> <cmd> <args...>
    Start a new screen session, running arbitrary <cmd> and <args...>
    inside.

  ./screener.sh restart <screen_id>
    Works only when the last command for <screen_id> failed.
    This will restart the old <cmd> and its <args...> as before.
    Use only when you want to repeat the same command once again.

  ./screener.sh kill <screen_id>
    Terminate the running screen session forcibly.

  ./screener.sh continue
  ./screener.sh continue <screen_id> [<screen_id_list>]
  ./screener.sh continue <number>
    Useful for MASS automation of processes involving critical sections
    such as customer downtime.
    When giving a numerical <number> argument, up to that number
    of sessions are resumed (ordered by age).
    When no further arugment is given, _all_ currently waiting sessions
    are continued.
    When --auto-attach is given, it will sequentially resume the
    sessions to be continued. By default, unless --force_attach is set,
    it uses "screen -r" skipping those sessions which are already
    attached to somebody else.
```

```
    This feature works only with prepared scripts which are creating
    an empty flagfile
    /home/schoebel/mars/mars-migration.git/screener-logdir-testing/running/$screen_id.waiting
    whenever they want to wait for manual intervention (for whatever reason).
    Afterwards, the script must be polling this flagfile for removal.
    This screener operation simply removes the flagfile, such that
    the script will then continue afterwards.
    Example: look into ./football.sh
    and search for occurrences of substring "call_hook start_wait".

  ./screener.sh wakeup
  ./screener.sh wakeup <screen_id> [<screen_id_list>]
  ./screener.sh wakeup <number>
    Similar to continue, but refers to delayed commands waiting for
    a timeout. This can be used to individually shorten the timeout
    period.
    Example: Football cleanup operations may be artificially delayed
    before doing "lvremove", to keep some sort of 'backup' for a
    limited time. When your project is under time pressure, these
    delays may be hindering.
    Use this for premature ending of such artificial delays.

  ./screener.sh up <...>
    Do both continue and wakeup.

  ./screener.sh auto <...>
    Equivalent to ./screener.sh --auto-attach up <...>
    Remember that only session without current attachment will be
    attached to.

Attach to a running session:

  ./screener.sh attach <screen_id>
    This is equivalent to screen -x $screen_id

  ./screener.sh resume <screen_id>
    This is equivalent to screen -r $screen_id

Communication:

  ./screener.sh notify <screen_id> <txt>
    May be called from external scripts to send emails etc.

Locking (only when supported by <cmd>):

  ./screener.sh lock
  ./screener.sh unlock
  ./screener.sh lock <screen_id>
  ./screener.sh unlock <screen_id>

Cleanup / bookkeeping:

  ./screener.sh clear-critical <screen_id>
  ./screener.sh clear-serious <screen_id>
  ./screener.sh clear-failed  <screen_id>
    Mark the status as "done" and move the logfile away.

  ./screener.sh purge [<days>]
```

```
    This will remove all old logfiles which are older than
    <days>. By default, the variable $screener_log_purge_period
    will be used, which is currently set to '30'.

  ./screener.sh cron
    You should call this regulary from a user cron job, in order
    to purge old logfiles, or to detect hanging sessions, or to
    automatically send pending emails, etc.

Options:

  --variable
  --variable=$value
    These must come first, in order to prevent mixup with
    options of <cmd> <args...>.
    Allows overriding of any internal shell variable.
  --help --verbose
    Show all overridable shell variables, also for plugins.


PLUGIN screener-email

  Generic plugin for sending emails (or SMS via gateways)
  upon status changes, such as script failures.
```

H.5. screener.sh --help --verbose

```
OVERRIDE verbose=1
./screener.sh: Run _unattended_ processes in screen sessions.
    Useful for MASS automation, running hundreds of unattended
    commands in parallel.
    HINT: for running more than ~500 sessions in parallel, you might need
    some system tuning (e.g. rlimits, kernel patches etc) for creating
    a huge number of file descritor / sockets / etc.
    ADVANTAGE: You may attach to individual screens, kill them, or continue
    some waiting commands.

Synopsis:
  ./screener.sh --help [--verbose]
  ./screener.sh list-running
  ./screener.sh list-waiting
  ./screener.sh list-failed
  ./screener.sh list-critical
  ./screener.sh list-serious
  ./screener.sh list-done
  ./screener.sh list
  ./screener.sh list-screens
  ./screener.sh run <file.csv> [<condition_list>]
  ./screener.sh start <screen_id> <cmd> <args...>
  ./screener.sh [<options>] <operation> <screen_id>

Inquiry operations:

  ./screener.sh list-screens
    Equivalent to screen -ls

  ./screener.sh list-<type>
```

```
    Show a list of currently running, waiting (for continuation), failed,
    and done/completed screen sessions.

  ./screener.sh list
    First show a list of currently running screens, then
    for each <type> a list of (old) failed / completed / sessions
    (and so on).

  ./screener.sh status <screen_id>
    Like list-*, but filter <sceen_id> and dont report timestamps.

  ./screener.sh show <screen_id>
    Show the last logfile of <screen_id> at standard output.

  ./screener.sh less <screen_id>
    Show the last logfile of <screen_id> using "less -r".

MASS starting of screen sessions:

  ./screener.sh run <file.csv> <condition_list>
    Commands are launched in screen sessions via "./screener.sh start" commands,
    unless the same <screen_id> is already running,
    or is in some error state, or is already done (see below).
    The commands are given by a column with CSV header name
    containing "command", or by the first column.
    The <screen_id> needs to be given by a column with CSV header
    name matching "screen_id|resource".
    The number and type of commands to launch can be reduced via
    any combination of the following filter conditions:

      --max=<number>
        Limit the number of _new_ sessions additionally started this time.

      --<column_name>==<value>
        Only select lines where an arbitrary CSV column (given by its
        CSV header name in C identifier syntax) has the given value.

      --<column_name>!=<value>
        Only select lines where the colum has _not_ the given value.

      --<column_name>=~<bash_regex>
        Only select lines where the bash regular expression matches
        at the given column.

      --max-per=<number>
        Limit the number per _distinct_ value of the column denoted by
        the _next_ filter condition.
        Example: ./screener.sh run test.csv --dry-run --max-per=2 --dst_network=~.
        would launch only 2 Football processes per destination network.

    Hint: filter conditions can be easily checked by giving --dry-run.

Start / restart / kill / continue screen sessions:

  ./screener.sh start <screen_id> <cmd> <args...>
    Start a new screen session, running arbitrary <cmd> and <args...>
    inside.
```

```
  ./screener.sh restart <screen_id>
    Works only when the last command for <screen_id> failed.
    This will restart the old <cmd> and its <args...> as before.
    Use only when you want to repeat the same command once again.

  ./screener.sh kill <screen_id>
    Terminate the running screen session forcibly.

  ./screener.sh continue
  ./screener.sh continue <screen_id> [<screen_id_list>]
  ./screener.sh continue <number>
    Useful for MASS automation of processes involving critical sections
    such as customer downtime.
    When giving a numerical <number> argument, up to that number
    of sessions are resumed (ordered by age).
    When no further arugment is given, _all_ currently waiting sessions
    are continued.
    When --auto-attach is given, it will sequentially resume the
    sessions to be continued. By default, unless --force_attach is set,
    it uses "screen -r" skipping those sessions which are already
    attached to somebody else.
    This feature works only with prepared scripts which are creating
    an empty flagfile
    /home/schoebel/mars/mars-migration.git/screener-logdir-testing/running/$screen_id.waiting
    whenever they want to wait for manual intervention (for whatever reason).
    Afterwards, the script must be polling this flagfile for removal.
    This screener operation simply removes the flagfile, such that
    the script will then continue afterwards.
    Example: look into ./football.sh
    and search for occurrences of substring "call_hook start_wait".

  ./screener.sh wakeup
  ./screener.sh wakeup <screen_id> [<screen_id_list>]
  ./screener.sh wakeup <number>
    Similar to continue, but refers to delayed commands waiting for
    a timeout. This can be used to individually shorten the timeout
    period.
    Example: Football cleanup operations may be artificially delayed
    before doing "lvremove", to keep some sort of 'backup' for a
    limited time. When your project is under time pressure, these
    delays may be hindering.
    Use this for premature ending of such artificial delays.

  ./screener.sh up <...>
    Do both continue and wakeup.

  ./screener.sh auto <...>
    Equivalent to ./screener.sh --auto-attach up <...>
    Remember that only session without current attachment will be
    attached to.

Attach to a running session:

  ./screener.sh attach <screen_id>
    This is equivalent to screen -x $screen_id

  ./screener.sh resume <screen_id>
    This is equivalent to screen -r $screen_id
```

```
Communication:

  ./screener.sh notify <screen_id> <txt>
    May be called from external scripts to send emails etc.

Locking (only when supported by <cmd>):

  ./screener.sh lock
  ./screener.sh unlock
  ./screener.sh lock <screen_id>
  ./screener.sh unlock <screen_id>

Cleanup / bookkeeping:

  ./screener.sh clear-critical <screen_id>
  ./screener.sh clear-serious <screen_id>
  ./screener.sh clear-failed  <screen_id>
    Mark the status as "done" and move the logfile away.

  ./screener.sh purge [<days>]
    This will remove all old logfiles which are older than
    <days>. By default, the variable $screener_log_purge_period
    will be used, which is currently set to '30'.

  ./screener.sh cron
    You should call this regulary from a user cron job, in order
    to purge old logfiles, or to detect hanging sessions, or to
    automatically send pending emails, etc.

Options:

  --variable
  --variable=$value
    These must come first, in order to prevent mixup with
    options of <cmd> <args...>.
    Allows overriding of any internal shell variable.
  --help --verbose
    Show all overridable shell variables, also for plugins.

  ## screener_includes
  # List of directories where screener-*.conf files can be found.
  screener_includes="${screener_includes:-/usr/lib/mars/plugins /etc/mars/plugins $script_dir/plugi

  ## screener_confs
  # Another list of directories where screener-*.conf files can be found.
  # These are sourced in a second pass after $screener_includes.
  # Thus you can change this during the first pass.
  screener_confs="${screener_confs:-/usr/lib/mars/confs /etc/mars/confs $script_dir/confs $HOME/.ma

  ## title
  # Used as a title for startup of screen sessions, and later for
  # display at list-*
  title="${title:-}"

  ## auto_attach
  # Upon start or upon continue/wakuep/up, attach to the
  # (newly created or existing) session.
```

```
auto_attach="${auto_attach:-0}"

## auto_attach_grace
# Before attaching, wait this time in seconds.
# The user may abort within this sleep time by
# pressing Ctrl-C.
auto_attach_grace="${auto_attach_grace:-10}"

## force_attach
# Use "screen -x" instead of "screen -r" allowing
# shared sessions between different users / end terminals.
force_attach="${force_attach:-0}"

## drop_shell
# When a <cmd> fails, the screen session will not terminated immediately.
# Instead, an interactive bash is started, so can later attach and
# rectify any probllems.
# WARNING! only activate this if you regulary check for failed sessions
# and then manually attach to them. Don't use this when running hundreds
# or thousand in parallel.
drop_shell="${drop_shell:-0}"

## session_timeout
# Detect hanging sessions when they don't produce any output anymore
# for a longer time. Hanging sessions are then marked as failed or critical.
session_timeout="${session_timeout:-$(( 3600 * 3 ))}" # seconds

## screener_logdir or logdir
# Where the logfiles and all status information is kept.
export screener_logdir="${screener_logdir:-${logdir:-$HOME/screener-logs}}"

## screener_command_log
# This logfile will accumulate all relevant $0 command invocations,
# including timestamps and ssh agent identities.
# To switch off, use /dev/null here.
screener_command_log="${screener_command_log:-$screener_logdir/commands.log}"

## screener_cron_log
# Since "$0 cron" works silently, you won't notice any errors.
# This logfiles gives you a chance for checking any problems.
screener_cron_log="${screener_cron_log:-$screener_logdir/cron.log}"

## screener_log_purge_period
# $0 cron or $0 purge will automatically remove all old logfiles
# from $screener_logdir/*/ when this period is exceeded.
screener_log_purge_period="${screener_log_purge_period:-30}" # Days

## dry_run
# Dont actually start screen sessions when set.
dry_run="${dry_run:-0}"

## verbose
# increase speakiness.
verbose=${verbose:-0}

## debug
# Some additional debug messages.
debug="${debug:-0}"
```

```
## sleep
# Workaround races by keeping sessions open for a few seconds.
# This is useful for debugging of immediate script failures.
# You have some short time window for attaching.
# HINT: instead, just inspect the logfiles in $screener_logdir/*/*.log
sleep="${sleep:-3}"

## screen_cmd
# Customize the screen command (e.g. add some further options, etc).
screen_cmd="${screen_cmd:-screen}"

## use_screenlog
# Add the -L option. Not really useful when running thousands of
# parallel screen sessions, because the automatically generated filenames
# are crap, and cannot be set in advance.
# Useful for basic debugging of setup problems etc.
use_screenlog="${use_screenlog:-0}"

## waiting_txt and delay_txt
# RTFS Don't use this, unless you know what you are doing.
waiting_txt="${waiting_txt:-SCREENER_waiting_WAIT}"
delayed_txt="${delayed_txt:-SCREENER_delayed_WAIT}"

## critical_status
# This is the "magic" exit code indicating _criticality_
# of a failed command.
critical_status="${critical_status:-199}"

## serious_status
# This is the "magic" exit code indicating _seriosity_
# of a failed command.
serious_status="${serious_status:-198}"

## less_cmd
# Used at $0 less $id
less_cmd="${less_cmd:-less -r}"

## date_format
# Here you can customize the appearance of list-* commands
date_format="${date_format:-%Y-%m-%d %H:%M}"

## csv_delimit
# The delimiter used for CSV file parsing
csv_delim="${csv_delim:-;}"

## csv_cmd_fields
# Regex telling the field name for 'cmd'
csv_cmd_fields="${csv_cmd_fields:-command}"

## csv_id_fields
# Regex telling the field name for 'screen_id'
csv_id_fields="${csv_id_fields:-screen_id|resource}"

## csv_remove
# Regex for global removal of command options
csv_remove="${csv_remove:---screener}"
```

```
  ## user_name
  # Normally automatically derived from ssh agent or from $LOGNAME.
  # Please override this only when really necessary.
  export user_name="${user_name:-$(ssh-add -l | grep -o '[^ ]+@[^ ]+' | sort -u | tail -1)}"
  export user_name="${user_name:-$LOGNAME}"

  ## tmp_dir and tmp_stub
  # Where temporary files are residing
  tmp_dir="${tmp_dir:-/tmp}"
  tmp_stub="${tmp_stub:-$tmp_dir/screener.$$}"

Running hook: email_describe_plugin

PLUGIN screener-email

  Generic plugin for sending emails (or SMS via gateways)
  upon status changes, such as script failures.

  ## email_*
  # List of email addresses.
  # Empty = don't send emails.
  email_critical="${email_critical:-}"
  email_serious="${email_serious:-}"
  email_failed="${email_failed:-}"
  email_warning="${email_warning:-}"
  email_waiting="${email_waiting:-}"
  email_done="${email_done:-}"

  ## sms_*
  # List of email addresses of SMS gateways.
  # These may be distinct from email_*.
  # Empty = don't send sms.
  sms_critical="${sms_critical:-}"
  sms_serious="${sms_serious:-}"
  sms_failed="${sms_failed:-}"
  sms_warning="${sms_warning:-}"
  sms_waiting="${sms_waiting:-}"
  sms_done="${sms_done:-}"

  ## email_cmd
  # Command for email sending.
  # Please include your gateways etc here.
  email_cmd="${email_cmd:-mailx -S smtp=mx.nowhere.org:587 -S smpt-auth-user=test}"

  ## email_logfiles
  # Whether to include logfiles in the body.
  # Not used for sms_*.
  email_logfiles="${email_logfiles:-1}"
```

I. GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not

allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If

there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements",

and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to

60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

    Copyright (c) YEAR YOUR NAME.

```
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```