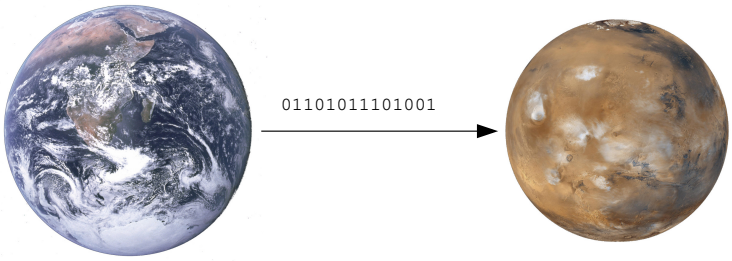
MARS User Manual

Multiversion Asynchronous Replicated Storage



Thomas Schöbel-Theuer (tst@1und1.de)

Version 0.1a-141

Copyright (C) 2013-16 Thomas Schöbel-Theuer
Copyright (C) 2013-16 1&1 Internet AG (see http://www.1und1.de shortly called 1&1 in the following).
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".

Abstract

MARS is a block-level storage replication system for long distances / flaky networks under GPL.

It is a key component for achieving **geo-redundancy** under Linux, for example Disaster Recovery (DR) at datacenter granularity, and/or Location Transparency (LT) at VM / LV granularity.

It can help to increase **reliability** via Sharding, and to **save cost** by optional support for local storage in addition to network storage.

It eases **load balancing** and **background migration of data**, even over long distances.

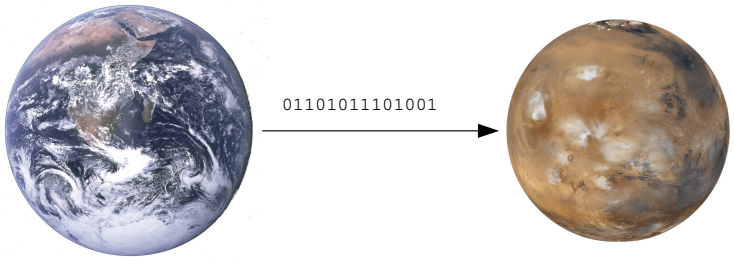
MARS runs as a Linux kernel module. The sysadmin interface is similar to DRBD, but its internal engine is completely different from DRBD: it works with transaction logging, similar to some database systems.

Therefore, MARS can provide stronger consistency guarantees. In case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but will not become inconsistent. In contrast to DRBD, MARS preserves the order of write operations even when the network is flaky (Anytime Consistency).

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

MARS supports a new method for building Cloud Storage / Software Defined Storage, called **LV Football**. It comes with some automation scripts, enabling a similar functionality than Kubernetes, but devoted to stateful LVs over **virtual LVM pools** in the petabytes range.

MARS is in production since 2014, and on thousands of Linux servers replicating petabytes of data.



Many thanks for constructive feedback which helped to improve this document series and related material like presentation slides:

- Philipp Reisner from Linbit
- Ewen NcNeill and Simon Lyall from the Australian / New Zealand Linux community
- Jens Clever and Jörg Mann, external freelancers working at 1&1
- Anders Henke and Christian Albert from 1&1 Ionos
- Olof Sandström-Herrera from Arsys

Please report any omissions in case I forgot somebody.

Preface

Introduction

MARS is a block-level storage replication system for long distances / flaky networks under GPL.

It is a key component for achieving **geo-redundancy** under Linux, for example Disaster Recovery (DR) at datacenter granularity, and/or Location Transparency (LT) at VM / LV granularity.

It can help to increase **reliability** via Sharding, and to **save cost** by optional support for local storage in addition to network storage.

It eases **load balancing** and **background migration of data**, even over long distances.

MARS 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 can provide stronger consistency guarantees. In case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but will not become inconsistent. In contrast to DRBD, MARS preserves the order of write operations even when the network is flaky (Anytime Consistency).

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

MARS supports a new method for building Cloud Storage / Software Defined Storage, called **LV Football**. It comes with some automation scripts, enabling a similar functionality than Kubernetes, but devoted to stateful LVs over **virtual LVM pools** in the petabytes range.

MARS is in production since 2014, and on thousands of Linux servers replicating petabytes of data.

Purpose

This document explains how to install, setup and run a storage replication system on a Linux based platform using MARS.

MARS is free software under GPL. The terms and conditions of the GPL apply. This documentation is *best effort* and may contain bugs (no warranty, according to the GPL).

Scope

The following topics are covered withing this document:

- preconditions: what you need.
- creating a Linux kernel module for MARS.
- creation of a MARS cluster.
- creation and operation of MARS resources.
- dynamic creation / deletion of additional replicas, and how migrate data this way.
- hints for monitoring.
- troubleshooting

Audience

This document is written for experienced sysadmins with working knowledge on the following methods and technologies:

- Setup and operation of LVM (Logical Volume Manager) under Linux.
- Operation of storage systems.
- Ability to patch and to compile and install a customized Linux kernel. Most patches are already provided ready-to-use, so no full developer knowledge is needed.

How to use this document

Please start with the very short introduction chapter 1 Briefing: how MARS works.

If you want to install MARS, read HOWTO setup MARS.

If you just want to operate a MARS installation which is already set up, go on to HOWTO operation of MARS resources.

If you already have some experiences with MARS and just need some details about marsadm commands, chapter Working with `marsadm` commands is a kind of "reference" for you.

Users who want to go deeper into tuning should read Tuning, tips and tricks.

Automation via interfacing to `systemd` is described in Advanced users: automation and the macro processor, as well as tips for writing your own automation scripts.

Appendix A on page 103 answers a few FAQs by the inventor of MARS.

Appendix B on page 105 contains some estimations for typical datacenter hardware, called Technical Data MARS.

Some non-standard expert tricks (e.g. for mass operation of thousands of instances) can be found in the following appendices.

Related documents

- `mars-architecture-guide.pdf`: explains usage scenarios.
- `football-user-manual.pdf`: for sysadmins and userspace developers who want to use Football.
- `mars-for-kernel-developers.pdf`: some infos for kernel developers.

Contents

1. Briefing: how MARS works **10**
- 1.1. Typical MARS replication setup . . . 10
- 1.2. The Transaction Logger . . . 11
- 1.3. The State of MARS . . . 12

2. HOWTO setup MARS **14**
- 2.1. Description: what you Need . . . 14
- 2.2. MARS Kernel Module . . . 16
- 2.3. Setup Primary and Secondary Cluster Nodes . . . 19
  - 2.3.1. Setup Hardware . . . 19
  - 2.3.2. Setup the Network . . . 19
- 2.4. Setup / Install OS . . . 20
  - 2.4.1. Setup LVM . . . 22
  - 2.4.2. Setup Cluster Nodes . . . 23
- 2.5. Setup Housekeeping Cron Job . . . 24
- 2.6. Creating and Maintaining Resources . . . 26

3. HOWTO operation of MARS resources **28**
- 3.1. Inspecting the State of MARS . . . 28
  - 3.1.1. Standard `marsadm view` . . . 28
- 3.2. Switch Primary / Secondary Roles . . . 33
  - 3.2.1. Intended Switching / Planned Handover . . . 34
  - 3.2.2. Forced Switching . . . 35
- 3.3. Split Brain Resolution . . . 37
- 3.4. Final Destruction of a Damaged Node . . . 39
- 3.5. Online Resizing during Operation . . . 40
- 3.6. Defending Overflow of `/mars/` . . . 41
  - 3.6.1. Countermeasures against overflow . . . 42
    - 3.6.1.1. Dimensioning of `/mars/` . . . 42
    - 3.6.1.2. Monitoring . . . 42
    - 3.6.1.3. Throttling . . . 43
- 3.7. Emergency Mode and its Resolution . . . 44

4. Working with `marsadm` commands **47**
- 4.1. Cluster Operations . . . 49
- 4.2. Resource Operations . . . 52
  - 4.2.1. Resource Creation / Deletion / Modification . . . 53
  - 4.2.2. Operation of the Resource . . . 55
  - 4.2.3. Logfile Operations . . . 60
  - 4.2.4. Consistency Operations . . . 60
- 4.3. Further `marsadm` Operations . . . 60
  - 4.3.1. Inspection Commands . . . 60
  - 4.3.2. Setting Parameters . . . 61
    - 4.3.2.1. Per-Resource Parameters . . . 61
    - 4.3.2.2. Global Parameters . . . 61
  - 4.3.3. Waiting . . . 62
  - 4.3.4. `systemd` Control Commands . . . 63
  - 4.3.5. Low-Level Expert Commands . . . 64
  - 4.3.6. Senseless Commands (from DRBD) . . . 65
  - 4.3.7. Forbidden Commands (from DRBD) . . . 65

5. Tuning, tips and tricks **66**
5.1. IO Performance Tuning . . . 66
5.2. Data Compression and Checksumming (Digests) . . . 70
5.2.1. Network Transport Compression . . . 71
5.2.2. Logfile Payload Compression . . . 71
5.2.3. Logfile Payload Digests . . . 71
5.2.4. Network Payload Digests . . . 72
5.3. The `/proc/sys/mars/` and other Expert Tweaks . . . 72
5.3.1. Tuning Network Performance . . . 72
5.3.2. Syslogging . . . 74
5.3.2.1. Logging to Files . . . 74
5.3.2.2. Logging to Syslog . . . 74
5.3.2.3. Tuning Verbosity of Logging . . . 75
5.3.3. Tuning the Sync . . . 75
5.3.4. Lowlevel TCP Tuning (Networking Experts Only) . . . 75

6. Advanced users: automation and the macro processor **77**
6.1. The `systemd` Template Generator . . . 77
6.1.1. Why `systemd`? . . . 77
6.1.2. Execution Model of systemd and marsadm . . . 78
6.1.3. Working Principle of the Template Generator for `systemd` . . . 80
6.1.4. Template Markers . . . 81
6.1.5. Special `.script` Pseudo Units . . . 82
6.1.6. Example `systemd` Templates . . . 85
6.1.7. Fully Automatic Handover using `systemd` . . . 87
6.2. The macro processor . . . 88
6.2.1. Predefined Primitive Macros . . . 89
6.2.1.1. Intended for Humans . . . 89
6.2.1.2. Intended for Scripting . . . 91
6.3. Creating your own Macros . . . 95
6.3.1. General Macro Syntax . . . 96
6.3.2. Calling Builtin / Primitive Macros . . . 97
6.3.3. Predefined Variables . . . 101
6.4. Scripting Advice . . . 102

A. FAQ **103**

B. Technical Data MARS **105**

C. HISTORIC Guide for Midnight Problem Solving **106**
C.1. Inspecting the State of MARS . . . 106
C.2. Replication is unexpectedly Stuck . . . 107
C.3. Standard Resolution of Emergency Mode and Split Brain . . . 108
C.4. Alternative Resolution of Split Brain / Emergency Mode / Defective Hardware . . . 109
C.5. Handover of Primary Role . . . 111
C.6. Failover = Emergency Switch of Primary Role . . . 112

D. HISTORICAL Methods for Split Brain Resolution **114**

E. Alternative De- and Reconstruction of a Damaged Resource **115**

F. Cleanup in case of Complicated Cascading Failures **116**

G. Experts only: Special Trick Switching and Rebuild **118**

H. Creating Backups via Pseudo Snapshots **120**

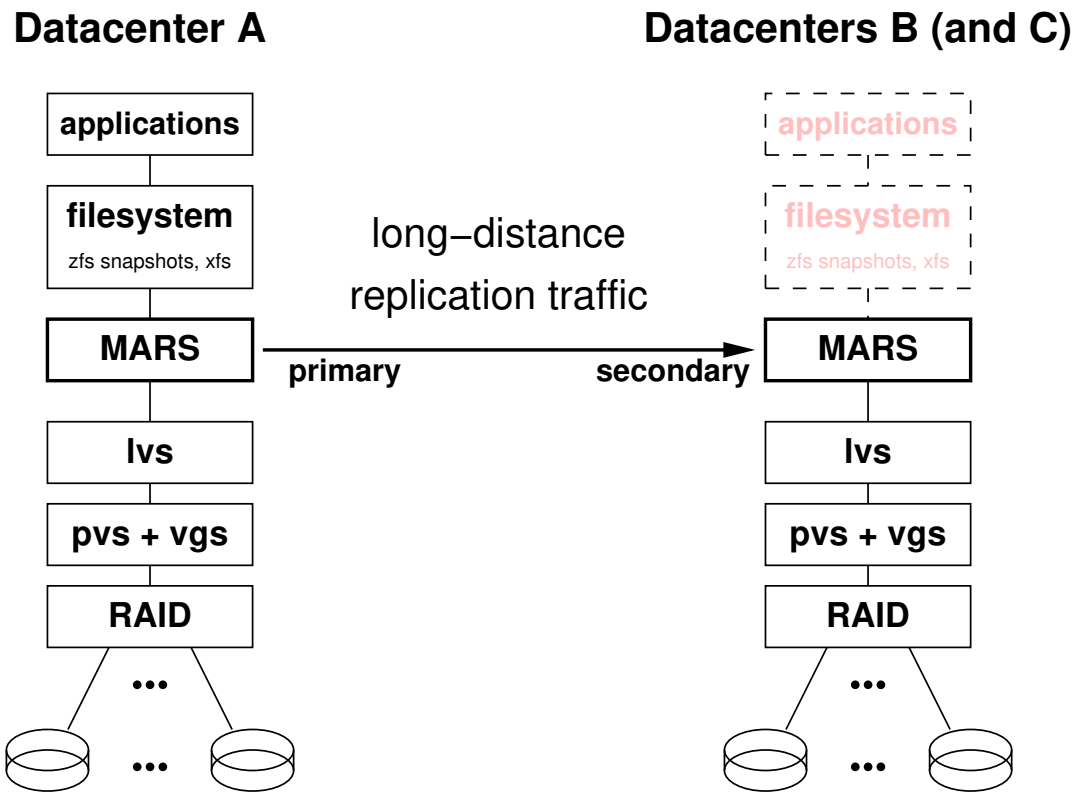
I. Command Documentation for Userspace Tools **121**
I.1. `marsadm --help` . . . 121

J. GNU Free Documentation License 134

1. Briefing: how MARS works

1.1. Typical MARS replication setup

Typical recommended usage is replication of multiple Logical Volumes (LVs) directly at bare metal (never inside of VMs), similar to DRBD:



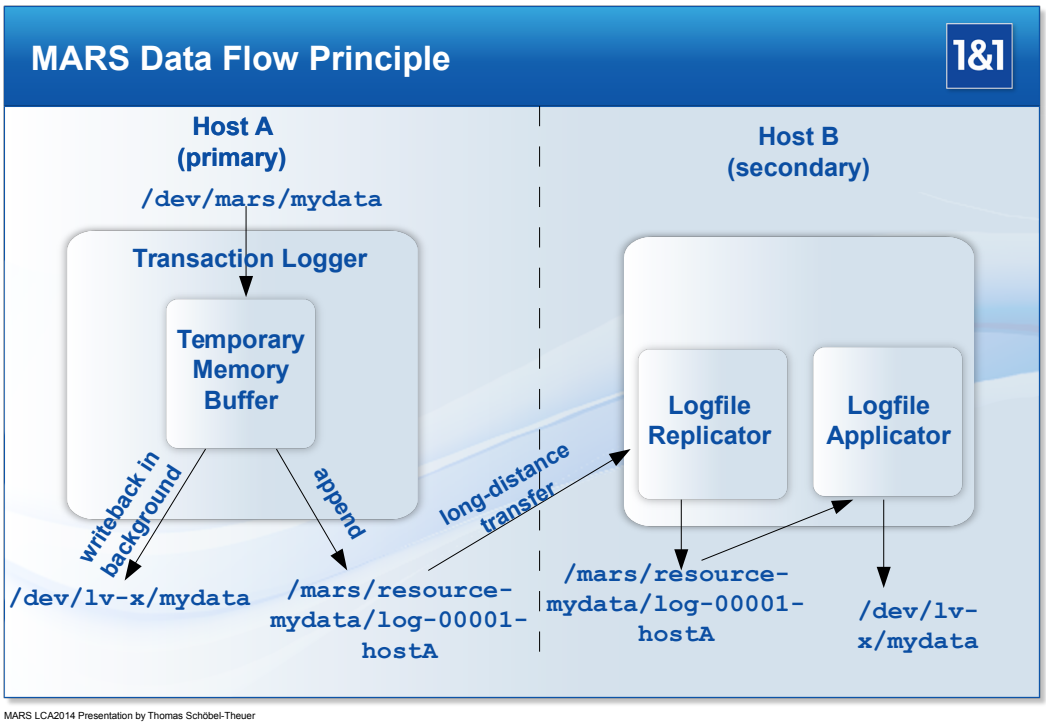
At the primary (active) side A, the applications are running. At each of the secondary (passive) sides B and C, only the underlying LV replicas are updated via the replication network traffic. The filesystem is not mounted at any secondary side, and the applications are not running there. However, the roles may be switched at any time, and then the application will for example run at Datacenter B in primary role, while the corresponding LV replicas will then be in secondary (passive) role at datacenters A and C.

An advantage of multiple LV replication is that primary and secondary roles can be ***individually* switched at runtime**. For example, if you have 10 LVs in each of your servers, 6 LVs may currently run in datacenter A in primary role, while the other 4 LVs are running in datacenter B, while datacenter C is dimensioned for less CPU power, and is mainly intended for additional "emergency backup" replicas. For instance, such a 3-datacenter configuration may be used for load balancing during overload peaks, or for switchover due to kernel security updates, and much more.

Further setups are also possible. For example, you might replicate physical disks. However, this would be less flexible because your volumes must then uniformly run in the *same* datacenter at the same time.

In addition, MARS' replicated block devices may be exported via iSCSI or other protocols. Filesystems residing on top of MARS may be exported via NFS or glusterfs, etc. For more details, please consult `mars-architecture-guide.pdf`.

1.2. The Transaction Logger



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

This is similar in concept to some database systems (c.f. MySQL replication), 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.

Write requests directed at the virtual block device `/dev/mars/mydata` are first buffered in a *temporary* kernel memory buffer.

The temporary kernel memory buffer serves multiple purposes:

- It keeps track of the **order** of write operations.
- Additionally, it keeps track of the positions in the underlying LV `/dev/lv-x/mydata`. In particular, it detects when the same block is overwritten multiple times.
- Writeback to the underlying LV may occur in a different order than submission order. On magnetic disk media, this may lead to a noticeable **performance boost**, as shown in section 5.1.
- During pending write operations, any concurrent reads from the same locations are served from the temporary memory buffer.

After the write has been buffered in the temporary memory buffer, the internal transaction logger kernel thread 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 LV, 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 an MD5 checksum or compression information.

Once the log entries[1] have been written through to the `/mars/` filesystem via `fsync()`, the

[1] 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. At least on magnetic media, appending to a logfile is almost purely sequential IO, it runs much faster than random IO.

application waiting for the write operation at `/dev/mars/mydata` is signalled that the write was successful[2].

This usually happens *before* the writeback to the underlying LV `/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 will be reconstructed from there after restart from power loss (Recovery phase). This is similar to Recovery of database systems after unexpected power loss.

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. Thus any temporary inconsistency will be *healed* [3]by logfile application. Good news for desperate sysadmins forced to work with old or flaky hardware!

The basic idea of asynchronous replication by MARS is rather simple: just transfer the logfiles to your secondary nodes, and replay them onto their LV replica (aka copy of the disk data, aka mirror).

Replay is always in the same write 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**[4]**.**

As you can see in the picture, the logfile transfer process is *independent* from the replay process. 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 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*[5] 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 are planned to also allow "fast-backward" rewinding, of course at some cost.

1.3. 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.

[2]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 is *sorted* according to ascending sector ´numbers. This reduces 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.

[3]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 even **heal** some defects.

[4]Your secondary nodes are always **strictly consistent** in themselves. Notice that this kind of consistency is a *local* consistency model. At global level, MARS is **eventually consistent**. Strict global consistency is generally not possible over long distances. Reasons are (1) Einstein's law (speed of light), and (2) the CAP theorem and its sister theorems. The front-cover pictures showing the planets Earth and Mars tries to lead your imagination away from global consistency models, in order to prepare you mentally for local consistency as in "MARS Think$^{(tm)}$".

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

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[6].

[6] 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.

2. HOWTO setup MARS

This chapter is for impatient but experienced sysadmins. For more detailed information, refer to chapter Working with `marsadm` commands.

2.1. Description: what you Need

This section describes the hardware you will need to buy and deploy, and which software components to install. Step-by-step setup instructions are following in the next section (starting with section 2.2).

Typically, you will install MARS at many bare metal servers for replication of many LVs *between*[1] multiple datacenters. Do *not* use MARS inside of VMs (see explanation of Dijkstra's layering rules in `mars-architecture-guide.pdf`).

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

Here is a list of software to be installed at your servers (with distro-specific tools like `dpkg` / `aptitude` / `rpm` / `yum` / `zypper` / etc):

- `ssh`
- `ssh-agent` (such that `ssh root@hostA` will work without password)
- `rsync`
- `perl`
- `lvm`
- Further standard Linux tools like `modprobe`, typically already present at servers. When necessary, you typically can install them from typical OpenSource / Linux Distro repos.
- Only if you don't have an already pre-built MARS kernel module, and only at your workstation, not necessarily at your server: everything you need for compiling a customized kernel. Optionally, the tools for building a Debian or rpm package. Details are distro-specific.

In order to protect your server data from low-level disk failures, you should use a **hardware RAID controller with BBU**. Software RAID is currently *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).

For many application workloads, RAID-6 provides a good compromize between cost and performance. Reads are very fast due to RAID-6 striping, while the slow RAID-6 writes are partially compensated by the MARS kernel memory buffer (see section 5.1).

For almost double the cost per TiB, you can speed up write operations by RAID-10. However, checkout RAID-6 first. A good tool to measure your *real* application performance is `blktrace` plus blkreplay, see https://github.com/schoebel/blkreplay/raw/master/doc/blkreplay.pdf.

For much higher cost per TiB, typically by about a factor of 10, you can of course also use SSDs in place of HDDs. While relatively small-sized database workloads are nowadays typically on SSDs, big mass data is typically remaining on HDDs for cost reasons.

[1] Many other solutions, even from commercial storage vendors, will not work reliably over distances greater than ≈ 50 km, and/or when your network is not *extremely* reliable, and/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. More information can be found in `mars-architecture-guide.pdf`.

Typically, you should build more than one RAID set[2] if you have more than 12 to 15 spindles in total. Therefore, the step-by-step HOWTO from this best-effort manual will show you some examples with LVM striping over 2 physical volumes (PVs).

LVM is highly recommended[3] for maximum flexibility. When used in static space allocation mode (as opposed to thin provisioning mode), LVM involves no measurable overhead (within the measurement tolerances of `blkreplay`). Although LVM thin provisioning could potentially save some cost, it may lead to massive performance degradation as observed with certain types of application behaviour. In order to stay at the safe side of operations, you should dimension your RAID storage size accordingly.

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. An example: in 1&1 Shared Hosting Linux (ShaHoLin), we found that only few applications are writing more than 1 TB per day during ordinary[4] operations. Most are writing even less than 100 GB per day, because the observed average filesystem data change rate is only about 1% per day[5]. Of course, there exist other applications like backup where the write rate is much higher. Please try to determine your actual write rates from system tools like `sar`. Usually, you want to dimension `/mars/` such that you can survive a network loss lasting 3 days / about one weekend.

This can be achieved rather easily, in one of the following ways:

1. Create an LV for `/mars` on top of your application VG, typically named `/dev/vg/mars` or similar (see step-by-step instructions in section 2.4.1). This is the easiest solution if you are anyway using LVM on top of a hardware BBU. This is also most flexible: it can be **resized during operation**. Therefore, you may start with a size of around 500 GiB, and later be extended with increasing demands.
   This variant is also recommended if you have very expensive SSD storage. Depending on write rates, you could for example start with 100 GiB, and extend dynamically as far as needed, for example by some alerting scripts, or even using some cron job.

2. Alternatively, you may use one **dedicated HDD** with a capacity of 4 TB or more. Typically, this will provide you with plenty of headroom even for bigger networking incidents. Performace of a single HDD over a BBU is typically good enough for `/mars` because the transaction logs are involving mostly *sequential* reads and writes in larger chunks. However, there exist some workloads where striping could be necessary for maximizing sequential throughput.

3. Alternatively, if you are concerned about both performance and reliability, use two dedicated spindles over hardware RAID-1 with BBU. For maximum flexibility, put another VG on top of the dediactedRAID-1 set. For example, if `/dev/sdc` is your RAID-1 set, create a PV and a VG called `mars` on top of it. This is most flexible, since you might later migrate your `/mars` even during runtime, for example when replacing small disks with bigger ones, or when replacing HDDs with SSDs during runtime.

4. For extemely high performance, separate SSD sets for the user data VG and for `/mars` might be beneficial. However, check whether it really pays off. Notice that a hardware BBU is nothing but a RAM cache, which is faster than any SSD, and there *exist* some workloads where sequntial IO to HDDs is faster than to SSDs. Sometimes, there are hidden performance bottlenecks, such as SAS busses, or some old-generation RAID controllers.

Dedicated HDDs for `/mars/` have another advantage: their mechanical head movement is completely independent from your data head movements. For best performance, attach the cor-

[2] 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).

[3] In principle, you may combine MARS with commercial storage boxes connected over Fibrechannel or iSCSI. At 1&1, there is not yet operational experience with such setups.

[4] Exception: restores from backup.

[5] Within some limits, the distribution is an exponential one, according to Zipf's law.

responding disks 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 configured as a relatively small RAID-1 set. For extremely high performance demands, you may consider (and check) RAID-10 and/or SSD storage. However, SSDs are reported as less reliable. While failures of HDDs are typically detectable in advance by upcoming SMART media error counts, SSDs are typically failing suddenly and unexpectedly[6]. And their failure is not statistically independent in general. Building a RAID-1 on top of SSDs bears an increased risk that *both* SSDs are unexpectedly failing both at the same time[7].

If you want to build extremely cheap low-cost storage, for example for low-performance backup systems or similar use cases: cheap but high-capacity nearline-SAS[8] disks may be sufficient, because the transaction logfiles are highly sequential in their access pattern. However, check with `blkreplay` that performance is *really* sufficient, when compared with "better" disks.

Do not import the block device for `/mars/` over iSCSI. This would sacrifice both reliability and performance. MARS is constructed for exploiting a hardware BBU cache with a typical IO parallelism degree of 1000 parallel IO requests, over fast local DMA. See also section IO Performance Tuning.

Consequence: never run MARS inside of a VM (other than for functional component testing). See also Dijkstra's layering rules in `mars-architecture-guide.pdf`.

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

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

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

Currently, MARS provides no shared secret like DRBD, because a simple shared secret is way too weak to provide any real security (potentially misleading people about the real level of security). Future versions of MARS might provide some 2-factor authorization, and encryption via dynamic session keys. Until that is implemented[9], use a secured VPN instead. And don't forget to *audit* it for security holes.

2.2. MARS Kernel Module

Always use the newest stable version (master branch) from https://github.com/schoebel/mars. Please consult the file `ChangeLog` there.

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

[6] Notice: the component failure rate is not the crucial point. Even if some types of SSDs have a better MTBF than typical HDDs: when you can detect failure in advance, you can prevent

[7] Preliminary replacement of SSDs after a certain amount of write may help. But it will increase cost.

[8] Even cheaper SATA disks are not recommended for professional datacenter usage. Typically, they are not rated for 24/7/365 usage. Even for some use cases like backup, experiences are worse.

[9] There is fundamental argument: network traffic between datacenters belongs to a higher level than a single component like MARS. Thus its security requirements must be solved at that higher level, but not at the lower level of MARS.

- As an external Debian or rpm kernel module, as provided by a package contributor (or hopefully by standard distros in the future).
- Via `dkms`. Although there is an example file `contrib/mars-dkms.dkms` in the official MARS repo at https://github.com/schoebel/mars which could serve as a base for Linux distro vendors, it would suffer from *serious performance degradation* if it would be compiled *without* the MARS pre-patch. Don't use such a version for any serious application. With pre-patch, `dkms` would be no problem. You can check whether some already built `mars.ko` kernel module has been compiled with pre-patch or not:
  `modinfo mars`
  which displays the version for the currently active kernel, or any other version via
  `modinfo /path/to/mars.ko`
  This should display something like

  ```
  io_driver:      aio
  prepatch:       has_prepatch
  ```

  Do not use a kernel module for production if the io_driver either reports `sio` in place of `aio`, or if no pre-patch is detected.
- As a *separate* kernel module, only recommended for *experienced*[10] sysadmins: see file `Makefile.dist` (tested with some older versions of Debian; may need some extra work with other distros).
- Following are recommended build instructions for senior sysadmins or developers, inplace in the kernel source tree. Look into the subdirectory `pre-patches/` of the MARS repo for the right version of the pre-patches.

Here are example instructions for LTS kernel 4.4, building everything from scratch at your Linux workstation, using `git`. Actions marked "for safety" are not generally necessary, but may be appropriate for recovery from previous build failures.

1. `git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git linux-stable.git`
2. `cd linux-stable.git`
3. `git checkout linux-4.4.y`
4. For safety:
   `git pull`
5. Get an appropriate old `.config` file. For example at some OpenSuSE distro:
   `cp /boot/config-4.12.14-lp151.28.13-default .config`
   Notice: there are several other methods which are outside the scope of this manual. When in doubt, consult somebody with kernel build experience.
6. `make oldconfig || make olddefconfig`
7. `cd block/`
8. `git clone --recurse-submodules https://github.com/schoebel/mars`
9. For safety, ensure you always have the newest MARS version:
   `cd mars/`
   `git checkout master`
   `git pull`
   `cd ..`
10. Go back to the root of Linux git:
    `cd ..`

[10] 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.

11. To avoid mixup of later actions with upstream patches:
    `git branch -D mars-patches-for-4.4 || echo IGNORE THIS ERROR`
    `git checkout -b mars-patches-for-4.4`

12. Apply *all(!)* the patches from `block/mars/pre-patches/vanilla-$version`. For example with kernel 4.4:
    a) For safety:
       `git reset --hard; git clean -f`
    b) `git am block/mars/pre-patches/vanilla-4.4/*.patch`

13. `make menuconfig` (or another variant like `make xconfig`)
    a) Go to "Enable the block layer"
    b) Select MARS as a module (by moving the cursor and then typing `m`)
    c) A lot of sub-options for MARS will pop up. Leave them at their default.
    d) Save the Kconfig options to `.config` and exit.

14. Build the kernel, typically something like one of:
    a) For Debian / Ubuntu / etc:
       `make deb-pkg`
    b) For Redhat / CentOS / SuSE / etc:
       `make rpm-pkg`
    c) Classical with local install:
       `make -j 12 && make install modules_install`
    d) ... see some more variants and make targets:
       `make help`

15. Copy and install the new Debian or rpm package to the destination servers, activate them (system-dependent, like `update-grub` or similar), and reboot your servers with the modified kernel.

16. Check that the MARS kernel module is installed (but do not yet load it):
    `modinfo mars`

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

Installing marsadm by hand is rather simple: just copy it to `/usr/bin/` or `/usr/local/bin`. For example:

- `cp -a block/mars/userspace/marsadm /usr/local/bin/`
- Alternatively, create a Debian or rpm package (named `mars-utils` or `mars-tools` or similar). Since `marsadm` is a basic Perl script which requires no other Perl modules, the only package dependency is from the Perl interpreter. Since Perl is typically already installed on virtually every Linux server, leaving out this dependency won't be a show stopper.

With newer versions of MARS, a prepatch for vanilla kernels 3.2 through 4.9 (or even later) is no longer needed, at least in theory. However, **IO performance** is currently *much* worse when the pre-patch is not applied. This will be hopefully addressed in a future release. At the moment, don't use pre-patch-less MARS on any production system. It has extremely limited IO parallelism because some performance-critical kernel interfaces are not accessible without the pre-patch. The command `modinfo mars` will report whether the pre-patch is present at your kernel, or not (see description of `dkms` above).

Pre-patches for various kernel version can be found in the `pre-patches/` subdirectory of the MARS source tree. Following are the types of pre-patches:

- 0001-mars-minimum-pre-patch-for-mars.patch or similar. Please prefer this one (when present for your kernel version) in front of any old / deprecated 0001-mars-generic-pre-patch-for-mars.patch or similar. The latter should not be used anymore, except for testing or as an emergency fallback.

- 0002-mars-SPECIAL-for-in-tree-build.patch or similar. This is *only* needed when building the MARS kernel module together with all other kernel modules in a single `make` pass. For separate external module builds (e.g. with `dkms`, or with `rpmbuild` etc), this patch *must not* be applied (but the pre-patch *strongly should* if somehow possible). When using this patch, please apply the aforementioned pre-patch also, because your kernel will be patched anyway.

- For certain kernels like 4.14, some additional fixes may be necessary. These are in the respective `vanilla-*/` subdirectory, indicated by filename `*fix*`. Sometimes, these are *absolutely* needed. For example, an endless loop in unfixed upstream vanilla kernels >= 4.10 may occur upon network timeouts.

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

2.3. Setup Primary and Secondary Cluster Nodes

If you already have some production data on your bare metal servers via LVM, you may skip some of the following subsections.

In case your data is already replicated with DRBD, you may migrate to MARS (or even back from MARS to DRBD) if you use *external*[11] DRBD metadata (which is not touched by MARS). Internal DRBD metadata is reported to also work, because it resides at the end of the block device. However, you will waste some small amount of storage.

Migrating back to DRBD is also possible, provided you re-initialize the DRBD meta-data again.

For the following instructions to work, you must be `root` on your servers.

2.3.1. Setup Hardware

Do not use MARS inside of VMs. Only use at bare metal!

When using hardware RAID controllers with hardware BBU (as is highly recommended), you will need to build your RAID sets with the corresponding tools.

Don't set your hardware BBU cache to "writethrough" mode. This may lead to tremendous performance degradation. Use the default "writeback" strategy instead. It should be operationally safe, because in case of power loss the BBU cache content will be preserved thanks to the battery, and/or thanks to goldcaps for saving the cache content into some flash chips.

In the following sections, we assume that two RAID sets are already built, and are accessible as `/dev/sdb` and `/dev/sdc`.

2.3.2. Setup the Network

Here are only brief recommendations. Network setup is outside the scope of this manual. Recommended are the following basics:

[11] *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.

- Avoid layer 2 coupling *between* datacenters. MARS requires only TCP/IP (typically over IPv4 default ports 7776...7779) for replication traffic, so layer 4 coupling (aka routing) is sufficient. Of course, the lower layers are always present inside the *same* datacenter, so just avoid *unnecessary* lower-layer coupling *between* datacenters. Any problems caused by the network and its setup are up to your own.

- As explained in `architecture-guide-geo-redundancy.pdf`, dedicated replication networks are *recommended* for *long-distance* replication of hundreds or thousands of servers.

- Best practice: ensure that *each* of your cluster hosts can `ping` to *each* other (which means $O(k^2)$ potential network connections), via their *pure hostname.*
  Example on hostA: `ping hostB`

- If you have only 1 server IP over 1 physical ethernet interface, classical datacenter-internal DNS (as typically used for sysadmin `ssh` access etc) is sufficient. If you have a separate replication network, e.g. a separate physical ethernet interface `eth1` in addition to classical `eth0`, you *might* omit another DNS entry *theoretically.* Although several `marsadm` commands are supporting separate `$host_ip` parameters for circumvention of DNS, working directly on IP addresses is *not* a best practice. Out of many alternatives, try to *avoid* separate DNS names for the `eth1`-specific master IP, but *consider* to use *local routing* for the MARS ports 7776 to 7779 over `eth1`, while other ports may remain on `eth0`. Such a port-specific routing setup will make you *independent* from changes in the network or hardware setup, and it will make the DNS less complex. Your scripting will also benefit from simplicity. On the other hand, beware of (internal or external) routing problems. However, operations of professional datacenters needs to deal with suchalike playgrounds anyway. This MARS-specific guide cannot dive into details.

- Firewalling is also OT = Off Topic here. Recommendation: KISS = Keep It Simple and Stupid.

- Avoid FQDNs for any testing, and do not encode domain names into any scripting. See also section 2.4 Setup / Install OS.

2.4. Setup / Install OS

Installing a Linux distribution is outside the scope of this manual. There are dozens of methods, each of them working differently in detail. Here are only some rough hints for avoidance of problems with MARS:

- Hostnames need to be entered into typical installers, e.g. as used by major Linux distros. In case you have selected a bad name, you can typically change it, e.g. via your favourite tool like `yast` or `zypper` or `yum` or the equivalent Debian / Ubuntu tools, and os on, or manally like `vi /etc/hostname` followed by reboot, or similar. However, do this **only *before* MARS is configured**. Here are **pitfalls you need to know**:

  1. As is best practice since decades: only use *plain* hostnames, *separated from* domain names. This means: the shell command `hostname` (and siblings like `uname -n`) **must** deliver exactly 1 word, without any dot. Only use *this* name in later marsadm commands.

  2. Of course, you *may* also set a **domain name** as is typically reported by the shell command `domainname`.

Do not enter any domain name into the `hostname` field of several major Linux distros. Typically, the entered domainname will then become *part of* the hostname, which will not only ENDAGER YOUR DATA, but is also a well-known fault in the Linux and UNIX ecosystems since decades.

Always enter any domainname *separately*, typically into a different field, or via a different shell command, or similarly. If this is not possible during installation, do it later, and *do not forget it*. Ensure that the shell command `hostname` (or any other host-based command) will never report *any part* of any / the domainname!

Why? because there exist OS setups where a host can be member of *multiple domains*, all in parallel.

Historically, there were *several types of domains*, e.g. from YP ~ NIS driven Sun setups. Some alternate domain setups are used today, e.g. by Microsoft, or by non-DNS driven network protocols, by CIFS, or by some printer domains, and so on.

3. *Never* use domain names in later `marsadm` commands! This will pay off, because you typically can change the domainname(s) later, e.g. when your machine is expedited to a different location (like moves of whole datacenters), or you may configure *multiple* domainnames under certain circumstances. Only the *hostname* has to be treated as a *unique* **immutable ID of your machine**. Other hardware or software configurations like MAC addresses or IP addresses or DNS zones etc will be typically reconfigurable (when done properly).

   Q: why is the `hostname` used as a dot-less machine ID, although there exist a *multitude* of UUIDs at hardware level?
   A: because defective hardware is actually repaired in big datacenters, e.g. new mainboards, new CPUs, or even (partially) **preconfigured spare machines** where *nothing* has survived from the old hardware. MARS supports a **variety of repair strategies** by placing MARS' machine ID into the *traditional* `hostname`, independently from any number of domain or DNS names etc, and their contradictory shortcomings. This is a *feature*, not a bug, and you *agreed* to this built-in feature of MARS by *usage*, see the GPL and its license terms and conditions like FITNESS FOR A PARTICULAR PURPOSE.

4. Best practice: start your immutable hostname with a lowercase letter from `a` to `z`, and avoid uppercase letters completely. Avoid non-English letters from other languages. Avoid utf-8 etc. You may include some numbers, and the `marsadm` tool will later automatically sort these parts numerically, even when the number of digits is varying. Do not use the underscore symbol `_` . When separators are necessary, simply replace it with an *infix* dash.

5. Hint: use *short* but *recognizable* hostnames according to the Unix Philosophy. You will likely type these names very often in the following years! In a huge company, prefer a *systematic* naming + numbering scheme which is easyly usable by a bigger sysadmin team.

6. Hint: for long-distance replication, you should encode the *datacenter name* resp the *datacenter region* (or similar) with 2 or 3 lowercase letters. This makes it **less risky** for humans during midnight incidents to get caught by **typos**!
   Example: pair the hostname `cpu-wc1234` with `cpu-me1234` where cpu denotes the machine class, `wc` means the "west coast" datacenter while `me` means the "middle east" datacenter, and 1234 is the *cluster* number, not a sequential machine number. Thus

you don't need to lookup partner machine names or numbers from some database, even when you just woke up from your sleep at midnight.

7. Do not use silly[12] hostnames like `none` / `all` / `any` / `undefined` / `local` / `localhost` which are reserved specifiers for `marsadm`. When unsure, consult `marsadm --help`.

8. NEVER EVER modify the hostname *after* and/or *during* MARS is already installed, or is already running! **Any (human) error is your fault!**
See also the GPL: NO WARRANTY
11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

9. You are FULLY RESPONSIBLE if you do any of the following. There is NO WARRANTY according to the GPL. If you *really* need to change the hostname, first stop all of your applications, then decommission MARS *fully* via `marsadm leave-resource` and `marsadm leave-cluster` commands (see later descriptions), and unload the MARS kernel module via `rmmod mars`. Afterwards, you MUST recreate the `/mars` filesystem for safety, typically via `mkfs.ext4`. Be **100% sure** that all of this has *fully succeeded*. After decommisioning and deconfiguration of the stack parts *above* of `/dev/lv/mydata` (see the following sections), you can *try* to change the hostname. CHECK YOUR WHOLE SETUP, and finally you are *fully responsible* for re-configuration of the formerly deconfigured parts under the new hostname, as described in the following sections.

10. Hint: by not decommissioning your disks `/dev/lv/mydata`, you may keep your valuable data. However, you are also fully responsible for any error in device naming, device location, device content, etc.

2.4.1. Setup LVM

Execute the following instructions only once after bare metal hardware deployment, or if you want to *fully* re-install your server. Otherwise, you may delete existing data.

1. First step is create the LVM meta-information on the RAID sets `/dev/sdc` and `/dev/sdc`:
```
pvcreate /dev/sdb
pvcreate /dev/sdc
```

2. Check your physical volumes:
```
pvs
```

3. Create a volume group:
```
vgcreate vg /dev/sdb /dev/sdc
```

[12] As a protection against sillyness, the list of silly basic names in `marsadm` may be extended in future.

4. Check your volume group:
   `vgs`
5. Create a LV for /mars:
   `lvcreate -i 2 -L 100G -n mars lv`
6. Check your list of LVs:
   `lvs`

2.4.2. Setup Cluster Nodes

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

1. On each of hostA and hostB, create the `/mars/` mountpoint:
   `mkdir /mars`
2. On each host, create an `ext4` filesystem on your separate disk / RAID set via `mkfs.ext4`[13]
   `mkfs.ext4 /dev/vg/mars`
3. On each host, mount that filesystem to `/mars/`. It is advisable to add an entry to `/etc/fstab`., or to create a systemd unit `mars.mount`. Here we just mount it by hand:
   `mount /dev/vg/mars /mars`
4. For security reasons, execute `chmod 0700 /mars` everyhwere after `/mars/` has been mounted. If you forget this step, any following `marsadm` command will drop you a warning, but will fix the problem for you.
5. On hostA: `marsadm create-cluster`
   This must be done *exactly once*, on exactly one node of your cluster (the *first* node). Never do this twice or on different hosts, 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 2.

   Nodenames (see `uname -n` and `man 1 uname` and `man 2 uname`) are the most critical part of your setup. They **must** be **globally unique**, and they **must never change, perpetually. Any (human) error is your fault!**
   See also the GPL: NO WARRANTY
   11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
6. Except for historic versions[14] of MARS, execute on *both* hosts hostA and hostB:
   `modprobe mars`

[13] Don't use `xfs` for `/mars`. Its late allocation strategy may lead to deadlocks and other problems, at least with some elder kernel versions.

[14] Old versions of MARS before mars0.1astable101 needed a working `ssh` connection from hostB to hostA (as `root`), and also in the opposite direction, and between *all* (current and future) cluster members. The following is an advice for a historic method. Test ssh on hostB:

- `ssh hostA w`

  This needs to work without entering a password. Ensure that it also works in the opposite direction. In addition, `rsync` must be installed.

7. On hostB:
   `marsadm join-cluster hostA`
8. When not yet done, do on each of your hosts:
   `modprobe mars`
9. Check that mars is running everywhere:
   `marsadm view all`
10. Ignore any warnings that no resources are yet defined. But you should check that no warnings about network connections are appearing. Both cluster nodes should be able to communicate with each other over the MARS ports (default 7777 to 7779).
11. Additional checks, to be executed on *all* of your hosts:
    `netstat -lp --tcp | grep 777`
    `netstat --tcp | grep 777`
    Both variants should show up some healty connections. If not, fix your network configuration and/or firewalling etc. Details are outside of the scope of this manual.

Beware of *asymmetric connections*, e.g. caused by inappropriate networking or firewall rules. *Any* host must be able to communicate with *any* other host, *at least* in the *same* cluster. See also the big footnote 14 starting on the previous page (ending on this page).

Do not shoot yourself in your foot by the ill-belief that it would be easy to control the (replication) network traffic and/or to manage fine-granular firewalling on hundreds or thousands of machines, whether a single huge BigCluster or many smaller clusters, e.g. pairwise and/or according to the current LV replica situation etc. Big systems (as such) are not only prone to sporadics like defective hardware, they also tend to *some dynamic behaviour* like growth and hardware lifecyle. Thus they need *updates* and housekeeping in an *incremental*[15] manner. They aren't static piles of metal and fibres. Networking and its configuration should also obey: KISS = Keep It Simple and Stupid.

Well-done *coarse* granularity at network level is your friend.

If you try to reduce the risk, or are already hit by asymmetric MARS connections, e.g. for some historic reasons or due to sporadic ARP-cache overflows etc: regularly check (e.g. via monitoring, and/or via long-running background jobs) that *all* MARS ports are operational, and in *all combinations* from each server to each other server.

2.5. Setup Housekeeping Cron Job

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

Hints: *very useful* is `ssh-agent` and `ssh -A` preconfigured via `/etc/ssh/ssh{,d}_config`. Hint 2 (experiences from the `football` project): if you don't use `ssh-agent` (or if you *disallow* it explicitly by default and allow it only exceptionally), then you will waste a lot of time and energy with trivial basics. `marsadm` has got some provisionary workarounds, like internal fallback to an internal list of `ssh` ports, but suchalike isn't recommended. Just configure your `ssh` infrastructure in such a ways that it works *smoothly*.

Similar waste of time and energy will occur if you follow the ill-belief that (static or dynamic) firewalling on the MARS ports 7776 to 7779 would be a "clever" idea, and/or if you "sell" some "features" like *port knocking* on MARS ports to the management. The quality of such ideas could be disguised if you noticed that your dedicated replication network *is* already separated by construction, or it *could* be done (e.g. via *simple* network-level firewalling) with *less effort*. Simply, and frankly: **do not shoot yourself in your foot**.

Another way for damaging yourself is usage of old MARS versions. Notice that MARS has drastically improved in functional and non-functional aspects during the last years.

Some historic hint for those who *want to* shoot themselves, or are *forced* to non-productively *test* something from the ancient world: in old MARS versions, you *must not* modprobe before join-cluster is executed. In newer versions, it is vice versa.

[15] Big systems are often **close to 24/7/365** und thus need *incremental* updates / housekeeping at ***every* layer**, including networking and many sub-components.

time. In order to avoid filesystem overflow, the following must be executed in regular time intervals:

1. `marsadm cron`

Best practice is to run `marsadm cron` in a `cron` job, such as `/etc/cron.d/mars`. An example cronjob can be found in the `userspace/cron.d/` subdirectory of the git repo. Of course, alternatives to `crontab` may be also used, but please ensure that `marsadm cron` processes aren't invoked in high parallelism to each other on the *same* cluster host. On *different* hosts / distributed systems, parallelism is of course fully OK. Just avoid any *unnecessary* local parallelism.

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 3.6.

Here is some more background information if you want to configure your system cronjob manually. In most installations, a 10 minute cron interval should be sufficient. Here is an example line, to be placed in a file like `/etc/cron.d/mars`:

1. `*/10 * * * * root if [ -L /mars/uuid ] ; then marsadm cron ; fi > /dev/null 2>&1`

Here is some background explanation about some internal intermediate steps, as executed by `marsadm cron`. The following is DEPRECATED. Use cron for operations. Skip the following except for debugging:

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.

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

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

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

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

Optional, only when running *masses* of `marsadm leave-cluster` or `leave-resource` or `split-cluster` in *huge* masses (e.g. when running `Football` for hardware lifecyle or for long-term load balancing over a very long time): newer versions of `marsadm` can also delete some very old remains, which could accumulate over a very long time (typically years). Just run `marsadm cron --autoclean` everywhere once per month, or once per year.

2.6. Creating and Maintaining Resources

For the sake of simplicity, the underlying LV 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 suchalike is not recommended, since it may easily lead to confusion in larger installations.

You may have some already pre-existing `/dev/lv/mydata` at the initially primary hostA. 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.

If `/dev/lv/mydata` already exists and contains some data (e.g. previously used by DRBD), you should skip the first three steps, otherwise you may destroy your data.

1. On both hostA and hostB, create a LV. In this example, its size is 50G:
   `lvcreate -i 2 -L 50G -n mydata lv`

2. On each node, check that the new LV is occurring in each of the following lists:
   `lvs`

3. Only on hostA, you may create a new filesystem:
   `mkfs.xfs /dev/vg/mydata`
   Alternatively, here is a variant for creation of a zfs filesystem:
   `zpool create /dev/vg/mydata`
   `zpool export /dev/vg/mydata`

4. Only on hostA, say
   `marsadm create-resource mydata /dev/lv/mydata`
   As a result, a directory `/mars/resource-mydata/` will be created on hostA, containing some symlinks. hostA will automatically start in the primary role for this resource. Therefore, a new pseudo-device `/dev/mars/mydata` will appear after a few seconds.
   Note that the initial content of `/dev/mars/mydata` will be *exactly* the same as in your pre-existing LV `/dev/lv/mydata`.
   If you like, you may now 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 do so even 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.

5. On hostB:
   `marsadm wait-cluster`
   Check that the directory `/mars/resource-mydata/` and its symlink content is also appearing on hostB. If not, check your network and/or firewall setup.

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

7. On hostB, check that the sync is running after a few seconds:
   `watch marsadm view all`

Of course, now any old content `/dev/lv/mydata` at hostB (and *only* there!) is overwritten by the version from hostA. This is just what you wanted to do by setting up MARS replication. If you didn't check that your old contents at hostB didn't contain any valuable data (or if you accidentally provided a wrong LV device argument), it is too late now. Therefore, double-check that you are running `create-resource` and `join-resource` at the right sides of your cluster, and with the right block device names. Accidental confusion of the right sides, or accidental confusion of LV names may overwrite valuable data with wrong data, or even with uninitialized trash[16].

In order to reduce suchalike risks, `marsadm` does some basic checks. It checks that the disk device argument is really a block device, and that exclusive access to it is possible, as well

[16] Trying to mount unintitialized LV data is bad practice. It may even crash your kernel.

as some further safety checks, e.g. whether the size is big enough. Notice that bigger replica device sizes are allowed at secondaries, although then you will waste some space. In such a case, `marsadm view all` will display a warning. This behaviour is necessary as an intermediate step for online resizing via `marsadm resize`.

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.

Check that state `Orphan` is left after a while on hostB. Notice that `join-resource` is only *starting* a new replica, but does not wait for its completion.

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.

1. 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[17]. Then, and only then, you may say `marsadm fake-sync mydata` in order to abort the sync operation.

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

[17] 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`.

3. HOWTO operation of MARS resources

3.1. Inspecting the State of MARS

The main command for viewing the current state of MARS is

- `marsadm view mydata`

or its more specialized variant

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

where *`$macroname`* is one of the macros described in the following section Standard `marsadm view`, or in section 6.2, or another macro which has been written by yourself.

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

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

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

3.1.1. Standard `marsadm view`

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.

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

`%{res}` resource name.

`[`*`this_count`*`/`*`total_count`*`]` number of replicas of this resource (see join-resource & co), total number of cluster peers (see join-cluster & co).

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

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

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

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

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

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

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

Customization via your own macros (see section 6.3) is explicitly encouraged for experienced sysadmins and userspace developers. It would be nice if a vibrant user community would emerge, helping each other by exchange of macros.

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

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

`device-info` When present, shows the status of /dev/mars/mydata in human-readyble form. Shows the empty string when /dev/mars/mydata is not present.

`diskstate` Shows the status of the underlying disk device, in the following order of precedence[1]:

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

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

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

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

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

1. When the damage is only at one of your secondaries, you should first ensure that the primary has a good logfile after a `marsadm cron`, then try `marsadm invalidate` at the damaged secondary. It is crucial that the primary has a fresh correct logfile behind the error position, and that it is continuing to operate correctly.
2. When *all* of your secondaries are reporting `DefectiveLog`, the primary could have *produced* a damaged logfile (e.g. in RAM, in a DMA channel, etc) while continuing to operate, and all of your secondaries got that defective logfile. After `marsadm cron`, you can check this by comparing the `md5sum` of the first primary logfile (having the lowest serial number) with the versions on your replicas. The problem is that you don't know whether the primary side has a silent corruption on any of its disks, or not. You will need to take an operational decision whether to switchover to a secondary via `primary --force`, or

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

whether to continue operation at the primary and `invalidate` your secondaries.

3. When the original primary is affected in a very bad way, such that it crashed badly and afterwards even recovery of the *primary* is impossible[2] due to this error (which typically occurs extremely rarely, observed two times during 7 millions of operating hours on defective hardware), you need to take an operational decision between the following alternatives:

   a) switchover to a former secondary via `primary --force`, producing a split brain, and producing some (typically small) data loss. However, integrity is more important than actuality in such an extreme case.

   b) deconstruction of the resource at *all* replicas via `leave-resource --force`, running `fsck` or similar tools by hand at the underlying disks, selecting the best replica out of them, and finally reconstructing the resource again.

   c) restore your backup.

`Orphan` The secondary cannot replay data anymore, because it has been kicked out for avoidance of emergency mode. The data is not recent anymore. Typically, `marsadm invalidate` needs to be done.
There is an execption: shortly after `join-resource` or `invalidate`, it may take some time until state `Orphan` may be left, and until the newest logfile has appeared at your secondary site (depending on the size of logfiles, and on your network). In case of network problems, this may take very long.

This state tells you that your replica is not current, and currently not being updated at all. Don't forget to **monitor** for longer occurrences of this state! Otherwise you may get a big surprise when you need a forceful emergency failover, but your replica is very old or even does not really exist at all.

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

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

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

Only in case of a primary node crash, and *only* after attempts have failed to become primary again (e.g. IO errors, etc), this *can* (but need not) mean that something went wrong. Even in such an extremely unlikely event, chances are high that `fsck` can fix any remaining problems (and, of course, you can also switchover to a former secondary).

When this message appears, simply start MARS again (e.g. `modprobe`

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

`mars; marsadm up all`), in whatever role you are intending. This will *automatically* go into phase `Recovery`, i.e. try to replay any necessary transaction logfile(s) in order to fix any inconsistency. Only if the automatic fix fails and this message persists for a long time without progress, you *might* have a problem. Typically, as observed at a large installation at 1&1, this happens extremely rarely, and even then it just indicated that hardware was defective.

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

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

`WriteBack[`*amount*`]` (cf `%is-primary{}` and amount via `%writeback-rest{}`) Appears only at actual primaries (whether designated or not), when the writeback from the RAM buffer is active (see section 1.2). The *amount* is displayed in human readable form, and may be used for a very rough estimation of recovery time after a primary crash.

`Recovery` (cf `%todo-primary{}`) Appears only at the designated primary before it actually has become primary. Similar to database recovery, this indicates the recovery phase after a crash[3].

`EmergencyMode` (cf `%is-emergency{}`) A current designated primary exists, and it is known that this host has entered emergency mode. See section 3.7.

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

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

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

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

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

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

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

  `EmergencyMode` (cf. `%is-emergency{}`) See section 3.7.

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

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

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

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

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

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

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

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

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

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

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

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

`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.

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

`role` Shows the *actual* state: `None`, `NotYetPrimary`, `Primary`, `RemainsPrimary`, `ForcedPrimary` 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 . `ForcedPrimary` indicates that *multiple* cluster hosts (see `%nr-primary{}`) are claiming to be in actual primary role, e.g. when another one is in role `RemainsPrimary`, or when network interruption is preventing role change information from propagating.

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

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

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

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

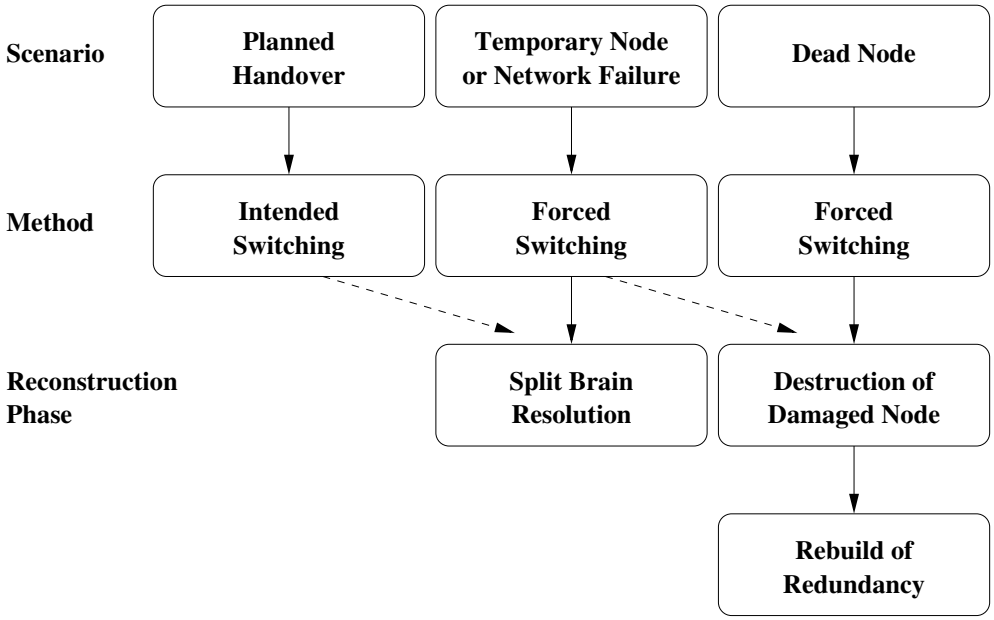
`replinfo` Shows an informational progress bar when fetch is running. This should not be used for scripting at all, because it contains realtime information in human-readable form.

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

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

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

3.2. Switch Primary / Secondary Roles



MARS distinguishes between *intended* and *forced* switching. This distinction is necessary due to the communication architecture (asynchronous communication vs synchronous communication, see explanation of Lamport Clock in `mars-for-kernel-developers.pdf`).

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 (as compared to a synchronous system like DRBD).

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

3.2.1. Intended Switching / Planned Handover

Before starting a planned handover from your old primary `hostA` to a new primary `hostB`, you should check the replication of the resource. As a human, use `marsadm view mydata`. For scripting, use the macros from section 6.2.1 (see also section 6.4; an example can be found in `contrib/example-scripts/check-mars-switchable.sh`). The network should be OK, and the amount of replication delay should be as low as possible. Otherwise, handover may take a very long time.

Best practice is to **prepare**[4] **a planned handover** by the following steps:

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

2. Only when the `systemd` method from section 6.1 is *not* used: stop your application on hostA, then say on hostA:
   `umount /dev/mars/mydata`

   If you use the automatic handover method provided by systemd templates (see section 6.1), this step is *not needed*.

3. Only when systemd templates are *not* used, and only for increased safety on hostA:
   `marsadm wait-umount mydata`
   This will reduce the risk of **hanging umounts** leading to long-lasting waits at the future primary hostB. Such problems will be detected earlier, so you have more possibilties for fixing them.

   Also good practice: use `lsof /dev/mars/mydata` before umount for even earlier detection of hanging processes.

4. Optionally, and only when the `systemd` method from section 6.1 is *not* used: on host `B`, wait until `marsadm view-diskstate mydata` shows `UpToDate`. This way, you are gaining more control over the *duration* of the handover. In case of unexpected network problems, disk space problems, etc, you can script a compensation action like giving up much earlier, and restarting your application at the old primary hostA much earlier.

5. On hostB:
   `marsadm primary mydata`
   When combined with the `systemd` method (see section 6.1), this will even automatically stop the application at hostA, wait for handover, and start the application at hostB.

The most important difference to DRBD: don't use an intermediate `marsadm secondary mydata` at hostA. Although it is possible, there are severeral *disadvantages*[5] from losing the

[4] Precondition for a plain `marsadm primary` (without `systemd`) is that you are up, that means in attached and connected state (cf. `marsadm up`), that you are no sync target anymore, and (only when `systemd` isn't configured to automatically stop the application at the old site) that any old primary (in this case `A`) does not use its `/dev/mars/mydata` device any longer, and that the network is healthy. If some (parts of) logfiles are not yet (fully) transferred to the new primary, you will need enough space on `/mars/` at the target side. If one of the preconditions described in section 4.2.2 is violated, `marsadm primary` may refuse to start.

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

[5]
`marsadm secondary` is *discouraged* for several reasons. It tells the *whole cluster* that *nobody* is designated as primary anymore. *All* nodes should go into secondary mode, globally. In the current version of MARS, the secondaries will no long fetch any logfiles, since in split brain situations they don't know which version is the "right" one. When a primary host is designated, this is the "right" one by definition. Syncing is also not possible when there is no designated primary. When the device `/dev/mars/mydata` is in use somewhere, it will remain in *actual* primary mode during that time, and the secondaries will sync therefrom.

primary state. In case of an unexpected crash at the wrong moment, nobody might know anymore where the primary was running before. Best practice is to always switch *directly* from the old primary hostA to the new primary hostB.

If you need the local device `/dev/mars/mydata` to disappear *everywhere* in the whole cluster, you don't need the discouraged `marsadm secondary` command. `marsadm detach` or `marsadm down` can do it also, without destroying knowledge about the former designated primary. There is only one use case where `marsadm secondary` is really needed: final destruction of a resource before `marsadm delete-resource` is executed.

In contrast to DRBD, MARS remembers the designated primary, even when your system crashes and reboots. With DRBD, you typically will have to re-setup the DRBD roles with (scripted) commands like `drbdadm up ...; drbdadm primary ...`. Instead, MARS will **automatically resume** its former roles just by saying `modprobe mars`. When combined with a proper `systemd` setup (see section 6.1), this will even automatically re-start your application after the crash.

Another fundamental difference to DRBD: when the network is healthy, there can only exist *one* designated primary at a time. By saying `marsadm primary mydata` on host`B`, **all other** hosts (including `hostA`) will **automatically go into secondary role** after a while. You don't need to tell them explicitly, because MARS is automatically propagating the information for you.

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 be flickering.

Planned handover is refused *by default* when some sync is running somewhere, even at a third hostC. By adding the option `--ignore-sync`, you are no longer protected by this *safety measure*, and you are willing to accept that any already running sync at any hostC or hostD will restart from point 0, in order to ensure consistency.

Tip: newer versions of `mars` + `marsadm` are supporting the option `--parallel` combined with `all`, e.g. `marsadm primary all --parallel`. Instead of waiting until *all* the resources have left the primary role at the old primary (barrier synchonization), the handover speed of each resource is treated *individually*. Slow resources will no longer retard fast ones, minimizing total downtime. However, check that your cluster manager can deal with a rather high parallelism degree. At the moment, the `systemd` interface is not yet prepared for this.

3.2.2. Forced Switching

In case of an incident, the connection to the old primary hostA may be lost for several reasons. Then, at hostB, we just don't know anything about its *current* state (which may deviate from its *last known* state). The following command sequence will skip many checks (essentially you just need to be attached and you must not be a current sync target) and tell hostB to become primary forcefully:

1. On hostB:
   `marsadm pause-fetch mydata`

   notice that this is similar to `drbdadm disconnect mydata` as you might be 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. In contrast, `marsadm pause-fetch` is equivalent to

As soon as the local `/dev/mars/mydata` is released, the node will *actually* go into secondary mode if it is no longer designated as primary.

`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, potentially distributed over multiple routes, and potentially even from any *other* member of the resource, if they think they can get the data from there. In order to instruct[6] *all* members of the resource to stop fetching logfiles, you may use `marsadm pause-fetch-global mydata` instead (cf section 4.2.2).

2. On hostB:
   `marsadm primary --force mydata`

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

   When `systemd` is configured properly (see section 6.1), your application will start automatically at the new primary site.

   when the replication network is interrupted while the old primary hostA continues[7] running, it cannot know that hostB is the new designated primary. Therefore hostA will continue running by default. This means that your application will run twice! Only when the metadata exchange is working again (by default on port 7777), the old hostA will be automatically shut down by its local `systemd` configuration, when configured properly (see section 6.1). In difference to the *planned* handover from section 3.2.1, this may happen much later. In case of very long-last network outages, it may take even days or weeks.

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

3. For safety on hostB:
   `marsadm resume-fetch mydata`

   The new primary would 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.

   Newer marsadm versions, starting from `mars0.1astable113`, do not need this step anymore. After successful activation of `/dev/mars/mydata`, the equivalent of `marsadm up mydata` is executed automatically.

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

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

Split brain is always an **erroneous state** which should be never entered without reason! Once you have entered it accidentally, you **must** resolve it ASAP (see section 3.3), otherwise you cannot operate your resource in the long term.

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

[7] Notice: in certain network outage scenarios, you may not be able to remotely login to the console and to check whether a server is running. Therefore it may happen that you erronously think hostA is dead, while in reality it continues running. Even if you would know it, you might not be able to remotely kill it in a STONITH-like manner.

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

Some Background (may be skipped) In contrast to DRBD, split brain situations are handled differently by MARS . When two primaries are accidentally active at the same time, each of them writes into different logfiles `/mars/resource-mydata/log-000000001-hostA` and `/mars/resource-mydata/log-000000001-hostB` 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 cron (historic `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.

Split brain situations are detected *passively* by secondaries. Whenever a secondary detects that somewhere a split brain has happend, it refuses to replay any logfiles behind the split point (and also to fetch them when possible), or anywhere where something appears suspect or ambiguous. This tries to keep its local disk state always being "as consistent as possible", but outdated with respect to any of the split brain versions.

`marsadm primary -force` is rejected in newer marsadm versions[8] if your replica is a current sync target. This is not a bug: it should prevent you from forcing an inconsistent replica into primary mode, which will *certainly* lead to inconsistent data. However, in extremely rare cases of severe damage of *all* of your replicas, you may be desperate. Only in such a rare case, and only then, you might decide to force any of your replicas (e.g. based on their last sync progress bar) into primary role although none of the re-syncs had finished before. In such a case, and only if you really know what you are doing, you may use `marsadm fake-sync` to first mark your inconsistent replica as UpToDate (which is a **lie**) and then force it to primary as explained above. Afterwards, you will certainly need `fsck` or another type of repair before you can restart your application. Good luck! And don't forget to check the size of `lost+found` afterwards. This is really your *very last* chance if nothing else had succeeded before.

Tip: newer versions of `mars` + `marsadm` are supporting the option `--parallel` combined with `all`, e.g. `marsadm primary --force all --parallel`. This may potentially speed up startup. However, check that your cluster manager can deal with a rather high parallelism degree. At the moment, the `systemd` interface is not yet prepared for this.

3.3. 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.

In general, **split brain is unavoidable** in *any* distributed system, even if you use a passive crossover cable with DRBD, and when the crossover cable fails at the wrong moment. Please search Wikipedia for the CAP theorem, or read the corresponding section in `mars-architecture-guide.pdf`.

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

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

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.
   Newer versions of `marsadm view` are supporting your decision by telling you the amount of logfile data you would destroy when destroying a certain version of a certain host. Typically, the smaller version is a candidate for destruction. However, there are situations where this may be wrong, such as amok-running applications running like endless loops, or Spammer attacks against databases, or similar. MARS cannot know about this.

2. If you did not already switch your primary to the final destination determined in the previous step, do it now (see description in section 3.2.2). Don't use an intermediate `marsadm secondary` command (as known from DRBD): *directly* switch to the new designated primary!

3. Unless `systemd` is configured properly (see section 6.1), do the following manually: on each non-right version (which you don't want to retain) which had been primary before, umount your `/dev/mars/mydata` or otherwise stop using it (e.g. stop iSCSI or other users of the device). Wait until each of them has actually left primary state and until their local logfile(s) have been fully written back to the underlying disk.

4. Wait until the network works again. All your (surviving) cluster nodes *must*[9] be able to communicate with each other. If this is not possible, or if it would take too long, fall back to one of the method described in appendix .... section 3.4, but do this only when necessary.

The next steps are different for different use cases:

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

5. On **all** affected secondary nodes hostX where SPLIT BRAIN is reported:
   `marsadm invalidate mydata`

Typically, no split brain is reported anymore after that (via `marsadm view all`), and you are done.

In rare cases (when `/mars` is almost full somewhere, or when emergency mode has occured somewhere), you may need to run `marsadm cron` at the primary host, and to repeat `marsadm invalidate` on any SPLIT BRAIN host. In extremely rare of overloaded nodes, you may need to repeat this several times.

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

Check that state `Orphan` is left after a while. Notice that `invalidate` is only *restarting* an existing replica, but does not wait for its completion.

Retaining a Split Brain Version (optionally, typically not needed, may be skipped) On those cluster nodes where you want to retain some SPLIT BRAIN version (e.g. for inspection or debugging purposes, or as a kind of "emergengy backup"):

5. `marsadm down mydata`

6. `marsadm leave-resource mydata`

[9] 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 any not-yet-consistent intermediate states), you may deviate from this requirement, at your own risk.

7. After having done this on *all* those cluster nodes, check that the split brain is gone (e.g. by saying `marsadm view mydata` at the primary). In very rare cases, you might also need a `log-purge-all` at the primary (see page 59).

8. Rename the underlying local `/dev/lv/mydata` is into something like `/dev/lv/mydata-backup`. For example:
   `lvrename lv mydata mydata-backup`
   For details, see `man lvrename`. This is *extremely* recommended to avoid confusion with the old resource name.

9. For safety: check that each underlying local disk `/dev/lv/mydata-backup` is really usable afterwards, e.g. by test-mounting it (and `fsck` when needed). If all is OK, don't forget to umount it before proceeding with the next step.

10. Finally, you may either delete your backup somewhen via `lvremove`, or you may create a completely new MARS resource out of it, but *under a different name*. See description in section 2.6 on page 26.

11. Optionally, if you have enough disk space (check with `vgs`): re-create your replica by freshly creating a new `/dev/vg/mydata` with the right size, and `marsadm join-resource mydata /dev/vg/mydata`.

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

Keeping a Good Version (typically no actions needed) 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 will typically[10] resume replay after the SPLIT BRAIN has been resolved at the other nodes. Then you don't need to do any action for it.

When all SPLIT BRAIN versions have disappeared from the cluster (by `invalidate` or `leave-resource` as described before), the confusion should be over, and the secondary should automatically resume tracking of the new unique version.

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

Hint / advice for $k > 2$ replicas: it is a good idea to start split brain resolution *first* with those (few) nodes which had been (accidentally) primary before, but are not the new designated primary. Usually, you had 2 primaries during split brain, so this will apply only to *one* of them. Leave the other one intact, by not umounting `/dev/mars/mydata` at all, and keeping your applications running. Even during emergency mode, see section 3.7. *First* resolve the problem of the "wrong" primary(s) via `invalidate` or `leave-resource`. Wait for a short while. Then check the rest of your secondaries, whether they now are already following the new (unique) primary, and finally check whether the split brain warning reported by `marsadm view all` is gone everywhere. This way, you can often skip unnecessary invalidations of replicas.

3.4. Final Destruction of a Damaged Node

When a node has eventually died (e.g. defective hardware), **do not forget**[11] 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

[10] 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.
[11] If you forget this, `/mars` will fill up forever. Finally, emergency mode will be triggered.

memory corruption, disk corruption, or whatever. Although MARS has some provisions like md5 checksums in its transaction logfiles: don't risk any **unpredictable behaviour**!

2. **Manually** check which of the surviving versions will be the "right" one. Any human error is up to you: resurrecting an unnecessarily old / outdated version and/or decommissioning the productive primary server will be *your* fault.
3. If you did not already switch your primary to the final destination determined in the previous step, do it now (see description in section 3.2.2).
4. On a surviving node, give the following commands:
   a) `marsadm --host=your-damaged-host down mydata --force`
   b) `marsadm --host=your-damaged-host leave-resource mydata --force`

   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[12] node!
5. Repeat the same with *all* resources which were formerly present at `your-damaged-host`.
6. Finally, say
   `marsadm --host=$your_damaged_host leave-cluster --force`

Now all 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. For safety, check this via `marsadm view` everywhere.

As another safeguard, any re-attempt to "surrect" a "zombie" from its "claimed death" and/or from its "real death" should *spit at you* with some tons of warnings. These warnings are a *feature*, not a bug. If you *really* are an expert and if you *really* know what you are doing, you may ignore this AT YOUR OWN RISK.

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

Further advice for complicated cases of destruction are in appendix E and F.

3.5. Online Resizing during Operation

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

1. Increase your local disks (usually `/dev/vg/mydata`) *everywhere* in the whole cluster. In order to avoid wasting space, increase them *uniformly* to the same size (when possible). For example, on both hostA and hostB:
   `lvresize -L +100G /dev/vg/mydata`
2. For safety, say on both hostA and hostB
   `marsadm up mydata`
3. For safety, say on both hostA and hostB
   `marsadm wait-cluster`
4. Only at the primary hostA:
   `marsadm resize mydata`

[12] That said, MARS appears to be rather tolerant of human errors. As long as your `/dev/vg/mydata` is not removed at LVM level, you have a chance for recovery. Once a sysadmin destroyed a whole cluster by accident, including all of its resources, and while it was continuously running in primary role. Even transaction logging did continue on some orphan logfiles, but `/mars` was filling up "unexpectedly". Fortunately, this behaviour led to a monitoring alert and to detection of the problem. It was early enough for a correction without causing any extraordinary customer downtime outside of accepted SLAs, and no data loss at all.

5. A *partial* fast full-sync will start at hostB. Only the *new* portion of the block device will be synced. Check that sync is running, or has already finished.

6. If you have intermediate layers such as iSCSI, you may need some `iscsiadm` update or other commands.

7. Generally not needed, only for *extreme* safety / paranoia: you may wait until the partial sync has finished. This is not really needed, but it may *slightly(!)* reduce the risk in case of an unplanned incident at the primary side. If you execute the last step before the sync has finished, some data might have been already written to the new portions of the underlying LV. These writes will be written to the transaction log, and will be replicated. Thus there is no real danger, and your secondary will be **logically consistent** in reality, although it will be *reported* as `InConsistent` by marsadm in the meantime. If you want to avoid confusion about state `Consistent`, and/or if you want extremely high protection against damaged logfiles by hardware defects at the wrong moment (although the transaction logfiles are already protected by md5 checksums), and/or if you are paranoid, it may be helpful to wait until the sync has finished. Normally, this is not needed, and you may immediately proceed to the last step:

8. Now you may increase your filesystem. This is specific for the filesystem type and documented elsewhere. Some filesystems are able to increase their size while they are mounted and while the applications are running on top of them, but others cannot do this. For example, an xfs online resize during operation can be triggered at primary hostA where `/dev/mars/mydata` is currently mounted:
   `xfs_growfs /mountpoint/of/mydata`

Hint: in general, the sync of the new portions is not really needed, because the new junk data just does not care at filesystem level. If you are not paranoid and if you know what you are doing, you may use `marsadm fake-sync mydata` to abort unnecessary network traffic.

3.6. 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?

In practice, no harm will occur to your data. MARS will automatically go into the so-called emergency mode. Resolution of emergency mode is very similar to resolution of split brain (section C.4): at all of your secondaries, type (repeatedly)

```
marsadm invalidate all
```

This is all you need to know. If you are impatient, you may now skip the rest of this section.

For some background explanations, keep reading on.

Overflow and its treatment is *unavoidable* for long-distance replication. If you want a system which can survive long-lasting network outages, while keeping your replicas consistent as long as possible (called **anytime consistency**), then you *need* dynamic storage. It is *impossible* to solve with static pre-allocated memory[13]. A true solution would need *infinite memory*. But suchalike does not exist on earth.

It would be an even worse idea to statically pre-allocate a lot of space for *each* of your resources. The latter would waste a lot of space, because some resources will likely fill much more quickly than others. MARS deals with this by using a *common* filesystem /mars which is *shared* by the transaction logs of *all* of your resources.

Although the size of /mars is statically allocated at cluster generation time, there is a workaround for the problem. When `/mars` fills up during a network outage, and you have

[13] The bitmaps used by DRBD cannot 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.

some spare space on your VG, and when the network outage will be repaired shortly, you may decide to dynamically extend /mars during operation.

Because of these fundamental differences, DRBD and MARS have different application areas. If you just want a simple system for mirroring your data over short distances via passive[14] crossover cable, and when failures of the crossover cables are very unlikely, 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 it, but at some *cost*.

3.6.1. Countermeasures against overflow

3.6.1.1. 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, preferably 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!

3.6.1.2. Monitoring

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

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

1. Regular userspace monitoring must throw an INFO if a certain freespace limit l_1 of `/mars/` is undershot. Typical values for l_1 are 30%. Typical actions are automated calls of `marsadm cron`.

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 `cron`) alarming human supervisors via SMS and/or further stronger automated actions.

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

   Another source of space hogging is a "forgotten" `pause-sync` or `disconnect`. Therefore, a simple `marsadm up 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.

   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.

[14] Notice: newer generation 10GBit technologies like SFP+ are no longer passive. They involve some active chips, which may fail independently from your servers. In case of a failure, the CAP theorem property P is violated, and you only have the choice between C and A. For details, see `mars-architecture-guide.pdf`.

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

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

6. Third-level kernelspace action, automatically executed when /proc/sys/mars/required_free_space_2_gb + /proc/sys/mars/required_free_space_1_gb is undershot:
   all locally secondary resources will stop fetching transaction logfiles. This is a more desperate action of the kernel module. You don't want to get there (except for testing).

7. Last desperate kernelspace action when all else has failed and /proc/sys/mars/required_free_space_1_gb is undershot:
   all locally primary resources will enter **emergency mode** (see description below in section 3.7). 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.

3.6.1.3. Throttling

This not generally recommended. It may harm the IO performance from the viewpoint of your customers. Thus use it only as a *desperate* defense against overflow, by **throttling your performance pigs**.

Motivation: in rare cases, some users with ssh access can do *very* silly things. For example,

- some users 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. 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.

- another user wrote his own shell script for his own private backup of his website, although there already is a daily system backup. He regularly made a complete copy of his entire webspace (more than 60GiB) via cp -a, then created a tarball out of the copy, uploaded it into the cloud, finally removed both the tarball and the complete filesystem copy. Each time, about 100GB was temporarily allocated (and replicated via MARS).

Typically, your hard disks / 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. 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*.

MARS has some mechanism for throttling bulk writers whenever the network is weaker than your IO subsystem. It is off by default.

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 3.6.1.1), MARS may be used for throttling 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*. Defaul value is 0, which means "off". Practical values for this coule be around 80%.

`write_throttle_end_percent` Maximum throttling will occur once this space threshold is reached, i.e. the throttling is now at its maximum effect. A practical value is 90%, which is the default. 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[15] 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[16] throttling, this parameters determines the input flow into `/mars/`. The default value is 10.000 KB/s. Please adjust this value to your application needs and to your environment.

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

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

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

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

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

`write_throttle_maxwindow_ms` This parameter must be set sufficiently much greater than `write_throttle_minwindow_ms`. In case the flow of throttled operations pauses for some natural reason (e.g. switched off, low load, etc), this parameter determines when a completely new rate calculation should be started over[17].

3.7. Emergency Mode and its Resolution

This section explains some implementation details. You may skip it.

[15] Read requests are never throttled at all.

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

[17] 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.

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 is written to it (for now). The hole in the numbering will prevent any secondaries from replaying any logfiles behind the hole (should they ever contain some data, e.g. because the emergency mode has been left again). This works because the secondaries are regularly checking the logfile numbers for contiguity, and they will refuse to replay anything which is not contiguous. As a result, the secondaries will be left in a consistent, but outdated state (at least if they already were consistent before that).

2. The kernel module writes back all data present in the temporary memory buffer (see figure in section 1.2). This may lead to a (short) delay of user write requests until that has finished (typically fractions of a second or a few seconds). The reason is that the temporary memory buffer must not be increased in parallel during this phase (race conditions).

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

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

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

1. Free enough space. For example, delete any foreign files on `/mars/` which have nothing to do with MARS, or resize the `/mars/` filesystem, or whatever.

2. The following control is intended for testing. If `/proc/sys/mars/mars_reset_emergency` is off, now is the time to set it. By default, it should be already set.

3. Notice: as long as not enough space has been freed, a message containing `''EMEGENCY MODE HYSTERESIS''` (or similar) will be displayed by `marsadm view all`. As a consequence, any sync will be automatically halted. This applies to freshly invoked syncs also, for example created by `invalidate` or `join-resource`.

4. On the secondaries, use `marsadm invalidate $res` in order to request updating your outdated mirrors.

5. On the primary: `marsadm cron`

6. As soon as emough space has been freed everywhere to leave the `EMEGENCY MODE HYSTERESIS`, sync should really start. Until that it had been halted.

7. Recommendation: check at secondaries that state `Orphan` has been left after a while.

Alternatively, there is a more complicated method, which keeps more intermediate emergency backup replicas:

1. On *all* of your secondaries hostX:
   `marsadm leave-resource mydata`

2. At the primary hostA:
   `marsadm cron`

3. Wait until `df /mars` shows no longer an overflow.

4. On the first secondary hostB:
   `marsadm join-resource mydata /dev/lv/mydata`

5. Wait until sync has finished at hostB.
6. If you have more than 2 replicas in total: proceed with step 4 at hostC, and so on. This time, you could join multipe resources in parallel, because you already have a life replica at hostB.

Expert advice, if you have only 2 replicas, and provided you have enough VG space: analogously to paragraph 3.3 on page 38 you may use `lvrename` for keeping an outdated emergency backup before creating a new LV with the old name, and before re-joining the latter. Don't forget to remove your backup LV after sync has finished!

4. Working with `marsadm` commands

This chapter is a kind of reference about the `marsadm` tool. The sub-commands of `marsadm` are grouped according to the topic they deal with.

Since MARS work *asynchronously* at metadata propagation level (which is *necessary* for long-distance replication over flaky networks), several commands are only *triggering* an action, but do not wait for its completion.

Such cases are indicated by the term "after a while". 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, typically working in combination with (almost) any `marsadm` command (unless documented differently):

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 internal waiting logic for thois commands which emulate synchronous behaviour on top of the asynchronous communication paradigm. Many commands are waiting until the desired effect has succeeded. 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*!
`--parallel`	no	Only makes sense in combination with `all`. This is roughly equivalent to forking a bunch of parallel `marsdm` processes, like in pseudo shell script notation: `for i in $resource_list; do marsadm $parameters $i & done; wait`. Several cluster managers are not re-entrant and may deadlock. First check whether this option is usable in your concrete environment!
`--parallel=$number`	no	Like `--parallel`, but limit the parallelism degree to a maximum number of parallel processes. This may be useful for limiting the parallelism of startup processes, e.g. when kernel caches are cold, so the machine would get *overloaded* when too many resources would be starting in parallel. Several cluster managers are not re-entrant and may deadlock. First check whether this option is usable in your concrete environment!
Option	Cmp	Description

Option	Cmp	Description
--ignore-sync	no	Use this for a *planned* handover instead of --force. Only one precondition is relaxed: some sync may be running somewhere. Careful when using this on extremely huge LVs where the sync may take serveral days, or weeks. It is your sysadmin decision what you want to prefer: restarting the sync, or planned handover.
--ignore-deleted-peers=$number	no	Only for experts - usually not needed - may be harmful - read the sourcecode!
--verbose	no	Some (few) commands will become more speaky.
--timeout=$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 been transferred from the other host since more than the window time, the host is considered dead. Default is 60s.
--stuck-seconds=$seconds	no	Silencing period. When replay is hanging *unecpectedly* for a longer time, spit a WARNING. Nevertheless, there might be valid reasons, such as completely overloaded hardware. Therefore the default is much higher than --window, around 3600s (1 hour).
--keep-backup-hours=$hours	no	Only relevant for cron and link-purge-all. Old remains from dead / unreachable machines, and some backup data produced by join-cluster and split-cluster (potentially useful for experts), will be purged after this age. Default is 24 * 7 hours.
--autoclean	no	Only relevant for cron. Very old *historic* remains from *decommissioned* machines will be purged after this age. Default is an age of the remains of more than 1 month.
--threshold=$size	no	The macros containing the substring -threshold- or -almost- are using this as a default value for approximation whether some data transfer (e.g. logfile and/or sync) has approximately completed. Default is 10MiB. Notice: when data is continuously appended to the logfile, completeness may *never* be reached. Some data may always fly around somewhere in the network transfer channels. The $size argument may be a number optionally followed by one the lowercase characters k m g t p for indicating kilo mega giga tera or peta bytes as multiples of 1000. When using the corresponding uppercase character, multiples of 1024 are formed instead.
--host=$host	no	The command acts as if the command were executed on another host $host. This option should not be used regularly, because the local information in the symlink tree may be outdated or even wrong. Additionally, some local information like remote sizes of physical devices (e.g. remote disks) is not present in the symlink tree at all, or is wrong (reflecting only the *local* state). THIS OPTION IS DANGEROUS! Use it only for final destruction of dead cluster nodes, see section 3.4.
--ssh-port=$number	no	(deprecated) Only useful when the old ssh-based or rsync-based {join,merge,split}-cluster or join-resource commands are used. When newer mars.ko and marsadm versions are installed throughout the whole cluster, this is not needed anymore.
--no-ssh	no	Avoid any potential timeouts / hangs caused by networks or firewalls, by explicitly disabling the old ssh-based communication method, and relying on the new MARS communication protocol (by defaut on port 7777).
Option	Cmp	Description

Option	Cmp	Description
`--ip=$ip` or `--ip-$peer=$ip`	no	Override the IP information for the local host or `$peer` at the command line. When this option is *not* given, the following rules apply in the following order: 1. lookup the IP for `$peer` from the symlink tree in directory `/mars/ips/`. 2. so-called *probe data* from other hosts in the cluster. This tries to retrieve preliminary information as best as possible. It can however only work when the other peers are reachable, which also implies that in turn *their* currently configurred local peer IP must be correct. 3. Backups in `/mars/backup-$timestamp/` as automatically created by several commands like `merge-cluster` and `split-cluster`. When multiple historic backups are available, the *youngest* version will always win. 4. fallback to a DNS query via `/usr/bin/host`. 5. all local network interfaces are scanned by `/sbin/ip` 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 this type of error-prone automatic IP detection and to explicitly tell the IP address of your *storage network* (which might be different from the ordinary IP address of your host), please use this option for maximum safety. Usually you will need this only at `{create,join,merge}-cluster` to determine any not yet known addresses. Typically, `{leave,split}-cluster` are able to automatically detect historic information from the backups.
`--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 2.4.2). The `mars.ko` 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 network interfaces. Example on hostA: `marsadm --ip=192.168.2.101 create-cluster` Nodenames (see `uname -n` and `man 1 uname` and `man 2 uname`) are the most critical part of your setup. They **must** be **globally unique**, and they **must never change, perpetually. Any (human) error is your fault!** **See also the GPL: NO WARRANTY** **11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.**
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`join-cluster` `$host` -or- `join-cluster` `$host` `$host_ip`	no	Preconditions: the cluster must have been already created with `create-cluster` at another node `$host` (optionally supplying an alternative IP address `$host_ip`). At your local node, the `/mars/` filesystem must be mounted and it must be empty (`mkfs.ext4`, see instructions in section 2.4.2). , The `mars.ko` kernel module must be loaded[a]. Postcondition: the initial symlink tree `/mars/` is replicated from the remote host `$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 on your *local* hostB, and thus the local IP detection is not unique. Be sure to use the right IP, e.g. if you have a dedicated replication network. Similarly, use the optional `$host_ip` parameter if the current primary hostA *also* has *multiple* IP addresses, and thus the partner IP (in the replication network) is also not uniquely deducable from the hostname. Full example on hostB: `marsadm --ip=192.168.2.102 join-cluster hostA 192.168.2.101` [a] In ancient MARS versions before mars0.1astable101 the kernel module *must not* be loaded, and a working ssh connecttion to `$host` must work as root (without password), and `rsync` must be installed at all cluster nodes. PROVISIONARY: in *some* newer MARS versions >= mars0.1astable101, the old ssh-based method is automatically used as a fallback when the kernel module is forgotten to load; however this provisionary workaround shall disappear in future.
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 cluster 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 (rmmod mars). The local `/mars/` filesystem should be manually destroyed thereafte,r, e.g. for decommissioning of hardware. This is *strongly receommended* for preventing "zombies" to resurrect by accident (human error, which is always AT YOUR RSIK). In case of an unintended hardware destruction (e.g. fire, water, ...) this command should be used on another healty cluster node $helper in order to finally remove $damaged from the cluster via the command `marsadm leave-cluster --host=$damaged --force`. An example is explained in section 3.4 on page 39. Before leave-cluster, ensure that all other cluster nodes know that it is no longer participating in *any* resource! Hint: this can be usually achieved by `marsadm leave-resource $resource --host=$damaged --force` In case you cannot use `leave-cluster` for any reason (e.g. complete network shutdown, no communication anymore possible at all), here is an ADVICE for a last resort AT YOUR RISK: 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 re-joining the old cluster, or for creating / joining a new / different cluster, you *must* create a fresh filesystem (see instructions in section 2.4.2) via `mkfs.ext4`. [a] Reason: `leave-cluster` removes only its *own* IP address from `/mars/ips/`, but does not destroy the usual symmetry of the symlink tree by leaving the other IPs intact. Therefore, the local node will continue fetching updates from all nodes present in `/mars/ips/`. As an effect, the local node will *passively* mirror the symlinks of other cluster members, but not vice versa. There is no communication from the local node to the other ones, turning the local node into a **whitness** according to some terminology from Distributed Systems. This is a feature, not a bug. It could be used for porst-mortem analysis, or for monitoring purposes. However, *deletions* of symlinks are not guaranteed to take place, so your whitness may *accumulate* thousands of old symlinks over a long time. If you want to eventually stop all communication to the local node, just run `rmmod`. [b] This is a feature, not a bug. The `uuid` is created once, but alterered anywhere. An exception is `marsadm merge-cluster` (see there). The only way to get rid of the uuid is *external* deletion (not by `marsadm`) *together(!)* with all other contents of `/mars/`. This prevents you from accidentally merging half-dead remains which could have survived a disaster for any reason, such as snapshotting filesystems / VMs or whatever.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`merge-cluster` `$host`	no	Preconditions: The set of resources at the local cluster (transitively) and at the other cluster as addressed by some foreign member `$host` (transitively) must be disjoint. *All(!)* hosts must be mutually reachable via the MARS ports (default 7777 to 7779). Since `mars0.1astable114, ssh` and `rsync` are no longer required, provided that *both* clusters have been fully updated. Otherwise ssh must be working between all members of both clusters, without password (e.g. via ssh-agent). Create the union of both clusters, consisting of the union of all participating machines (transitively). Resource memberships are unaffected. This is useful for creating a "virtual LVM cluster" where resources can be migrated later via `join-resource` / `leave-resource` operations. Usage examples can be found in the Football sub-project of MARS. Attention! use a newer version of MARS. The old branch `0.1.y` does not scale well in number of cluter members, because it evolved from a lab prototype with $O(n^2)$ behaviour at metadata exchange. Never exceed the maximum cluster members as described in appendix B on page 105. For safety, you should better stay at 1/2 of the numbers mentioned there. Use `split-cluster` for going back to smaller clusters again after your background data migration has completed. Future versions of MARS will be constructed for very big clusters in the range of thousands of nodes. Development has not yet stabilized there, and operational experiences are missing at the moment. Be careful until official announcements are appearing in the ChangeLog, reporting of operational experiences from the 1&1 big cluster at metadata level.
`merge-cluster-check` `$host`	no	Check in advance whether the set of resources at the local cluster and at the other cluster `$host` are disjoint.
`split-cluster`	no	This is almost the inverse operation of `merge-cluster`: it determines the minimum sub-cluster groups participating in some common resources. Then it splits the cluster memberships such that unnecessary connections between non-related nodes are interrupted. Use this for avoidance of too big clusters. Since `mars0.1astable114, ssh` and `rsync` are no longer required, provided that *all* hosts are mutually reachable over the default metadata communication port 7777.
`wait-cluster`	no	See section 4.3.3.
`update-cluster`	no	See section 4.3.3.
`create-uuid`	no	Deprecated. Only for compatibility with very old version light0.1beta05 or earlier. Will disappear somewhen in future. Precondition: the `/mars/` filesystem must be mounted. A `uuid` (such as automatically created by recent versions of `marsadm create-cluster`) must not already exist; i.e. you have a very old and outdated symlink tree. Postcondition: the `/mars/uuid` symlink is created for later distribution in the cluster. It uniquely indentifies the cluster in the world. This must be called at most once at the current primary.
Command / Params	Cmp	Description

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 (including a unique `uuid`), that your current node is a valid member of the cluster, and that the kernel module `mars.ko` is loaded. When communication is impossible due to network outages or bad firewall rules, most commands will succeed, but other cluster nodes may take a long time to notice your changes.

Instead of executing `marsadm` commands serveral times for each resource argument, you may give the special resource argument `all`. This work even when combined with `--force`, but be cautious when giving dangerous command combinations like `marsadm delete-resource --force all`.

In newer versions of `marsadm`, you may give a comma-separated list of resource names in place of `all`. This way, you have more fine-grained control over the set of resource names you want to use.

Beware when combining this with `--host=somebody`. In some very rare cases, like final destruction of a whole datacenter after an earthquake, you might need a combination like

`marsadm --host=defective delete-resource --force all`. Don't use such combinations if you don't need them *really*! You can easily shoot yourself in your head if you are not carefully operating such commands!

4.2.1. Resource Creation / Deletion / Modification

Command / Params	Cmp	Description

`create-resource`
`$res`
`$disk_dev`
`[$mars_name]`
`[$size]`
	no	Precondition: the resource argument `$res` must denote a *new* (not yet existing) resource name in the cluster. The argument `$disk_dev` must denote an absolute path to a usable local block device, its size must be greater zero. When the optional `$mars_name` is given, that name must not already exist on the local node; when not given, `$mars_name` defaults to `$res`. When the optional `$size` argument is given, it must be a number, optionally followed by a lowercase suffix `k`, `m`, `g`, `t`, or `p` (denoting size factors as multiples of 1000), or an uppercase suffix `K`, `M`, `G`, `T` or `P` (denoting size factors as multiples of 1024). The given size must not exceed the actual size of `$disk_dev`. It will specify the future resource size as shown by `marsadm view-resource-size $res`.
		Postcondition: the resource `$res` is created, the inital role of the current node is primary. The corresponding symlink tree information is asynchonously distributed in the cluster (in the background). The device `/dev/mars/$mars_name` should appear after a while.
		Notice: when `$size` is strictly smaller than the size of `$disk_dev`, you will unnecessarily waste some space..
		This must be called exactly once for any new resource.

`join-resource`
`$res`
`$disk_dev`
`[$mars_name]`
	no	Precondition: the resource argument `$res` must denote an already existing resource in the whole cluster (i.e. its symlink tree information must have been received; use `marsadm wait-cluster` for achieving this). The resource must have a designated primary, and it must no be in emergency mode. There must not exist a split brain in the cluster. The local node must not be already member of that resource. The argument `$disk_dev` must denote an absolute path to a usable (but currently unused) local block device, its size must be greater or equal to the logical size of the resource. When the optional `$mars_name` is given, that name must not already exist on the local node; when not given, `$mars_name` defaults to `$res`.
		Postcondition: the current node becomes a member of resource `$res`, the inital role is secondary. The initial full sync should start after a while.
		Notice: when the size of $disk_dev is strictly greater than the size of the resource, you will unnecessarily waste some space.
		 After a while, state `Orphan` should be left. Don't forget to regularly monitor for longer occurrences of `Orphan`!

`leave-resource` `$res`	no	Precondition: the local node must be a member of the resource `$res`; its current role must be secondary. It must be detached (see `marsadm down`). The kernel module should be loaded and the network should be operating in order to also propogate the effect to the other cluster nodes. Postcondition: the local node is no longer a member of `$res`. Notice: as a side effect for other nodes, their `cron` (or historic `log-delete`) may now become possible, since the current node does no longer count as a candidate for logfile application. As another side effect, split brain situation may be (partly) resolved by this. Please notice that this command *may likely* resolve split brain (but cannot guarantee in general). 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 needs to 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`. Details are in section 3.4 on page 39.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`delete-resource` `$res`	no	Precondition: the resource must be empty (i.e. all cluster members must have left via `leave-resource`). This precondition is overridable by `--force`, increasing the danger to maximum! It is even possible to combine `--force` with an invalid resource argument and an invalid `--host=somebodyelse` argument in order to desperately try to destroy remains of incomplete or pysically damaged hardware. Postcondition: all cluster members will somewhen be forcefully removed from `$res`. In case of network interruptions, the forced removal may take place far in the future. THIS COMMAND IS *VERY* DANGEROUS! Use this only in desperate situations, and only manually. Don't call this from scripts. You are forcefully using a sledgehammer, even without `--force`! The danger is that the *true* state of other cluster nodes cannot be known in general, e.g. network problems etc. 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 **STONITH**, but on cluster level, affection all known resource members. 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" zombie nodes (beware of shapshot / restores on virtual machines!!). MARS does its best to avoid problems even in case the new resource name should equal the old one, but there can be *no guarantee* in all possible failure scenarios / usage scenarios. Whenever possible, prefer `leave-resource` over this kind of sledgehammer!
`activate-guest` `$res`	no	Precondition: the current host must be a cluster member (see commands `join-cluster` and `merge-cluster`), but need not (yet) be a resource member. The resource must exist somewhere else in the cluster. No additional storage is needed, except a few kilo or megabytes (typically) for the symlink tree. Postcondition: symlink updates with other resource members and/or guests are more frequently. Consequently, marsadm commands and macros with the `--host=` option may be used for remote state inspection, etc. `marsadm view all` will display the guest and its status.
`deactivate-guest` `$res`	no	Precondition: the resource must exist. Postcondition: any previous pure `activate-guest` is rolled back. After about a month,, `marsadm cron` will also remove the guest relationship. This is to protect you from long-term accumulation of unnecessary guest relationships, which are intended only for *temporary* purposes (in constrast to *full* resource memberships requiring storage space for keeping *persistent* replicas).
`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).

With newer versions of `marsadm`, you can also give a list of comma-separated resource names in place of `all`.

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. Another difference to DRBD: the fetch / replay processes etc 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 1.3). Notice: only *local* transfer operations *to* the local disk are paused by a detach. When another node is remotely running a sync *from* your local disk, it will likely remain in use for remote reading. The reason is that the server part of MARS is operating purely passively, in order serve all remote requests as best as possible (similar to the original Unix philosophy). In order to really stop all accesses, do a `pause-sync` on all other resource member where a sync is currently running. You may also try `pause-sync-global`. WARNING! After this, and ather having paused any remote data access, you might use the underlying disk for your own purposes, such as test-mounting it in *readonly* mode. **Don't modifiy** its contents in any way! Not even by an `fsck`[a]! Otherwise, you will have inconsistencies *guaranteed*. MARS has no way for knowing of any modifications to your disk when bypassing `/dev/mars/*`. In case you accidentally modified the underlying disk at the *primary* side, you may choose to resolve the inconsistencies by `marsadm invalide $res` on *each* secondary. [a] Some (but not all) `fsck` tools for some filesystems have options to start only a test repair / verify mode / dry run, without doing actual modifications to the data. Of course, these modes *can* be used. But be really sure! Double-check for the right options!
Command / Params	Cmp	Description

Command / Params	Cmp	Description
pause-sync $res	partly	Equivalent to pause-sync-local.
pause-sync-local $res	partly	Precondition: none additionally. Postcondition: any sync operation targeting the local disk (when not yet completed) is paused after a while (cf section 1.3). 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: additionally, a primary must be designated, and it must not be in emergency mode. Postcondition: any sync operation targeting the local disk (when not yet completed) is resumed after a while. When completed, this operation will remember the switch state forever and become relevant if a sync is needed again (e.g. invalidate or resize).
resume-sync-global $res	partly	Like *-local, but operates on all members of the resource.
pause-fetch $res	partly	Equivalent to pause-fetch-local.
pause-fetch-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any transfer of (parts of) transaction logfiles which are present at another primary host to the local /mars/ storage are paused at their current stage. This switch works independently from {pause,resume}-replay.
pause-fetch-global $res	partly	Like *-local, but operates on all members of the resource.
resume-fetch $res	partly	Equivalent to resume-fetch-local.
resume-fetch-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any (parts of) transaction logfiles which are present at another primary host shouldl be transferred to the local /mars/ storage as far as not yet locally present. This works independently from {pause,resume}-replay.
resume-fetch-global $res	partly	Like *-local, but operates on all members of the resource.
pause-replay $res	partly	Equivalent to pause-replay-local.
pause-replay-local $res	partly	Precondition: none additionally. The resource *should* be in secondary role. Otherwise the switch has *no immediate* effect, but will come (possibly unexpectedly) into effect whenever secondary role is entered later for whatever reason. Postcondition: any local replay operations of transaction logfiles to the local disk are paused at their current stage. This works independently from {pause,resume}-fetch resp. {dis,}connect.
pause-replay-global $res	partly	Like *-local, but operates on all members of the resource.
resume-replay $res	partly	Equivalent to pause-replay-local.
resume-replay-local $res	partly	Precondition: must be in secondary role. Postcondition: any (parts of) locally existing transaction logfiles (whether replicated from other hosts or produced locally) are started for replay to the local disk, as far as they have not yet been applied.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`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 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 updates running in background (default port 7777) are *not* stopped, in order to always propagate as much metadata as possible throughout the cluster. In case of a later incident, chances will be 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 $res	almost	There are three variants: **Variant 1: planned handover (no --force)** Precondition: sync must have finished at any resource member. All relevant transaction logfiles must be either already locally present, or be fetchable (see marsadm up, or low-level commands resume-fetch and resume-replay). When some logfile data is locally missing, there must be enough space on /mars/ to fetch it. Any replay must not have been interrupted by a replay error (see macro %replay-code{} or diskstate DefectiveLog). The current designated primary must be reachable over network. When there is no designated primary (i.e. marsadm secondary had been executed before, which is explicitly *not recommended*), at least the old primary must be reachable. The (old) primarie's virutal device /dev/mars/mydata must not be in use any more (see marsadm wait-umount). A split brain must not already exist. Postcondition: the current host is in primary role; /dev/mars/$dev_name appears locally and is usable; the equivalent of marsadm up $res has been executed as a safeguard against any forgotten previous pause-* operations. Switches the **designated primary**. Description of the **Handover** protocol (when --force is not given): when another host is currently primary, it is first asked to leave its primary role. When systemd templates are active, this will be automatically triggered via systemctl stop $stop_unit. Otherwise, you are resposible for stopping the load yourself, and you dhoulf use marsadm wait-umount in advance for checking. Anyway, the handover procol s waiting until the former primary has actually become secondary. After that, the local host is requested to become primary. Before actually becoming primary, all relevant logfiles are transferred over the network and replayed, in order to avoid accidental creation of split brain as best as possible[a]. Only after that, /dev/mars/$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. 3. use the option --ignore-sync, which leads to a restart of the running sync from position 0. **Variant 2: planned handover (no --force) with sync abort (--ignore-sync)** 2) **Handover ignoring running syncs,** by adding the option --ignore-sync. Any running syncs will restart from scratch, in order to ensure consistency. Use this only when the planned handover is more important than the sync time. **Variant 3: unplanned failover (--force)** 3) **Forced switching**: by giving –force while pause-fetch is active (but not pause-replay), most preconditions are ignored, and MARS does its best to actually become primary even if some logfiles are missing or incomplete or even defective. primary --force is a potentially harmful variant, because it will provoke a split brain in most cases, and therefore in turn will lead to **data loss** because one of your split brain versions must be discarded later in order to resolve the split brain (see section 3.3). **Never** call primary --force when planned handover via 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 have some *chance* for avoiding a split brain even with primary --force. However, there is no guarantee. primary --force switches the *designated* primary. In some extremely rare cases, when *multiple* faults have accumulated in a *weird* situation, it *might* be impossible becoming the / an actual primary. Typically you may be *already* in a split brain situation. This has not

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** and you **should not use** this command during normal operation, including handover. Any resource member which is *not* designated as primary will *automatically* go into secondary role. For example, if you have $k = 4$ replicas, only *one of them* can be designated as a primary. When the network is OK, all other 3 nodes will know this fact, and they will *automatically* go into secondary mode, following the transaction logs from the (new) primary. Hint: avoid this command. It turns off *any* primary, **globally**[a]. You cannot start a sync after that (e.g. `invalidate` or `join-resource` or `resume-sync`), because it is *not unique* wherefrom the data shall be fetched. In split brain situations (when the network is OK again), this may have further drawbacks. It is much better / easier to ***directly* switch the designated primary** from one node to another via the `primary` command. See also section 3.2.2. There is only one valid use case where you *really* need this command: before finally destroying a resouce via the *last* `leave-resource` (or before *forcefully killing* your resource via the dangerous `delete-resource`). [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. Usually, you don't need this. `leave-resource` and `invalidate` are already doing a similar logfile cleanup for you. Use this only as a desperate last resort 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, as could be the case during very awkward disaster scenarios, such as corrupted `/mars` filesystems. MARS does its best for checking anything, but there cannot be an absolute guarantee. That said, no single incident has been observed during millions of operation hours. Hint: use `--dry-run` beforehand for checking!
`err-purge-all` `$res`	no	Precondition: none additionally. Postcondition: errors reported by `marsadm view $res --verbose` are deleted after a while. However notice that some error reports may soon re-appear in case the error condition is persisting.
`link-purge-all` `$res`	no	Precondition: none additionally. Postcondition: all deletable links withspeical value `.deleted`, which have been fully replicated throughout the whole cluster, will be deleted eventually. This is necessary to prevent inode overflow on `/mars`. Notice: `marsadm cron` will do this also.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`resize` `$res` `[$size]`	almost	Precondition: The local host must be primary. All disks in the cluster participating in `$res` must be physically larger than the logical resource size (e.g, by use of `lvm`; can be checked by macros `%disk-size{}` and `%resource-size{}`). When the optional `$size` argument is present, it must be smaller than the minimum of all physical sizes, but larger than the current logical size of the resource. Postcondition: the logical size of `/dev/mars/$dev_name` will reflect the new size after a while.
Command / Params	Cmp	Description

4.2.3. Logfile Operations

Command / Params	Cmp	Description
`cron`	no	Do all necessary housekeeping tasks. This needs to be regularly called by an external cron job or similar.
`log-rotate` `$res`	no	Historic. Please use `cron` instead.
`log-delete` `$res`	no	Historic. Please use `cron` instead.
`log-delete-one` `$res`	no	Historic. Please use `cron` instead.
`log-delete-all` `$res`	no	Historic. Please use `cron` instead.
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, and it must be reachable over network. Postcondition: the local disk is marked as `Inconsistent`, and a fast fullsync from the designated primary will start after a while. Any *local* split brain (deviation from the designated primary) will be resolved, but any *other* split brain at other secondaries will *not* be affected. When the fullsync has finished successfully, the local node will be consistent again. After a while, state `Orphan` should be left. Don't forget to regularly monitor for longer occurrences of `Orphan`!
`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. THIS IS HIGLY DANGEROUS FOR DATA CONSISTENCY! ONLY USE THIS IF YOU REALLY KNOW WHAT YOU ARE DOING! See the WARNING in section 2.6 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 `marsadm` Operations

4.3.1. Inspection Commands

Command / Params	Cmp	Description
`view-`*`macroname`* `$res`	no	Display the output of a macro evaluation. See section 3.1 for a thorough description.
`view` `$res`	no	Equivalent to `view-default`.
`role` `$res`	no	Deprectated, will vanish. Use `view-role` instead.
`state` `$res`	no	Deprectated, will vanish. Use `view-state` instead.
`cstate` `$res`	no	Deprectated, will vanish. Use `view-cstate` instead.
`dstate` `$res`	no	Deprectated, will vanish. Use `view-dstate` instead.
`status` `$res`	no	Deprectated. Use `view-status` instead.
`show-state` `$res`	no	Deprectated, will vanish. Don't use it. Use `view-state` instead, or other macros.
`show-info` `$res`	no	Deprectated, will vanish. Don't use it. Use `view-info` instead, or other macros.
`show` `$res`	no	Deprectated, will vanish. Don't use it. Implement your own macros instead.
`show-errors` `$res`	no	Deprectated, will vanish. Use `view-the-err-msg` or `view-resource-err` similar macros.
`cat` `$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` `$res` *`n`*	no	The argument n must be percentage between 0 and 100 %. When the remaining store space in `/mars/` undershoots the given percentage, the resource will go *earlier* into emergency mode than by the global computation described in section 3.6. 0 means unlimited.
`get-emergency-limit` `$res`	no	Inquiry of the preceding value.
`set-connect-pref-list` `$res host1,host2,hostn`	no	Provisionary and deprecated - this command should hopefully disappear somewhen in future. Avoid it. Set the order of preferences for connections (in DRBD-like speak) when there are more than 2 hosts participating in a resource (aka peers). Of course, these peers need also to be cluster members. The last argument must be a comma-separated list of node names (aka peer names).
`get-connect-pref-list` `$res`	no	Provisionary and deprecated - this command should hopefully disappear somewhen in future. Inquiry of the preceding value.
Command / Params	Cmp	Description

4.3.2.2. Global Parameters

Command / Params	Cmp	Description
`set-global-sync-limit-value` *`n`*	no	Limit the concurrency of sync operations to some maximum number. 0 means unlimited.
`get-global-sync-limit-value`	no	Inquiry of the preceding value.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
set-global-enabled-log-compressions $features	no	Tell the whole cluster which compression features to use globally for logfile compression. The effective value can be checked via marsadm view-enabled-log-compressions. See marsadm view-potential-features and marsadm --help for a list of compression feature names, which must be separated by | symbols. Details are described in section Data Compression and Checksumming (Digests).
set-global-enabled-net-compressions $features	no	Tell the whole cluster which compression features to use globally for network transport compression. The effective value can be checked via marsadm view-enabled-net-compressions. See marsadm view-potential-features and marsadm --help for a list of compression feature names, which must be separated by | symbols. Details are described in section Data Compression and Checksumming (Digests).
set-global-disabled-digests $features	no	Tell the whole cluster which digests to disable globally for checksumming of transaction logfile data. The effective value can be checked via marsadm view-disabled-digests. See marsadm view-potential-features and marsadm --help for a list of compression feature names, which must be separated by | symbols. Details are described in section Data Compression and Checksumming (Digests).
Command / Params	Cmp	Description

4.3.3. Waiting

For scripting, these commands are often needed for race avoidance. MARS' symlink are not propagated *immediately* throughout the whole cluster, but will take some time (also called **eventually consistent** using a so-called Lamport clock). The following commands can be used for triggering a status update, and then waiting until information is recent enough.

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 relevant* nodes in the cluster have sent a status update of their version of the symlink tree, or until timeout. The default timeout is 30 s (exceptionally) and may be changed by --timeout=$seconds Use this for avoidance of rance conditions in the cluster, for nodes pariticipating in some of the local resources. This command does its best to get the current status from the other cluster members. This works only for resources which have been *already* joined. It you want to do a join-resource, use update-cluster instead for fetching / updating the not-yet-joined symlink information also.
update-cluster	no	Precondition: the /mars/ filesystem must be mounted everywhere and it must contain a valid MARS symlink tree produced by the other marsadm commands. The network must be healthy. The kernel module must be loaded everywhere. Postcondition: none. Wait until *all* nodes in the *whole* cluster have sent a status update of their *full* symlink tree, including any joined or non-joined information, or until timeout. The default timeout is 30 s (exceptionally) and may be changed by --timeout=$seconds Use this before join-resource to ensure that *all* symlink information is recent.
Command / Params	Cmp	Description

Command / Params	Cmp	Description
`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 $res`	almost	This is an alias for `wait-cluster` waiting until only those nodes are reachable which are particiapting at `$res` (instead of waiting for *all* hosts participating in *any* of the locally joined resources).
`wait-umount $res`	no	Precondition: none additionally. Postcondition: `/dev/mars/$dev_name` is no longer in use (e.g. umounted).
Command / Params	Cmp	Description

4.3.4. `systemd` Control Commands

These are optional commands when you want to use systemd for control of your services running on top of MARS, and for more automated handover / failover. The concept is described in section The `systemd` Template Generator.

Command / Params	Cmp	Description
`systemd-trigger`	no	Tell the macro processor engine of marsadm that some systemd templates and/or configurations have changed. This will automatically re-compute any necessary systemd units residing in `/run/systemd/system/` from template files provided in one of the directories from `@MARS_PATH/$SYSTEMD_SUBDIR/`, and will automatically remove any *instances* (not templates) which are no longer needed. Finally, `systemctl reload-daemon` is executed, and any necessary units are started / stopped according to the new situation. The macro language behind the template engine is described in section The `systemd` Template Generator. When combined with `--force`, this will forcefully re-compute any template instances, even when already created before. Without `--force`, there is an internal check whether re-creation is necessary. This check might be fooled by manual intervention into `/run/systemd/system/`, so `--force` will repair it.
`set-systemd-unit $res $start_unit $stop_unit`	no	Assign new start and unit names to a resource, or delete any existing names when the empty string `""` is provided as `$start_unit`. Afterwards, a `systemd-trigger` is executed, which causes start / stop operations according to the new situation.
`get-systemd-unit $res`		Report the current assignment of start and stop unit names to the given resource.
`set-systemd-want $res $hostname`		Manually override the host where $start_unit should appear. This is useful for a temporary stop of the applicatiom stack when $hostname has the special value `"(none)"`. See the `fsck` example in section The `systemd` Template Generator. In split brain situations, it *might* be useful for manual override, but notice that `marsadm primary -force` should be preferred because it switches *both* the designated primary *and* the application stack. Notice that the application stack won't actually start at a location where `/dev/mars/$res` is not present.
`get-systemd-want $res`	no	Report the current systemd start location for the given resource.
Command / Params	Cmp	Description

4.3.5. Low-Level Expert Commands

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

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

The following commands are for manual setup / repair of cluster membership. Only to be used by experts who know what they are doing! In general, cluster-wide operations on IP addresses needs to be repeated at all hosts in the cluster iff the communication is not (yet) possible and/or not (yet) actually working (e.g. firewalling problems etc).

Command / Params	Cmp	Description
`lowlevel-ls-host-ips`	no	List all configured cluster members together with their currently configured IP addresses, as known in `/mars/ips/` *locally*.
`lowlevel-set-host-ip` `$hostname` `$ip`	no	Change the assignment of IP addresses in `/mars/ips/` at least *locally*, and *try* to push the information also to other known peers (which is *unreliable* in general). This may be used when hosts are moved to different network locations, or when different network interfaces are to be used for replication (e.g. dedicated replication IPs). Notice that the names of hosts must not change at all, only their IP addresses may be changed, or new peers may be manually added this way. Tip: check active connections with `netstat` & friends. Updates may need some time to proceed (socket timeouts etc). Hint: for safety, call this on *all* members of a cluster to ensure consistency. Otherwise it may happen that some cluster members do not know the *new* IP address where to fetch the *new* information from. See also the description of the `--ip-$peer=$peer_ip` option. For creation of new cluster memberships, always prefer `join-resource`. It checks for any uuid mismatches and for any resource name clashes / violations of resource name uniqueness, which could be **extremely dangerous** for your data.
`lowlevel-delete-host` `$hostname`	no	Please prefer `leave-resource` followed by `leave-cluster` instead. This is only a fallback for *experts*. Useful for decommissioning of dead / physically destroyed hosts after a disaster. Removes a host from the cluster membership in `/mars/ips/` at least *locally*, together with its IP address assignment. It also *tries* to push the deletion to other cluster members (which is *unreliable* in general). This does not remove any further information. In particular, resource memberships are untouched. Please use `leave-resource --host=$name --force` first. Repeat this for *all* former resource memberships. Otherwise you may produce left-over "zombie resource memberships", which in turn may prevent `marsadm cron` from deleting logfile data, and consequently filling up `/mars/` forever. When *all* replicas of any such resource are decommissioned eventually, also use `delete-resource --force` and friends, and/or `marsadm cron --autoclean` after a retention period of 1 month or more. Have a *serious look* at `leave-resource --host=$hostname`, first without `--force`, and also `leave-cluster --host=$hostname`, first without `--force`, which are checking for some common pitfalls.
Command / Params	Cmp	Description

4.3.6. Senseless Commands (from DRBD)

For completeness, here is a list of commands which do not make sense with MARS. Some of them are syntactically parsed for scrpting compatibility, but are not doing anything.

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

4.3.7. Forbidden Commands (from DRBD)

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

Command / Params	Cmp	Description
`invalidate-remote`	no	This would be dangerous in case you have multiple secondaries. A similar effect can be achieved with the `--host=` option.
`verify`	no	This would cause unintended side effects due to races between log-file transfer / application and block-wise comparison of the underlying disks. However, `marsadm join-resource` or `invalidate` will do the same as DRBD verify followed by DRBD resync, i.e. this will automatically correct any found errors;. Note that the fast-fullsync algorithm of MARS will minimize network traffic.
Command / Params	Cmp	Description

5. Tuning, tips and tricks

5.1. IO Performance Tuning

There *exist* some use cases where MARS *can* deliver better IO performance than a raw block device. However, this cannot be expected *in general*. In some *other* cases the performance may be *lower* than with a *single* local raw device.

For demonstration, we use the `blkreplay` tool from http://blkreplay.org and a load which has been captured from a **real datacenter** (1&1 Ionos ShaHoLin = Shared Hosting Linux). The load already contains a parallelism degree of 20 LXC containers running in parallel at the same iron. This corresponds to about 60,000 web spaces running on 20 Apache instances, already in parallel. In difference to artificial benchmarks (like pure random IO or pure sequential IO), this benchmark is much more close to real server operations, while artificial benchmarks are not meaningful for practice in general, because they can deviate from real server operations by *factors* or even by **orders of magnitude**.

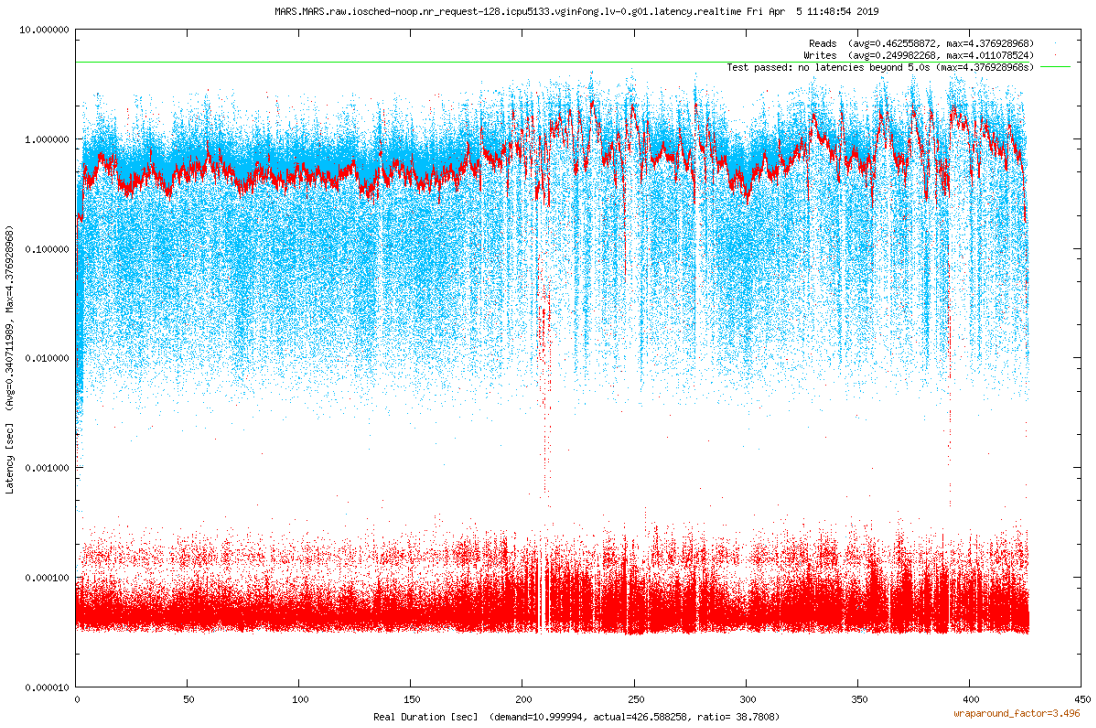
In order to determine the limits of the test candidates, the timing of the original workload was converted to a linear ramp-up, simulating an **overloaded** system. Otherwise benchmarking would not be possible.

The following `blkreplay` benchmarks were executed on an otherwise unloaded Dell R630 with 40 CPU threads on 2 sockets, 192 GB RAM, a Dell R730 hardware RAID controller with 2 GB BBU cache, and 10 spindles Dell 1.8 TB 2.5 inch SAS disks configured as RAID-6. All data, including the `/mars` directory, was located on the hardware RAID via LVM2. `/dev/vginfong/lv-0` was assigned a size of 8 TiB. For testing, vanilla kernel 4.9.x with the MARS pre-patch and `mars0.1astable72` was used.

The `blkreplay` parameters were as follows:

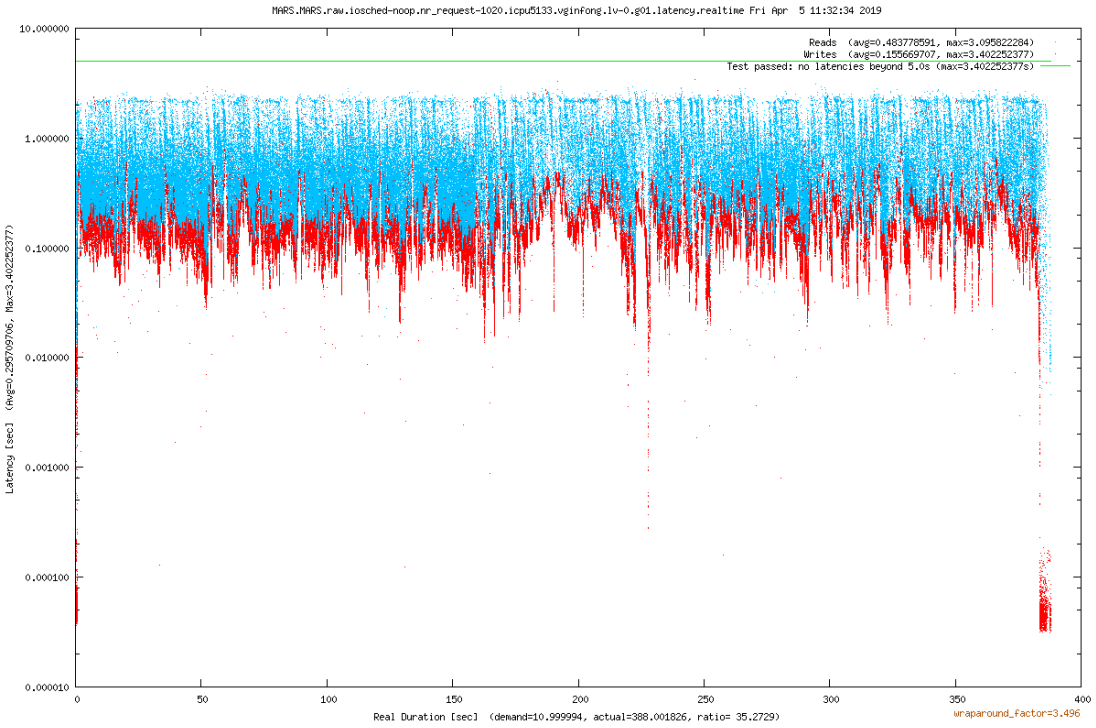
```
output_label="MARS"

# input description
input_file_list="http://blkreplay.org/loads/natural/1and1/shared-hosting/2016/ShaHoLin_
replay_duration=110
speedup=10
threads=512
cmode=with-conflicts
scheduler="noop"

# hardware setup
replay_host_list="icpu5133"
replay_device_list="/dev/vginfong/lv-0"

# output description
enable_graph=1
graph_options="--no-static --dynamic"
```

We start with the **raw** device `/dev/vginfong/lv-0` which had a size of 8 TiB. The throughput is about 1418 IOPS, and the latency diagram shows that the system is overloaded, but can cope with that overload:



As you can see in the filename, the NOOP kernel IO scheduler was used, and the kernel parameter `nr_requests` was left at its default value of 128. When you read the specs of the Dell R730 hardware RAID controller, you will notice that it can handle a much higher IO request parallelism of almost 1024 requests in parallel.

So the first natural tuning attempt is `nr_requests=1020`, in order to release the "kernel IO handbrake". This results in an improved throughput of 1562 IOPS, and even the *maximum* latencies are improved, but the *average* latencies are becoming a little bit worse:



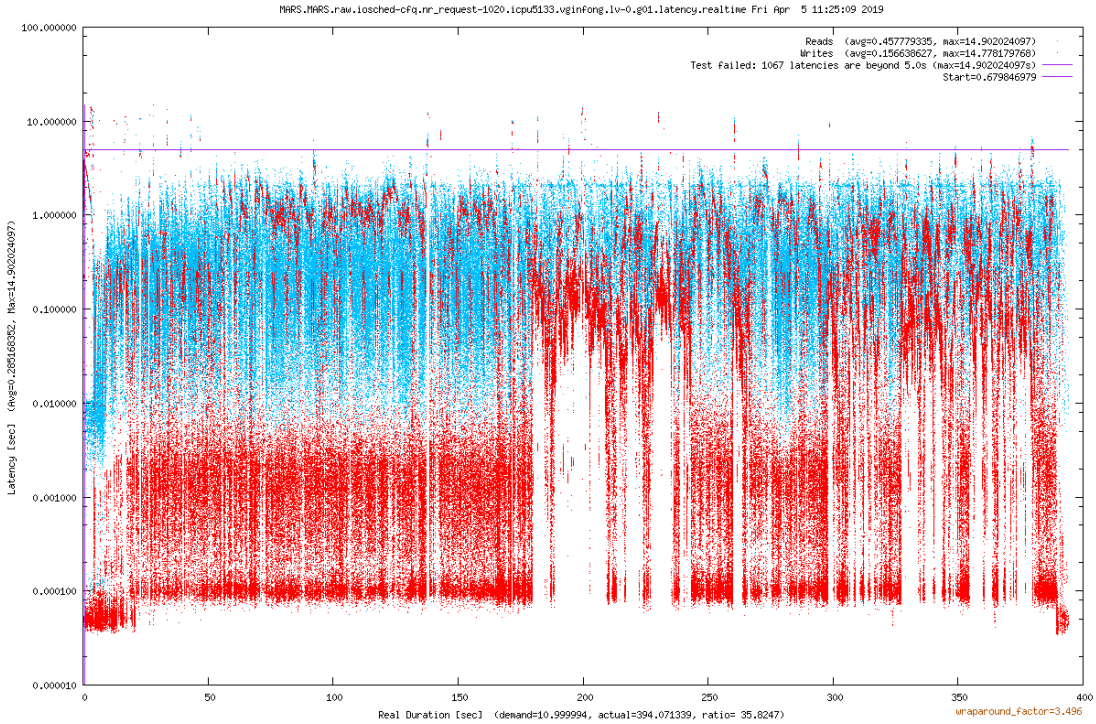
It is well known since decades that there is a principal tradeoff between throughput and latencies in IO systems. Thus it is not a surprising result.

On servers, overload situations should be rare, and during overload throughput is typically

much more important than latencies, as long as latencies are not exceedingly high. Thus we can recommend `nr_requests=1000` for production.

However, some sysadmins might be tempted to question why the NOOP scheduler has been used. On the internet, there are a ton of claims that other IO schedulers like CFQ are much better.

Well, testing with CFQ instead of NOOP is no problem for `blkreplay`. However, the result is very surprising. While the IOPS are 1539, which is only a slight decrease which could result from measurement tolerances, the latencies are now turning almost into a disaster:



In production, you should never encounter IO latencies of almost 15 seconds. So what is going wrong here?

Here is an explanation. A hardware RAID controller *already* has an *internal* IO scheduler. This IO scheduler is hidden in a black box, such that many sysadmins don't know of its existence. If you add another IO scheduler at kernel level, you will have **two different** IO schedulers running in parallel, and sometimes taking **contradictory decisions**.

These contradictory IO scheduling decisions may lead to problems in certain cases and scenarios.

There is another risk of interference with a third IO scheduler, which is MARS' internal asymmetric writeback scheduler. The latter is currently well-tuned for co-working with a BBU cache and its internal scheduler, running on bare metal.

Never use MARS inside of VMs! There you will have *several* additional IO schedulers and further types of IO bottlenecks[1] inbetween, even if you try to disable some of them.

While kernel-level IO schedulers like CFQ certainly have their merits at improving your workstation's IO behaviour, they are counter-productive at servers with hardware RAID controllers.

So the advice is clear: **switch them off** *in such a case.*

Even if you have a software RAID, check with `blkreplay` that any IO schedulers are *really* improving things. Notice that device driver restrictions like `nr_requests` may also work similarly to an IO scheduler. When possible, use your real workload, captured with `blktrace`.

Never use a benchmark which only delivers IOPS! As demonstrated, inappropriate IOPS

[1] Example: data container formats like `qcow2` can act as serious bottlenecks. Never place `/mars` on top of them! Potential exceptions (after well-founded investigations) are functional testing, and certain non-critical workstation-like workloads. In general, never plan to place unknown or enterprise-critical workloads on top of them!

tuning (or choice of inappropriate components) can worsen latencies so much that production can be endangered!

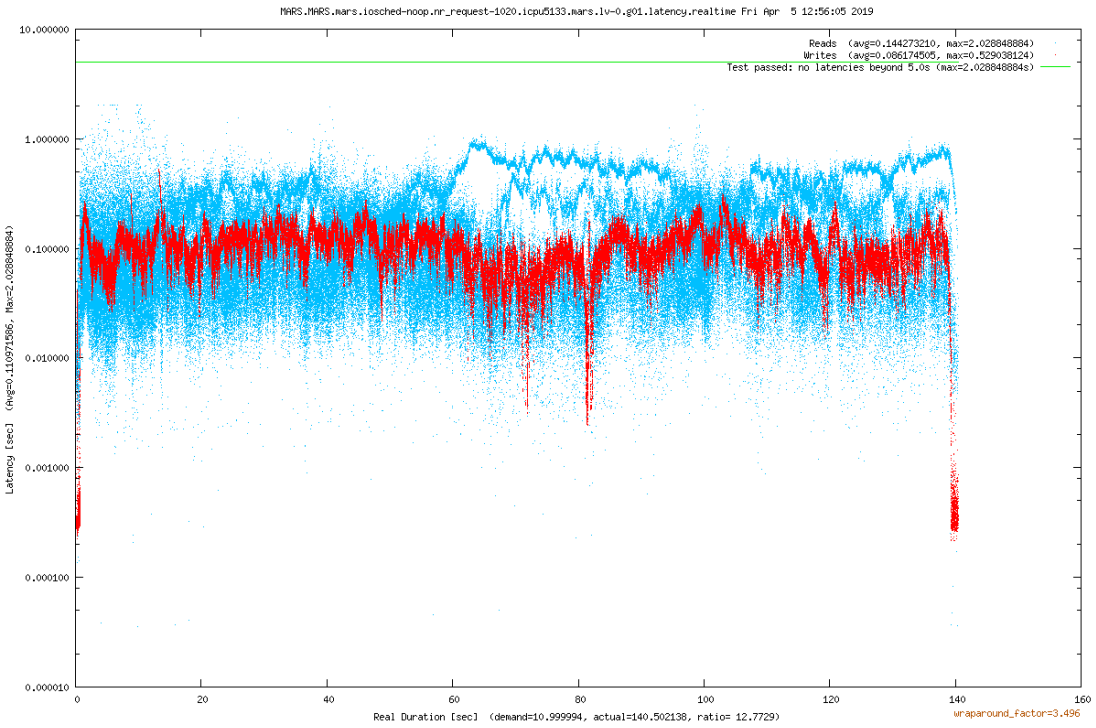
Always look at *both* IOPS *and* latencies!

Average latencies, even when enriched with *standard deviation*, are not enough. Classical statistics does not clearly describe operational problems like **hangs** and **exceptionally high latency requests**, which may occur only rarely, but can then lead to **serious incidents**. Use a tool which can clearly display *any* faulty behaviour, such as `blkreplay`'s **latency diagrams**!

Now we come to benchmarking `/dev/mars/lv-0` placed on top of `/dev/vginfong/lv-0`. Notice that MARS needs to write all write requests twice: once into the transaction logfile, and a second time by writeback into `/dev/vginfong/lv-0`.

So you might expect that performace of `/dev/mars/lv-0` could be worse than at the underlying raw device.

Nevertheless, the **throughput** is now measured 4338 IOPS, which means that performance has **more than doubled**. You can also see it by the duration of the benchmark at the x axis. Even the latencies have improved in many cases:



How is it possible to be faster than a RAW device? How can this be explained?

Look at the graphics and at the explanations from section 1.2 on page 11. The key to local IO performance is the **re-ordering of writeback** according to ascending sector numbers. This can reduce mechanical seek times of hard disks considerably, and even by factors, such that it can over-compensate the doubled writes to the transaction logfile, and even when both are residing at the same RAID set.

Since RAID-6 is has more expensive write operations due to its CRC computations and updates, RAID-6 will much more profit from this effect. Other RAID modes like RAID-10 may show a lower throughput improvement.

Notice: this effect is not only dependent from total RAM size and from the maximum size of the MARS temporary memory buffer (tuning parameter `/proc/sys/mars/mars_mem_percent` which defaults to a limit of 20%). It is also highly dependent from the actual seek behaviour of the **workload**.

For example, if you use `dd` for sequentially overwriting /dev/mars/lv-0 with a parallelism degree of 1, the writeback optimization of MARS cannot be exploited. However, `dd` is no appropriate benchmarking tool, and has almost nothing to do with real workloads occuring in

datacenters, which typically are neither sequential, nor do they have a parallelism degree of only 1.

Please don't try to lead any discussions about this: simply use `blktrace` to capture your real server workload, and compare it to a run of dd. Only if you encounter the same behaviour as `dd`, only then you can really claim that your workload is like `dd`.

Any **assumptions about workloads** are very dangerous: they can deviate from practice not only by factors, but sometimes even by *orders of magnitude*. There no substitute for real measuments of **actual workload behaviour**.

Notice: the writeback optimization of MARS can typically only improve performance of HDDs, but not of SSDs.

By placing `/mars` onto its own physical device with appropriate speed, you can compensate the doubled writes to some degree.

Depending on the workload and on RAID parameters, `/mars` may be better placed onto SSDs, or better be placed on HDDs. There is no general rule. Just use `blktrace` on your real workload, and check several configuration alternatives (also different RAID levels etc) with `blkreplay`.

5.2. Data Compression and Checksumming (Digests)

Data compression can reduce the amount of data which needs to be piped through long-distance or other network bottlenecks. It is available in newer MARS versions, starting from mars0.1astable91. You also need to install the corresponding new version of `marsadm` across the whole cluster.

The locally compiled-in compression and checksumming features as compiled into your currently running `mars.ko` can be queried via

```
marsadm view-implemented-features
```

The output may depend on your kernel compile options, such as the enabled crypto algorithms of your kernel. Typical output should look like

```
CHKSUM_MD5_OLD|CHKSUM_MD5|CHKSUM_CRC32C|CHKSUM_CRC32|CHKSUM_SHA1|COMPRESS_
LZO|COMPRESS_LZ4|COMPRESS_ZLIB
```

In case you get less options, check your kernel `.config` for the corresponding crypto algorithms, which can be compiled into your kernel firmly, or as a module. When necessary, re-compile your kernel with more crypto options enabled (see build instructions in section section §2.2).

When the compile-time option `CONFIG_MARS_BENCHMARK=y` is enabled, `modprobe mars` will show you a list of benchmark results for each enabled crypto algorithm, in units of nanoseconds. Smaller numbers are better. Notice that results may depend on your processor model, and on availability of hardware acceleration (as supported by the crypto infrastucture of your kernel).

Take the benchmark results with a grain of salt. The performance of some crypto algorithms may heavily depend on the *compressibility* of the data to be compressed. `CONFIG_MARS_BENCHMARK` uses a rather artifical test data pattern, which may deviate from the compressibility of your real productive data. Take the results with similar caution than BOGOMIPS, which are also not comparable with other benchmarks in general.

In order to work properly, *all* cluster members must have loaded a newer version of `mars.ko`. During rolling upgrade to newer MARS versions, mixed operation of different MARS versions is supported, even in combination with some old versions supporting only the traditional CHKSUM_MD5_OLD (which has some shortcomings and should not be used anymore in future). Only *common* features are actually usable. You can query the commonly usable options via the commands

```
marsadm view-usable-compressions
```

and

```
marsadm view-usable-digests
```

These should should show you a (possibly empty) list of those options which are really usable *at the moment*. By installing newer / better versions of `mars.ko` and `marsadm`, the list may become longer.

An overview of currently usable options, as well as the actually used algorithms, are displayed at the headings produced by `marsadm view all`.

5.2.1. Network Transport Compression

By default, network transport compression is disabled, since it may worsen the CPU consumption. You can enabled it for the whole cluster via

```
marsadm set-global-enabled-net-compressions\
                              ‘‘COMPRESS_LZO|COMPRESS_LZ4|COMPRESS_ZLIB’’
```

(or a shorter list of compress options), and you can disable it globally by supplying an empty list:

```
marsadm set-global-enabled-net-compressions ‘‘’’
```

Notice: this will compress the *data payloads* of network traffic, both for (incremental) logfile traffic (by default on port 7778), and for sync traffic (by default on port 7779).

5.2.2. Logfile Payload Compression

By default, logfile data compression is disabled, since it may worsen the CPU consumption, and may worsen local IO performance. You can enable it for the whole cluster via

```
marsadm set-global-enabled-log-compressions\
                              ‘‘COMPRESS_LZO|COMPRESS_LZ4|COMPRESS_ZLIB’’
```

(or a shorter list of compress options), and you can disable it globally by supplying an empty list:

```
marsadm set-global-enabled-log-compressions ‘‘’’
```

In difference to network compression, this does not apply to sync data. It compresses the logfile payload *before* it is written to the transaction logfile. As a side effect, it also reduces network traffic, because the logfiles are usually smaller. Additionally, your `/mars` directory may run out of space less quickly.

However, as a major drawback, this may slow down the IO latencies of writes considerably, and thus may drastically reduce local IO performance (depending on performance of your crypto hardware, and on compressibility of data, etc). In particular, ZLIB is known to be a very slow algorithm (but to compress somewhat better than others), while LZO is a very old but very fast algorithm. In many cases, LZO or LZ4 are preferable. Do not enable this option blindly. Always observe the performance of your system afterwards.

5.2.3. Logfile Payload Digests

By default, all of these options are *enabled*, because most users want to checksum the logfile data for detection of hardware errors, such as BBU cache failures, or silent corruption during the network transport of logfile data. When your secondaries encounter a checksum mismatch, they will ***refuse* to apply the defective data**, and will report `DefectiveLog` in the `diskstate` part of `marsadm view all` (see section Standard `marsadm view`).

Most people view this behaviour as a *feature*. It protects you from some types of data corruption.

If you want to disable some or all of the logfile digest algorithms, you can do via

```
marsadm set-global-disabled-log-digests\
   ''CHKSUM_MD5_OLD|CHKSUM_MD5|CHKSUM_CRC32C|CHKSUM_CRC32|CHKSUM_SHA1''
```

Disabling *all* of these options may improve local IO performance, but at the cost of less reliability. However, several compression algorithms are already doing some internal checksumming upon decompression. For maximum performance on weak hardware, it may pay off to enable compression, while disabling separate digesting. Please check what is the best combination for your hardware, your load, etc.

If you decide to keep the logfile digests, e.g. when HA SLAs are more important than maximum performance: notice that checksumming is done at the *input* data *before* any compression is applied. This increases safety against (potential / theoretical) problems with compression / decompression errors.

5.2.4. Network Payload Digests

By default, all of these options are *enabled*, because checksumming over the network at fast full-sync cannot be disabled by concept. At least one of the network digests must always remain enabled. If you try to disable all of them, an automatic fallback to `CHKSUM_MD5_OLD` will occur. Since this a rather slow and non-optimum algorithm, disabling the faster ones (such as `CHKSUM_CRC32C`) is no good idea.

If you want to disable some of the network digest algorithms, you can do similarly to

```
marsadm set-global-disabled-net-digests\
                  ''CHKSUM_MD5|CHKSUM_CRC32C|CHKSUM_CRC32|CHKSUM_SHA1''
```

5.3. The /proc/sys/mars/ and other Expert Tweaks

In many cases, you will not need to deal with tweaks in `/proc/sys/mars/` because everything should already default to reasonable predefined values. This is not a "stable" interface. It may change during development of MARS. It allows access to some *internal* kernel variables of the `mars.ko` kernel module at *runtime*.

This means, all values modified via /proc/ are not persistent. They will be reset to default at `rmmod mars` or at reboot. If you need some persistence, implement it by yourself, e.g. at startup scripts.

This section describes only those tweaks which could be helpful for sysadmins, but not those for developers / very deep internals.

5.3.1. Tuning Network Performance

Since a few years, a feature called "socket bundling" is available.

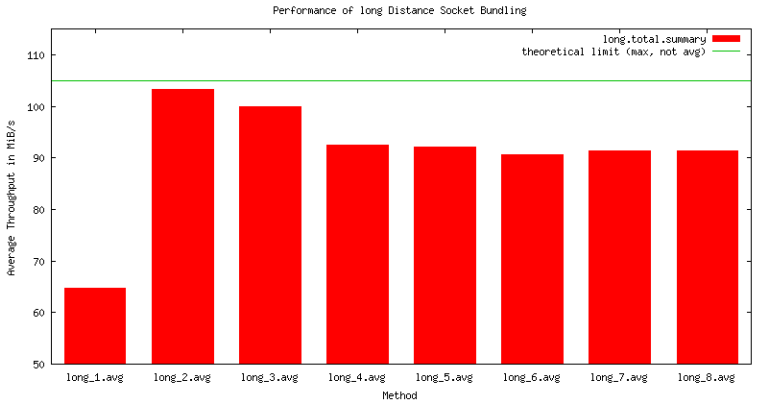
It is mostly intendend for lines showing high packet loss. By using multiple TCP sockets in parallel for emulating a single logical connection, throughput can be significantly increased.

Example for setting the socket parallelism to 4:

- `echo 4 > /proc/sys/mars/parallel_connections`

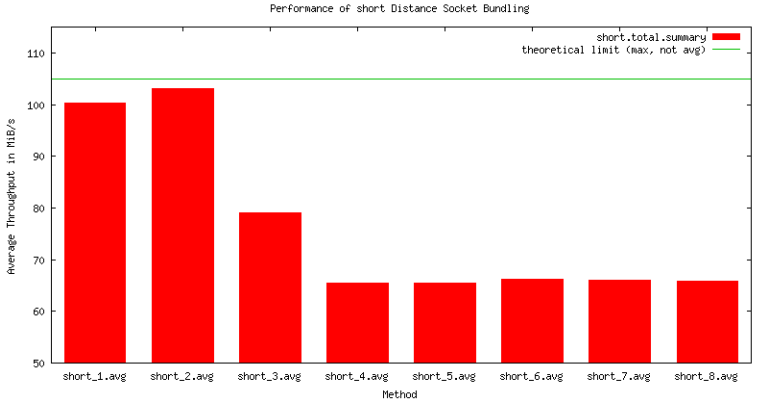
The following graphics shows the throughput of a non-fast[2] fullsync of a *single* 100GiB resource over a loaded long-distance line between Europe/Germany and USA/Midwest. In order to compensate highly varying load at the line, all the experiments were repeated more than 10 times and averaged. Each bar shows the throughput for a particular socket parallelism.

[2] The fast fullsync algorithm would not saturate the `eth0` link with traffic from a single resource.



Notice that the uplinks of the two servers are only 1 GBit/s respectively. When the uplink is saturated, about 100 MByte/s is the maximum possible peak throughput in theory. You can easily recognize that the peak throughput is almost reached with a parallelism degree of 2, but using even more sockets appears to be slightly counter-productive. One of the reasons is that more sockets will increase contention on the line, and thus increasing packet loss. Another potential reason is that higher parallelism at sockets will lead to higher parallelism in disk reads, in turn leading to more permutations of disk read positions (more *random* reads instead of purely sequential reads), which is counter-productive for disk readahead strategies.

The next graphics shows the same, but over a medium distance of about 50km. This line is even more heavily loaded with respect to the number of TCP connections running in parallel (probly some 10,000 or even 100,000 if not more), and there is some kind of "traffic shaping" at some intermediate network gear which will "punish" those traffic sources disproportionally increasing overall packet loss. This can explain the even higher counter-productive effect of using too much sockets and thus injecting additional packet loss:



In general, the optimum value for `/proc/sys/mars/parallel_connections` may depend on many runtime factors such as other load running over some (parts of) physical equipment. You will need to determine optimum values yourself.

Notice that socket bundling is conceptually the "opposite" of traffic shaping. You are trying to get *more* bandwidth, at the cost of *other* traffic competing for the same network resources.

If you are operating masses of servers, don't set the MARS socket parallelism **too high** everywhere. You might "steal" too much bandwidth from other applications when starting masses of syncs in parallel, e.g. after an incident. Best practice is to start with a default value of 1, and to increase it only *on demand*, and/or preferably *only* at those servers where high load really occurs or where some urgent actions need a *temporary* boost.

Experts in networking may try to load-balance the parallel TCP connections over multiple

physical paths, for example by hashing over the dynamic source port numbers. However, we currently have no experience with suchalike setups.

5.3.2. 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)

5.3.2.1. Logging to Files

This feature will likely disappear when MARS goes to kernel upstream. It was mostly intended for debugging during early beta phases and is no longer needed for stable operation. Developers may use it for spotting potential problems.

The classes may be 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 3.6 will appear. Use `*.log` only for a **limited time**, and **only for debugging!**

5.3.2.2. 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.

5.3.2.3. Tuning Verbosity of Logging

`show_debug_messages` Boolean switch, 0 or 1. Mostly useful only for kernel 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.

5.3.3. Tuning the Sync

`sync_flip_interval_sec` (rw) The sync process must not run in parallel to logfile replay, in order to easily guarantee consistency of your disk. If logfile replay would be paused for the full duration of very large or long-lasting syncs (which could take some days over very slow networks), your `/mars/` filesystem could overflow because no replay would be possible in the meantime. Therefore, MARS regulary flips between actually syncing and actually replaying, if both is enabled. You can set the time interval for flipping here.
Increasing this value may improve overall sync throughput, at the cost of some more space required by `/mars`.

`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 and/or your local IO system with too many sync processes running *in parallel*.

`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.

5.3.4. Lowlevel TCP Tuning (Networking Experts Only)

When `CONFIG_MARS_SEPARATE_PORTS` and `CONFIG_MARS_IPv4_TOS` are enabled, MARS uses the following types of traffic:

`MARS_TRAFFIC_META` (by default on port 7777 with `IPTOS_LOWDELAY`) This can be tuned in directory `/proc/sys/mars/tcp_tuning_0_meta_traffic/`.

`MARS_TRAFFIC_REPLICATION` (by default on port 7778 with `IPTOS_RELIABILITY`) This can be tuned in directory `/proc/sys/mars/tcp_tuning_1_replication_traffic/`.

`MARS_TRAFFIC_SYNC` (by default on port 7779 with `IPTOS_MINCOST`) This can be tuned in directory `/proc/sys/mars/tcp_tuning_2_sync_traffic/`. Attention: since the advent of `DSCP`, this bit (hex `0x2` in host byte order) is suppressed by the kernel, and yields `DS0`.

In each of these directories, the following tunables are available (only for networking experts who know what they are doing):

`ip_tos` As explained above. Notice: hex constants from `/usr/include/linux/ip.h` must be converted to decimal before forwarding to the `/proc` interface.

`tcp_window_size` Current default is 8 * 1024 * 1024.

`tcp_nodelay` Current default is 0.

`tcp_timeout` Current default is 2.

`tcp_keepcnt` Current default is 3.

`tcp_keepintvl` Current default is 3.

`tcp_keepidle` Current default is 4.

Further tuning parameters are in the standard Linux kernel. Notice that `IP_TOS` is internally converted to `DSCP`, which in turn can be further manipulated by `netfilter` / `iptables` and/or by `qdisc` (`tc`) and/or by further (external) networking components. The ancient TOS settings are meant as a default *starting point* for further customization to your needs.

Typically, *public* internet transports are flattening / ignoring or otherwise manipulating[3] the TOS / DSCP fields. There it will not work. Anyway, you should never route unencrypted MARS traffic over public transports, for obvious security reasons. Notice: MARS replication is meant for company-*internal* networks like *internal* **replication networks** (or storage networks) where some networking department has control of.

Playing with the above settings can easily tear down your whole (replication) network if you don't know exactly what you are doing. Please test any changes in the lab first. Mass rollout should be done in incremental phases, each in power of 10 units. There might be unexpected effects like packet storms, or packet loss, etc. Some of these effects may only show up when a certain number of hosts is exceeded, or when certain load conditions are hammering the overall Distributed System. Some very old routers / switches are known to break down unexpectedly when overloaded in certain ways. Be careful in a production environment!

[3] DSCP markings can be only made reliable on private networks (possibly requiring some effort). Public Internet service and transit providers do not necessarily treat the TOS values or DSCP markings with any form of priority and may also remove or change them without any notice. Some internet service or transit providers also do use specific DSCP markings to mark packets for being dropped, which may result in hard to find transmission errors.

If want to use MARS on a public internet connection, you should use **encrypted VPN** with different DSCP markings, and coordinate them with your network services provider.

6. Advanced users: automation and the macro processor

6.1. The `systemd` Template Generator

Starting with `mars0.1astable79` (much better with `mars0.1astable119`), you may use `systemd` as a cluster manager at the Mechanics Layer. MARS will replicate any `systemd`-relevant state information across the (big) cluster, achieving some remote control. In particular, **automated handover** triggered by `marsadm primary $resource` is supported. More features are likely to be added to future releases.

6.1.1. Why `systemd`?

All major Linux distributions are now `systemd` based. It is the new quasi standard. Although there have been some discussions in the OpenSource community about its merits and its shortcomings, now it appears to be accepted in large parts of the Linux world.

Systemd has a few advantages:

1. It is running as `init` process under the reserved `pid=1`. If it ever would die, then your system as a whole would die. There is no need for adding yet another MARS cluster-manager daemon `marsd` or similar, which could fail *independently* from other parts of the system.

2. Although `systemd` has been criticised as being "monolithic" (referring to its internal software architecture), its *usage* by sysadmins is easily decomposable into many plugins called ***units***.

3. Local LXC containers, local VMs, `iSCSI` exports, `nfs` exports and many other parts of the system are often already controlled by `systemd`. Together with `udev` and other parts, it already controls devices, LVM, mountpoints, etc. Since MARS is only a particular *component* in a bigger complicated stack, it is an advantage to use the same (more or less standardized and well-integrated) tools for managing the whole stack.

4. Personal experience: the **unit dependency engine** of systemd is extremely elaborated[1] and useful. Management of complex transitive dependencies is relatively easy (though not always fully intuitive). Writing your own dependency engine would be a huge effort, which can be saved by just using standard systemd and learning how to configure it properly for your applications.

In the opinion of the author, systemd has a few disadvantages, such as:

1. It is not **accepted** everywhere. Therefore the `systemd` template extensions of `marsadm` are *not mandatory* for MARS operations. You can use or implement your own alternatives when necessary.

2. Interfacing to **third-party software** may become hairy. systemd appears to assume that more or less *everything* is controlled by systemd. Pre-existing software would sometimes need to be adapted to systemd, but this isn't always possible in practice. For example, `systemd-notify` assumes that you can alter some third-party-controlled executables or complex third-party systems, which often isn't easily possible in practice[2].

[1] The author misses only one feature: a new dependency type, or an object type like a "semaphore" or "mutex" for expressing "mutual exclusion", without any other side effects as for example caused by `Conflicts=` .

[2] The problem is that `systemd-notify` needs to be inserted into the *control flow* of your third-party software. This doesn't work if your third-party software doesn't have the right hooks in the right places, or when there exists no local control flow at some (failure) conditions. Therefore, any status inquiry would need to be

3. Sometimes it can be **messy** to deal with. In particular, it can sometimes *believe* that some parts of the system *were* in a particular state, although in reality they aren't. The current version of systemd lacks an important property called **Idempotence**[3], which is more or less standard in industrial automation and control (e.g. big industry plants). Compensation via native systemd units and dependencies may become hairy, if possible at all[4]. MARS can workaround these shortcomings via a pseudo unit type `.script` which allows you to directly call some (wrapper) scripts, or to write some **adaptors** to third-party software.

4. **Usability** / reporting: in my experience it is less usable for getting an overview over a bigger local system, and less usable (out-of-the-box) for managing a bigger cluster at cluster level. Monitoring needs to be done separately.

6.1.2. Execution Model of systemd and marsadm

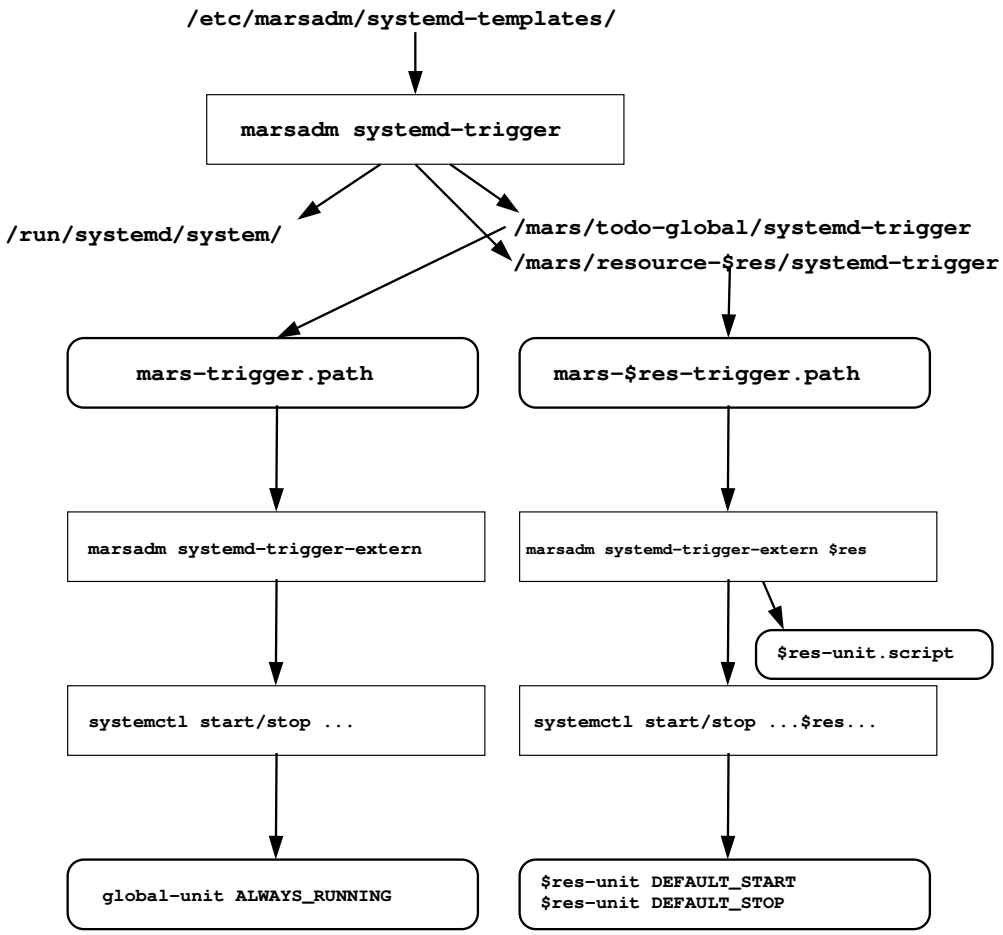
marsadm and systemd are playing together and communicating with each other in the following way:

configurable at systemd units, not by any modifications of third-party software. Unfortunately, the current systemd does not have appropriate features like `ExecCheckState=/my/state-reporting-script.pl` or `.py` or similar.

[3] In industrial automation and control, it is quite standard that you specify the **desired target state**, without having to obey the current state. For example, if a big fan or a big pump is already running, or if a valve is already open, there will be no error if somebody tries to "start" it (for whatever reason, or when induced from a higher level in the control hierarchy). Unfortunately, `systemctl` often reports an error code if you try to start a unit when it is currently running, or when it *believes* that it *were* already running (whether this is actually true or not). *Sometimes*, there is a *workaround* by first stopping, and then restarting, or similar. However, this is clearly an **absolute no-go** for HA where uptime and interruption-free service is often a MUST. For humans who have worked in complex industry plants, it is easy to become *desperate* about this confusing (and sometimes "*unpredictable*") behaviour of systemd, with no easy chance to compensate these deficiencies.

[4] According to `man sd_notify`, there exists a clearly defined message type `STOPPING=1`, but I cannot find a clearly defined and reliable equivalent of STOPPED=1 for reporting that the unit has actually completed the stopping, and that now it is actually (re-)startable (again). Notice that some third-party software needs some external preconditions to be met before it can be started. From the `sd_notify` docs, it is unclear to me whether `ERRNO=0` could be *misused* for such a report. By reading some systemd sourcecode, I cannot easily tell, because the internal state model of systemd appears to be extremely complex, and the message passing model looks even more complex, like a huge finite state automaton with additional non-trivial transitional conditions. Semantically unspecified generic messages like STATUS="freetext_message" cannot not help me for achieving a clearly defined and reliable behaviour.

Notice that some third-party software does not use long-running daemons at all, or only in certain places. Then messages like `MAINPID=` are not useful at all. There exist use cases where **dynamic polling of actual state** would be a very simple and easy solution in place of a complex state-keeping / state transition / message passing model, for example a new unit directive like `ExecCheckState=/my/state-reporting-script.pl` or `.py` or similar.



Communication between marsadm to systemd is done in the following ways:

- by instanting a template residing in /etc/marsadm/systemd-templates (or in another configurable location) and generating an instantiated unit into /run/systemd/system/ as documented in `man systemd.unit`. The method from `man systemd.generator` is currently not yet supported (but possibly in a future release).
- by replicating the following triggers across the mars cluster:
- by touching the cluster-global file `/mars/userspace/systemd-trigger` which in turn is watched at each cluster members by the systemd unit `mars-trigger.path`, which in turn activates `mars-trigger.service`, which in turn executes `ExecStart=/usr/bin/marsadm systemd-trigger-extern` everywhere in the cluster.
- similarly for multiple per-resource triggers, by touching any of the resource-specific trigger files `/mars/resource-$res/systemd-trigger` which is turn is watched by systemd units `mars-$res-trigger.path` at each resource member, which then in turn activates `mars-$res-trigger.service`, which finally executes `ExecStart=/usr/bin/marsadm systemd-trigger-extern $res` everywhere in the cluster.
- In turn, `marsadm systemd-trigger-extern $res` will start or stop the units as configured via `marsadm set-systemd-unit $res $start_unit_name $stop_unit_name`, as described later.

This architecture looks somewhat complicated, but it was found to be necessary to assure lockless parallel starting and stopping of multiple resources without conflicts. Apparently, systemd is not a fully asynchronous system, while a Distributed System (like the MARS replication) is always a fully asynchronous system by its very nature. The above triggers are used to translate from the fully asynchronous Distributed System to the less asynchronous systemd execution model.

Fortunately, you don't need to deal too much with these details if you take the example templates from `systemd-icpu/` and adapt them to your application (see section Example `systemd` Templates).

6.1.3. Working Principle of the Template Generator for `systemd`

systemd already has some basic templating capabilities. It is possible to create unit names containing the `@` symbol, which can then be expanded under certain circumstances, e.g. to tty names etc. However, automatic expansion is only done when somebody knows the instance name already *in advance*. The author has not found any way for creating instance names out of "thin air", such as from dynamically created MARS resource names. Essentially, an *inference machine* for systemd template names does not yet exist.

This lacking functionality is completed with the following macro processing capabilities of `marsadm` (see section 6.2 on page 88):

Some ordinary or templated `systemd` unit files (see `man systemd.unit`) can be installed into one of the following directories: `./systemd-templates`, `$HOME/.marsadm/systemd-templates/`, `/etc/marsadm/systemd-templates/`, `/usr/lib/marsadm/systemd-templates/`, `/usr/local/lib/marsadm/systemd-templates/`. Futher places can be defined by overriding the `$MARS_PATH` environment variable.

From these directories, ordinary systemd unit files will be just copied into `/run/systemd/system/` (configurable via `$SYSTEMD_TARGET_DIR`) and then picked up by `systemd` as ordinary unit files.

Template unit files are nothing but unit files, optionally containing `@{`*`varname`*`}` or `@escvar{`*`varname`*`}` parts or other macro definitions in their filename, and possibly also in their bodies, at arbitrary places. These `@{...}` parts are substituted by the `marsadm` macro processing engine.

The following macro capabilities are currently defined:

`@{`*`varname`*`}` Expands to the value of the variable. This can be used both in template filenames and and in content of template files. Predefined are the following variables:

- `@{res}` The MARS resource name.
- `@{resdir}` The MARS resource directory `/mars/resource-$res/`.
- `@{host}` The local host name as determined by `marsadm`, or as overridden by the `--host=` parameter.
- `@{cmd}` The `marsadm` command as given on the command line (only reasonable for debugging or for error messages).
- `@{`*`varname`*`}` Further variables as defined by the macro processor, see section 6.3.3 on page 101, and as definable by `%let{`*`varname`*`}{...}` statements, see also section 6.2.1 on page 89.

`@eval{`*`text`*`}` Calls the MARS macro processor as explained in section 6.2 on page 88, and substitutes its output. Notice that systemd template variables occurring in the macro processor *`text`* must be accessed via the macro processor syntax `%{varname}`, because the macro processor uses `%` as an escape symbol, while the systemd template engine uses `@` instead. This is necessary for distinction of both layers. Notice that variables defined via the macro processor syntax `%let{varname}{value}` can be afterwards accessed by the template engine via `@{varname}` syntax, once the macro engine has finished working on *`text`*.

`^{`*`varname`*`}` This is a matching operator, binding a string to a variable. It can be used in template filenames *only*. First, any `@{`*`othername`*`}` are substituted. Finally, any `^{`*`varname`*`}` are matched against the actual filename like a shell wildcard `*`. The matching part of the filename is assigned to *`varname`*, and can be later used at `@{`*`varname`*`}` substitutions occurring in the *content* of the file.

`@esc{`*`text`*`}` Calls the `systemd-escape` tool for conversion of pathnames following the `systemd` naming conventions (see `man systemd-escape`). For example, a dash is converted to `\x2d`.

Omission of `systemd-escape` can lead to problems when your resource names are containing special characters like dashes or other special symbols (in the sense of `systemd`). Bugs of this kind are hard to find and to debug. Either forbid special characters in your installation, or don't forget to test everything with some crude resource names!

Example snippet from a `.path` unit. Please notice where escaping is needed and where it must not be used (also notice that a dash is sometimes a legal part of the `.mount` unit name, but except from the resource name part):

```
[Path]
PathExists=/dev/mars/@{res}
Unit=vol-@escvar{res}.mount
```

Another source of crude bugs is the backslash character in the `systemd-escape` substitution, such as from `\x2d`. When passed to a shell, such as in certain `ExecStart=` statements like `/bin/bash -c $args`, the backslash will be removed. Therefore, don't forget to either replace any single backslash with two backslashes, or to put the whole pathname in single quotes, or similar. Always check the result of your substitutions! It depends on the *target* (such as `bash`, as opposed to `systemd`) whether further escaping of the escapes is needed, or whether it *must not* be applied.

Become a master of the escaping hell by inserting debug code into your scripts (reporting to `/dev/stderr` or to log files) and do thorough testing like a devil.

`@escvar{`*`varname`*`}` Equivalent to `@esc{@{`*`varname`*`}}`. This is often used in template filenames to ensure that MARS resource names containing special symbols like dashes, are properly converted to systemd naming conventions where a dash has a different pre-defined meaning.

When creating a new resource via `marsadm create-resource`, or when adding a new replica via `marsadm join-resource` or similar, the template system will automatically create new instances for the new resource or its replicas. Conversely, `marsadm leave-resource` and its friends like `delete-resource` etc will automatically remove the corresponding template instances from `/run/systemd/system/`.

6.1.4. Template Markers

Starting with `mars0.1astable119`, you may use some special markers in comments of systemd templates. These comments are ignored by systemd, but interpreted by the marsadm template engine. The marker syntax is somewhat stricter than in usual comments:

> Exactly one hash symbol exactly at the start of the line, optionally followed by whitespace, followed by the marker in CAPITAL letters.

The requirement of exactly one hash symbol allows you to comment out markers by using two hash symbols or more. Here is a table of markers and their meaning:

Marker	Meaning
# ALWAYS_DISABLED	The unit instance will be generated, but neither enabled nor started by marsadm (except when exclicitly using it as an argument to `set-systemd-unit`). Useful for generation of units you want to control by some other tools.
# ALWAYS_START	This is the opposite of ALWAYS_DISABLED: this unit will always be enabled and started, regardless what `marsadm get-systemd-want` is telling you. If it is no longer in use after leave-resource, only then it will be stopped. Useful for permanent tasks such as `.path` path watcher units, etc.

# KEEP_RUNNING	This unit will always be enabled and started. If the unit is no longer in use after leave-resource, it will be only be disabled (protection against accidental restart), but otherwise stay untouched by marsadm. In particular, it will *not* be stopped if it is currently running. Useful for important HA-like services which need to be controlled by another means. Also useful for *global* units which are independent from any resources.
# ${NAME}_START	This can be used for simplification of the `set-systemd-unit` command. In place of providing a full template name with all of its escape characters and all of its risky typos, simply say `marsadm set-systemd-unit mydata DEFAULT`. This will search for templates marked with DEFAULT_START and DEFAULT_STOP, and use the corresponding `@escvar{res}` substitutions for starting and stopping. HINT: by inventing other marker ${NAME}s, you may ease discrimination of multiple operation modes. Notice: `^{...}` pattern matching is currently NYI, only `@{res}` and variants like `@escvar{res}` are currently implemented.
# ${NAME}_STOP	See description of # ${NAME}_START.

6.1.5. Special `.script` Pseudo Units

Howto use .script When a template name ends in `.script` and has execute permissions, the marsadm template generator will write the instantiated script (with all `@{...}` variables substituted) into the directory `/etc/marsadm/systemd-generated/` directory in place of `/run/systemd/systemd/`.

In place of `systemctl start $unit`, the script `/etc/marsadm/systemd-generated/$unit.script start` is called with single parameter `start`, and similarly for `stop`.

This way, the author found it easy to program idempotent scripts, like adaptors to third-party software.

Motivation for .script The author has spent a lot of time (several months) for getting standard systemd units to work in a fully asynchronous Distributed System consisting of hundreds of machines, where *any* type of event may appear at *any* time.

Although this big effort resulted in a somewhat working system, the result is was not fully HA.

Notice that HA is defined by a single number like 99.99% (see mars-architecture-guide.pdf). In practice, true HA *typically* means that the *total system*, including its controlling components, must be **more reliable** than (some or any) of its worker sub-components. Notice that systemd is one one of these components.

The problem with systemd was that it created failures on its own. Compensation of these failures became very hairy with the current version, as used for the experiments (Debian Buster).

Apparently, systemd does not tolerate certain types of parallelism. For example, while `systemd daemon-reload` is currently executing, a parallel `systemctl start $other_unit` may fail, although it normally works without problems.

Even more unfortunate, sometimes systemd did no longer know the *actual state* of a unit, as the following example output snippet from `systemctl status mnt-test-lv\x2d0.mount` is demonstrating:

```
Dec 24 17:00:48 c1 systemd[1]:  Mounting MARS TESTING local mount on /mnt/test/lv-0...
Dec 24 17:00:58 c1 systemd[1]:  mnt-test-lv\x2d0.mount:  Mount process finished, but there is no mount.
Dec 24 17:00:58 c1 systemd[1]:  mnt-test-lv\x2d0.mount:  Failed with result 'protocol'.
Dec 24 17:00:58 c1 systemd[1]:  Failed to mount MARS TESTING local mount on /mnt/test/lv-0.
```

This failure report was wrong, because the mount had actually succeeded in reality, but systemd wrongly assumed the opposite. Neither `start` nor `stop` was possible afterwards, even after `systemctl reset-failed`.

Possibly this may be a bug, which could be fixed. However, HA systems are expected to even work in the presence of bugs, as best as possible.

If you ever encounter a similar problem, you may get stuck in a dead end, at least for a while. Apparently, systemd is repairing the state of failed mount units after some timeout, but in the meantime true HA with many nines is endangered. Other unit types appear to show similar failure behaviour.

What is the reason for this behaviour?

The author is no systemd expert[5], but here is an *attempt* to analyze the problems at *concept level*, and to suggest some solutions.

The following may be of interest for systemd developers, and for very advanced users of systemd. All others may skip the rest of this paragraph, and just use the `.script` workaround as explained before.

From observation, and from `man systemctl` it appears to me that a `systemctl` command invocation (aka job) is not fully equivalent to an `ExecStart=` directive. According to the docs, `Exec=` and some sister directives are obeying dependencies to any queued or already running jobs. However, I found no obvious way for obeying dependencies when starting jobs via `systemctl` commands *manually*. There is an option `--job-mode=` which defaults to the value `replace`, documenting the bad behaviour I have observed: it may ***disturb*** a currently running or conflicting job.

In worst case, the conflicting unit may fail "unexpectedly" from a user's viewpoint due to the default `--job-mode=replace`, and I have observed cases where it was neither startable nor stoppable immediately afterwards (maybe this could depend on further properties of called `Exec=` scripts, which may stem from a third party, and are not constructed for being called by systemd: eventually I got desperate after investing a lot of time).

Conclusions: there appear multiple non-trivial HA problems with the current version of systemd. While some of them may be bugs which could be fixed, others appear to reside at *concept level*:

1. Jobs seem to be *uniquely identified* by unit names. Thus it is not possible to *address* multiple different jobs belonging to the *same* unit.
   An example: `systemctl start unitA` is currently running, taking a long time to execute. During this, another command `systemctl stop unitA` is started *in parallel* (e.g. from another commandline shell). According to the documented default behaviour `--job-mode=replace`, the old job may be *interrupted* in favour of the new one. However, interruption of a job may be interpreted as a job failure, possibly leading to state `failed`. However, a failed unit can no longer be stopped, it seems to be treated like if it were *already stopped*. It is possible to reset a failed unit via `systemctl reset-failed unitA`, but then I found that it is also treated as if it were already stopped, such that `systemctl stop unitA` just reports "success" without doing anything. This may lead to *serious* problems in a HA world.
   An example: unitA was executing a script `scriptA.sh` in oneshot mode. Unit failure typically meant that the script was killed by a signal. Even if `scriptA.sh` had a signal handler doing proper application cleanup, and even if there were no final SIGKILL which eventually killed the running signal handler, the *application* may get *stuck* in some **undefined application state**. But afterwards, one is unable to queue a stop job for cleanup[6], as explained before. Notice that queuing a start job in place of a cleanup job is no option in general, at least in a HA world.
   According to the docs, there exists no `--job-mode=append` or similar option, which would just queue / delay *any* new job for *any* unit, until all conflicting previous jobs are gone. Notice that this would be always mathematically possible (e.g. resulting in a strictly sequential job order). An implementation should be technically possible, but currently it seems not implemented in systemd according to the docs. This lack of full dependency awareness for entering new jobs causes numerous other problems, not only the previously

[5] The author might have overlooked further possibilities for solutions. This rises the question how the **usability** of systemd could be improved.

[6] Possibly, a new directive `ExecCleanup=` could help. However, this could mean that a new command variant `systemctl cleanup unitA` would need to be introduced. This would complicate the user interface. Instead, the author would prefer Idempotence, because Idempotences allows to use `systemctl stop unitA` for cleanup. This results in a much simpler end-user interface.

described ones.
Possibly (the author has checked[7], which resulted in some behavioural improvements) the `--job-mode=fail` option can be used for a self-programmed busy-wait submission loop, in order to *approach* (but not to *guarantee* in HA sense) disturbance-free addition of new systemctl jobs obeying all specified dependencies, and without causing side effects on already queued / running jobs. However, this trick did not always help in the presence of the above mount unit failure example, triggered by *transitive* dependencies. Another remaining problem is the exit code: there is no documented difference showing the *reason* of the failure. Was it a conflict with another queued job caused by `--job-mode=fail`, or did the unit *itself* fail?

2. Not only in the presence of multiple jobs for the same unit (possibly submitted in parallel to each other), query operations like `systemctl is-active unitA` (or `is-failed` etc) are prone to races. A potential future solution could be externally visible job IDs for identifying jobs. Similar concepts are for example used in printer job spoolers for decades. Of course, information about terminated jobs have to be kept for some time, or until some maximum job entries are exhausted. In addition, true asynchronous invocation via `--no-block` and later polling the job ID would also become possible.

3. Systemd lacks an important property called Idempotence. Idempotence is a very common feature in big industry plants, where hundreds of human workers may act on controlling hundrets of facilities. Each alarm call may cause a different person to try to "fix" a problem. Idempotence is a *must* in such an industry environment. Idempotence means that "start" and "stop" are always working when possible at all, **regardless of the previous state** of the *machinery* (*not* of some *software* state). For operations stuff, there is no need to obey any previous state. When pressing a "start" button, it means "ensure that the machine including all of of its sub-components will be running", regardless whether it was running before, or not, or only half-running, or only an arbitrary subset of sub-components was running before. So the button press does not declare an operation, it declares a **target state**.
It is the task of the Idempotence controller software to ensure that the target state is being reached *somehow*. This does not specify a particular way for reaching the target state. In general, there might be multiple ways. Very sophisticated controller software may even try multiple ways, until one of them succeeds in reaching the specified target state.

A future Idempotence feature of systemd could be implemented as follows:

1. In the unit file, an option for declaration of the Idempotence property could be added. Backwards compatibility: the semantics of conventional non-idempotent units should not be changed unless idempotence is declared explicitly.

2. When Idempotence is declared, `systemctl start` or `systemctl stop` will *always* execute a start/stop job somewhen later, obeying all dependencies. Optionally, this could also mean that in *worst case* all conflicting previous jobs might need to terminate first. More sophisticated variants might provide mechanisms for **controlled abortion** of startup operations, provided that they usually do not lead to failure at all. Care must be taken that failures caused by aborts will not occur too frequently for HA. When failures caused by aborts are occuring too frequently, the concept of abort should be disabled.

3. When `ExecStart=` or `ExecStop=` is declared in an Idempotent unit, the script will be *always* executed, regardless of the previous internal systemd state of the unit. It is up to the script writer to ensure idempotence at his/her script code. This means, the *script* has to approach the application's target state independently from the previous application state, and has to report in its exit code whether the *target state* has been actually reached.

[7] The documentation `man systemd` explains a subtle difference. The conflicts causing `--job-mode=fail` to fail seem to be different from general conflicts in unit dependencies. Although not precisely documented, the observed behaviour luckily appears to make HA more likely. There remains some uncertainity caused by the documented failure possibility. A new option called `--job-mode=append` or `--job-mode=wait` could resolve this, for example by resulting in strict sequential operations in place of "unnecessary" failures.

4. A very helpful addition to *any* systemd unit (not only to Idempotent ones) would be a new directive `ExecCheck=` . In place of keeping a copy of the application state inside of systemd, it would allow to directly **poll** the actual *application* state via callback, whenever it becomes necessary. The user could provide a state-checking script which has no side effects, other than querying the actual state of the application. This feature would be extremely helpful for turning *any* conventional unit into an idempotent one, likely without need for changing anything else (e.g. in third-party software, even when it lacks idempotence). Only when actual state checking is available and when it does not fail, Idempotent units may be allowed to skip the execution of other `ExecStartOrStop=` actions.

In ideal case, `ExecCheck=` or another measure for achieving true HA idempotence might make the above `*.script` workaround superfluous.

6.1.6. Example `systemd` Templates

These can be found in the MARS repo in the `systemd-testing/` and the `systemd-icpu/` subdirectories. Until systemd is supporting idempotence natively, any newbee is advised to take one of the examples `systemd-testing/mnt-test-@escvar{res}-testload.script` and `systemd-icpu/nodeagent-@escvar{res}.script` as a basis for own modifications.

At the moment, the following templates are available (subject to further extension and improvements without notice):

`daemon-reload.service` A helper unit which ensures that `systemctl daemon-reload` is not triggered in parallel to itself. It also reduces the risk for parallel execution with some other units, but unfortunately cannot provide full mutual exclusion with everything else. Hopefully a future version of systemd will allow specification of full mutual exclusion.

`mars.path` This ensures that the mountpoint `/mars/` is already mounted before `mars.service` is started.

`mars.service` This starts and stops the MARS kernel module, provided that `/mars` is (somehow) mounted. The latter can be ensured by classical `/etc/fstab` methods, or by `.mount` units like your own hand-crafted `mars.mount` unit.

`mars-trigger.path` This is used for remote triggering of the marsadm template engine from another MARS cluster member, e.g. when executing `marsadm join-resource` or `leave-resource` or `set-systemd-unit`. Local triggering is also possible via `touch /mars/userspace/systemd-trigger`. When triggered, the command `marsadm systemd-trigger-extern` (without `--force`) will be executed. In turn, this will recompute all `systemd` templates when necessary.

`mars-trigger.service` An intermediate helper unit, sitting inbetween `mars-trigger.path` and `marsadm systemd-trigger-extern`. It also ensures that the marsadm command is never called in parallel to itself.

`mnt-test-@escvar{res}.path` This path unit is used for generic triggering of any `resource-specific systemd` unit as set by `marsadm set-systemd-unit $res $unit` (see below in section 6.1.7).

Pitfall: systemd's path unit watchers are based on the `inotify` infrastructure of the kernel. By default, many kernels are configured to a rather low number of inotify watches. When using more than 20 MARS resources, or when userspace also consumes inotify watches, some of the path watches may not start up (typical `systemctl status` messages are hinting at "resource" or similar). You may need some kernel tweaks, such as `echo 4096 > /proc/sys/fs/inotify/max_user_instances` and/or relatives.

`mnt-test-@escvar{res}.service` Similarly to `mars-trigger.service`, this sits inbetween the per-resource trigger and the executed command `marsadm systemd-trigger-extern @{res}` , which in turn either calls `systemctl` for actually starting / stopping the per-resource units, or in turn it directly calls any `.script` workaround. See the overview picture in section Execution Model of systemd and marsadm.

`mnt-test-@escvar{res}-testload.script` This is an academic example for testing and for inspection, not intended for production. It shows the currently recommended script workaround for achieving idempotence. Fully automatic activation / deactivation of this target during handover via `marsadm primary $res` can be configured via one of the following commands: `marsadm set-systemd-unit mydata DEFAULT` or by the long form `marsadm set-systemd-unit mydata mnt-test-mydata.script`.

Note that the *previous* `*trigger*{path,service}` native units remain necessary for getting `*.script` to work. Any potential races in *their* activation are automatically healed by idempotent re-triggering of this .script workaround.

The following native templates might be used in place of this workaround, but I have to warn that the following native systemd units are not fully passing various stress tests, while the corresponding .script workaround has passed them.

`mnt-test-@escvar{res}.mount` **or** `^{mntname}-@escvar{res}.mount` This is one of the possible native systemd execution targets configurable by `marsadm set-systemd-unit`. For fully automatic activation / deactivation of this alternative target during handover via `marsadm primary $res`, you can configure a very basic test with something like `marsadm set-systemd-unit mydata vol-mydata.mount` or similar.

Notice: the template notation `^{mntname}` can be used for mounting to an *arbitrary* mountpoint, such as `/another/mountdir/mydata`, by using the corresponding systemd template syntax in `marsadm set-systemd-unit mydata another-mountdir-mydata.mount`.

Look into the template file `^{mntname}-@escvar{res}.mount`. In the first line, there is the following macro call:
`@eval{%let{mntpath}{%subst{%{mntname}}{-}{/}}}`
This is a trick for conversion of any systemd template name `mntname` into into an ordinary filesystem pathname `mntpath`. While subdirectories in a path are separated by slashes, the systemd unit naming conventions (as required by systemd) are using dashes in place of slashes.

Do not confuse `@{mntname}` with `@{mntpath}`. Depending on the *type* of argument to be substituted, you may need either systemd unit naming conventions, or classical Unix pathname conventions.

`mnt-test-@escvar{res}-testload.service` This is an academic example for testing and for inspection, not intended for production. Here you can see in comments how a **transitive dependency chain** could be configured. In its body, this template contains a `BindsTo=` plus `After=` reference to another template `^{mntname}-@escvar{res}-delay.service`, which in turn contains a `BindsTo=` plus `After=` reference to `mnt-test-@escvar{res}.mount`.

Do not confuse `Requires=` with `BindsTo=` (see `man systemd.unit`). If you want to automatically stop your *entire* unit stack via a *single* command `systemctl stop vol-mydata.mount`, then you most likely need the stronger `BindsTo=` directive plus `After=` in place of weaker ones like `Requires=` or similar.

In most cases (but not always), you also need an `After=` directive. Otherwise you will unintentionally program a hard to debug **race condition**, which can extinct your last hair. Be sure to understand the corresponding details in the systemd documentation.

In general, it is good practice to have a *consistent* name scheme. Always use the *logically same name* (modulo some escaping conventions for special characters), e.g. for the underlying

LV (called disk in MARS terminology), equal to the MARS resource name, equal to the last part of the mountpoint, equal to the IQN of an iSCSI export, equal to the NFS share name, equal to the LXC container name, equal to the KVM/qemu virtual machine name, and so on. Messing around with non-systematic naming conventions can easily result in a hell.

6.1.7. Fully Automatic Handover using systemd

First, you need to install your systemd templates into one of the template directories mentioned in section 6.1.3. In case you have never used the template engine before, you can create the first instantiation via `marsadm systemd-trigger`. Afterwards, inspect `/run/systemd/system/` for newly created template instances (and `/etc/marsadm/systemd-generated/` for any .script workarounds) and check them.

For each resource `$res`, you should set systemd targets in one of the following variants:

- short form: `marsadm set-systemd-unit $res DEFAULT`
- long form, using the .script workaround: `marsadm set-systemd-unit $res mnt-test-$res-testload.script`
- most general form: `marsadm set-systemd-unit $res ‘‘$start_unit’’ ‘‘$stop_unit”`.

Except for .script workarounds, `$start_unit` and `$stop_unit` will typically denote *different* targets for start and stop (with few exceptions) for the following reason:

Example: assume your native systemd-controlled stack consists of `vol-@escvar{res}.mount` and `nfs-export-@escvar{res}.service`. Before the filesystem can be exported via `nfs`, it *first* needs to be mounted. At startup, `systemd` can do this automatically for you: just add a `Requires=` or `BindsTo=` dependency between both targets, or similar. Then, simply use `nfs-export-mydata.service` as your start unit. Whenever it (thinks that it) needs to be started, systemd will automatically analyze its dependencies and automatically start `vol-mydata.mount`. However, stopping is different. Theoretically, `systemctl stop nfs-export-mydata.service` *could* work in some cases, but in general it doesn't work this way. Reason: there might be other *sister* units which *also* depend on the mount. In some cases, you may not necessarily notice any sisters, because systemd can add further (internal) targets *automatically*. The problem is easily solvable by using `BindsTo=` and/or `PartOf=` dependencies, peferably augmented with `After=`, and then `systemctl stop vol-mydata.mount` will automatically tear down *all* dependencies in reverse order. Therefore, use the *bottom* of the stack (usually a mount unit) as your stop unit.

For maximum safety, `$start_unit` should always point at the *tip* of your stack, while `$stop_unit` should always point at the *bottom* (but one level higher than `/dev/mars/$res`).

Removing any systemd targets is also possible via `marsadm set-systemd-unit $res ‘‘’’`.

Tip: groups of units can be controlled via .target units, see `man systemd.target`.

When everything is set up properly, the following should work:

1. Issue `marsadm primary $res` on another node which is currently in secondary role.
2. As a consequence, `systemctl stop ‘‘$stop_unit’’` should be automatically executed at the old primary side.
3. After a while, the MARS kernel module will notice that `/dev/mars/$res` is no longer opened. You can check this manually via `marsadm view-device-opened $res` which will tell you a boolean result.

   In case the device is not closed for any reason, ordinary handover *cannot* proceed, because somebody could (at least potentially) write some data into it, even after the handover, which would lead to a split brain. Therefore MARS *must* insist that the device is closed before *ordinary* handover can proceed.

   In order to not leave you with a failed service, umount failures will be detected after

a timeout. Handover by marsadm will then *automatically* restart the old start unit at the old primary side where the the device was not released.

In case an ordinary handover is not possible due to hanging device openings, you have the following options:

a) Check your `systemd` configuration or other sources of error why the device is not closed. Possible reasons could be hanging processes or hanging sessions which might need a `kill` or a `kill -9` or similar. Notice that `lsof` does not catch *all* possible sources like (recursive or bind-) mounts occupied by foreign kernel namespaces.

b) Do a failover via `primary --force`, which will likely provoke a split brain.

4. Once `/dev/mars/$res` has disappeared, the ordinary MARS handover from the old primary to the new site should proceed as usual.

5. After `/dev/mars/$res` has appeared at the new site, `systemctl start ''$start_unit''` should be executed automaticy. In turn, this should bring up your configured services.

Details depend on your `systemd` configuration / templates. For example, you can configure systemd targets for activation of VMs, or for `LXC` containers, or for `iSCSI` exports, or for `nfs` exports, or for `glusterfs` exports, or for whatever you need. For true geo-redundancy, you will likely have to include some `quagga` or `bird` or other BGP configurations into your systemd unit stack.

In general, marsadm tries to keep your services running whenever a handover failure occurs, or when you re-attach after a detach, or when your machine reloads `mars.ko` after a crash reboot, etc. This is regarded as a *feature*.

However, this feature could become boring if you *intentionally(!)* want to stop your services, for example when you need to run an `fsck`. Do not use `marsadm secondary`, because this would make `/dev/mars/mydata` to disappear. Although `marsadm set-systemd-unit mydata ''` would solve the problem, this could make you forget the old start and stop unit names (if you don't use markers like DEFAULT etc). You could workaround by some wrapper script remembering the old names via `marsadm get-systemd-unit`, but this is not necessary:

There is a simple solution: `marsadm set-systemd-want ''(none)''` will *temporarily* stop the whole systemd unit stack, while keeping `/dev/mars/mydata` accessible. After your `fsck /dev/mars/mydata` has finished, simply use an idempotent `marsadm primary mydata` for restart of your services.

6.2. The macro processor

The macro processor is a very flexible and versatile tool for **customizing**. Conceptually, two levels of macros are discriminated:

1. primitive macros: these are firmly built into `marsadm`.

2. complex macros: these can be defined via the **macro language** of `marsadm`.

Some complex macros are already pre-defined, for example the standard `marsadm view all` (see section 3.1.1 on page 28).

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

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

6.2.1. Predefined Primitive Macros

6.2.1.1. Intended for Humans

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

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

`all-err-msg` Like before, but show all information including those which are `OK`. This way, you get a list[8] 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).

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

`todo-{attach,sync,fetch,replay,primary,secondary}` 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 1.3. Exceptions: `todo-primary` is not reporting the boolean value of a switch, but means that the designated primary string as reported by `get-primary` is equal to the current host. Similary, `todo-secondary` means that no designated primary exists throughout the cluster, indicating that `get-primary` equals to the special string value `(none)`.

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

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

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

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

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

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

`is-{attach,sync,fetch,replay,primary,secondary,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. Notice: `is-primary` (or its negation `is-secondary`) means that the transaction logger has (resp. not) reached a working state, but the corresponding `/dev/mars/mydata` prosumer device need not (yet) have appeared (somewhere else). More explanations can be found in section 1.3.

`is-split-brain` Shows whether split brain (see section 3.3) 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[9]. Don't confuse this with the consistency of `/dev/mars/mydata`, which is by construction *always* locally consistent once it has appeared[10]. By construction of MARS, the disk of secondaries will *always* remain in a locally consistent state once the initial sync has finished as well as the initial logfile replay. Notice that local consistency does not necessarily imply actuality.

`is-emergency` Shows whether emergency mode (see section 3.7) has been entered for the named resource, or not.

`nr-{attach,sync,fetch,replay,primary,secondary}` Show the *total number* of resource members which are in corresponding state `%is-`*`$something`*`{}`.

`rest-space` (global, no resource argument necessary) Shows the *logically* available space in `/mars/` measured and rounded down in GiB, which may deviate from the physically available space as indicated by the `df` command. The difference to `df` is intended for safeguarding. Computational details may change in future MARS releases.

`total-space` (available since mars0.1astable140, global, no resource argument necessary) Shows the *logically present* space in `/mars/`, which may deviate from the physically available space as indicated by the `df` command. Similarly to `rest-space` but independently from it, this is also safeguarded and rounded down in units of GiB. Computational details may change in future MARS releases.

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

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

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

`device-{ops,amount}-rate` Show the number of current IOPS, esp. the current throughput in KiB/s.

`device-nrflying` Show the number of currently flying IO requests. This is an indicator of queueing at the low-level device. When it is permenantly very high, it may point at IO problems, such as RAID degradation.

`disk-error` Show the negative Linux errno code of the last open() error on the underlying disk. It should be always zero. When < 0 according to kernel return-code conventions, this typically indicates a hardware or LVM problem, etc.

[9] 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.

[10] 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.

`device-error` Show the negative Linux errno code of the last IO error, as reported upwards to applications. It should be always zero. When < 0 according to kernel return-code conventions, this typically indicates a hardware (or network) problem.

{potential,implemented,usable}-features Show a list of flag names, indicating the compression / digest features (see description in section section §5.2) as either as known to the current version of marsadm, or as implemented in the currently running kernel module, or as the minimum feature set currently available in the whole cluster.

{implemented,usable}-{digests,compressions} Same as before, but more specifically related to either compressions or digests.

enabled-{log|net}-compressions Show which compression features have been set by `marsadm set-global-enabled-*-compressions`.

disabled-{log|net}-digests Show which digest features have been disabled by `marsadm set-global-disabled-*-digests`.

used-{log,net}-{digest,compression} Show which digest or compression features are currently actually used by $host, either for logfile or for network purposes.

6.2.1.2. Intended for Scripting

While complex macros may output a whole bunch of information, the following primitive macros are outputting exactly one value. They are intended for script usage (cf. section 6.4). Of course, curious humans may also use 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).

Memberships, Name Querying and their Counts

`is-member` Boolean, indicating whether `%{host}` is a storage member of the resource `%{res}`.

`is-guest` Boolean, indicating whether `%{host}` is currently a *dynamic guest* of resource `%{res}`.

`cluster-peers` Show a newline-separated list of all host names appearing in `/mars/ips/ip-$peername`. Please prefer this macro in place of direct access to filesystem data: future MARS features might need some changes in the low-level naming conventions, and/or in the hierarchical filesystem layout, or even the replacement of the `/mars/` filesystem by different data container formats, etc.

`resource-peers` Show a newline-separated list of all host names participating in the particular resource `%{res}`. Notice that this is typically a subset of `%cluster-peers{}`.
Note: stray data, e.g. as produced by forgotten / interrupted `leave-resource` & co may lead to some mismatches.

`guest-peers` Show a newline-separated list of all host names which are currently dynamically added as *guests* to resource `%{res}`.

`count-{cluster,resource,guest}-peers` Show the corresponding *number* of hosts, accordingly.

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

`{my,all}-members` Show a newline-separated list of storage members existing in the cluster. There is a very subtle difference to `*-resources`: there may exist resources which have no storage members. This may for example occur when all storage members have left via leave-resource, but delete-resource has not yet been executed.

`{my,all}-guests` Show a newline-separated list of currently dynamically added guests.

`count-{my,all}-{resources,members,guests}` Show the corresponding *number* of resources or storage members or guests, accordingly.

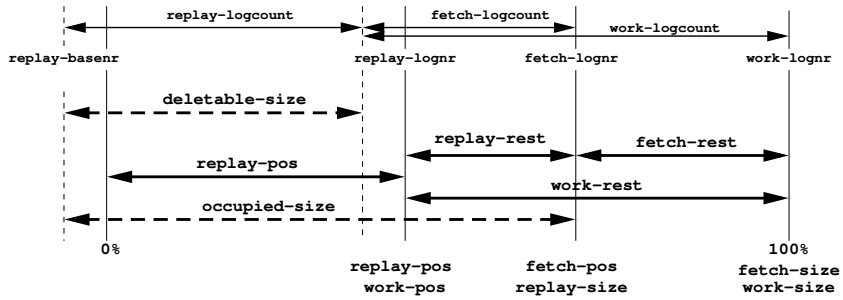


Figure 6.1.: overview on amounts / cursors

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 cron`. 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 3.7).

`disk-size` Show the size of the underlying local disk in bytes.

`resource-size` Show the logical size of the resource in bytes. When this value is lower than `disk-size`, you are wasting space.

`device-size` At a primary node, this may differ from `resource-size` only for a very short time during the `resize` operation. At secondaries, there will be no difference.

The following macros are only meaningful for resources in primary mode:

`writeback-rest` Show the amount of data which is already in the transaction logfile, but has not yet been written back to the underlying disk. This may be used for estimation of recovery time after a potential primary crash. The writeback buffer is explained by the graphics at 1.2 on page 11.

The following macros are only meaningful for resources in secondary mode. By information theoretic limits, they can only tell what is *locally known*. They **cannot** reflect the "true (global) state[11]" of a cluster, in particular during network partitions.

`{sync,fetch,replay,work}-size` Show the total amount of data which is / was to be processed by either sync, fetch, or replay. `work-size` is equivalent to `fetch-size`. `replay-size` is equivalent to `fetch-pos` (see below). Units are *bytes*, not kilobytes.

`{sync,fetch,replay,work}-pos` Show the total amount of data which is already processed (current "cursor" position). `work-pos` is equivalent to `replay-pos`.

[11] 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.

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 `cron`. 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[12].

`{fetch,replay,work}-threshold-reached` Boolean value indicating whether `*-rest` dropped down to `%{threshold}`, which is pre-settable by the `--threshold=`*`size`* command line option (default is 10 MiB). In asynchronous use cases of MARS, this should be preferred over `*-reached` for *human display*, because it produces less flickering by the inevitable replication delay.

`{fetch,replay,work}-almost-reached` Boolean value indicating whether `*-rest` *almost / approximately* dropped down to zero. The default is that at lease 990 permille are reached. In asynchronous use cases of MARS, this can be preferred over `*-reached` for *human display* only, because it produces less flickering by the inevitable replication delay. However, don't base any decisions on this!

`{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[13] 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.

`{sync,fetch,replay}-{ops,amount}-rate` Show the current IOPS of sync / fetch / replay, or its corresponding throughput in KiB/s.

`summary-vector` Show the colon-separated CSV value `%replay-pos{}:%fetch-pos{}:%fetch-size{}`.

`replay-basenr` Get currently first reachable logfile number (see figure 6.1 on the facing page). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{pos,size}` macros instead.

[12] MARS can 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.

[13] 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.

`{replay,fetch,work}-lognr` Get current logfile number of replay or fetch position, or of the currently known last reachable number (see figure 6.1 on page 92). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{pos,size}` macros instead.

`{replay,fetch,work}-logcount` Get current number of logfiles which are already replayed, or are already fetched, or are to be applied in total (see figure 6.1 on page 92). Only for curious humans or for debugging / monitoring - don't base any decisions on this. Use the `*-{rest}` macros instead.

`alive-timestamp` Tell the Lamport Unix timestamp (seconds since 1970) of the last metadata communication to the designated primary (or to any other host given by the first argument). Returns −1 if no such host exists.

`{fetch,replay,work}-timestamp` Tell the Lamport Unix timestamp (seconds since 1970) when the last progress has been made. When no such action exists, −1 is returned. `%work-timestamp{`*`hostname`*`}` is the maximum of `%fetch-timestamp{`*`hostname`*`}` and `%replay-timestamp{`*`hostname`*`}`. When the parameter *`hostname`* is empty, the local host will be reported (default). Example usage: `marsadm view all --macro='%replay-timestamp{%todo-primary{}}'` shows the timestamp of the last reported[14] writeback action at the designated primary.

`{alive,fetch,replay,work}-age` Tell the number of seconds since the last respective action, or −1 if none exists.

`{alive,fetch,replay,work}-lag` Report the time difference (in seconds) between the last *known* action at the local host and at the designated primary (or between any other hosts when 2 parameters are given). Returns −1 if no such action exists at any of the two hosts. Attention! This need not reflect the *actual* state in case of networking problems. Don't draw wrong conclusions from a high `{fetch,replay}-lag` value: it could also mean that simply no write operation at all has occurred at the primary side for a long time. Conversely, a low lag value does not imply that the replication is recent: it may refer to *different* write operations at each of the hosts; therefore it only tells that *some* progress has been made, but says nothing about the amount of the progress.

Device Information

`get-device` Tell the device name, which is `/dev/mars/%{res}` in the current MARS implementation.

`device-present` Boolean, telling whether `/dev/mars/%{res}` is currently appearing at `%{host}` or not.

`device-opened` Tell the number of times `/dev/mars/%{res}` is currently opened (e.g. mounted) at `%{host}`. Upon non-exclusive access by multiple readers / writers in parallel (which is potentially very dangerous), the number may grow greater than 1. You may exploit this for monitoring / supervision.

`device-ops-rate` Tell the current request throughput, aka IOPS. This is actually changing much more frequently than can be reported by the kernel, but anyway may be useful for getting some impression on what is going on.

`device-error` Tell the Unix error code when any IO error has occurred in the past, or 0 when no error is known. Useful for debugging and fault analysis.

`device-nrflying` Tell the number of currently flying IO requests (i.e. submitted, but not yet completed). This is changing in much higher frequency that can be ever reported by the kernel, but may be useful for bottleneck analysis, and when the system is stuck (e.g. defective RAID).

[14] Updates of this information are occurring with lower frequency than actual writebacks, for performance reasons. The metadata network update protocol will add further delays. Therefore, the accuracy is only in the range of minutes.

`device-completion-stamp` Tell the realtime timestamp of the last completed IO request. Useful for detection of a hanging system (e.g. defective disks, etc).

`device-completion-age` Similar to before, but report the *relative* age (compared to the current time) in seconds.

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 1.3 also apply to `get-primary` versus `actual-primary` analogously.

`is-alive` Boolean value indicating whether all other nodes participating in `mydata` are reachable / healthy.

`uuid` (global) Show the unique identifier created by `create-cluster` or by `create-uuid`. Hint: this is immutable, and it is firmly bound to the `/mars/` filesystem. It can only be destroyed by deleting the whole filesystem (see section 4.2).

`tree` (global) Indicate symlink tree version (see `mars-for-kernel-developers.pdf`).

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,secondary}-{on,off}` This may be used to program some useful waiting conditions in advanced macro scripts. It works via busy wait, and does not support disjoint waiting conditions. Use at your own risk! Hint: for disjoint and/or more complex waiting conditions, and/or for programming your own finite state transition machines etc, please prefer the non-blocking `{is,todo}-*` and sisters, and program any busy wait yourself (or try to avoid busy-wait at all).

6.3. 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`.

6.3.1. 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 6.3.3).
- Of course, parameter calls may be (almost) arbitrarily nested.
- Of course, the *correctness* of nesting of braces must be generally obeyed, as usual in any other macro processor language. General rule: for each opening brace, there must be exactly one closing brace somewhere afterwards.

These rules are hopefully simple and intuitive. There are currently no exceptions. In particular, there is no special infix operator syntax for arithmetic expressions, and therefore no operator precedence rules are necessary. You have to write nested arithmetic expressions always in the above prefix syntax, like `%*{7}{%+{2}{3}}` (similar to non-inverse polish notation).

When deeply nesting macros and their braces, you may easily find yourself in a feeling like in the good old days of Lisp. Use the above backslash-newline syntax to indent your macros in a readable and structured way. Fortunately, modern text editors like (x)emacs or vim have modes for dealing with the correctness of nested braces.

6.3.2. Calling Builtin / Primitive Macros

Primitive macros can be called in two alternate forms:

- `%primitive-`*`macroname`*`{`*`something`*`}`
- `%`*`macroname`*`{`*`something`*`}`

When using the `%primitive-*{}` form, you *explicitly disallow* interception of the call by a `*.tpl` file. Otherwise, you may override the standard definition even of primitive macros by your own template files.

Notice that `%call{}` conventions are used in such a case. The parameters are passed via `%{0}` ...`%{n}` variables (see description below).

Standard MARS State Inspection Macros These are already described in section 6.2.1. When calling one of them, the call will simply expand to the corresponding value.

Example: `%get-primary{}` will expand to the hostname of the current designated primary node.

Further MARS State Inspection Macros

Variable Access Macros

- `%let{`*`varname`*`}{`*`expression`*`}` Evaluates both *`varname`* and the *`expression`*. The *`expression`* is then assigned to `varname`.
- `%let{`*`varname`*`}{`*`expression`*`}` Evaluates both *`varname`* and the *`expression`*. The *`expression`* is then appended to `varname` (concatenation).
- `%{`*`varname`*`}` Evaluates *`varname`*, and outputs the value of the corresponding variable. When the variable does not exist, the empty string is returned.
- `%{++}{`*`varname`*`}` or `%{`*`varname`*`}{++}` Has the obvious well-known side effect e.g. from C or Java. You may also use `--` instead of `++`. This is handy for programming loops (see below).
- `%dump-vars{}` Writes all currently defined variables (from the currently active scope) to `stderr`. This is handy for debugging.

CSV Array Macros

- `%{`*`varname`*`}{`*`delimiter`*`}{`*`index`*`}` Evaluates all arguments. The contents of *`varname`* is interpreted as a comma-separated list, delimited by *`delimiter`*. The *`index`*'th list element is returned.
- `%set{`*`varname`*`}{`*`delimiter`*`}{`*`index`*`}{`*`expression`*`}` Evaluates all arguments. The contents of the old *`varname`* is interpreted as a comma-separated list, delimited by *`delimiter`*. The *`index`*'th list element is the assigend to, or substituted by, *`expression`*.

Arithmetic Expression Macros The following macros can also take more than two arguments, carrying out the corresponding arithmetic operation in sequence (it depends on the operator whether this accords to the associative law).

- `%+{`*arg1*`}{`*arg2*`}` Evaluates the arguments, inteprets them as numbers, and adds them together.
- `%-{`*arg1*`}{`*arg2*`}` Subtraction.
- `%*{`*arg1*`}{`*arg2*`}` Multiplication.
- `%/{`*arg1*`}{`*arg2*`}` Division.
- `%%{`*arg1*`}{`*arg2*`}` Modulus.
- `%&{`*arg1*`}{`*arg2*`}` Bitwise Binary And.
- `%|{`*arg1*`}{`*arg2*`}` Bitwise Binary Or.
- `%^{`*arg1*`}{`*arg2*`}` Bitwise Binary Exclusive Or.
- `%<<{`*arg1*`}{`*arg2*`}` Binary Shift Left.
- `%>>{`*arg1*`}{`*arg2*`}` Binary Shift Right.
- `%min{`*arg1*`}{`*arg2*`}` Compute the arithmetic minimum of the arguments.
- `%max{`*arg1*`}{`*arg2*`}` Compute the arithmetic maximum of the arguments.

Boolean Condition Macros

- `%=={`*arg1*`}{`*arg2*`}` Numeral Equality.
- `%!={`*arg1*`}{`*arg2*`}` Numeral Inequality.
- `%<{`*arg1*`}{`*arg2*`}` Numeral Less Then.
- `%<={`*arg1*`}{`*arg2*`}` Numeral Less or Equal.
- `%>{`*arg1*`}{`*arg2*`}` Numeral Greater Then.
- `%>={`*arg1*`}{`*arg2*`}` Numeral Greater or Equal.
- `%eq{`*arg1*`}{`*arg2*`}` String Equality.
- `%ne{`*arg1*`}{`*arg2*`}` String Inequality.
- `%lt{`*arg1*`}{`*arg2*`}` String Less Then.
- `%le{`*arg1*`}{`*arg2*`}` String Less or Equal.
- `%gt{`*arg1*`}{`*arg2*`}` String Greater Then.
- `%ge{`*arg1*`}{`*arg2*`}` String Greater or Equal.
- `%=~{`*string*`}{`*regex*`}{`*opts*`}` or `%match{`*string*`}{`*regex*`}{`*opts*`}` Checks whether *string* matches the Perl regular expression *regex*. Modifiers can be given via *opts*.

Shortcut Evaluation Operators The following operators evaluate their arguments only when needed (like in C).

- `%&&{`*arg1*`}{`*arg2*`}` Logical And.
- `%and{`*arg1*`}{`*arg2*`}` Alias for `%&&{}`.
- `%||{`*arg1*`}{`*arg2*`}` Logical Or.
- `%or{`*arg1*`}{`*arg2*`}` Alias for `%||{}`.

Unary Operators

- %!{*arg*} Logical Not.
- %not{*arg*} Alias for %!{}.
- %~{*arg*} Bitwise Ńegation.

String Functions

- %length{*string*} Return the number of ASCII characters present in *string*.
- %toupper{*string*} Return all ASCII characters converted to uppercase.
- %tolower{*string*} Return all ASCII characters converted to lowercase.
- %append{*varname*}{*string*} Equivalent to %let{*varname*}{%{*varname*}*string*}.
- %subst{*string*}{*regex*}{*subst*}{*opts*} Perl regex substitution.
- %sprintf{*fmt*}{*arg1*}{*arg2*}{*argn*} Perl sprintf() operator. Details see Perl manual.
- %human-number{*unit*}{*delim*}{*unit-sep*}{*number*1}{*number*2}... Convert a number or a list of numbers into human-readable B, KiB, MiB, GiB, TiB, as given by *unit*. When *unit* is empty, a reasonable unit will be guessed automatically from the maximum of all given numbers. A single result string is produced, where multiple numbers are separated by *delim* when necessary. When *delim* is empty, the slash symbol / is used by default (the most obvious use case is result strings like "17/32 KiB"). The final unit text is separated from the previous number(s) by *unit-sep*. When *unit-sep* is empty, a single blank is used by default.
- %human-seconds{*number*} Convert the given number of seconds into hh:mm:ss format.

Complex Helper Macros

- %progress{20} Return a string containing a progress bar showing the values from %summary-vector{}. The default width is 20 characters plus two braces.
- %progress{20}{*minvalue*}{*midvalue*}{*maxvalue*} Instead of taking the values from %summary-vector{}, use the supplied values. minvalue and midvalue indicate two different intermediate points, while maxvalue will determine the 100% point.

Control Flow Macros

- %if{*expression*}{*then-part*} or %if{*expression*}{*then-part*}{*else-part*} Like in any other macro or programming language, this evaluates the expression once, not copying its outcome to the output. If the result is non-empty and is not a string denoting the number 0, the *then-part* is evaluated and copied to the output. Otherwise, the else-part is evaluated and copied, provided that one exists.
- %unless{*expression*}{*then-part*} or %unless{*expression*}{*then-part*}{*else-part*} Like %if{}, but the expression is logically negated. Essentially, this is a shorthand for %if{%not{expression}}{...} or similar.
- %elsif{*expr1*}{*then1*}{*expr2*}{*then2*}... or %elsif{*expr1*}{*then1*}{*expr2*}{*then2*}...{*odd-else-p*
  This is for simplification of boring if-else-if chains. The classical if-syntax (as shown above) has the drawback that inner if-parts need to be nested into outer else-parts, so rather deep nestings may occur when you are programming longer chains. This is an alternate syntax for avoidance of deep nesting. When giving an odd number of arguments, the last argument is taken as final else-part.
- %elsunless... Like %elsif, but *all* conditions are negated.

- `%while{`*`expression`*`}{`*`body`*`}` Evaluates the *`expression`* in a while loop, like in any other macro or programming language. The *`body`* is evaluated exactly as many times as the *`expression`* holds. Notice that endless loops can be only avoided by a calling a non-pure macro inspecting external state information, or by creating (and checking) another side effect somewhere, like assigning to a variable somewhere.

- `%until{`*`expression`*`}{`*`body`*`}` Like `%while{`*`expression`*`}{`*`body`*`}`, but negate the expression.

- `%for{`*`expr1`*`}{`*`expr2`*`}{`*`expr3`*`}{`*`body`*`}` As you will expect from the corresponding C, Perl, Java, or (add your favorite language) construct. Only the syntactic sugar is a little bit different.

- `%foreach{`*`varname`*`}{`*`CSV-delimited-string`*`}{`*`delimiter`*`}{`*`body`*`}` As you can expect from similar `foreach` constructs in other languages like Perl. Currently, the macro processor has no arrays, but can use comma-separated strings as a substitute.

- `%eval{`*`count`*`}{`*`body`*`}` Evaluates the *`body`* exactly as many times as indicated by the numeric argument *`count`*. This may be used to re-evaluate the output of other macros once again.

- `%protect{`*`body`*`}` Equivalent to `%eval{0}{`*`body`*`}`, which means that the body is not evaluated at all, but copied to the output verbatim[15].

- `%eval-down{`*`body`*`}` Evaluates the *`body`* in a loop until the result does not change any more[16].

- `%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 `mars-for-kernel-developers.pdf`), in units of seconds since the Unix epoch.

- `%real-time{}` Return the current system clock timestamp, in units of seconds since the Unix epoch.

- `%sleep{`*`seconds`*`}` Pause the given number of seconds.

- `%timeout{`*`seconds`*`}` Like `%sleep{`*`seconds`*`}`, but abort the `marsadm` command after the total waiting time has exceeded the timeout given by the `--timeout=` parameter.

[15] TEX or LATEX fans usually know what this is good for ;)
[16] Mathematicians knowing Banach's fixedpoint theorem will know what this is good for ;)

Misc Macros

- `%warn{`*text*`}` Show a WARNING:
- `%die{`*text*`}` Abort execution with an error message.

Experts Only - Risky The following macros are unstable and may change at any time without notice.

- `%get-msg{`*name*`}` Low-level access to system messages. You should not use this, since this is not extensible (you must know the name in advance).
- `%readlink{`*path*`}` Low-level access to symlinks. Don't misuse this for circumvention of the abstraction macros from the symlink tree!
- `%setlink{`*value*`}{`*path*`}` Low-level creation of symlinks. Don't misuse this for circumvention of the abstraction macros for the symlink tree!
- `%fetch-info{}` etc. Low-level access to internal symlink formats. Don't use this in scripts! Only for curious humans.
- `%is-almost-consistent{}` Whatever you guess what this could mean, don't use it, at least never in place of `%is-consistent{}` - it is risky to base decisions on this. Mostly for historical reasons.
- `%does{`*name*`}` Equivalent to `%is-`*name*`{}` (just more handy for computing the macro name). Use with care!

6.3.3. Predefined Variables

- `%{cmd}` The command argument of the invoked `marsadm` command.
- `%{res}` The resource name given to the `marsadm` command as a command line parameter (or, possibly expanded from `all`).
- `%{resdir}` The corresponding resource directory. The current version of MARS uses `/mars/resource-%{res}/`, but this may change in future. Normally, you should not need this, since anything should be already abstracted for you. In case you *really* need low-level access to something, please prefer this variable over `%{mars}/resource-%{res}` because it is a bit more abstracted.
- `%{mars}` Currently the fixed string `/mars`. This may change in future, probably with the advent of MARS Full.
- `%{host}` The hostname of the local node.
- `%{ip}` The IP address of the local node.
- `%{timeout}` The value given by the `--timeout=` option, or the corresonding default value.
- `%{threshold}` The value given by the `--threshold=` option, or the corresonding default value.
- `%{window}` The value given by the `--window=` option, or the corresonding default value (60s).
- `%{force}` The number of times the `--force` option has been given.
- `%{dry-run}` The number of times the `--dry-run` option has been given.
- `%{verbose}` The number of times the `--verbose` option has been given.
- `%{callstack}` Same as the `%callstack{}` macro. The latter gives you an opportunity for overriding, while the former is firmly built in.

6.4. Scripting Advice

Both the **asynchronous communication model** of MARS including the Lamport clock, and the **state model** (cf section 1.3) is something you *definitely* should have in mind when you want to do some scripting. Here is some 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 possible to produce a mess. `marsadm` has no internal locking. There is no cluster-wide locking at all, because if would cause trouble during long-distance network outages. 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!

A. FAQ

Following answers are from Thomas Schöbel-Theuer, the inventor of the MARS components and the responsible maintainer of the out-of-tree OpenSource project under GPL.

Q: Why are the units of `marsdm view-rest-space` or `marsdm view-total-space` in GiB?

A: This is for your protection. Contemporary servers have RAM up to TiB, not only GiB. In worst case, e.g. certain types of userspace bugs / attacks / intrusions, large parts of this RAM can be filled via the *page cache of the kernel* by (masses of) wild-running user processes *like hell*, e.g. GiB/ms. Exceptional RAM usage / rates can trigger the **OOM killer**, which in turn may ***randomly*** **kill** almost anything, **including daemons**[1].

Q: Why isn't `marsadm` a daemon, but follows the process paradigm from UNIX?

A: Same story: reasonable protection against whoever kills processes for whatever reasons. Daemons can **loose their runtime state**, and this may happen even in *critical sections*. Thus your **(distributed) system(s) may get (widely) stuck**. More explanations are in section Why `systemd`? on page 77. This can become *evil* when it happens just during an operational incident, requiring a (mass) failover, and just when you *desperately* want to *rely* on MARS (see also the CAP theorem). In contrast, `marsadm` (as a component) is **stateless**[2] *between* invocations. Thus your system(s) can **recover more easily**.

Q: Can I create a fork of MARS?

A: Yes, given that you **conform to the license**. Please read the GPL, and follow it. This is not only about kernel code. `marsadm` will remain under GPL. I have *created* it under European personal rights, and even during my precious spare time. Personally, I will *not* publish anything which is *not* under an OSI and/or OIN-compliant license. In addition, it is *wise* to obey the customs of the OpenSource **community**. I **firmly recommend** to take the time to read **Homesteading the Noosphere** (a ground-breaking essay from Eric Raymond, see original web document at `http://catb.org/~esr/writings/homesteading/homesteading/`), and to think about its consequences. Just stating the obvious, do not break important basics from there. Finally, keep in mind that it is mandatory to follow the written and unwritten rules from the Linux kernel community.

Q: Will MARS become part of the official Linux kernel as published at `www.kernel.org`?

A: Hopefully. The decision is up to Linus Torvalds and the kernel community. If possible, I will try to do my best, in order to get it there[3]. After acceptance, I am willing to support it further as a member of the community, up to my death, under my personal constraints after my retirement. I am also planning to continuously support the community, and to consent to passing over the baton to somebody *from the Linux*

[1] Exceptional fill rates are dangerous by itself. Neither the kernel nor MARS can *reliably* prevent such "use cases". We altogether can only *try* to *deal with it* and to protect us *as best as possible* under certain conditions. Detailed high-frequency monitoring would be difficult in practice. Via units of GiB, I try to *hint* you on this large problem field.

[2] `marsadm` does not *functionally* depend on network traffic, at least for non-network operations. The *kernel(!) and its GPLed code* will *automatically* cleanup most remains upon process kill. **Repeatability**: afterwards, the next `marsadm` invocation should not depend on former (OOM-like) problems.

[3] Hopefully, the *out-of-tree* kernel part of MARS will no longer be needed by anyone in the long term. Any (professional) support for the MARS kernel module, and also for `marsadm`, should hopefully migrate to the official Linux kernel and/or to the OpenSource community of all interested Linux distros, in a reasonable timeframe.

kernel community. My current problem is lack of time. I estimate that I will need more than 70% of my precious working hours for getting it into the kernel. Please help me. Please talk *directly* with me about perspectives.

Q: What is the **business value** of any `OpenSource` for my company?

A: Short story for managers / controllers / consultants: talk with **experts from the Linux Foundation**.
Some recommended details top-down via checklist:

1. Consider http://catb.org/~esr/writings/homesteading/homesteading/ a *successful* management strategy for *creation* of *globally shared commons* value[4].
2. Understand the fundamental differences between commons *cultures* and company *cultures*, and their relationships to management *cultures*.
3. Map typical *usage patterns* of `OpenSource` down to your company, like follows. Cross-compare your **savings** (if you would need to *develop* and to *maintain* the software *fully yourself*, and/or to *buy* and *integrate* and operate comparable products on non-`OpenSource` markets), and compute the corresponding **earnings**[5] via best-practice management methods.
4. Finally, take the **future** of business value alternatives, spread over the **actual lifetime**, into account.
5. Ultimately, take the long-term **ecosystem effects** of *all* your downloaded and/or contributed `OpenSource` into account, without overlooking so-called *hidden champions*.

Obviously, delegate OpenSource (strategy) only to persons who have *proven*[6] that they *actually master* the OpenSource playgrounds according to its intrinsic rules. Do not put the fox in charge of the henhouse[7].

This FAQ will be continued / updated only when necessary, and according to my time constraints.

[4] Recommendation: do not follow a common misbelief that something would not deliver any value because it is *for free*. Also, do not mix up English meanings of "free". When unsure, please consult the well-known publications from Richard Stallman, who explained the fundamentals.

[5] Optionally, further effects like **scaling effects** may be also relevant.

[6] Recommendations: do not forget to take the *actual* OpenSource **affinity** of people into account, in which **culture** they have **grown up**. Just believing some personal claims is not enough. Check for any people who *grew up during their youth* in a non-affine culture, and do not trust a counter-culture where fighting *against* OpenSource is or was a fashion. Beware of (hidden) ***clash of cultures***. Check for contradictions. Guard your company against hidden agendas and against *fakes*. Valuable OpenSource is *not* created by management affinity, but *only* by *certain hand-selected* **individuals**. Carefully select the *right* individual(s) for your company. Do not forget to ask for **personal references**, best on *actual individual creations*, and *check* them.

[7] Similar saying: do not trust the cat to keep the cream.

B. Technical Data MARS

Do not use MARS inside of VMs. Only use at bare metal!

MARS has some built-in limitations which should be overcome[1] by future MARS versions. Please don't exceed the following limits:

- maximum 10 nodes per cluster
- maximum 30 resources per cluster
- maximum 100 logfiles per resource

[1] Some internal algorithms are quadratic. The reason is that MARS evolved from a lab prototype which wasn't originally intended for enterprise grade usage, but should have been succeeded by the fully instance-oriented MARS Full much earlier.

C. HISTORIC Guide for Midnight Problem Solving

Here is a *generic guide* (**not to be treated as instructions**) for the *generic* `marsadm` and commandline level. Do not forget the terms and conditions from the GPL.

This is ***not* an operational documentation**. There are too many other application-dependent components in a real production system. Any other levels / layers (e.g. different types of cluster managers, PaceMaker, control scripts / `rc` scripts / `upstart` or `systemd` scripts, etc) *need* to be described elsewhere. Midnight problem solving means to engage sysadmins familiar with (1) *all* of these components, and (2) with any *potential interferences* between them.

Aristoteles taught us that a system is more than the sum of its components. You need your own *maintained* handout for problem solving at *full-system* level.

C.1. Inspecting the State of MARS

For manual inspection, please prefer the new `marsadm view all` over the old `marsadm view-1and1 all`. It shows more appropriate / detailed information.

```
# watch marsadm view all
```

Checking the low-level network connections at runtime:

```
# watch "netstat --tcp | grep 777"
```

Meaning of the port numbers (current *default* as configured into the kernel module, may change in future):

- 7776 = prosumer device traffic (realtime requirements, only at new MARS versions)
- 7777 = metadata / symlink propagation (low traffic)
- 7778 = transfer of transaction logfiles (load dependent)
- 7779 = transfer of fast-fullsync traffic (only when fast-fullsync is running)

7777 must be always active on a healthy cluster. All others will appear only on demand, when resources are configured accordingly and/or when some data is transferred.

Hint: when one of the columns Send-Q or Recv-Q are constantly at high values, you might have a network bottleneck.

When unsure or desperate: check that at least the *server-part* sockets 7777 to 7779 are *always* present after modprobe mars and in state LISTEN, e.g. via

```
# netstat -lp --tcp | grep 777
```

Hint: I have seen strange incidents where somebody else had *occupied* some of these server sockets via some bugs in some userspace scripts. There might be other network-level problems, e.g. network hardware has firmware bugs, or when network devices are misconfigured, or when some strange firewalling rules are kicking in, etc. Or even *trivial* problems, like electrical problems, faulty cables or contacts, bad temperatures, humidity, etc.

C.2. Replication is unexpectedly Stuck

Indications for a stuck:

- `marsadm view all` displays some human-readable *warnings* containing the keywords `age` or `hint` and some hours:minutes:seconds numbers.
- Other warnings containing the keyword error and some errno code / text (see `man errno` but notice that the Linux kernel uses negative numbers in place of positives). You need to read and understand the meaning of these `errno` numbers in the context of you Linux version and `mars.ko`.
- One of the flags shown by `marsadm view all` or `marsadm view-flags all` contain a symbol `"-"` (dash). This means that some switch is currently switched off (deliberately). Please check whether there is a valid reason why somebody else switched it off. If the switch-off is just by accident, use the following command to fix the stuck:

  ```
  # marsadm up all
  ```

  (or replace `all` by a particular resource name if you want to start only a specific one). Note: `up` is equivalent to the sequence `attach; resume-fetch; resume-replay; resume-sync`. Instead of switching each individual knob, use `up` as a shortcut for switching on anything which is currently off.
- `netstat --tcp | grep 777` does not show anything. Please check the following:
  - Is the kernel module loaded? Check `lsmod | grep mars`. When necessary, run `modprobe mars`.
  - Is the network interface down? Check `ifconfig`, and/or `ethtool` and friends, and fix it when necessary. Use `journalctl -k` to detect any potential kernel messages, such as network driver problems.
  - Is a `ping <partner-host>` possible? If not, fix the network / routing / firewall / etc. When fixed, the MARS connections should automatically appear after about 1 minute.
  - When `ping` is possible, but a MARS connection to port 7777 does not appear after a few minutes, try to connect to remote port 7777 by hand via `telnet`. But don't type anything, just abort the connection immediately when it works! Typing anything will almost certainly throw a harsh error message at the other server, which could unnecessarily alert other people.
- Check whether `marsadm view all` shows some progress bars somewhere. Example:

  ```
  istore-test-bap1:~# marsadm view all
  ---------- resource lv-0
   lv-0 OutDated[F] PausedReplay dCAS-R Secondary istore-test-bs1
      replaying: [>...................]  1.21%  (12/1020)MiB
  logs: [2..3]
         > fetch: 1008.198 MiB rate: 0 B/sec
  remaining: --:--:-- hrs
         > replay: 0 B rate: 0 B/sec  remaining: 00:00:00 hrs
  ```

  At least one of the `rate:` values should be greater than 0. When none of the `rate:` values indicate any progress for a longer time, try `marsadm up all` again. If it doesn't help, check and repair the network. If even this does not help, check the hardware for any IO hangups, or kernel hangups via `journalctl` and sisters. Check the RAID controllers. Often (but not certainly), a stuck kernel can be recognized when many processes are *permanently* in state "D", for a long time: `ps ax | grep " D" | grep -v grep` or similar. Please check whether there is just an overload, or *really* a true kernel problem. Discrimination is not easy, and requires experience (as with any other system; not limited to MARS). A truly stuck kernel can only be resurrected by rebooting. The same holds for many hardware problems.

- Check whether `marsadm view all` reports any lines like `WARNING: SPLIT BRAIN at '' detected`. In such a case, check that there is *really* a split brain, before obeying the instructions in section C.4. Notice that network outages or missing `marsadm cron` may continue to report an old split brain which has gone in the meantime.

- Check whether `/mars/` is too full. For a rough impression, `df /mars/` may be used. For getting authoritative values as internally used by the MARS emergency-mode computations, use `marsadm view-rest-space` (the unit is GiB). In practice, the differences are only marginal, at least on bigger `/mars/` partitions. When there is only few rest space (or none at all), please obey the instructions in section C.3.

- Check `journalctl -k` specifically for any kernel-level problems.

C.3. Standard Resolution of Emergency Mode and Split Brain

Emergency mode occurs when `/mars/` runs out of space, such that no new logfile data can be written anymore.

In emergency mode, the primary will write any write requests *directly* to the underlying disk, as if MARS were not present at all. Thus, your application will continue to run. Only the *replication* as such is stopped.

Notice: emergency mode means that your secondary nodes are usually in a *consistent*, but *outdated* state (exception: when a sync was running in parallel to the emergency mode, then the sync will be automatically started over again). You can check consistency via `marsadm view-flags all`. Only when a local disk shows a lower-case letter `"d"` instead of an uppercase `"D"`, it is known to be inconsistent (e.g. during a sync). When there is a dash instead, it usually means that the disk is detatched or misconfigured or the kernel module is not started. Please fix these problems first before believing that your local disk is unusable. Even if it is really inconsistent (which is very unlikely, typically occurring only as a consequence of hardware failures, or of the above-mentioned exception), you have a big chance to recover most of the data via `fsck` and friends.

A currently existing emergency mode can be detected by

```
primary:~# marsadm view-is-emergency all
secondary:~# marsadm view-is-emergency all
```

Notice: this delivers the current state, telling nothing about the past.

Emergency mode *may* also show something like `WARNING: SPLIT BRAIN at '' detected` because both problems may sometimes appear in combination. Anyway, the standard resolution process is equal.

Preconditions:

- The network **must** be working. Check that the following gives an entry for each secondary:

  ```
  primary:~# netstat --tcp | grep 7777
  ```

  When necessary, fix the network first (see instructions above).

- The hardware must be healthy, and any RAID subsystems *should not* be in any degraded state

  It is your *operational decision* whether to prefer relatively fast RAID rebuild after handover of any primary to some secondary and/or pausing some secondary-side IO for some time, or whether you drastically increase the TTR = Time To Repair by loading your RAID *unnecessarily*.

  MARS is constructed for working "almost as best as possible" even on slow IO systems and over slow networks, but *do not expect miracles*. The laws of physics are stronger than any human expectations!

Emergency mode should now be resolved, e.g. by checking `marsadm view all`, and/or via commands like

```
primary:~# marsadm view-is-emergency all
primary:~# du -s /mars/resource-* | sort -n
```

Remember the affected resource names and their *logfile storage s*ize. Do *not* invalidate on *any* non-SPLIT_BRAIN or non-Emergency resources. Do *not* invalidate at the target where you **might need to failover** somewhen in the next time! It is your *responsible operational decision* which resource on which secondary host you want to repair via `invalidate` at a certain point in time!

Be aware of the *officially documented fact* that `marsadm invalidate` (similarly to `drbdadm invalidate`) will make your secondary replica `Inconsistent` until the following fast-fullsync has successfully finished!

Depending on operational strategies for certain use cases and/or hardware defects and/or load situations etc, it *might* be better to wait some (relatively short) time to start `marsadm invalidate` *later*.

Once you have decided to use `invalidate` on certain resources, best practice is to start with the *biggest* resource as shown by `du -s /mars/resource-* | sort -n` in reverse order:

```
secondary1:~# marsadm invalidate <res1>
secondary1:~# marsadm view <res1>
primary:~# marsadm cron
```

Typical non-overloaded servers allow you to invalidate about 3 to 5 (or *sometimes(!)* even tens of) resources in parallel, but highly overloaded machines may require `marsadm pause-sync $some_resource` for load reduction. When more resources are affected by SPLIT BRAIN, regularly check whether `df /mars/` shows enough space to wait some time until you need to invalidate more resources according to `du -s /mars/resource-* | sort -n` .

Hint: during the resolution process, some other resources might have gone into emergency mode concurrently. In addition, it is possible that some secondaries are stuck at particular resources while the corresponding primary has *not yet* entered emergency mode. Please repeat the steps in such a case, and look for any emergency modes at secondaries additionally. When necessary, extend your list of *affected* resources and/or secondary hosts.

Hint: be patient. Deleting large bulks of logfile data may take a long time, at least on highly loaded systems. You should give the cleanup processes at least 5 minutes before concluding that `marsadm invalidate` followed by primary-side `marsadm cron` had no effect! When unsure, use `marsadm cron` at *all* cluster nodes, even when seemingly unaffected.

Notice: neither a secondary-side `marsadm invalidate` can be noticed by the current primary side, nor `marsadm cron` can work *fully* when the metadata update over port 7777 is extremely slow, or when it does not work for whatever reason. Check your replication speed!

In very complex scenarios, when the primary roles of different resources are spread over diffent hosts (aka mixed operation), you may need to repeat the whole cycle iteratively for a few cycles until the complex jam is fully resolved in the full Distributed System.

If it does not disappear fully, you have another chance by the following split-brain and emergency-mode resolution process:

C.4. Alternative Resolution of Split Brain / Emergency Mode / Defective Hardware

Hint: in most cases, the resolution of emergency mode and/or split brain via `marsadm invalidate` (plus some additional `marsadm cron`) should resolve practically almost everything,

at least when done in the right order and on the right hosts / their resources. However, it cannot always work miracles on certain types of **defective hardware**.

Here is a more capable but more complex alternative, typically useful in **complex failure scenarios**, e.g. after **cascading disasters** & co and after the hardware + software + network had been **repaired** successfully.

The following can only work when the *original content* of `/mars/` has survived your incident. Otherwise, you may need a *variant* of this method as explained later, and probably you should not do this at midnight. Read this *completely* and understand it, before starting any action!

Never run MARS inside of VMs, see `architecture-guide-geo-redundancy.pdf`. If you do so anyway, or if you place MARS *on top of* some LV snapshot(s) or similar, you are running a **high risk**: any restore of an *old snapshot* affecting the content of `/mars/` and/or of some underly resource disk / *underlying* LV and/or its content can **provoke a disaster!**

That said, the following method often works when the underlying resource data had been somewhat damaged or even lost at *some(!)* of the replicas (due to the incident), but `/mars/` has survived and the storage is operational again. For exampley, any defective RAID disks have been already replaced and the underlaying RAID is now *rebuilt* and *no longer* is in **degraded RAID** mode – check your RAID first!

Precondition: the hardware + software + network must be working, and any RAID is no longer degraded. Check that the following gives an entry for each secondary:

```
primary:~# netstat --tcp | grep 7777
```

When necessary, fix the network first (see instructions above). Be sure the hardware and any RAID is fully operational.

Inspect the split brain situation:

```
primary:~# marsadm view all
primary:~# du -s /mars/resource-* | sort -n
```

Remember those resources where a message like `WARNING: SPLIT BRAIN at '' detected` appears. Do the following only for *affected* resources, starting with the biggest one (before proceeding to the next one).

Do the following with only *one* resource at a time (before proceeding to the next one), and repeat the actions on that resource at every secondary (if there are multiple secondaries):

```
secondary1:~# marsadm leave-resource $res1
```

ONLY WHEN some secondaries are FINALLY LOST FOREVER and WILL NEVER RE-APPEAR like a "zombie", try the following instead:

```
alternative-secondory-or-primary:~# marsadm leave-resource $res1 \
                                    --force --host=$defective_secondary1
```

Check whether the split brain has vanished everywhere, and the replication is actually working on any non-stucked secondary. Startover with other resources at their secondaries when necessary.

Finally, when no split brain is reported at any (former) secondary, do the following on the primary:

```
primary:~# marsadm cron
primary:~# sleep 30
primary:~# marsadm view all
```

Now, any split brain and/or emergency mode should be gone even at the primary. If not, repeat this step (and some of the previous ones).

In case even this should fail on some `$res` (which is very unlikely), read the PDF manual before using `marsadm log-purge-all $res`.

ONLY WHEN some secondary had been FINALLY LOST FOREVER: before running `marsadm join-cluster` on any new replacement hardware where `/mars/` is initially empty, you should first remove the old hardware from the cluster:

```
alternative-secondary-or-primary:~# marsadm leave-cluster \
                                    --force --host=$defective_secondary1
```

ONLY WHEN a former primary had been LOST FOREVER: before you can forcibly remove a defective primary as described above, you must first **decide** which surving secondary will be the surviving new primary, and you *need* a **successful failover** as described in section C.6 Failover = Emergency Switch of Primary Role.

Do not forget to *ensure* that suchalike zombies cannot "resurrect" by itself! Read *all* relevant parts of this manual!

Finally, when the split brain + emergency mode is gone everywhere, rebuild the redundancy at every (newly deployed *fully*, including `marsadm join-cluster`) secondary host via

```
secondary1:~# marsadm join-resource $res1 /dev/<lv-x>/$res1
```

Here is the LAST RESORT if you are *completely desperate*, and if even the previous methods did not help after *multiple attempts*, and when you are *sure* that a hardware / software / kernel / etc upgrade / downgrade cannot help: setup the whole cluster afresh by `rmmod mars` everywhere, and creating a fresh `/mars/` filesystem everywhere, followed by the same procedure as installing MARS for the first time (which is outside the scope of this guide).

C.5. Handover of Primary Role

When there exists a *reliable* method for primary handover in higher layers such as cluster managers, please prefer that method (e.g. `cm3` or other tools).

If suchalike doesn't work, or if you need to handover some resource `$res1` by hand, do the following:

- Stop the any *customer* network traffic first. This is important for overload prevention, and for security reasons, and for minimization of any potential split brain data in case of any failures (see also the famous CAP theorem).
- Stop the load / application processes corresponding to `$res1` on the old primary side.
- `umount /dev/mars/$res1`, or otherwise close any openers such as iSCSI.

  Do not skip this step: otherwise a *handover* (not to be confused with *failover*) is *not possible*.

  When `umount` & co does not work for whatever reason: then it is your *responsible operational decision* whether you want to (a) restart the full application stack again at the old primary side (via whatever method, including reboot), or (b) accept a *potential* split brain and its potential redundancy loss, and proceed with section C.6 Failover = Emergency Switch of Primary Role.

  Informed operational decisions can be supported by familiarity with the famous CAP theorem.
- At the new primary: `marsadm primary $res1`
- Restart the application at the new site (in reverse order to above). In case you want to switch *all* resources which are not yet at the new side, you *cautiously may* use `marsadm primary all`.

When overload of the new primary side is possible, e.g. when the **kernel caches are cold**, it is *your operational decision* whether to (a) handover all resources *in parallel* (which is a tested functionality of MARS), or (b) to avoid overload caused by applications, initiate a *sequential* handover resource-by-resource, obeying that the new primary side often needs to have **warmed its kernel caches** sufficiently in order to meet your SLAs.

Important: by definition, *handover* is *not* an emergency operation like *failover* is. It is *your operational decision* how many resources you want to handover from any location X to any other location Y in whatever parallelism degree. Do not blame MARS if doesn't run faster than a local reboot and its inevitable cache warming phases.

- Finally, check that the network is up again (e.g. BGP) and that everything is running fine at the new location (from a customer's perspective).

C.6. Failover = Emergency Switch of Primary Role

Failover = emergency switching is necessary when your primary is no longer reachable over the network for a *longer* time, or when the hardware is defective.

Do not blame MARS for anything which is outside its scope residing at kernel level. Wheter and when a failover is needed for whatever reason, and in which parallelism degree, is clearly outside the scope of a *component* like MARS is. The same would be true for DRBD in place of MARS.

Failover = emergency switching will often lead to a split brain, see the famous CAP theorem. It *may* require manual actions to resolve (see above), depending on your type of automation and its automation degree, and depending on the **management-level risks** caused by improper automation and/or bugs / misinformation / influences from the famous CAP theorem in whatever higher-level component.

Therefore, try to avoid failover = emergency switching when possible!
Hint: MARS can automatically recover after a primary crash / reboot, as well as after secondary crashes, just by executing `modprobe mars` after `/mars/` had been mounted. Please *consider* to wait until your system comes up again, instead of risking a split brain.

The decision between failover = emergency switching and continuing operation at the same primary side is an operational one. MARS can support your decision by the following information at the potentially new primary side (which was in secondary mode before):

```
istore-test-bap1:~# marsadm view all
---------- resource lv-0
lv-0 InConsistent Syncing dcAsFr Secondary istore-test-bs1
syncing: [====>.............] 27.84% (567/2048)MiB rate: 72583.00 KiB/sec remaining: 0
> sync: 567.293/2048 MiB rate: 72583 KiB/sec remaining: 00:00:20 hrs
replaying: [>:::::::::::::::::::] 0.00% (0/12902)KiB logs: [1..1]
> fetch: 0 B rate: 38 KiB/s remaining: 00:00:00
> replay: 12902.047 KiB rate: 0 B/s remaining: --:--:--
```

When your target is syncing (like in this example), you cannot switch to it (same as with DRBD). When you had an emergency mode before, you should first resolve that (whenever possible). When a split brain is reported, try to resolve it first (same as with DRBD). Only in case you *know* that the primary is *really* damaged, or it is *really* impossible to the run the application there for some reason, failover = emergency switching is desirable for typical use cases like webhosting or cloud computing (see `architecture-guide-geo-redundancy.pdf`).

Hint: in case the secondary is inconsistent for some reason, e.g. because of an incremental fast full-sync, you have a last chance to recover most data after forceful switching by using a filesystem check or suchalike. This might be even faster than restoring data from the backup. But use suchalike only if you are *really* desperate and *have no other chance*!

The amount of data which is *known* to be missing at your secondary is displayed by `marsadm view all` after the `> fetch:` in human-readable form. However, in cases of networking problems this information may be *outdated*.

You *always* need to consider further facts which cannot be known by MARS.

When there exists a method for failover = emergency switching of the primary in higher layers such as cluster managers, please prefer that method in front of the following one.

If suchalike doesn't work, or when a handover attempt has failed several times, or if you *really need* forceful switching of some resource `$res1` by hand, you can do the following:

- When possible, stop the *customer* network traffic + application corresponding to `$res1` on the old primary side.
- When possible, `umount /dev/mars/$res1`, or otherwise close any openers such as iSCSI.
- When possible (if you have some time), wait until as much data has been propagated to the new primary as possible (watch the `fetch:` indicator).
- At the new primary: `marsadm pause-fetch $res1; marsadm primary --force $res1`
- Restart the application at the new site (in reverse order to above).
- Check that everything is up again (e.g. BGP and customer traffic etc) and that everything is running fine at the new location (from a customer's perspective).
- After the application is known to run reliably, check for split brains at MARS level and cleanup them when necessary.

D. HISTORICAL Methods for Split Brain Resolution

HISTORICAL / DO NOT USE ANYMORE: Instead of `marsadm invalidate`, the following steps may be used. In preference, start with the old "wrong" primaries first:

1. `marsadm leave-resource mydata`
2. After having done this on one cluster node, check whether the split brain is already gone (e.g. by saying `marsadm view mydata`). There are chances that you don't need this on all of your nodes. Only in very rare[1] cases, it might happen that the preceding `leave-resource` operations were not able to clean up all logfiles produced in parallel by the split brain situation.
3. Read the documentation about `log-purge-all` (see page 59) and use it.
4. If you want to restore redundancy, you can follow-up a `join-resource` phase to the old resource name (using the correct device name, double-check it!) This will restore your redundancy by overwriting your bad split brain version with the correct one.

It is important to resolve the split brain *before* you can start the `join-resource` reconstruction phase! In order to keep as many "good" versions as possible (e.g. for emergency cases), don't re-join them all in parallel, but rather start with the oldest / most outdated / worst / inconsistent version first. It is recommended to start the next one only when the previous one has sucessfully finished.

[1] When your network had partitioned in a very awkward way for a long time, and when your partitioned primaries did several `cron` 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`.

E. Alternative De- and Reconstruction of a Damaged Resource

In case `leave-resource --host=` does not work, you may use the following fallback. On the surviving new designated primary, give the following commands:

1. `marsadm disconnect-all mydata`
2. `marsadm down mydata`
3. Check by hand whether your local disk is consistent, e.g. by test-mounting it readonly, `fsck`, etc.
4. `marsadm delete-resource mydata`
5. Check whether the other vital cluster nodes don't report the dead resource any more, e.g. `marsadm view all` at *each* of them. In case the resource has not disappeared anywhere (which may happen during network problems), do the `down ; delete-resource` steps also there (optionally again with `--force`).
6. Be sure that the resource has disappeared *everywhere*. When necessary, repeat the `delete-resource` with `--force`.
7. `marsadm create-resource newmydata ...` at the *correct* node using the *correct* disk device containing the *correct* version, and further steps to setup your resource from scratch, preferably under a different name to minimize any risk.

In any case, **manually check** whether a split brain is reported for any resource on any of your *surviving* cluster nodes. If you find one there (and only then), please (re-)execute the split brain resolution steps on the affected node(s).

F. Cleanup in case of Complicated Cascading Failures

MARS does its best to recover even from multiple failures (e.g. **rolling disasters**). Chances are high that the instructions from sections 3.3 3.4 or appendix D E will work even in case of multiple failures, such as a network failure plus local node failure at only 1 node (even if that node is the former primary node).

However, in general (e.g. when more than 1 node is damaged and/or when the filesystem `/mars/` is badly damaged) there is no general guarantee that recovery will *always* succeed under *any* (weird) circumstances. That said, your chances for recovery are *very* high when some disk remains usable at least at one of your surviving secondaries.

It should be very hard to finally trash a secondary, because the transaction logfiles are containing `md5` checksums for all data records. Any attempt to replay currupted logfiles is refused by MARS. In addition, the sequence numbers of rotated logfiles (e.g. via `cron`) are checked for contiguity. Finally, the *sequence path* of logfile applications (consisting of logfile names plus their respective length) is additionally secured by a `git`-like incremental checksum over the whole path history (so-called "version links"). This should detect split brains even if logfiles are appended / modified *after* a (forceful) switchover has already taken place.

That said, your risk of final data loss is very high if you remove the **BBU** from your hardware RAID controller before all hot data has been flushed to the physical disks. Therefore, never try to "repair" a seemingly dead node before your replication is up again somewhere else! Only unplug the network cables when advised, but never try to repair the hardware instantly!

In case of desperate situations where none of the previous instructions have succeeded, your last chance is rebuilding all your resources from intact disks as follows:

1. Do `rmmod mars` on all your cluster nodes and/or reboot them. Note: if you are less desperate, chances are high that the following will also work when the kernel module remains active and everywhere a `marsadm down` is given instead, but for an *ultimate* instruction you should eliminate *potential* kernel problems by `rmmod` / `reboot`, at least if you can afford the downtime on concurrently operating resources.

2. For safety, physically remove the storage network cables on *all* your cluster nodes. Note: the same disclaimer holds. MARS really does its best, even when `delete-resource` is given while the network is fully active and multiple split-brain primaries are actively using their local device in parallel (approved by some testcases from the automatic test suite, but note that it is impossible to catch all possible failure scenarios). Don't challenge your fate if you are desperate! Don't *rely* on this! Nothing is absolutely fail-safe!

3. **Manually** check which surviving disk is usable, and which is the "best" one for your purpose.

4. Do `modprobe mars` *only* on that node. If that fails, `rmmod` and/or reboot again, and start over with a completely fresh `/mars/` partition (`mkfs.ext4 /mars/` or similar) *everywhere* on *all* cluster nodes, and continue with step 7.

5. If your old `/mars/` works, and you did not already (forcefully) switch your designated primary to the final destination, do it now (see description in section 3.2.2). Wait until any old logfile data has been replayed.

6. Say `marsadm delete-resource mydata --force`. This will cleanup all internal symlink tree information for the resource, but will leave your disk data intact.

7. Locally build up the new resource(s) as usual, out of the underlying disks.
8. Check whether the new resource(s) work in standalone mode.
9. When necessary, repeat these steps with other resources.

Now you can choose how the rebuild your cluster. If you rebuilt `/mars/` anywhere, you *must* rebuild it on *all* new cluster nodes and start over with a fresh `join-cluster` on each of them, from scratch. It is not possible to mix the old cluster with the new one.

10. Finally, do all the necessary `join-resource`s on the respective cluster nodes, according to your new redundancy scenario after the failures (e.g. after activating spare nodes, etc). If you have $k > 2$ replicas, start `join-resource` on the worst / most damaged version first, and start the next preferably only after the previous sync has completed successfully. This way, you will be permanently retaining some (old and outdated, but hopefully potentially usable) replicas while a sync is running. Don't start too many syncs in parallel.

Never use `delete-resource` twice on the same resource name, after you have already a working standalone primary[1]. You might accidentally destroy your again-working copy! You *can* issue `delete-resource` multiple times on different nodes, e.g. when the network has problems, but doing so *after* re-establishment of the initial primary bears some risk. Therefore, the safest way is first deleting the resources everywhere, and then starting over afresh.

Before re-connecting any network cable on any non-primary (new secondaries), ensure that all `/dev/mars/mydata` devices are no longer in use (e.g. from an old primary role before the incident happened), and that each local disk is detached. Only after that, you should be able to safely re-connect the network. The `delete-resource` given at the new primary should propagate now to each of your secondaries, and your local disk should be usable for a re-`join-resource`.

When you did not rebuild your cluster from scratch with fresh `/mars/` filesystems, and one of the old cluster nodes is supposed to be removed permanently, use `leave-resource` (optionally with `--host=` and/or `--force`) and finally `leave-cluster`. After `leave-cluster`, you *must* re-create the `/mars` filesystem.

[1] Of course, when you don't have created the *same* resource anew, you may repeat `delete-resource` on other cluster nodes in order to get rid of local files / symlinks which had not been propagated to other nodes before.

G. Experts only: Special Trick Switching and Rebuild

The following is a further alternative for **experts** who really know what they are doing. The method is very simple and therefore well-suited for coping with mass failures, e.g. **power blackout of whole datacenters**.

In case a primary datacenter fails as a whole for whatever reason and you have a backup datacenter, do the following steps in the backup datacenter:

1. Fencing step: by means of firewalling, **ensure** that the (virtually) damaged datacenter nodes **cannot** be reached over the network. For example, you may place REJECT rules into all of your local iptables firewalls at the backup datacenter. Alternatively / additionally, you may block the routes at the appropriate central router(s) in your network.

2. Run the sequence `marsadm disconnect all; marsadm primary --force all` on all nodes in the backup datacenter.

3. Restart your services in the backup datacenter (as far as necessary). Depending on your network setup, further steps like switching BGP routes etc may be necessary.

4. Check that *all* your services are *really* up and running, before you try to repair anything! Failing to do so may result in data loss when you execute the following restore method for *experts*.

Now your backup datacenter should continue servicing your clients. The final reconstruction of the originally primary datacenter works as follows:

1. At the damaged primary datacenter, ensure that nowhere the MARS kernel module is running. In case of a power blackout, you shouldn't have executed an automatic `modprobe mars` anywhere during reboot, so you should be already done when all your nodes are up again. In case some nodes had no reboot, execute `rmmod mars` everywhere. If `rmmod` refuses to run, you may need to umount the `/dev/mars/mydata` device first. When nothing else helps, you may just mass reboot your hanging nodes.

2. At the failed side, do `rm -rf /mars/resource-$mydata/` for all those resources which had been primary before the blackout. Do this *only* for those cases, otherwise you will need unnecessary `leave-resource`s or `invalidate`s later (e.g. when half of your nodes were already running at the surving side). In order to avoid unnecessary traffic, please do this only as far as really necessary. Don't remove any other directories. In particular, `/mars/ips/` *must* remain intact. In case you accidentally deleted them, or you had to re-create `/mars/` from scratch, try `rsync` with the correct options.

   Caution! before doing this, check that the corresponding directory exists at the backup datacenter, and that it is *really* healthy!

3. Un-Fencing: restore your network firewall / routes and check that they work (`ping` etc).

4. Do `modprobe mars` everywhere. All missing directories and their missing symlinks should be automatically fetched from the backup datacenter.

5. Run `marsadm join-resource $res`, but only at those places where the directory was removed previously, while using the same disk devices as before. This will minimize actual traffic thanks to the fast full sync algorithm.

It is **crucial** that the fencing step **must** be executed *before* any `primary --force`! This way, no split brain will be *visible* at the backup datacenter side, because there is simply no chance for transferring different versions over the network. It is also crucial to remove any (potentially diverging) resource directories *before* the `modprobe`! This way, the backup datacenter never runs into split brain. This saves you a lot of detail work for split brain resolution when you have to restore bulks of nodes in a short time.

In case the repair of a full datacenter should take so extremely long that some `/mars/` partitions are about to run out of space at the surviving side, you may use the `leave-resource --host=failed-node` trick described earlier, followed by `cron`. 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 6.4.

H. Creating Backups via Pseudo Snapshots

When all your secondaries are all homogenously located in a standby datacenter, they will be almost idle all the time. This is a waste of computing resources.

Since MARS is no substitute for a full-fledged backup system, and since backups may put high system load onto your active side, you may want to utilize your passive hardware resources in a better way.

MARS supports this thanks to its ability to switch the `pause-replay` *independently* from `pause-fetch`.

The basic idea is simple: just use `pause-replay` at your secondary site, but leave the replication of transaction logfiles intact by deliberately *not* saying `pause-fetch`. This way, your secondary replica (block device) will stay frozen for a limited time, without loosing your redundancy: since the transaction logs will continue to replicate in the meantime, you can start `resume-replay` at any time, in particular when a primary-side incident should happen unexpectedly. The former secondary will just catch up by replaying the outstanding parts of the transaction logs in order to become recent.

However, some *details* have to be obeyed. In particular, the current version of MARS needs an additional `detach` operation, in order to release exclusive access to the underlying disk `/dev/lv/$res`. Future versions of MARS are planned to support this more directly, without need for an intermediate `detach` operation.

Beware: `mount -o ro /dev/vg/$res` can lead to **unnoticed write operations** if you are not careful! Some journalling filesystems like `xfs` or `ext4` may replay their journals onto the disk, leading to *binary* differences and thus **destroying your consistency** later when you re-enable `resume-replay`!

Therefore, you may use small LVM snapshots (only in such cases). Typically, `xfs` journal replay will require only a few megabytes. Therefore you typically don't need much temporary space for this. Here is a more detailed description of steps:

1. `marsadm pause-replay $res`
2. `marsadm detach $res`
3. `lvcreate --size 100m --snapshot --name ro-$res /dev/vg/$res`
4. `mount -o ro /dev/vg/ro-$res /mnt/tmp`
5. Now draw your backup from `/mnt/tmp/`
6. `umount /mnt/tmp`
7. `lvremove -f /dev/vg/ro-$res`
8. `marsadm up $res`

Hint: during the backup, the transaction logs will accumulate on `/mars/`. In order to avoid overflow of `/mars/` (c.f. section 3.6), don't unnecessarily prolong the backup duration.

I. Command Documentation for Userspace Tools

I.1. marsadm --help

```
Thorough documentation is in mars-user-manual.pdf. Please use the PDF manual
as authoritative reference! Here is only a short summary of the most
important sub-commands / options:

marsadm [<global_options>] <command> [<resource_names> | all | <args> ]
marsadm [<global_options>] view[-<macroname>] [<resource_names> | all ]

<global_option> =
  --force
    Skip safety checks.
    Use this only when you really know what you are doing!
    Warning! This is dangerous! First try --dry-run.
    Not combinable with 'all'.
  --ignore-sync
    Allow primary handover even when some sync is running somewhere.
    This is less rude than --force because it checks for all else
    preconditions.
  --ignore-deleted-peers=<number>
    0 = off
    1 = only ignore deleted peers at reports (default)
    2 = Only for EXPERTS. This may be dangerous.
    Ignorance about the existence of a peer may be very harmful
    in a Distributed System.
  --dry-run
    Don't modify the symlink tree, but tell what would be done.
    Use this before starting potentially harmful actions such as
    'delete-resource'.
  --verbose
    Increase speakyness of some commands.
  --parallel
    Only resonable when combined with "all".
    For each resource, fork() a sub-process running independently
    from other resources. May seepd up handover a lot.
    However, several cluster managers are known to have problems
    with a high parallelism degree (up to deadlocks).
    Only use this after thorough testing in combination with your
    whole operation stack!
    Turns off --singlestep.
  --parallel=<number>
    Like --parallel, but limit the parallelism degree to the given
    number of parallel processes.
    Turns off --singlestep.
  --singlestep
    Debugging aid for  multi-phase commands.
    Interactively step through the various phases of commands.
    Turns off --parallel.
```

```
--error-injection-phase=<number>
  Only for testing. NEVER use in production.
--delete-method=<code>
  EXPERIMENTAL! Only for testing! This option will disappear again!
  <code> == 0: Use new deletion method
  <code> == 1: Use old deletion method
  default is 1 for compatibility.
--logger=/path/to/usr/bin/logger
  Use an alternative syslog messenger.
  When empty, disable syslogging.
--max-deletions=<number>
  When your network or your firewall rules are defective over a
  longer time, too many deletion links may accumulate at
  /mars/todo-global/delete-* and sibling locations.
  This limit is preventing overflow of the filesystem as well
  as overloading the worker threads.
--thresh-logfiles=<number>
--thresh-logsize=<number>
  Prevention of too many small logfiles when secondaries are not
  catching up. When more than thresh-logfiles are already present,
  the next one is only created when the last one has at least
  size thresh-logsize (in units of GB).
--timeout=<seconds>
  Current default: 600
  Abort safety checks and waiting loops after timeout with an error.
  When giving 'all' as resource agument, this works for each
  resource independently.
  The special value -1 means "infinite".
--window=<seconds>
  Current default: 60
  Treat other cluster nodes as healthy when some communcation has
  occured during the given time window.
--stuck-seconds=<seconds>
  Current default: 3600
  Some warnings, like stucking fetch or replay, will appear in
  "marsadm view" after this silence period.
--keep-backup-hours=<hours>
--keep-backups=<hours>
  link-purge-all and cron will delete old backup files and old
  symlinks after this number of hours.
  Current default: 168
--threshold=<bytes>
  Some macros like 'fetch-threshold-reached' use this for determining
  their sloppyness.
--systemd-enable=<0|1>
  Enable / disable any systemd actions.
  On by default.
--host=<hostname>
  Act as if the command was running on cluster node <hostname>.
  Warning! This is dangerous! First try --dry-run
--backup-dir=</absolute_path>
  Only for experts.
  Used by several special commands like merge-cluster, split-cluster
  etc for creating backups of important data.
--ip-<peer>=<ip>
  Override the IP address of <peer> from the symlink tree, or as determined
  from old IP backups, or as determined from the list of network interfaces.
  Usually you will need this only at 'create-cluster' or
```

```
    'join-cluster' / 'merge-cluster' / 'split-cluster' for resolving
    ambiguities, or for telling the IP address of yet unknown peers.
    It is also useful at 'lowlevel-set-host-ip' for updating an
    already existing IP address.
    Hint: this option may be given multiple times for different <peer>
    parts.
  --ip=<ip>
    Equivalent to --peer-$host=<ip>
    where $host is usually the same as $(hostname), but you may
    use --host=<hostname> as an _earlier_ argument for overriding
    the default <hostname>.
  --ssh-port=<port_nr>
    Override the default ssh port (22) for ssh and rsync.
    Useful for running {join,merge}-cluster on non-standard ssh ports.
  --no-ssh
    Equivalent to --ssh-port=0
    Disable ssh and rsync completely.
    Dead peers / interrupted networks / firewalling may lead to (temporary)
    hangs of ssh probes, which are used by default for backwards compatibility.
    Hint: ssh_config options like ConnectTimeout may also help.
    Use this to disable any probes, and no time loss.
  --ssh-opts="<ssh_commandline_options>"
    Override the default ssh commandline options. Also used for rsync.
  --macro=<text>
    Handy for testing short macro evaluations at the command line.

<hostname> =
  [a-z][-a-z0-9]*
  with the exception of reserved names: none|all|any|local|localhost

<command> =
  activate-guest
    usage: activate-guest <resource_name>
    Conditional update-cluster, so that <resource_name> will be locally
    known at the local machine, and mark the resource as a guest.
    Useful inbetween create-resource and join-resource.
    A guest will receive any symlink updates much more frequently.
    Prefer this over update-cluster when interested in a resource.

  attach
    usage: attach <resource_name>
    Attaches the local disk (backing block device) to the resource.
    The disk must have been previously configured at
    {create,join}-resource.
    When designated as a primary, /dev/mars/$res will also appear.
    This does not change the state of {fetch,replay}.
    For a complete local startup of the resource, use 'marsadm up'.

  cat
    usage: cat <path>
    Print internal debug output in human readable form.
    Numerical timestamps and numerical error codes are replaced
    by more readable means.
    Example: marsadm cat /mars/5.total.status

  connect
    usage: connect <resource_name>
    See resume-fetch-local.
```

```
connect-global
  usage: connect-global <resource_name>
  Like resume-fetch-local, but affects all resource members
  in the cluster (remotely).

connect-local
  usage: connect-local <resource_name>
  See resume-fetch-local.

create-cluster
  usage: create-cluster (no parameters)
  This must be called exactly once when creating a new cluster.
  Don't call this again! Use join-cluster on the secondary nodes.
  Please read the PDF manual for details.

create-resource
  usage: create-resource <resource_name> </dev/lv/mydata>
  (further syntax variants are described in the PDF manual).
  Create a new resource out of a pre-existing disk (backing
  block device) /dev/lv/mydata (or similar).
  The current node will start in primary role, thus
  /dev/mars/<resource_name> will appear after a short time, initially
  showing the same contents as the underlying disk /dev/lv/mydata.
  It is good practice to name the resource <resource_name> and the
  disk name identical.

cron
  usage: cron (no parameters)
  Do all necessary regular housekeeping tasks.
  This must be regularly called by a cron job or similar, in order
  to prevent overflow of the /mars/ directory.
  For details and best practices, please refer to the PDF manual.

deactivate-guest
  usage: deactivate-guest <resource_name>
  Precondition: the resource must not have local storage assigned.
  Useful for cleaning up a pure guest relastionship.

delete-resource
  usage: delete-resource <resource_name>
  CAUTION! This is dangerous when the network is somehow
  interrupted, or when damaged nodes are later re-surrected
  in any way.

  Precondition: the resource must no longer have any members
  (see leave-resource).
  This is only needed when you _insist_ on re-using a damaged
  resource for re-creating a new one with exactly the same
  old <resource_name>.
  HINT: best practice is to not use this, but just create a _new_
  resource with a new <resource_name> out of your local disks.
  Please read the PDF manual on potential consequences.

detach
  usage: detach <resource_name>
  Detaches the local disk (backing block device) from the
  MARS resource.
```

```
  Caution! you may read data from the local disk afterwards,
  but ensure that no data is written to it!
  Otherwise, you are likely to produce harmful inconsistencies.
  When running in primary role, /dev/mars/$res will also disappear.
  This does not change the state of {fetch,replay}.
  For a complete local shutdown of the resource, use 'marsadm down'.

disconnect
  usage: disconnect <resource_name>
  See pause-fetch-local.

disconnect-global
  usage: disconnect-global <resource_name>
  Like pause-fetch-local, but affects all resource members
  in the cluster (remotely).

disconnect-local
  usage: disconnect-local <resource_name>
  See pause-fetch-local.

down
  usage: down <resource_name>
  Shortcut for detach + pause-sync + pause-fetch + pause-replay.

err-purge-all
  usage: err-purge-all <resource_name>
  Remove any err message from the given resources.

get-emergency-limit
  usage: get-emergency-limit <resource_name>
  Counterpart of set-emergency-limit (per-resource emergency limit)

get-global-sync-limit-value
  usage: get-sync-limit-value (no parameters)
  For retrieval of the value set by set-global-sync-limit-value.

get-systemd-unit
  usage: get-systemd-unit <resource_name>
  Show the system units (for start and stop), or empty when unset.

get-systemd-want
  usage: get-systemd-want <resource_name>
  Show the current hostname where the complete systemd unit stack
  between start- and stop-unit should appear.
  Reports empty when unset, or "(none)" when stopped.

invalidate
  usage: invalidate <resource_name>
  Only useful on a secondary node.
  Forces MARS to consider the local replica disk as being
  inconsistent, and therefore starting a fast full-sync from
  the currently designated primary node (which must exist;
  therefore avoid the 'secondary' command).
  This is usually needed for resolving emergency mode.
  When having k=2 replicas, this can be also used for
  quick-and-simple split-brain resolution.
  In other cases, or when the split-brain is not resolved by
  this command, please use the 'leave-resource' / 'join-resource'
```

```
    method as described in the PDF manual (in the right order as
    described there).

  join-cluster
    usage: join-cluster <hostname_of_primary>
    Establishes a new cluster membership.
    This must be called once on any new cluster member.
    This is a prerequisite for join-resource.

  join-resource
    usage: join-resource <resource_name> </dev/lv/mydata>
    (further syntax variants are described in the PDF manual).
    The resource <resource_name> must have been already created on
    another cluster node, and the network must be healthy.
    The contents of the local replica disk /dev/lv/mydata will be
    overwritten by the initial fast full sync from the currently
    designated primary node.
    After the initial full sync has finished, the current host will
    act in secondary role.
    For details on size constraints etc, refer to the PDF manual.

  leave-cluster
    usage: leave-cluster (no parameters)
    This can be used for final deconstruction of a cluster member.
    Prior to this, all resources must have been left
    via leave-resource.
    Notice: this will never destroy the cluster UID on the /mars/
    filesystem.
    Please read the PDF manual for details.

  leave-resource
    usage: leave-resource <resource_name>
    Precondition: the local host must be in secondary role.
    Stop being a member of the resource, and thus stop all
    replication activities. The status of the underlying disk
    will remain in its current state (whatever it is).

  link-purge-all
    usage: link-purge-all <resource_name>
    Remove any .deleted links.

  log-purge-all
    usage: log-purge-all <resource_name>
    This is potentially dangerous.
    Use this only if you are really desperate in trying to resolve a
    split brain. Use this only after reading the PDF manual!

  lowlevel-delete-host
    usage: lowlevel-delete-host <hostname>
    Delete cluster member.

  lowlevel-ls-host-ips
    usage: lowlevel-ls-host-ips
    List cluster member names and IP addresses.

  lowlevel-set-host-ip
    usage: lowlevel-set-host-ip <hostname> [<new_ip>]
    Set IP address <new_ip> for host.
```

```
  When <new_ip> is not given, try to determine the old address
  from the symlink tree, or from old backups.
  Often, you want to set a new IP address in place of an old one.
  Hint: you may also use the --ip-<hostname>=<new_ip> option.

merge-cluster
  usage: merge-cluster <hostname_of_other_cluster> [<host_ip>]
  Precondition: the resource names of both clusters must be disjoint.
  Create the union of two clusters, consisting of the
  union of all machines, and the union of all resources.
  The members of each resource are _not_ changed by this.
  This is useful for creating a big "virtual LVM cluster" where
  resources can be almost arbitrarily migrated between machines via
  later join-resource / leave-resource operations.

merge-cluster-check
  usage: merge-cluster-check <hostname_of_other_cluster>
  Check whether the resources of both clusters are disjoint.
  Useful for checking in advance whether merge-cluster would be
  possible.

merge-cluster-list
  usage: merge-cluster-list
  Determine the local list of resources.
  Useful for checking or analysis of merge-cluster disjointness by hand.

pause-fetch
  usage: pause-fetch <resource_name>
  See pause-fetch-local.

pause-fetch-global
  usage: pause-fetch-global <resource_name>
  Like pause-fetch-local, but affects all resource members
  in the cluster (remotely).

pause-fetch-local
  usage: pause-fetch-local <resource_name>
  Stop fetching transaction logfiles from the current
  designated primary.
  This is independent from any {pause,resume}-replay operations.
  Only useful on a secondary node.

pause-replay
  usage: pause-replay <resource_name>
  See pause-replay-local.

pause-replay-global
  usage: pause-replay-global <resource_name>
  Like pause-replay-local, but affects all resource members
  in the cluster (remotely).

pause-replay-local
  usage: pause-replay-local <resource_name>
  Stop replaying transaction logfiles for now.
  This is independent from any {pause,resume}-fetch operations.
  This may be used for freezing the state of your replica for some
  time, if you have enough space on /mars/.
  Only useful on a secondary node.
```

```
pause-sync
  usage: pause-sync <resource_name>
  See pause-sync-local.

pause-sync-global
  usage: pause-sync-global <resource_name>
  Like pause-sync-local, but affects all resource members
  in the cluster (remotely).

pause-sync-local
  usage: pause-sync-local <resource_name>
  Pause the initial data sync at current stage.
  This has only an effect if a sync is actually running (i.e.
  there is something to be actually synced).
  Don't pause too long, because the local replica will remain
  inconsistent during the pause.
  Use this only for limited reduction of system load.
  Only useful on a secondary node.

primary
  usage: primary <resource_name>
  Promote the resource into primary role.
  This is necessary for /dev/mars/$res to appear on the local host.
  Notice: by concept there can be only _one_ designated primary
  in a cluster at the same time.
  The role change is automatically distributed to the other nodes
  in the cluster, provided that the network is healthy.
  The old primary node will _automatically_ go
  into secondary role first. This is different from DRBD!
  With MARS, you don't need an intermediate 'secondary' command
  for switching roles.
  It is usually better to _directly_ switch the primary roles
  between both hosts.
  When --force is not given, a planned handover is started:
  the local host will only become actually primary _after_ the
  old primary is gone, and all old transaction logs have been
  fetched and replayed at the new designated priamry.
  When --force is given, no handover is attempted. A a consequence,
  a split brain situation is likely to emerge.
  Thus, use --force only after an ordinary handover attempt has
  failed, and when you don't care about the split brain.
  For more details, please refer to the PDF manual.

resize
  usage: resize <resource_name>
  Prerequisite: all underlying disks (usually /dev/vg/$res) must
  have been already increased, e.g. at the LVM layer (cf. lvresize).
  Causes MARS to re-examine all sizing constraints on all members of
  the resource, and increase the global logical size of the resource
  accordingly.
  Shrinking is currently not yet implemented.
  When successful, /dev/mars/$res at the primary will be increased
  in size. In addition, all secondaries will start an incremental
  fast full-sync to get the enlarged parts from the primary.

resume-fetch
  usage: resume-fetch <resource_name>
```

```
  See resume-fetch-local.

resume-fetch-global
  usage: resume-fetch-global <resource_name>
  Like resume-fetch-local, but affects all resource members
  in the cluster (remotely).

resume-fetch-local
  usage: resume-fetch-local <resource_name>
  Start fetching transaction logfiles from the current
  designated primary node, if there is one.
  This is independent from any {pause,resume}-replay operations.
  Only useful on a secondary node.

resume-replay
  usage: resume-replay <resource_name>
  See resume-replay-local.

resume-replay-global
  usage: resume-replay-global <resource_name>
  Like resume-replay-local, but affects all resource members
  in the cluster (remotely).

resume-replay-local
  usage: resume-replay-local <resource_name>
  Restart replaying transaction logfiles, when there is some
  data left.
  This is independent from any {pause,resume}-fetch operations.
  This should be used for unfreezing the state of your local replica.
  Only useful on a secondary node.

resume-sync
  usage: resume-sync <resource_name>
  See resume-sync-local.

resume-sync-global
  usage: resume-sync-global <resource_name>
  Like resume-sync-local, but affects all resource members
  in the cluster (remotely).

resume-sync-local
  usage: resume-sync-local <resource_name>
  Resume any initial / incremental data sync at the stage where it
  had been interrupted by pause-sync.
  Only useful on a secondary node.

secondary
  usage: secondary <resource_name>
  Promote all cluster members into secondary role, globally.
  In contrast to DRBD, this is not needed as an intermediate step
  for planned handover between an old and a new primary node.
  The only reasonable usage is before the last leave-resource of the
  last cluster member, immediately before leave-cluster is executed
  for final deconstruction of the cluster.
  In all other cases, please prefer 'primary' for direct handover
  between cluster nodes.
  Notice: 'secondary' sets the global designated primary node
  to '(none)' which in turn prevents the execution of 'invalidate'
```

```
    or 'join-resource' or 'resize' anywhere in the cluster.
    Therefore, don't unnecessarily give 'secondary'!

  set-emergency-limit
    usage: set-emergency-limit <resource_name> <value>
    Set a per-resource emergency limit for disk space in /mars.
    See PDF manual for details.

  set-global-disabled-log-digests
    usage: set-global-disabled-log-digests <features>
    Tell the whole cluster which checksumming digests to disable globally
    for the payload in transaction logfiles.
    The effective value can be checked via "marsadm view-disabled-log-digests".
    See "marsadm view-potential-features" and
    "marsadm --help" for a list of digest feature names,
    which must be separated by | symbols.

  set-global-disabled-net-digests
    usage: set-global-disabled-net-digests <features>
    Tell the whole cluster which checksumming digests to disable globally
    for cluster-wide data comparisons, like fast full-sync.
    The effective value can be checked via "marsadm view-disabled-net-digests".
    See "marsadm view-potential-features" and
    "marsadm --help" for a list of digest feature names,
    which must be separated by | symbols.

  set-global-enabled-log-compressions
    usage: set-global-enabled-log-compressions <features>
    Tell the whole cluster which compression features to use globally
    for logfile compression. The effective value can be checked via
    "marsadm view-enabled-log-compressions".
    See "marsadm view-potential-features" and
    "marsadm --help" for a list of compression feature names,
    which must be separated by | symbols.

  set-global-enabled-net-compressions
    usage: set-global-enabled-net-compressions <features>
    Tell the whole cluster which compression features to use globally for
    network transport compression. This is independent from log compression.
    The effective value can be checked via
    "marsadm view-enabled-log-compressions".
    See "marsadm view-potential-features" and
    "marsadm --help" for a list of compression feature names,
    which must be separated by | symbols.

  set-global-sync-limit-value
    usage: set-sync-limit-value <new_value>
    Set the maximum number of resources which should by syncing
    concurrently.

  set-systemd-unit
    usage: set-systemd-unit <resource_name> <start_unit_name> [<stop_unit_name>]
    This activates the systemd template engine of marsadm.
    Please read mars-user-manual.pdf on this.
    When <stop_unit_name> is omitted, it will be treated equal to
    <start_unit_name>.
    You may also use special keywords like DEFAULT, please read the manuals.
```

```
  set-systemd-want
    usage: set-systemd-want <resource_name> <host_name>
    Override the current location where the complete systemd unit stack
    should be started.
    Useful for a _temporary_ stop of the systemd unit stack by supplying
    the special hostname "(none)".
    For a _permanent_ stop, use "marsadm set-systemd-unit <resource>"
    instead.

  split-cluster
    usage: split-cluster (no parameters)
    NOT OFFICIALLY SUPPORTED - ONLY FOR EXPERTS.
    RTFS = Read The Fucking Sourcecode.
    Use this only if you know what you are doing.

  systemd-trigger
    usage: systemd-trigger [<resource>]

  up
    usage: up <resource_name>
    Shortcut for attach + resume-sync + resume-fetch + resume-replay.

  update-cluster
    usage: update-cluster [<resource_name>]
    Fetch all the links from all joined cluster hosts.
    Use this between create-resource and join-resource.
    NOTICE: this is extremely useful for avoiding races when scripting
    in a cluster.

  wait-cluster
    usage: wait-resource [<resource_name>]
    Waits until a ping-pong communication has succeeded in the
    whole cluster (or only the members of <resource_name>).
    NOTICE: this is extremely useful for avoiding races when scripting
    in a cluster.

  wait-connect
    usage: wait-connect [<resource_name>]
    See wait-cluster.

  wait-resource
    usage: wait-resource <resource_name>
                         [[attach|fetch|replay|sync][-on|-off]]
    Wait until the given condition is met on the resource, locally.

  wait-umount
    usage: wait-umount <resource_name>
    Wait until /dev/mars/<resource_name> has disappeared in the
    cluster (even remotely).
    Useful on both primary and secondary nodes.

<resource_names> = comma-separated list of resource names or "all" for all resources


<macroname> = <complex_macroname> | <primitive_macroname>

<complex_macroname> =
  1and1
```

```
  comminfo
  commstate
  cstate
  default
  default-footer
  default-global
  default-header
  default-resource
  device-info
  device-stats
  diskstate
  diskstate-1and1
  dstate
  fetch-line
  fetch-line-1and1
  flags
  flags-1and1
  outdated-flags
  outdated-flags-1and1
  primarynode
  primarynode-1and1
  replay-line
  replay-line-1and1
  replinfo
  replinfo-1and1
  replstate
  replstate-1and1
  resource-errors
  resource-errors-1and1
  role
  role-1and1
  state
  status
  sync-line
  sync-line-1and1
  syncinfo
  syncinfo-1and1
  todo-role


<primitive_macroname> =
  count-{cluster,resource,guest}-members
    deprecated
  count-{cluster,resource,guest}-peers
  count-{my,all}-{resources,members,guests}
  deletable-size
  device-{opened,nrflying,error,completion-{stamp,age}}
  device-{ops-rate,amount-rate,rate}
  disabled-{log|net}-digests
  disk-error
  enabled-{log|net}-compressions
  errno-text
    Convert errno numbers (positive or negative) into human readable text.
  get-log-status
  get-resource-{fat,err,wrn}{,-count}
  get-{disk,device}
  is-{alive}
  is-{member,guest}
```

```
  is-{split-brain,consistent,emergency,orphan}
  occupied-size
  present-{disk,device}
    (deprecated, use *-present instead)
  replay-basenr
  replay-code
    When negative, this indidates that a replay/recovery error has occurred.
  resource-possible-size
  summary-vector
  systemd-unit
  tree
  used-{log,net}-{digest,compression}
  uuid
  wait-{is,todo}-{attach,sync,fetch,replay,primary,secondary}-{on,off}
  writeback-rest
  {alive,fetch,replay,work}-{timestamp,age,lag}
  {all,the}-{pretty-,}{global-,}{{err,wrn,inf}-,}msg
  {cluster,resource,guest}-peers
  {cluster,resource}-members
    deprecated
  {disk,device}-present
  {disk,resource,device}-size
  {fetch,replay,work}-{lognr,logcount}
  {get,actual}-primary
  {implemented,usable}-{digests,compressions}
  {is,todo,nr}-{attach,sync,fetch,replay,primary,secondary}
  {my,all}-{resources,members,guests}
  {potential,implemented,usable}-features
  {rest,total}-space
  {sync,fetch,replay,work,syncpos}-{size,pos}
  {sync,fetch,replay,work}-{rest,{almost-,threshold-,}reached,percent,permille,vector}
  {sync,fetch,replay}-{ops-rate,amount-rate,rate,remain}
  {time,real-time}
  {tree,features}-version


<features> =
  CHKSUM_CRC32 |
  CHKSUM_CRC32C |
  CHKSUM_MD5 |
  CHKSUM_MD5_OLD |
  CHKSUM_SHA1 |
  COMPRESS_LZ4 |
  COMPRESS_LZO |
  COMPRESS_ZLIB
```

J. 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.
```