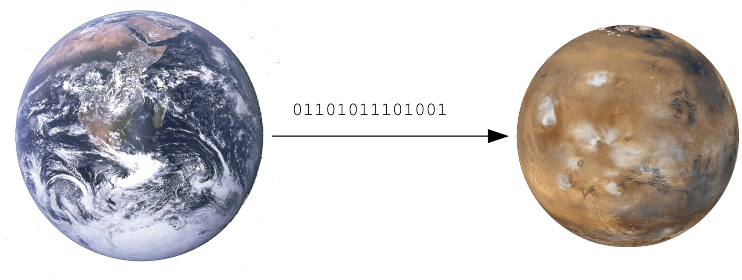
MARS Manual

Multiversion Asynchronous Replicated Storage



Thomas Schöbel-Theuer (`tst@1und1.de`)

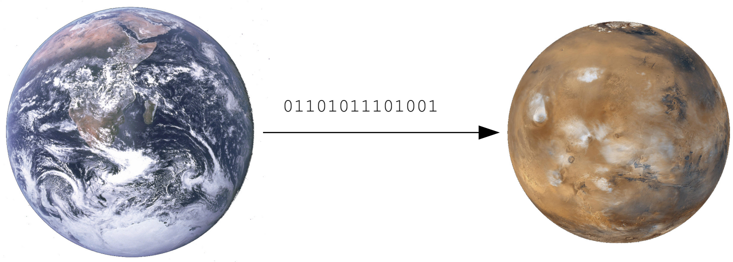
Version 0.18

Abstract

`MARS` Light 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 Light 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 Even in case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but never become inconsistent.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 Light preserves the **order of write operations** even when the network is flaky (**Anytime Consistency**).

The current version of MARS Light 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.



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

Contents

1. 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?

There exist *some* cases where DRBD is 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).

On the other hand, there exist other cases where DRBD did not work as expected, leading to incidents and other operational problems. We analyzed them for those 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.

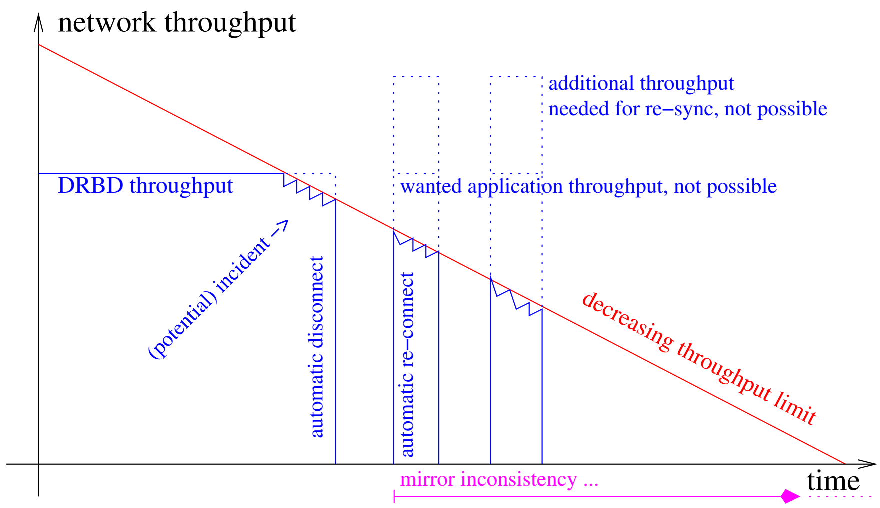
MARS and DRBD simply have **different application areas**.

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.

1.1. Network Bottlenecks

1.1.1. 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. The following picture illustrates an effect which is not only observable in practice, but is also reproducible by the MARS test suite[1]:



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.

[1]The effect has been demonstrated 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.

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[2] 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[3].

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[4] 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

---

[2] 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.

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

[4] 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.

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[5] either work, or not. The binary connect / disconnect behaviour of DRBD has no problems to cope with that.

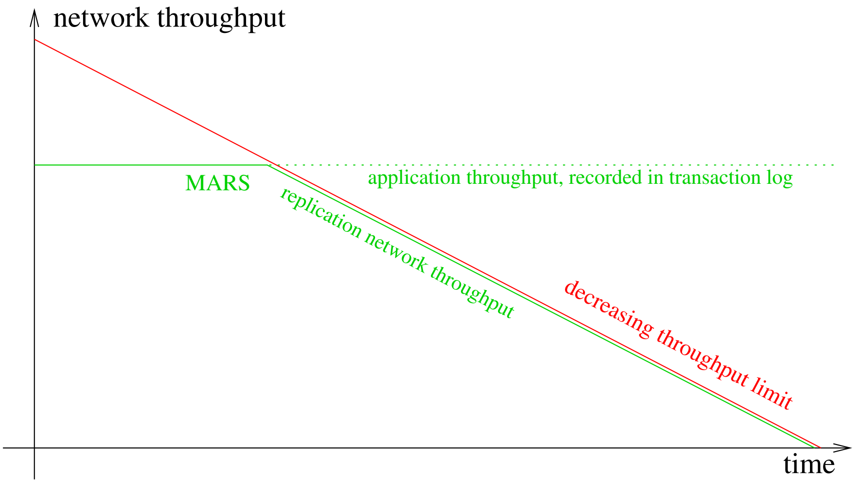
or Linbit recommends a **workaround** for the inconsistencies during re-sync: LVM snapshots. We tried it, but found a *performance penalty* which made it prohibitive for our concrete application. A problem seems to be the cost of destroying snapshots. LVM uses by default a BOW strategy (Backup On Write, 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.

1.1.2. Behaviour of MARS

The behaviour of MARS in the above scenario:

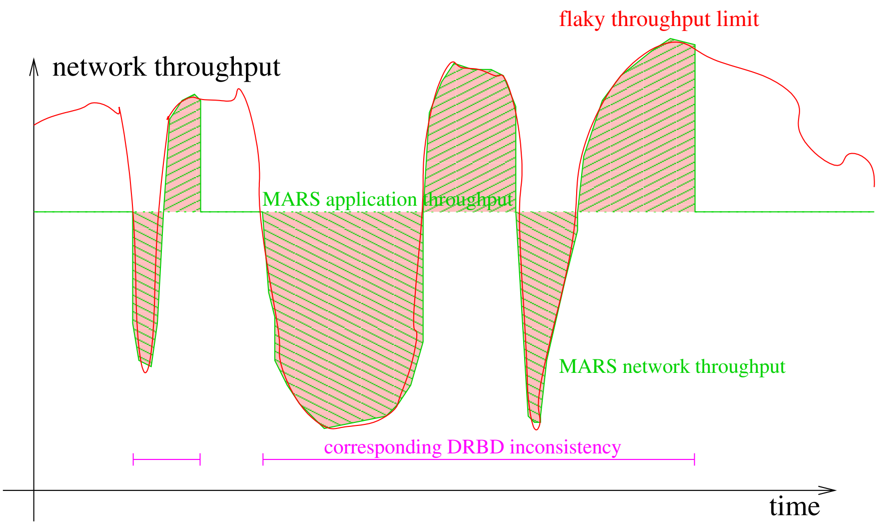
[5] Exceptions might be mechanical jiggling of plugs, or electro-magnetical interferences. We never noticed any of them.



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 3.4 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[6] 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

[6] 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**".

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[7] 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).

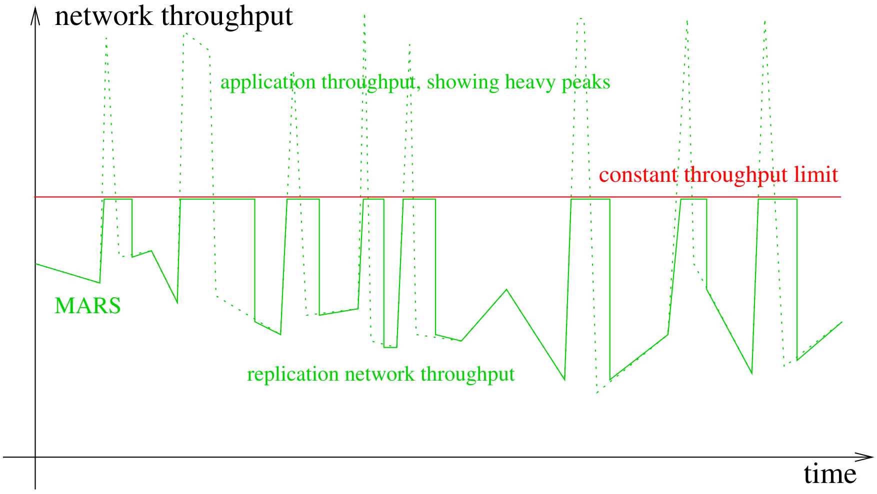
In case of lag-behind, the version of the data replicated to the secondary site corresponds to some time in the past. Since the data is always transferred in the same order as originally submitted at the primary site, the secondary never gets inconsistent. Your mirror always remains usable. Your only 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 network bottlenecks, there exists no better method than prescribed by the information theoretic limit (red line, neglecting data compression).

MARS' property of never sacrificing local data consistency (at the possible cost of actuality) is called **Anytime Consistency**.

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 3.4).

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

[7] 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.



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 capabilities for some 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.

1.2. 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, synchronous data replication (not limited to applications of DRBD) works reliably only over distances < 50 km.

There may be some exceptions, at least 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.

1.3. Higher Consistency Guarantees vs Actuality

We already saw in section 1.1 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.

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[8].

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*[9] 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[10]. 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 1.1.2 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 Light 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*

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

[9] 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?

[10] 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.

in order to restore consistency. According to POSIX[11] 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] The above argumentation also applies to Windows filesystems in analogous way.

2. 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/`)
.

2.1. Preparation: What you Need

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

Typically, you will use MARS Light 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 Light 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.

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!

2.2. 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).

2.2.1. Kernel and MARS Module

At the time of this writing, a small pre-patch for the Linux kernel is needed. It it trivial and consists mostly of `EXPORT_SYMBOL()` statements. The pre-patch must be applied to the kernel source tree before building your (custom) kernel. Future versions of MARS are planned to require no pre-patch anymore.

The MARS kernel module can be built in two different ways:

1. inplace in the kernel source tree: `cd block/ && git clone git://github.com/schoebel/mars`

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).

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.

2.2.2. 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`.

[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.

5. 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.

6. 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.

7. On node B, say `marsadm join-cluster A`

8. Only *after*[6] that, do `modprobe mars` on each node.

2.3. 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.

[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.

[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.

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 `/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 at any time by saying `marsadm invalidate mydata` (analogously to the corresponding DRBD command).

2.4. Keeping Resources Operational

2.4.1. 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[9], or once a day (depending on your load).

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`

[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`.

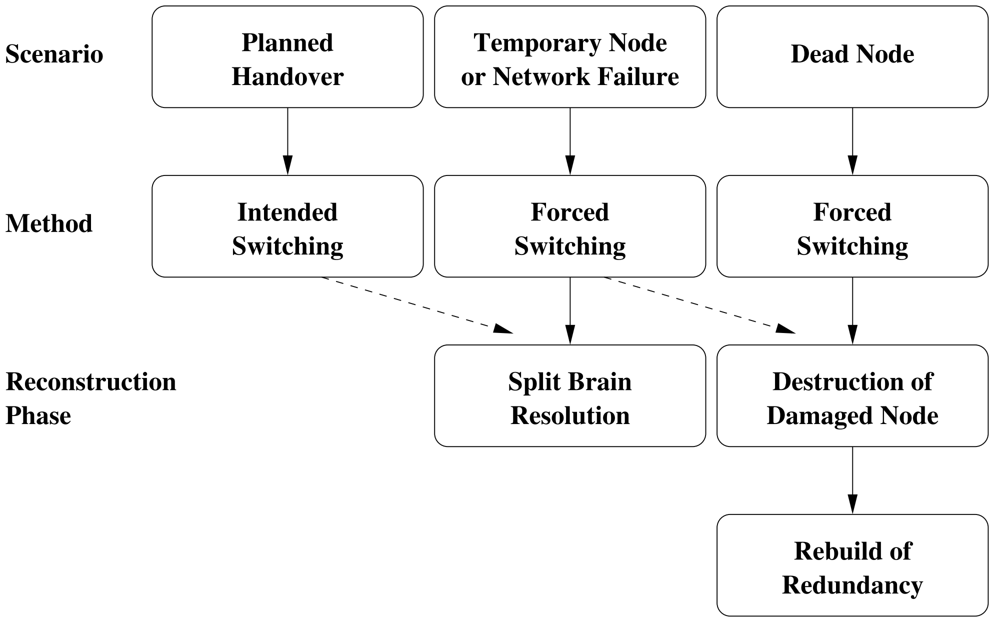
[9] Under *extremely* high load conditions, you might want to log-rotate serveral times an hour, in order to keep the size of each logfile under some practical limit. At 1&1 datacenters, we have not yet encountered conditions where that was really *necessary*.

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.

Best practice is to do both log-rotate and log-delete-all in a cron job. 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 3.4.

2.4.2. Switch Primary / Secondary Roles



In contrast to DRBD, MARS Light distinguishes between *intended* and *forced* switching. This distinction is necessary due to differences in the communication architecture (asynchronous communication vs synchronous communication, see sections 3.2 and 3.3).

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.

2.4.2.1. 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 2.6.1.2 (see also section 2.7). 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, or it may produce a split brain, or it may even fail.

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. Stop your application, then umount /dev/mars/mydata on host A.

3. When scripting, or when typing extremely fast, or for better safety, say `marsadm wait-umount mydata` host `B`. When your network is OK, the propagation of the device usage state[10] should take only a few seconds. Otherwise, check for any network problems or any other problems.

4. 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 2.4.2.2 and 2.4.3).

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

- `marsadm primary mydata`

on your formerly secondary node `B`.

The most important difference to DRBD: don't use an intermediate `marsadm secondary mydata` anywhere. Although it would be possible, it has *disadvantages*, such as increased risk of producing a split brain. 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`.

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 3.2). By saying `marsadm primary mydata` on host `B`, **all other** hosts (including `A`) will **automatically go into secondary role** after a while!

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

Precondition for `marsadm primary` is that you are up, that means in attached and connected state (cf. `marsadm up`), and 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 4.2.2 is violated, `marsadm primary` may refuse to start.

The 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

---

[10] 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 3.3). 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 2.7.

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 2.7). `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*[11] to be *really* `UpToDate` (see `marsadm wait-cluster` and `marsadm view` and other macros described in section 2.6).

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.

2.4.2.2. 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 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 re-connect 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[12] *all* members of the resource to stop fetching logfiles, you may use `marsadm pause-fetch-global mydata` instead (cf section 4.2.2).
- `marsadm primary mydata --force`
  - this is the forceful switchover. Use `--force` only if you know what you are doing!
- `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`, not only many precondition checks and other internal checks are skipped, but also the internal handover protocol for split brain avoidance.

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

[11] 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!

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

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 2.4.3), 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*[13], 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 2.4.3, or even destruction / reconstruction of a damaged node as described in section 2.4.4.

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.

Caveats In contrast to DRBD, split brain situations are handled differently by MARS Light. 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` 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.

When one of your multiple split brain nodes has left its actual primary role, e.g. after umounting its local `/dev/mars/mydata` device, and when the network is up (again), we cannot guarantee[14] that it is always possible to re-enter primary mode again, even when `primary --force` is given. Therefore, use `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. However, when the device `/dev/mars/mydata` is in use somewhere, it will remain in *actual* primary mode during that time, even if another host is now the designated primary, or if `(none)` is designated as primary as will result from a `secondary` command. As soon as a local `/dev/mars/mydata` is released, the node will *actually* go into secondary mode if it is no longer designated as primary. Thus, `marsadm secondary` can lead to a situation where noone is actually in primary role, and noone is able to re-enter it due to split brain. Such a situation can be avoided by *directly* switching over from one primary to another one, without intermediate `secondary` command. This behaviour is different from DRBD.

In case you have accidentally entered such a situation where all nodes are refusing to become primary due to split brain, you *have to* cleanup the split brain via `leave-resource` and friends, or use the method described in section 2.4.5. Remember that split brain is an ***erroneous*** state. Therefore it is **generally no good idea to (re-)enter it deliberately, or to stay in it any longer!**

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

[13] Most reasons will be displayed by `marsadm` when it is rejecting the switchover.

[14] In a few cases which are covered by the test suite, it is likely to work. Future versions of MARS Light might improve on this. It is generally no good idea to try to (forcefully) become primary in a split-brain situation starting from being secondary, because the result is likely to be **undefined at concept level**.

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 information about the former designated primary.

2.4.3. 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 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 2.4.2.2). Don't use an intermediate `marsadm secondary` command (as known from DRBD): *directly* switch to the new designated primary!

3. 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*[15] 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 2.4.4, but do this only as far as necessary.

5. If any of your (surviving) cluster nodes has already the "right" version and was not in a primary role when the split brain happened, you don't need to do the following step for it, of course. The following applies only to those nodes which *deviate* from the correct version:

6. It may happen that the "right" version you want to retain is *not* the version which is currently designated as primary for the whole cluster. **Only** in such a case, switch the primary role as described in sections 2.4.2.1 or 2.4.2.2. Here is a repetition of the necessary steps:

    a) First try `marsadm primary mydata` on the new designated primary host. Don't mix up your shell windows!

    b) Only if that refuses working *for no good reason*, do the following steps:

        i. `marsadm pause-fetch mydata`.

[15] 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 3.2 and 3.3), you may deviate from this requirement.

ii. `marsadm primary mydata --force`.

iii. `marsadm resume-fetch mydata`.

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 cannot retain its split-brain version, but start with the old "wrong" primaries first (see advice at the end of this section):

7. `marsadm leave-resource mydata`

8. 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[16] 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. Only in such rare cases, read the documentation about `log-purge-all` (see page 62) and try it.

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 should 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.

Alternatively, but only if you have only $k = 2$ replicas in total, you may use the following short procedure instead, which works in almost all $k = 2$ cases, but cannot resolve all (desperate, very scarce) split-brain situations (see documentation of `log-purge-all` on page 62):

7. On the single (new) secondary with a non-"right" version, and only if the split brain has not yet been resolved, say `marsadm invalidate mydata`.

Keeping a Split Brain Version This case starts indentical as before, but continues differently. On each of those cluster node(s) you don't want to retain:

7. `marsadm leave-resource mydata`

8. 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 62).

9. Check that each underlying local disk `/dev/lv-x/mydata` 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.

10. Create a completely new MARS resource out of the underlying disk `/dev/lv-x/mydata` having a different name, such as `mynewdata` (see description in section Creating and Maintaining Resources).

[16] 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`.

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[17] that you don't need to do any action for it. When all wrong versions have disappeared from the cluster (by `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: 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 leaving its primary state at all (if it is possible – notice that if you have enough space on `/mars/` it may be even possible to not only continue your application during the split brain without interruption, just by not umounting `/dev/mars/mydata` at all, but in addition to avoid invalidations caused by emergency mode, see section 3.4.2). *First* resolve the problem of the "wrong" primary(s) via `leave-resource`. Wait for a short while. Then check the rest of your secondaries (if you have $k > 2$ replicas in total), whether they now are already following the new (unique) primary, and finally check whether the split brain warning reported by `marsadm view all` is already gone. This way, you can often omit unnecessary invalidations of replicas.

2.4.4. 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 2.4.2.2).

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 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`.

[17] 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.

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!

In case `leave-resource --host=` does not work, you can start over with the following fall-back:

4. On the surviving new designated primary, give the following commands
   a) `marsadm disconnect-all mydata`
   b) `marsadm down mydata`
   c) Check by hand whether your local disk is consistent, e.g. by test-mounting it read-only, `fsck`, etc.
   d) `marsadm delete-resource mydata`
   e) 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`).
   f) Be sure that the resource has disappeared *everywhere*.
   g) `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).

2.4.5. Cleanup in case of Complicated Cascading Failures

MARS Light does its best to recover even from multiple failures (e.g. **rolling disasters**). Chances are high that the previous instructions 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) 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-rotated 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 "chances" for final loss of data are 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 2.4.2.2). 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 disk<8s<9.

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.

11. 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 successfully completed. This way, you will be retaining some very old and outdated, but hopefully potentially usable old 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[18]. 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`.

[18] 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.

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`.

2.4.6. 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 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-resources` or `invalidates` 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 2.7.

2.5. 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**.

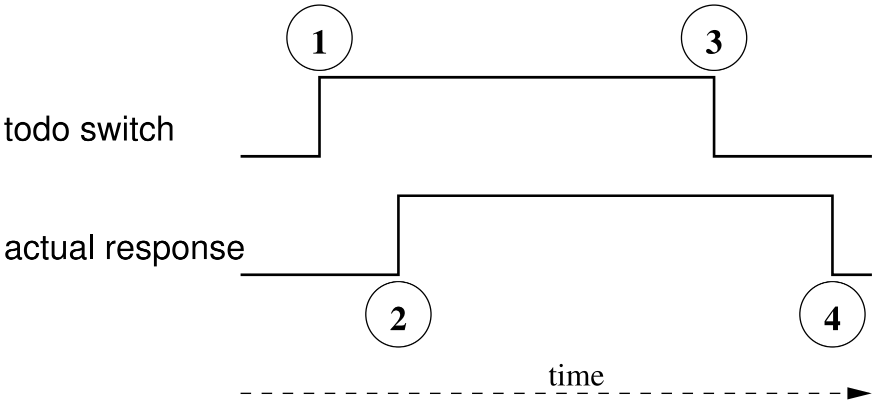
This is *necessary* because in a widely distributed long-distance system running on slow or even temporarily failing networks, actions may take a long time, and there may be many actions **started in parallel**.

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[19].

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:

[19] 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.



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[20]. 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.

2.6. Inspecting the State of MARS

The main command for viewing the current state of MARS Light is

- `marsadm view mydata`

or its more specialized variants

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

where *`macroname`* is one of the following macros described in the following sections, 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.

2.6.1. 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.

[20] 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.

2.6.1.1. 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 2.6.2, 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`[21].

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

- `%{res}` resource name.
- `%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.

`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[22] tool `marsview`[23]

Customization via your own macros (see section 2.6.2) 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.

`diskstate` Shows the status of the underlying disk device: `NotPresent`, `Detached`, `InConsistent`, `NeedsReplay`, `OutDated`, `UpToDate`.

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

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

[22] 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!**

[23] 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.

`replstate` Shows the status of the replication: `NotJoined`, `PrimaryUnreachable`, `PausedSync`, `Syncing`, `PausedReplay`, `Replaying`.

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

`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.

`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. Use cases are analogously to `syncinfo`.

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

2.6.1.2. Predefined Trivial Macros

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[24] 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.

`{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).

`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 2.5.

`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.

`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 2.5.

`is-split-brain` Shows whether split brain (see section 2.4.3) 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[25]. Don't confuse this with the consistency of `/dev/mars/mydata`, which is by construction *always* locally consistent once it has appeared[26]. By construction of MARS, the disk

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

[25] 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.

[26] 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.

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 1.1.2).

`is-emergency` Shows whether emergency mode (see section 3.4.2) 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.

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

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

Intended for Scripting While complex macros may output a whole bunch of information, the following "trivial" macros are outputting exactly one value. They are intended for script use (cf. section 2.7). 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).

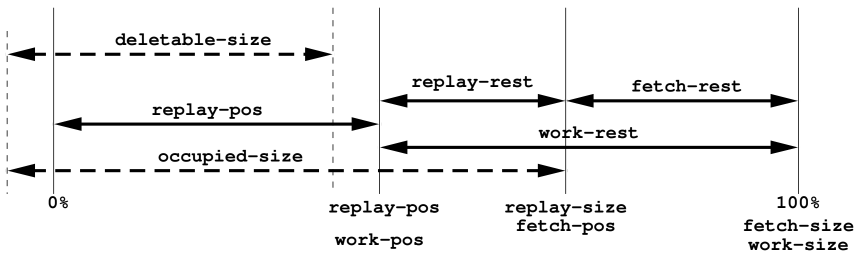


Figure 2.1.: overview on amounts / cursors

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 3.4.2).

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[27]" 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.

[27] 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.

{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[28].

{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 should be preferred over `*-reached` for *human display*, because it produces less flickering by the inevitable replication delay.

{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[29] 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{}`.

[28] Recall from chapter 1 that MARS Light (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.

[29] 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.

Misc Informational Status

`get-primary` Return the name of the current designated primary node as locally known.

`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 2.5 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 a unique identifier originally created at `create-cluster`. 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 4.2).

`tree` (global) Indicate symlink tree version (see section 3.3).

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!

2.6.2. 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`.

2.6.2.1. 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 2.6.2.2).
- 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.

2.6.2.2. 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 2.6.1.2. 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}` 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-pa`* 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[30].

- `%eval-down{`*body*`}` Evaluates the *body* in a loop until the result does not change any more[31].

- `%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 3.2), 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.

[30] TEX or LATEX fans usually know what this is good for ;)
[31] 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.
- `%fetch-lognr{}` etc. Get logfile numbers. Only for curious humans - don't use in scripts, don't base any decisions on this.
- `%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.
- `%does{`*name*`}` Equivalent to `%is-`*name*`{}` (just more handy for computing the macro name). Use with care!

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.
- `%{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.

2.7. Scripting HOWTO

Both the **asynchronous communication model** of MARS (cf section 3.2) including the Lamport clock, and the **state model** (cf section 2.5) 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!

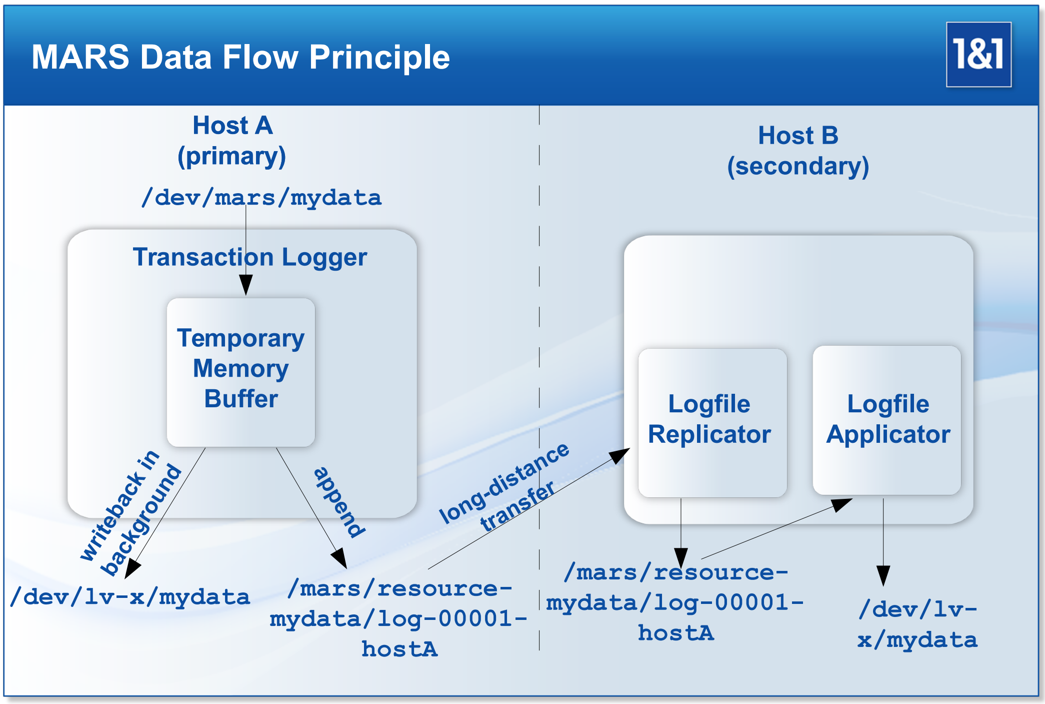
- Test your scripts in failure scenarios!

3. 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.

3.1. The Transaction Logger



MARS LCA2014 Presentation by Thomas Schöbel-Theuer

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 4.2.2). 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.

3.2. 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 1.1 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.

3.3. 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 2.6.1.2.

The `/mars/` filesystem contains not only transaction logfiles, but also acts as a generic storage for (persistent) state information. Both configuration information and runtime state informa-

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

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

Therefrom results a fundamentally different behaviour than DRBD. When your DRBD primary crashed before 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. Thus the `/mars/` directory forms some kind of *global namespace*.

Since the symlink replication works generically, you may use the `/mars/userspace/` directory in order to place your own symlink there (for whatever purpose, which need not have to do with MARS).

In order to avoid name clashes, each symlink created at node A should have the name A in its path name. Typically, internal MARS names follow the scheme `/mars/`*`something`*`/myname-A`, and you should follow the best practice of systematically using `/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).

Important: the convention of placing the **creator host name** inside your symlink names 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 symlinks as often as they want, but never modify them. This way, your cluster nodes are able to *communicate* with each other via symlink updates.

Although you may create (and change) your 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 two 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 symlinks) works on the *whole cluster*.

What's the difference? If you try to remove your symlink locally by hand via `rm -f`, you will be surprised: since the symlink has been replicated to 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 resurrect it whenever "necessary".

In order to solve the deletion problem, MARS Light 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 after that the auxiliary symlinks in `/mars/todo-global/` are also deleted eventually.

You may change your already existing symlink via `marsadm set-link some-other-value /mars/userspace/mykey-A` . The new value will be propagated in 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.

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

As a consequence, symlink updates made "by hand" via `ln -s` 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 the internal MARS Light symlinks residing in `/mars/` is documented in section 5.4.

3.4. 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[6].

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*.

3.4.1. Countermeasures

3.4.1.1. 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 2.1). 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!

3.4.1.2. 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 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

[6] 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.

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:
all locally secondary resources will stop fetching transaction logfiles. As a side effect, other nodes in the cluster may become unable to delete their logfiles also. This is a desperate action of the kernel module.

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 start removing any logfiles which are no longer used locally. This is a more 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 primary resources are checked for logfiles which are no longer needed *locally*. Locally unneeded files are deleted even when some secondary needs them. As a consequence, some secondaries may get stuck (left in consistent, but outdated state). In order to get them actual again, they will need a `marsadm invalidate` later (if there is no split brain; otherwise you might need the `leave-resource` ; `join-resource` method from section 2.4.3). This is an even more desperate action of the kernel module. You don't want to get there (except for testing).

7. Last desperate kernelspace action when all other 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 3.4.2). 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.

3.4.1.3. 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 1). 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 3.4.1.1), 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[7] 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.

`write_throttle_ratelimit_kb` Set the global IO rate in KB/s for those write requests which are throttled. In case of strongest[8] 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).

[7]Read requests are never throttled at all.

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

`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[9].

3.4.2. Emergency Mode

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.

2. The kernel module writes back all data present in the temporary memory buffer (see figure in section 3.1). 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 was not present anymore.

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. In consequence, the primary sides should continue transaction logging automatically.

[9] 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.

3. On the secondaries, and when there is no split brain, use `marsadm invalidate $res` in order to get your outdated mirrors uptodate. In case of split brain, follow the instructions from section 2.4.3. This will lead to temporarily inconsistent mirrors, so don't do this on all secondaries in parallel, but sequentially step by step. This way, if you have more than 1 mirror, you will always retain at least one consistent, but outdated copy.

   If you had only 1 mirror per resource before the overflow happened, you can now create a new one via `marsadm join-resource $res` on a third node (provided that your storage space permits it after the cleanup). After the initial full sync has finished there, do an `marsadm invalidate $res` on the outdated mirror (if you had no split brain; otherwise follow the instructions in section 2.4.3). 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 disk.

4. 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 3.2 and 3.3. 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 30s.
`--threshold=$size`	no	The macros containing the substring `-threshold-` use this as a default value for approximation whether something has been reached. Default is 10MiB.
`--host=$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 2.4.4.
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

4.1. 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 2.2.2). 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` `$host`	no	Precondition: the `/mars/` filesystem must be mounted and it must be empty (`mkfs.ext4`, see instructions in section 2.2.2). The kernel module must *not* be loaded. The cluster must have been already created at another node `$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 2.2.2 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 porstmortem 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.
`wait-cluster`	no	See section 4.3.3.
Command / Params	Cmp	Description

4.2. 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, that your current node is a 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.

4.2.1. Resource Creation / Deletion / Modification

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 suffix `k`, `m`, `g`, or `t` (denoting size factors in powers of two). The given size must not exceed the actual size of `$disk_dev`.
		Postcondition: the resource `$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 there must not exist a split brain. 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` `$res`	no	Precondition: the resource must be empty (i.e. all members must have left via `leave-resource`). This precondition is overridable by `--force`, increasing the danger to maximum! Postcondition: all cluster members will somewhen be forcefully removed from `$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 Light 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 4.3.3.

Command / Params	Cmp	Description

4.2.2. Operation of the Resource

Common preconditions are the preconditions from section 4.2, 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 2.5). 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 modify** 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`.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`pause-sync-local` `$res`	partly	Precondition: none additionally. Postcondition: any sync operation targeting the local disk (when not yet completed) is paused after a while (cf section 2.5). When successfully completed, this operation will remember the switch state forever and automatically become relevant if a sync is needed again (e.g. `invalidate` or `resize`).
`pause-sync-global` `$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: none additionally. 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.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`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` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`connect` `$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` `$res`	almost	Precondition: sync must have finished at any resource member. All relevant transaction logfiles must be either already locally present, or be fetchable (see `resume-fetch` and `resume-replay`). When some logfile data is locally missing, there must be enough space on `/mars/` to fetch it. The current designated primary must be reachable over network. When there is no designated primary (i.e. `marsadm secondary` had been executed before, which is explicitly *not recommended*), *all* other members of the resource must be reachable (since we have no memory who was the old primary before), and then they must also match the same preconditions. When another host is currently primary (whether designated or not), it must match the preconditions of `marsadm secondary` (that means, its local `/dev/mars/mydata` device must not be in use any more). A split brain must not already exist. Postcondition: `/dev/mars/$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`), many preconditions are skipped, and MARS does its best to actually become primary even if some logfiles are missing or incomplete. `primary --force` is a potentially harmful variant, because it will provoke a split brain in many 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 2.4.3). **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 only the *designated* primary, but actually becoming the / an actual primary may be impossible in case you are *already* in a split brain situation. In such a case, you *must* resolve the split brain immediately after giving this command (see section 2.4.3). Hint in case of $k > 2$ replicas: `marsadm invalidate` cannot resolve a split brain at other secondaries (which are neither the old nor the new designated primary). Therefore, use the `leave-resource` method described in section 2.4.3, starting with a `leave-resource` phase at the old primary, and proceeding to “unrelated” secondaries step by step, until the split brain is gone. Don't `join-resource` again before the split brain is gone! This way, all these replicas will remain consistent for now, but of course outdated (or potentially even a “wrong” split-brain

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 2.4.2.2. 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` `$res`	no	See section 4.3.3.
`log-purge-all` `$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 Light does its best for checking anything, but there is no guarantee. Hint: use `--dry-run` beforehand for checking!

`resize`
`$res`
`[$size]`
	almost	Precondition: all disks in the cluster participating in `$res` must be physically larger than the logical resource size (e.g. by use of `lvm`). When the optional `$size` argument is present, it must be smaller than the minimum of all physical sizes, but larger than the current logical size.
		Postcondition: at the (future) primary (if any), the logical size of `/dev/mars/$dev_name` will reflect the new size after a while.

Command / Params	Cmp	Description

4.2.3. Logfile Operations

Command / Params	Cmp	Description
`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` = `$old_nr` + 1.
`log-delete` `$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	Like `log-delete`, but mark *all* currently unreferenced logfiles for deletion.
Command / Params	Cmp	Description

4.2.4. 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 2.4.3). 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 2.3 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

4.3. Further Operations

4.3.1. Inspection Commands

Command / Params	Cmp	Description
`view-`*`macroname`* `$res`	no	Display the output of a macro evaluation. See section 2.6 for a thorough description.
`view` `$res`	no	Equivalent to `view-default`.
`role` `$res`	no	Deprectated. Use `view-role` instead.
`state` `$res`	no	Deprectated. Use `view-state` instead.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`cstate` `$res`	no	Deprectated. Use `view-cstate` instead.
`dstate` `$res`	no	Deprectated. Use `view-dstate` instead.
`status` `$res`	no	Deprectated. Use `view-status` instead.
`show-state` `$res`	no	Deprectated. Don't use it. Use `view-state` instead, or other macros.
`show-info` `$res`	no	Deprectated. Don't use it. Use `view-info` instead, or other macros.
`show` `$res`	no	Deprectated. Don't use it. Use or implement some macros instead.
`show-errors` `$res`	no	Deprectated. Use `view-the-err-msg` or `view-resource-err` similar macros.
`cat` `$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

4.3.2. Setting Parameters

4.3.2.1. Per-Resource Parameters

Command / Params	Cmp	Description
`set-emergency-limit` `$res` *n*	no	The argument *n* must be percentage between 0 and 100 %. When the remaining store space in `/mars/` undershoots the given percentage, the resource will go *earlier* into emergency mode than by the global computation described in section 3.4. 0 means unlimited.
`get-emergency-limit` `$res`	no	Inquiry of the preceding value.
Command / Params	Cmp	Description

4.3.2.2. 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

4.3.3. Waiting

Command / Params	Cmp	Description
Command / Params	Cmp	Description

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 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 (typicially, 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

4.3.4. Low-Level Helpers

These commands are for experts and advanced sysadmins only. The interface is not stable, i.e. the meaning may change at any time.

Command / Params	Cmp	Description
`set-link`	no	RTFS.
`get-link`	no	RTFS.
`delete-file`	no	RTFS.
Command / Params	Cmp	Description

4.3.5. 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

4.3.6. Forbidden Commands (from DRBD)

These commands are not implemented because they would be dangerous in MARS context:

Command / Params	Cmp	Description
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`invalidate-remote`	no	This is 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 logfile 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

4.4. 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 Light, but may also change between releases without notice.

This section describes only those tweaks intended for sysadmins, not those for developers / very deep internals.

4.4.1. 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)

4.4.1.1. 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 3.4 will appear. Use `*.log` only for a **limited time**, and **only for debugging!**

4.4.1.2. 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.

4.4.1.3. 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.

4.4.2. 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 Light 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.

5. 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.

5.1. 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 Light 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. Specifially, 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 Light 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.

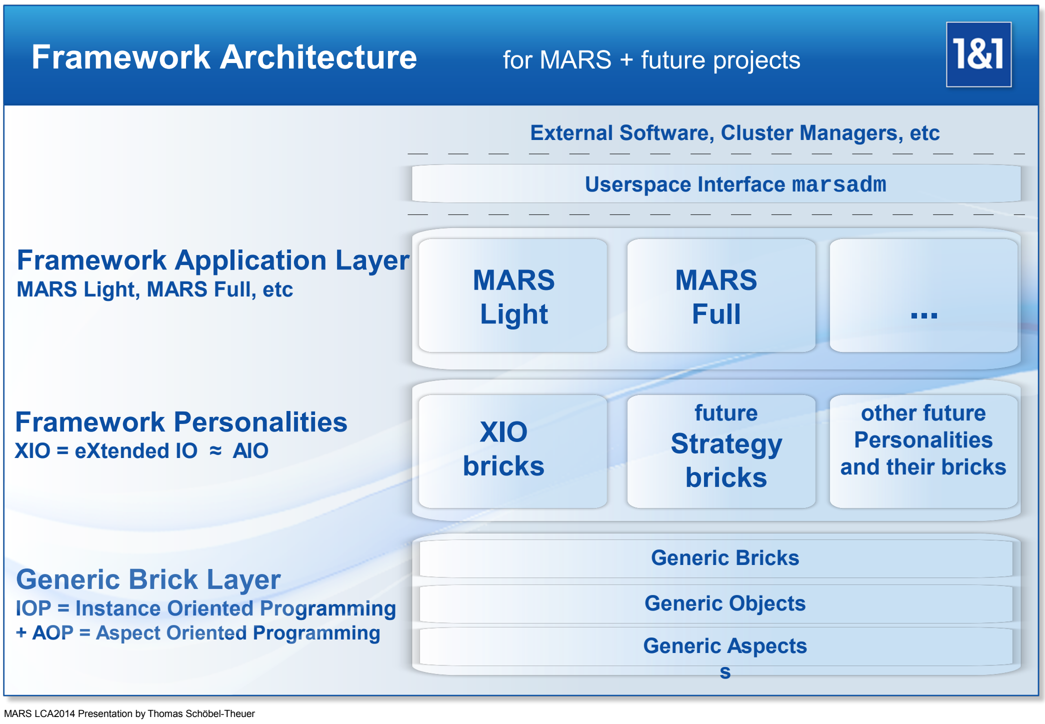
[8] See http://athomux.net/papers/paper_inst2.pdf

[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 Light 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.

5.2. Architecture Overview

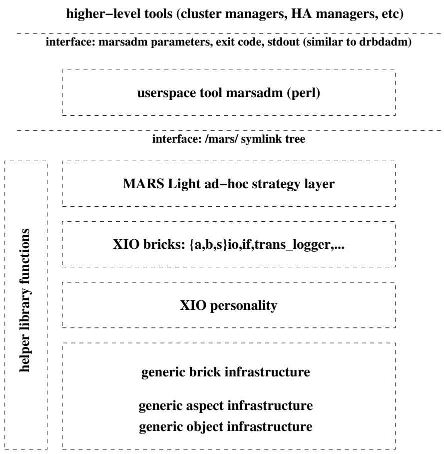


5.3. Some Architectural Details

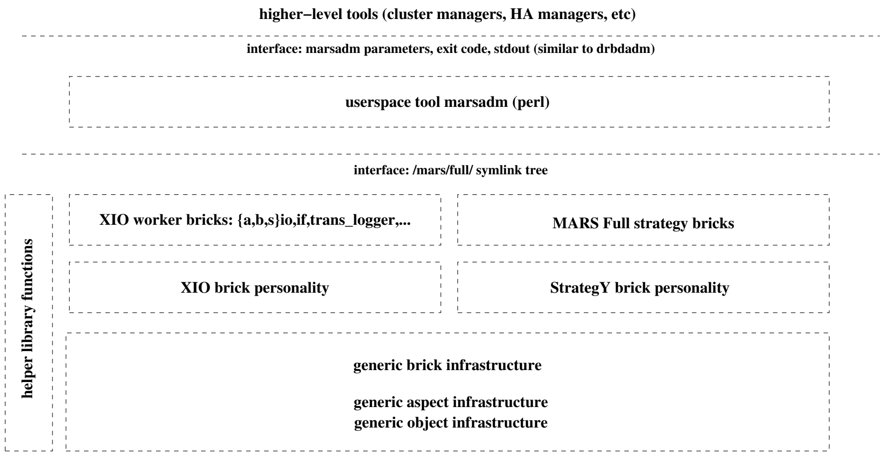
The following pictures show some "zones of responsibility", not necessarily a strict hierarchy (although Dijkstra's famous layering rules from THE are tried to be respected as much as possible). The construction principle follows the concept of **Instance Oriented Programming** (IOP) described in `http://athomux.net/papers/paper_inst2.pdf`. Please note that MARS Light is only instance-*based*[10], while MARS Full is planned to be fully instance-*oriented*.

5.3.1. MARS Light 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.



5.3.2. MARS Full Architecture (planned)



5.4. 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 3.2 and 3.3. 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.

5.4.1. Documentation of the MARS Light Symlink Tree

5.5. XIO Worker Bricks

5.6. StrategY Worker Bricks

NYI

5.7. The XIO Brick Personality

5.8. The Generic Brick Infrastructure Layer

5.9. The Generic Object and Aspect Infrastructure

A. Technical Data MARS Light

MARS Light 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 Light 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. GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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.