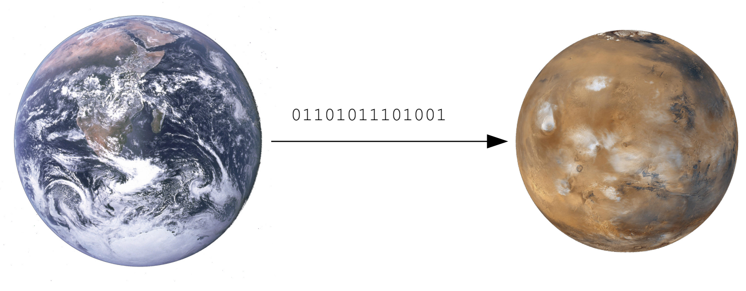
MARS Manual

Multiversion Asynchronous Replicated Storage



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

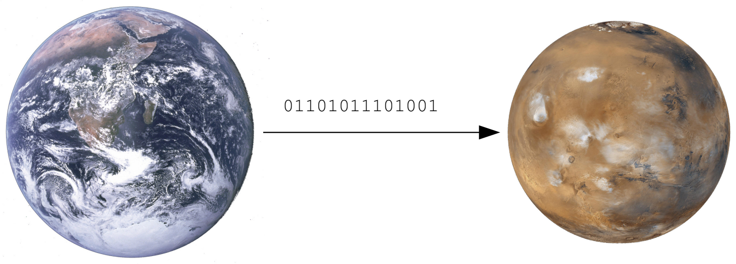
Version 0.7 (incomplete)

Abstract

`MARS` Light is a block-level storage replication system for long distances under GPL. It runs as a Linux kernel module. The sysadmin interface is similar to DRBD, 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stronger **consistency guarantees**: in case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but never become inconsistent.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**).

By default, MARS Light works **asynchronously**. Therefore, application performance is completely decoupled from any network problems.



Contents

1. 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`.

1.1. 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.

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, 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.

Use `ext3` or `ext4` for `/mars/`. Don't use `xfs`[3].

[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.

[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

1.2. 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).

1.2.1. 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. Hopefully, the patch will be integrated upstream some day.

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 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.

1.2.2. 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 `ext3` or `ext4` filesystem on your separate disk / RAID set (see description in section Preparation: What you Need).
4. On each node, mount that filesystem to `/mars/`. It is advisable to add an entry to `/etc/fstab`.
5. 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. 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.

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.

[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.

1.3. 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.

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 whatever.

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

4. Optionally: if you create a *new* filesystem on `/dev/mars/mydata` *after(!)* having created the MARS resource, you may skip the fast fullsync phase at all, because the old content of `/dev/mars/mydata` is just garbage not used by the freshly created filesystem. Just 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

[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.

the data! Otherwise, you are almost *guaranteed* to have produced harmful inconsistencies. If you accidentally issued `fake-sync`, you may startover the full sync at your secondary side at any time by saying `marsadm invalidate mydata` (analogously to the corresponding DRBD command).

1.4. Keeping Resources Operational

1.4.1. 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[8], 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* applied, on *all* relevant secondaries). All superfluous logfiles are then deleted, including all copies on all secondaries.

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

   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 2.4.

1.4.2. Switch Primary / Secondary Roles

MARS Light distinguishes betwenn *intended* and *emergency* switching. This distinction is necessary due to subtle differences in the communication architecture (synchronous communication vs synchronous communication, see sections 2.2 and 2.3).

1.4.2.1. Intended Switching

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

- `marsadm primary mydata`

on your formerly secondary node. Precondition is that the old primary must not use its `/dev/mars/mydata` device any longer. If that precondition is violated, `marsadm primary` refuses to run.

The reason for this check is that we want to avoid split brain situation as well as we can. Therefore, we distinguish between *intended* and *emergeny* switching. Intended switching will try to avoid split brain as best as it can.

Notice that the usage check for `/dev/mars/mydata` is based on the open count transferred from another cluster node. Since MARS is operating asynchronously (in contrast to DRBD), it

[8]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*.

may take some time our node knows that the device is no longer used at another node. This can lead to a race condition if you automate an intended takeover with a script like `ssh A ‘‘umount /dev/mars/mydata’’; ssh B ‘‘marsadm primary mydata’’` because your second ssh command may be faster than the internal MARS symlink tree propagation (cf section 2.3). In order to prevent such races, you should use the command

- `marsadm wait-umount mydata`

on node B before trying to become primary. The script should look like `ssh A ‘‘umount /dev/mars/mydata’’; ssh B ‘‘marsadm wait-umount mydata && marsadm primary mydata’’`.

1.4.2.2. Emergency Switching

In case the connection to the old primary is lost, we just don't know anything about its *current* state (which may deviate from its *last known* state). The following variant will skip almost all checks and tell your node to become primary forcefully:

- `marsadm primary mydata --force`

This may lead to split brain if the old primary continues to operate on its local `/dev/mars/mydata` device. Therefore, you should do this only after

1. `marsadm primary` without `--force` has failed, and

2. you are sure you really want to switch even when that leads to a split brain. Notice that in case of connection loss you might not be able to reliably detect whether a split brain will actually result, or not.

In contrast to DRBD, split brain situations are handled differently by MARS Light. When two primaries are active at the same time, each of them writes into different logfiles `/mars/resource-mydata/log-0000` and `/mars/resource-mydata/log-000000001-B` where the *origin* host is always recorded in the filename. Therefore, both nodes *can* run in primary mode indepently from each other, at least for some time. They may even `log-rotate` independently from each other. However, any other secondary node gets into some problems then: it simply does not not know whom it should follow.

Split brains are detected *passively* by secondaries. Whenever a secondary detects that somewhere a split brain has happend, it just refuses to to apply any logfiles behind the split point. This means that its local disk state will remain consistet, but outdated which respect to any of the split brain versions.

1.4.3. Split Brain Resolution

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

1.4.3.1. Destroying a Split Brain Version

Do the following steps:

1. Manually check which version is the “right” one.

2. It may happen that the “right” version is not the version which is currently designated as primary for the whole cluster. In this case (or if you want to ensure nothing can go wrong), say `marsadm primary mydata --force` on the node you want to become the surviving “right” version. This should always work, even it is already the “right” primary.

3. On each other (non-”right”) version, say `marsadm invalidate mydata`. Do this only when split brain has actually occurred at that node; otherwise an unnecessary full sync will start which was not really needed.

1.4.3.2. Keeping a Split Brain Version

Do the following steps:

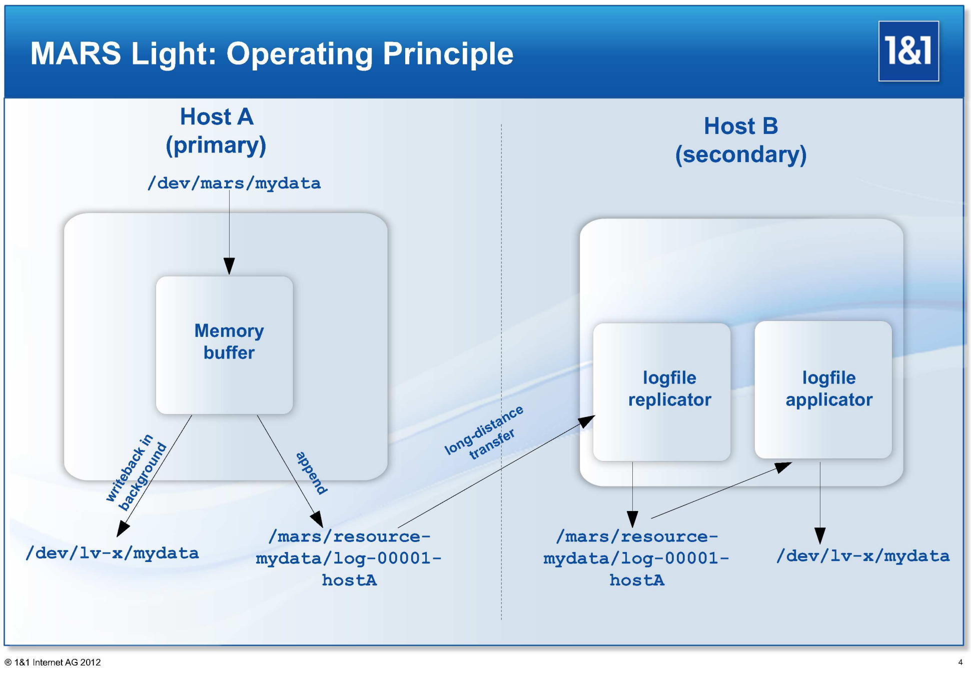
1. Manually check which version is the "right" one.
2. It may happen that the "right" version is not the version which is currently designated as primary for the whole cluster. In this case (or if you want to ensure nothing can go wrong), say `marsadm primary mydata --force` on *exactly that* node which you want to become the surviving "right" version.
3. On the non-right version which you want to retain, umount your `/dev/mars/mydata`.
4. Wait until it reaches secondary state and its local logfile has been fully applied. This should happen because of step 2.
5. `marsadm leave-resource mydata`
6. Check that the underlying disk `/dev/lv-x/mydata` is really usable, e.g. by test-mounting it. If all is OK, don't forget to umount it before proceeding with the next step.
7. 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).

2. 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.

2.1. The Transaction Logger



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

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

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

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

The temporary memory buffer serves multiple purposes:

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

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

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

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

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

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

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

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

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

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

In mathematics, the property that you can apply your logfile twice to your data (or even as often as you want), is called **idempotence**. This is a very desirable property: it ensures that nothing goes wrong when applying "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), applying 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 apply them to 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 applies the logfiles to 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 3.2.2). This may be *exploited*: for example, you may replicate your logfiles as soon as possible (to protect against catastrophic failures), but deliberately wait one hour until it is applied (under regular circumstances). If your data inside your filesystem `/mydata/` at the primary site is accidentally destroyed by `rm -rf /mydata/`, you have an

[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)".

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.

2.2. 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 is 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**.

In order to implement that consistency model, 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).

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

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

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.

2.3. The Symlink Tree

The `/mars/` filesystem contains not only transaction logfiles, but also acts as a generic storage for (persistent) state information. Information is stored in symlinks. Symlinks are "misused[5]" in order to represent some `key -> value` pairs.

(Almost) all symlinks appearing in the `/mars/` directory tree are replicated thoughout the whole cluster. Thus the `/mars/` directory forms some kind of *global namespace*.

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

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

timestamp comparison protocol: whenever node B notices that A has a *newer* version of some symlink (according to the Lamport timestamp), it will replace its elder version by the newer one. The opposite does *not* work: if B notices that A has an elder version, just nothing happens. This way, the timestamps of symlinks can only progress in forward direction, but never backwards in time.

As a consequence, symlink updates made "by hand" via `ln -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 4.2.

2.4. 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*.

2.4.1. Countermeasures

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 1.1). 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!

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.

[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.

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

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

If you just wanted to freeze a mirror at an outdated state for a very long time, you simply *cannot* do that without causing infinite growth of space consumption in `/mars/`. Therefore, a `marsadm leave-resource $res` at *exactly that(!)* secondary site where the mirror is frozen, can also work miracles. If you want to automate this in userspace, 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, automatic`/mars/`ally 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. 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 2.4.2). 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.

2.4.2. 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 applying 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 apply 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 2.1). 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.

3. On the secondaries, use `marsadm invalidate $res` in order to get your outdated mirrors uptodate. 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 that after the cleanup). After the initial full sync has finished there, do an `marsadm invalidate $res` on the outdated mirror. This way, you will always retain at least one consistent mirror somewhere. After all is up-to-date, you can delete the superfluous mirror by `marsadm leave-resource $res` and reclaim the disk space from its underlying disk.

3. The Sysadmin Interface `marsadm`

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 2.2 and 2.3. 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
`--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!
`--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 $seconds is 0, the command will not wait in case any precondition is not met, und abort without performing an action.. The default timeout is 5s.
`--host=$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, and some local information like sizes of physical devices (e.g. disks) is not present in the symlink tree. Use at your own risk!
`--verbose`	no	Some (few) commands will become more speaky.
Option	Cmp	Description

3.1. Cluster Operations

Command / Params	Cmp	Description
`create-cluster`	no	Precondition: the `/mars/` filesystem must be mounted and it must be empty. 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.
`join-cluster` `$host`	no	Precondition: the `/mars/` filesystem must be mounted and it must be empty. The kernel module must not be loaded. The cluster must have been already created at another node `$host`. A working ssh connecttion to $host must exit (without password). `rsync` must be installed at all cluster nodes. Postcondition: the initial symlink tree `/mars/` is replicated from the remote host `$host`, and the local host has been added as another cluster member. This must be called exactly once at every initial secondary.
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 kernel module must be loaded. The local node must no longe be member of any resource (see `marsadm leave-resource`). 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 local `/mars/` filesystem may be finally destroyed. In case of an eventual node loss (e.g. fire, water, ...) this may be used. on another node $helper in order to finally remove $damaged from the cluster via the command `marsadm leave-cluster --host=$damaged --force`.
`wait-cluster`	no	See section 3.3.2.
Command / Params	Cmp	Description

3.2. 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.

3.2.1. Resource Creation / Deletion / Modification

Command / Params	Cmp	Description
`create-resource` `$res` `$disk_dev` `[$mars_name]` `[$size]`	no	Precondition: the resource argument `$res` must not denote an already existing resource in the cluster. The argument `$disk_dev` must denote 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 `$res` must denote an already existing resource in the cluster (i.e. its symlink tree information must have been received). The local node must not be already member of that resource. The argument `$disk_dev` must denote a usable 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 if $disk_dev is strictly greater than the size of the resource, you will unnecessarily waste some space..
`leave-resource` `$res`	no	Precondition: the local node must be a member of the resource `$res`; its current role must be secondary. Thes disk must be detatched. Postcondition: the local node is no longer a member of `$res`. Notice: as a side effect for other noder, their log-delete .may now become possible, since the current node does no longer count as a candidate for logfile application. In case of an eventual node loss (e.g. fire, water, ...) this 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`.
`wait-resource` `$res` `{is-,}{attach,` `primary,` `device}{-off,}`	no	See section 3.3.2.
Command / Params	Cmp	Description

3.2.2. Operation of the Resource

Common preconditions are the preconditions from section 3.2, 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
`attach` `$res`	yes	Precondition: the local disk belonging to $res is not in use by anyone else. Postcondition: MARS uses the local disk and is able work with it (e.g. apply logfiles to it). Note: the local disk is opened in exclusive read-write mode. This should protect against most common misuse, such as opening the disk in parallel to MARS.
`detach` `$res`	yes	Precondition: the local host is in secondary role, `pause-sync` and `pause-replay` have been given.. Postcondition: the local disk belonging to $res is no longer in use. WARNING! After this, you might use the underlying disk for other purposes, such as test-mounting it in *readonly* mode.. **Don't modifiy** its contents in any way! Not even by an `fsck`! Otherwise, you will have inconsistencies *guaranteed*. MARS has no way for knowing of any modifications to your disk when not written via `/dev/mars/*`. In case you accidentally modified the underlying disk at the *primary* side, you may choose to resolve the inconsistencies by `marsadm invalide $res` on *each* secondary.
`pause-sync` `$res`	partly	Equivalent to `pause-sync-local`.
`pause-sync-local` `$res`	partly	Precondition: none additionally. Postcondition: any sync operation targeting the local disk (when not yet completed) is paused after a while. When completed, this operation will remember the switch state forever and become relevant if a sync is needed again (e.g. `invalidate` or `resize`).
`pause-sync-global` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`resume-sync` `$res`	partly	Equivalent to `pause-sync-local`.
`resume-sync-local` `$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` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`pause-replay` `$res`	partly	Equivalent to `pause-replay-local`.
`pause-replay-local` `$res`	partly	Precondition: must be in secondary role. Postcondition: any local apply operations of transaction logfiles to the local disk are paused at their current stage. This works independently from `{dis,}connect`.
`pause-replay-global` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`resume-replay` `$res`	partly	Equivalent to `pause-replay-local`.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`resume-replay-local` `$res`	partly	Precondition: must be in secondary role. Postcondition: any (parts of) locally existing transaction logfiles (whether replicated from other hosts or produced locally) are started for apply to the local disk, as far as they have not yet been applied.
`resume-replay-global` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`connect` `$res`	partly	Equivalent to `connect-local`.
`connect-local` `$res`	partly	Precondition: must be in secondary role. Postcondition: any (parts of) transaction logfiles which are present at another primary host will be transferred to the local `/mars/` storage as far as not yet present locally. This works independently from `{pause,resume}-replay`.
`connect-global` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`disconnect` `$res`	partly	Equivalent to `disconnect-local`.
`disconnect-local` `$res`	partly	Precondition: must be in secondary role. Postcondition: any transfer of (parts of) transaction logfiles which are present at another primary host to the local `/mars/` storage are paused at their current stage. This works independently from `{pause,resume}-replay`.
`disconnect-global` `$res`	partly	Like `*-local`, but operates on all members of the resource.
`up` `$res`	yes	Equivalent to `attach` followed by `connect` followed by `resume-replay` followed by `resume-sync`.
`down` `$res`	yes	Equivalent to `pause-sync` followed by `disconnect` followed by `pause-replay` followed by `detach`.
`primary` `$res`	almost	Precondition: all relevant transaction logfiles must be either already locally present, or be fetchable (see `connect` and `resume-replay`). When another host is currently primary, it must match the preconditions of `marsadm secondary`. Postcondition: `/dev/mars/$dev_name` appears and is usable; the current host is in primary role. When another host is currently primary, it is first asked to become secondary, and waited for to actually be secondary. After that, the local host is asked to become primary. Before actually becoming primary, all relevant logfiles are applied. Only after that, `/dev/mars/$dev_name` will appear. When netwrk transfers of the symlink tree are very slow (or currently impossible), this command may take a very long time. Therefore `--force` will skip all checks depending on remote state. In case a split brain is detected, the local host will refuse to become primary without `--force`.
`secondary` `$res`	almost	Precondition: the local `/dev/mars/$dev_name` is no longer in use (e.g. umounted). Postcondition: `/dev/mars/$dev_name` has disappeared; the current host is in secondary role.
`wait-umount` `$res`	no	See section 3.3.2.
`resize` `$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

3.2.3. 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

3.2.4. Consistency Operations

Command / Params	Cmp	Description
`invalidate` `$res`	no	Precondition: the local node must be in secondary role at `$res`. Postcondition: the local disk is marked as inconsistent, and a fast full-sync 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 1.3 Use this only *after* having created a fresh filesystem inside `/dev/mars/$res`.
`set-replay`	no	⚠ ONLY FOR ADVANCED HACKERS WHO KNOW WHAT YOU ARE DOING! This command is deliberately not documented. You need the competence level RTFS (“read the fucking sources”).
Command / Params	Cmp	Description

3.3. Further Operations

3.3.1. Inspection Commands

Command / Params	Cmp	Description
`role`	no	
`state`	no	
`cstate`	no	NYI
`dstate`	no	NYI
`status`	no	NYI
`show-state`	no	
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`show-info`	no	
`dstate`	no	
`show`	no	
`show-errors`	no	
`cat`	no	
Command / Params	Cmp	Description

3.3.2. Waiting

Command / Params	Cmp	Description
`wait-cluster`	no	Precondition: the `/mars/` filesystem must be mounted and it must contain a valid MARS symlink tree produced by the other `marsadm` commands. The kernel module must be loaded. Postcondition: none. Wait until *all* nodes in the cluster have sent a message, or until timeout. The default timeout is 30 s (exceptionally) and 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

3.3.3. Low-Level Helpers

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

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

3.3.4. 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	
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`outdate`	no	
`adjust`	yes	Implemented as NOP (not necessary with MARS).
`hidden-commands`	no	
Command / Params	Cmp	Description

3.3.5. Forbidden Commands (from DRBD)

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

Command / Params	Cmp	Description
`invalidate-remote`	no	This 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, MARS `invalide` will do the same as DRBD verify followed by DRBD resync, i.e. `marsadm invalidate` will automatically correct any found errors; note that the fast-fullsync algorithm of MARS will minimize network traffic.
Command / Params	Cmp	Description

3.3.6. Deprecated Operations

4. 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 (e.g., you may stop after reading the documentation on symlink trees or even use that documentation like an encyclopedia).

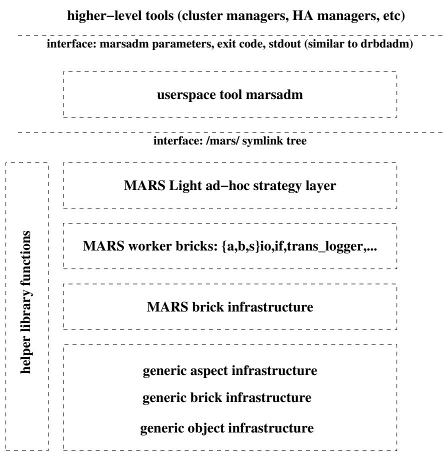
If you are a kernel developer and want to contribute code to the MARS community, please read it (almost) all. Due to the top-down organization, sometimes you will need to follow some forward references in order to understand details. Therefore I recommend reading this chapter twice in two different reading 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 exploit your knowlegde from the first pass for a deeper understanding of the details.

Alternatively, you may first read the first section 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.

4.1. General Architecture

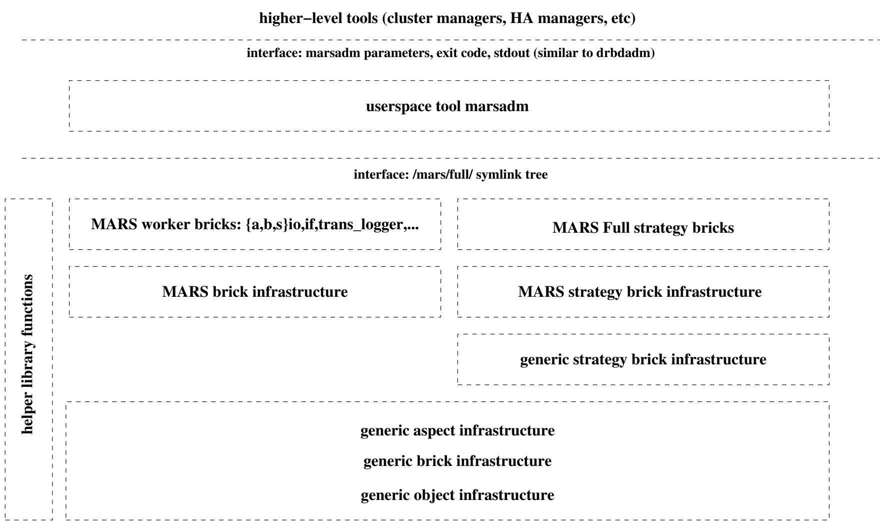
The following pictures show some "zones of responsibility", not necessarily a strict hierarchy (although Dijkstra's famous layering rules from THE are tried to be respected as much as possible). The construction principles follow the concepts of **Instance Oriented Programming** (IOP) described in `http://athomux.net/papers/paper_inst2.pdf`. Please note that MARS Light is only instance-based[1], while MARS Full is planned to be fully instance-oriented.

4.1.1. MARS Light Architecture


higher−level tools (cluster managers, HA managers, etc)
interface: marsadm parameters, exit code, stdout (similar to drbdadm)
userspace tool marsadm
interface: /mars/ symlink tree
MARS Light ad−hoc strategy layer
MARS worker bricks: {a,b,s}io,if,trans_logger,...
MARS brick infrastructure
generic aspect infrastructure
generic brick infrastructure
generic object infrastructure
helper library functions

4.1.2. MARS Full Architecture (planned)

[1]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.


higher–level tools (cluster managers, HA managers, etc)
interface: marsadm parameters, exit code, stdout (similar to drbdadm)
userspace tool marsadm
interface: /mars/full/ symlink tree
helper library functions
MARS worker bricks: {a,b,s}io,if,trans_logger,...
MARS Full strategy bricks
MARS brick infrastructure
MARS strategy brick infrastructure
generic strategy brick infrastructure
generic aspect infrastructure
generic brick infrastructure
generic object infrastructure

4.2. 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 2.2 and 2.3. 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.

4.2.1. Documentation of the MARS Light Symlink Tree

4.3. MARS Worker Bricks

4.4. MARS Strategy Bricks

4.5. The MARS Brick Infrastructure Layer

4.6. The Generic Brick Infrastructure Layer

4.7. The Generic Object and Aspect Infrastructure

A. GNU Free Documentation License

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.