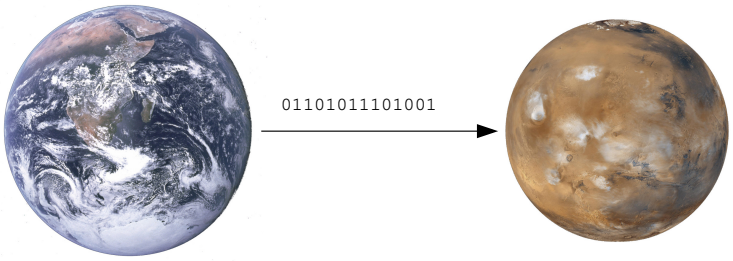
MARS Architecture Guide

Multiversion Asynchronous Replicated Storage



Thomas Schöbel-Theuer (tst@1und1.de)

Version 0.1a-77

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.

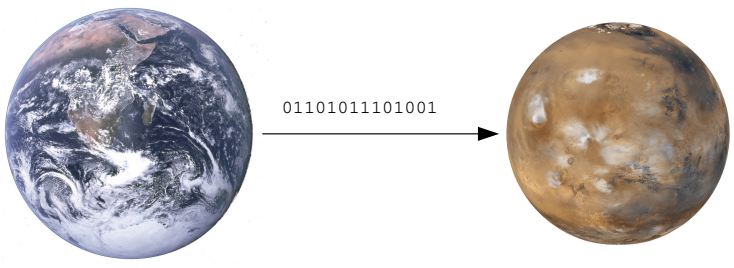
MARS runs as a Linux kernel module. The sysadmin interface is similar to DRBD, but its internal engine is completely different from DRBD: it works with transaction logging, similar to some database systems.

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 In case of network bottlenecks / problems / failures, the secondaries may become outdated (reflect an elder state), but will not become inconsistent.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 and discusses how to select the right storage architecture for typical use cases in big enterprises. Besides general storage architectures, pitfalls of geo-redundancy and long-distance replication are highlighted.

In addition to technical discussion, **cost and risks** are treated as well, addressing some **management needs** up to CTO level.

In contrast to several other publications, it is *not* an enumeration of sheer endless possibilites and components on the market. It provides **guidance** about the **structures and ideas** *behind* storage architectures and their connection to application processing. Particular attention is on **avoidance of pitfalls**.

It provides both *technical* and *management* guidance about selection of architectures as well as their implementation *classes*, and also about selection of suitable component *classes*.

Finally, it helps checking for use cases where MARS will be a good solution, and where other solutions will be better suited. It also addresses some unexpected problems when inappropriate types of cluster managers are selected for long-distance replication.

Scope

The following topics are covered within this document:

- Management Summary
- Architectures of Cloud Storage, and

  - their application area
  - their reliability / risks / pitfalls
  - their cost
  - scalability and performance of architectures
  - recommendations for managers and architects
- Selection of components
  - MARS vs DRBD
- Architecture and pitfalls of Cluster Managers

Audience

This document is mainly written for system architects. Technical decision makers / managers with technical background, up to CTO level, should also benefit from **risk reduction** and **cost saving**, when making clever investment and consolidation decisions.

Researchers in the field of storage systems are also addressed in the section about **reliability** and the appendix, by providing mathematical models of reliability.

How to use this document

Managers should start with chapter Management Summary. Then read the short chapter Important Concepts. For details, just follow the internal links within this document. In any case, the last chapter Recommendations for Managers is highly recommended.

Manager Hint 0.1:

These boxes are something you definitely should read as a manager. It explains **important key items** in a nutshell.

All others should read chapter 1 and 2 sequentially, and proceed to the other chapters when interested.

When MARS is already in use (or planned to be used), reading all of the chapters may pay off for **avoidance of pitfalls**.

Example 0.1:

Examples are marked with boxes like this. They can be skipped if you don't have much time. Examples will however help for understanding of complex material.

Details 0.1:

Detail explanations are marked like this. They are recommended for system architects for more elaborate methodology, and for deeper understanding of fundamentals.

Hint for research 0.1:

This document is no scientific work in a strong sense. However, it is based on scientific background. In a few places, hints like this could be fruitful for spawning research activity.

Related documents

- `mars-user-manual.pdf`: for sysadmins who want to install and run MARS.

- `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. Management Summary

This guide is about **investments and long-term follow-up cost** in the range of **millions** of € or $. It tries to guide you through the jungle of storage solutions and their features, by focussing at **fundamental principles** and high-level structures, called **architecture**.

For **HA enterprise-critical data** in the range of **petabytes**, different storage architectures are leading to very different properties in the **cost and risk dimensions**.

Manager Hint 1.1: **Provably best HA / Cloud Storage architecture**

By intuitive explanations as well as mathematical arguments, this guide shows that

- Permanent **minimization of the distances** between storage and the compute nodes will both **increase reliablity and reduce cost at the same time**.
- When applicable for a certain use case, the best architectural model is **sharding** on top of **local storage**. It can easily save a cost factor of about 2, while increasing **architectural reliability** at the same time.
- When the so-called **FlexibleSharding** variant of the sharding model is combined with a novel load balancing method called **Football**, it can deliver a very similar level of **flexibility** than network-centric BigCluster architectures are promising.
- By both intuitive and mathematical explanations, and contrary to some contemporary belief, you will learn ***why* BigCluster architectures are generally worse** in practically any dimension. There exist certain use cases where BigCluster cannot be explicitly recommended.
- When built and dimensioned properly, **cross-datacenter replication** and/or **geo-redundancy** will *not* double TCO = Total Cost of Ownership, but can cost roughly about the same as local redundancy in the same datacenter. The key is a certain class of **wide-area distribution of resources** *in place of* local replication.
- When cross-datacenter replication and/or geo-redundancy is required, the so-called **ability for butterfly** leads to further HA improvements.
- Object-based **Cloud Storage** can also be built on top of a sharding model.
- You will learn **why OpenSource component-based storage systems** are much cheaper than commercial storage appliances (up to *factors*), at least when you need a few petabytes of storage. Alone by relinquishing Vendor-Lock-In and going to RAID-based Linux storage, invest will typically decrease by factors between 3 and 10. By going to a `LocalSharding` or `FlexibleSharding` model, where possible, *another* decrease factor of about 2 is typically possible.

In addition, this guide explains the ideas behind the OpenSource components Football on top of MARS. It can be used for replication over short to very long distances, as well as for load balancing via background data migration while your services are running.

2. Important Concepts

This chapter is *very short*. Recommended reading for *everyone* is *each* of the definitions in *each* section, even if you think that you already know what each concept means.

In case you **notice a difference** between your former opinion about a concept and what you are reading here, then **don't skip the rest** of the corresponding section.

 Skipping anything in this chapter exposes you to serious risks:

- **Misunderstanding** of following important parts. This may become **expensive**. This guide is about investments and follow-up cost in the range of **millions** of €.
- **Second-order ignorance**: you probably don't know what you don't know. This is not only risky in **enterprise-critical** areas. You can also risk your **career**.

2.1. What is Architecture

From https://en.wikipedia.org/wiki/Software_architecture:

> Software architecture refers to the **high level structures** of a software system and the **discipline** of creating such structures and systems.

Throughout this document, the term "**architecture**" (without preceding "software") is strictly separated from "**implementation**" (without preceding "software"). Any of "architecture" or "implementation" can relate to both hard- and software.

 A certain architecture may have multiple implementations. An implementation is based on a *set* of **technologies**[1].

 Unfortunately, certain technologies are not suitable for certain architectures. There may be **restrictions**.

 Because of **hidden restrictions** which may show up later, you should not start with implementations or technologies. Always start top-down with architectural considerations, while trying to identify potential restrictions *as early as possible*.

Manager Hint 2.1:

 The biggest **potential for good solutions** is at architectural level. Exchanging a single component or a technology is typically much easier than changing a whole architecture, once it has been implemented. Often, changing an architecture is close to impossible.

 Starting with a particular implementation and/or with a particular technology in mind, and not sufficiently reasoning abouts its fundamental architecture, and/or **not seriously con-**

[1] Architectures are serving as aids for **classification of solutions**. An implementation is a solution which has *materialized* (in contrast to solutions which exist only on paper). Complex solutions / implementations are typically matching only one architecture. Thus the relationship between architectures and solutions / implementations is typically $1 : n$, while the relationship between solutions / implementations and technologies is $n : m$ in general. In case of a very simple solution, it may *exceptionally* match multiple architectures, but this is not typical for classification schemes.

sidering alternative architectures, is a major source of **costly ill-designs**. An example may be found in section Example Failures of Scalability.

 Confusion of "architecture" with "implementation" and/or "technology" is another major source of ill-designs, which then often cause major product flaws and/or operational problems. Be sure to understand the differences.

 Recommended best practice is to (1) look at the **problem space**, then (2) consider a *set* of **architectural solution classes**, and (3) look at each of the **mappings** between problem space and solution space. The *complexity* of such a mapping is a first hint.

Details 2.1:

 In somewhat more detail: start with **architectural requirements** for a particular **application area** (typically covering *multiple* use cases), then look at **multiple solution architectures**, and finally go down to a ***set*** of potential implementations, but only *after* the former has been understood. Selection of components and technologies should be the *last* step during the first iteration of this method. Then do a **quality check** at *concept* level. Often, this review will disguise some problems / limitations etc, which should be treated by further iterations, restarting top-down again.

 During this iterative concept work, you should **validate** your solution(s) several times, e.g. for **compatibility** (no conflicts caused by restrictions, etc).

 Starting at the bottom with a particular single solution in mind, and/or presuming a certain technology, is almost a *guarantee* for a non-optimum solution, or even a failed project, or even a disaster at company level when **enterprise-critical mass data** is involved. Always consider a *set of* candidate architectures, and for each of them, a *set of* solutions / technologies.

 Nevertheless, don't think in waterfall models. Always work **iteratively** and **evolutionary** by *re-considering architecture* whenever you find problems / contradictions induced by restrictions, similar to the **spiral model**[a].

 Be cautious when transferring *software* development methods to storage architectures, where operations involves masses of hardware. You need to find a balance between extreme waterfall-like and agile[b] methods.

[a] See https://en.wikipedia.org/wiki/Spiral_model.

[b] Purely agile methods are less suited for quality ensurance of storage architectures, because they are tempting people to start with simple approaches before the problem domain has been fully understood, increasing the **risk of architectural ill-designs**. Starting an implementation *too early* on basis of an ill-design can easily **lead into a dead end**. Agile methods are typically encouraging **early deliverables**., which can be counter-productive. Example: it is clearly a bad idea to plan for an early deliverable for some petabytes of storage. Thus architects and managers are tempted to *start small*, e.g. a BigCluster architecture with only 3 storage servers. This type of "early deliverable" cannot detect any **scalability problems** early enough, see section Example Failures of Scalability. So you are in a **dilemma**, whether you like it or not. Although you probably dislike it, the planning phase of big storage systems is unfortunately more like a waterfall process, by its very nature. Thus *workarounds* for the shortcomings of a pure waterfall model are needed. German readers may also check the V-model XT, as described in https://de.wikipedia.org/wiki/V-Modell_(Entwicklungsstandard). Unfortunately, the newer XT variant of the V-model is missing in the corresponding English Wikipedia article (retrieved autumn 2019), misleading readers with unfortunate opinions like the V-model being too similar to a waterfall model. Notice that the newer XT variant of the V-model, as well as some other variants (e.g. lecture notes from Professor Jochen Ludewig / University of Stuttgart), have adopted many ideas from the agile community, such as rework in loops and cycles, and thus should not be classifed as "linear waterfall" models. In particular,

early quality ensurance of concepts and architectures and **rework of architecture as early as possible** is something you definitely should borrow from the V-model and its modern variants, even if you dislike V-models otherwise.

 Serious bugs in an *architectural* ill-design (examples see section Example Failures of Scalability) are typically very hard by causing serious limitation and/or impact, and cannot be fixed by the best implementation, or by the best technology of the world. Be sure to understand the fundamental difference between architecture and its (multiple / alternative) implementations, as well as multiple technologies, and their respective restrictions, as well as their **reach**.

2.2. What is Backup

A Backup is a **copy of your data** at a **different location**. There are two distinct operations associated with backup:

1. **Creation** of backup. This creates a **copy**, or a new version of a copy. It involves some network traffic over various distances, e.g in simplest case over a USB cable, or from the application datacenter to a backup datacenter. Typically, this is done at **regular time intervals**, e.g. daily.

2. **Restore** from backup. This does the *opposite* of backup creation. It also involves network traffic, but in **reverse direction**. The **roles** of application datacenter and backup datacenter **do not change**. Restore is typically **triggered manually**, and only after some incident which led to **data loss**.

 It would be a *bad idea* to restore a backup although there is no data loss. This would likely overwrite your newest application data with an elder version, likely leading to *data loss*. Therefore, restore is **potentially dangerous operation**!

Manager Hint 2.2: **Summary: structural properties of backup**

Backup is **asymmetrical**. It involves two non-exchangable roles / locations, application location vs backup location.

 Confusion of these roles, or triggering an unnecessary restore is a **risk** for **data integrity**.

 Conversely, having no reasonable backup at all is an even higher risk. Backup is a **best practice**.

2.3. What is Replication

Intuitively, data backup and data replication are two different solution classes, addressing different problems.

However, there exist descriptions where both solution classes are overlapping, as well as their corresponding problem classes. For example, backup as explained in https://en.wikipedia.org/wiki/Backup could be seen as also encompassing some types of storage replications explained in https://en.wikipedia.org/wiki/Replication_(computing).

For this guide, we want a clearer discrimination, for better orientation in the solution jungle. As a rough comparison of *typical* implementations, see the following *typical* differences:

	Backup	Replication
Timely pattern	intervals	continuously
Fast handover (planned)	no, or cumbersome	yes
Fast failover (unplanned)	no, or cumbersome	yes
Protect for physical failures	yes	yes
Protect for logical data corruption	yes	typically no
Disaster Recovery Time (**MTTR**)	typically **very slow**	**fast**

There are some solutions implementing a *mixture*, by different combinations of some of these typical properties. Here we focus on fundamental principles.

Although **replication** as defined here **has much better properties** from a risk viewpoint on enterprise-critical data, there remains a gap in favour of backup: backup is typically implemented as a *logical copy*, which lowers risks from certain types of **data corruption**, such as filesystem corruption, for which only risky repair workarounds like `fsck` are the last resort when you don't have a backup *in addition*[2] to replication.

Because of these typical differences, enterprise-critical data typically deserves *both* solution classes at the same time.

Manager Hint 2.3: **Important requirements for replication**

A good replication solution is **symmetrical**. There a two (or more) copies at different locations. They are either active at the same time (which works reliably only rack-to-rack over crossover cables, see section Explanation via CAP Theorem), and/or they need to **switch their roles quickly**. Switching should have two different triggers: **planned handover**, vs **unplanned failover** in case of an incident.

Symmetry is an important precondition for **fast reaction** onto incidents. For **enterprise-critical data**, this is important for drastically **lowering** the expectance value of **losses by incidents**.

Confusion of solution classes replication vs backup and/or their corresponding problem classes / properties can be harmful to enterprises and to carreers of responsible persons.

Hint: the *granularity* of replication handover / failover is important for maximum flexibility. See section Flexibility of Handover / Failover Granularities.

2.4. What is Location Transparency

Replication as defined in the previous section works only reasonable fast enough when Location Transparency is implemented reasonably well, see also section Where to implement Location Transparency. Here is a brief explanation what it is.

Location Transparency is an extremely important and well-known **fundamental principle** in Distributed Systems, and has attracted research for decades.

[2] An integrated solution for continuous replication via logical copies would be difficult. There is a *concept mismatch* between logical copies and strict consistency requirements posed by fast handover, while at the same time compensation of logical data corruption would require the *opposite* of strict consistency. Notice that logical copies are residing at higher layers, e.g. filesystems or database records, while pure replication is easier done at block layer. See also section Performance Arguments from Architecture. Notice that snapshots at block layer cannot *reliably* protect against long-lasting **silent corruptions**. Even higher-layer ZFS snapshots treated in section Point-in-time Replication via ZFS Snapshots cannot provide the same protection level as a classical per-inode multi-generation backup onto a different filesystem type, thus lowering systematical risks from software bugs in filesystem code.

Simply stated, it means that the location of an object or of a service is never (part of) a primary key, but any access is via a *logical name* not depending on the location. Thus the location may (relatively easily) change at runtime.

There are numerous examples where this fundamental principle is obeyed. Unfortunately, there are also many examples where it is violated.

Example 2.1: **Phone numbers**

Phone numbers are *not* location transparent in general. For stationary phones, they contain a location-dependent prefix. In general, it is not possible to move to a different city while keeping the old stationary phone number. In case of mobile phones / cellphones, numbers are "more location transparent", but even there they are *not fully* location transparent: for international calls, they contain prefixes referring to the country, e.g +1 for US or +49 for Germany. In practice, it is not easily possible to permanently move from Germany to US, without giving up the old number after a while. In addition, often the *service provider* and/or the network technology (D-net vs E-net etc) may be also be encoded in cellphone numbers, e.g. somewhere as an infix, so changing the provider may have some restrictions. However, for *most practical purposes*, such as Europeans spending their holidays in US, mobile phone numbers are *sufficienctly* location transparent.


In practice, location transparency is not just a boolean property. As explained by the cellphone example, it may have various **degrees**. In addition, it can refer to different subsystems at different architectural layers. Some layers / some components may be (more or less) location transparent, but others not at all. Thus it is important to mention the **layer or the component** when talking about location transparency.

Interestingly, the Wikipedia article https://en.wikipedia.org/wiki/Location_transparency is an incomplete stub when this section was written (Autumn 2019). It seems that people are actually paying less attention to it.

Manager Hint 2.4:


Major violations of location transparency are almost always carrying some **technical debt**, likely causing future problems and impediments.


Therefore, establishment of reasonable location transparency needs to be seen as **best practice**.


It may happen that somebody thinks there would bot be enough time and/or resources for implementing certain kinds of location transparency. Although in many cases this is not really true, there might be some corner cases where it sometimes is true, or close to true.

Manager Hint 2.5:

Investments into location transparency are often ***longterm* investments**. Not doing it will likely **decrease your business opportunities** and **increase your risks** in the long term.


Location transparency is simply a certain type of **redirection mechanism**, which ***automatically* follows the current location** of a service and/or its storage. It makes you *independent* from various placement strategies.

Once you have established location transparency, a **multitude of placement strategies** for your services and/or your storage locations is possible. This opens up more **opportunities** for **higher efficiency**.

If anyone is arguing that location transparancy were *not needed* as a major feature, you should check whether such a person is really an expert. There needs to be a clear and valid justification for such an opinion.

Hints for implementation of location transparency are in section Where to implement Location Transparency.

2.5. What is HA = High Availability

HA is defined by a single number, denoting the *minimum percentage of uptime* of a certain system from a user's perspective. Some examples:

- 99% availability: a total downtime of more than 87.6 hours per year is not acceptable.
- 99.9% availability: a total downtime of more than 8.76 hours per year is not acceptable.
- 99.99% availability: a total downtime of more than 52.56 minutes per year is not acceptable.

HA is a **requirement**. Requirements are characterizations of the **problem space**. In software engineering, requirements are *strictly separated* from any measures, how a requirement can be met (solution space). In general, there may be *several* solutions for achieving a certain HA percentage.

Manager Hint 2.6:

Some of the potential solutions for the same HA percentage may be much more **expensive** than others, sometimes by *factors*. We will see some examples later.

Some people are arguing *incorrectly*, by claiming that *any* HA solution would *need* to be built up by *hardware redundancy*. Some people even believe that redundancy would be needed at *each and every single hardware component*, otherwise it would not be HA. This confuses requirements with solutions. It is wrong in general, because even a certain degree of hardware redundancy cannot guarantee a certain overall hard+software HA percentage in general, for example when certain components such as failover software are not reliable enough. See also section Detailed Explanation of `BigCluster` Reliability for a counter-example, where addition of more redundancy $> k$ does not help. Of course, higher degrees of HA are *typically(!)* built using certain types and degrees of redundancy, including variants like geo-redundancy. In general, however, there might be other means for achieving HA, like extremely quick automatic repair methods, self-healing[3] systems, etc.

2.6. What is Geo-Redundancy

From the technical viewpoint of HA, geo-redundancy belongs to the *solution space*. From the viewpoint of **government authorities**, and/or from **owners** of a company / rating agencies

[3] This is no joke. For example, certain spacecrafts need to run for years or even for decades, without any maintenance. Thus it helps enormously when some of their components are self-healing, for example certain surfaces or shields after a hit by micro meteorites.

determining the **business risk** and the **stock exchange value** of a company, it is also a *requirement*.

Geo-redundancy means that the **risk** of certain types of geo-localized **physical impacts**, such as earthquakes, floods, terrorist attacks, cascading mass power blackouts, etc, must be **compensated** by being able to run at least the **core business** from another geo-location within some reasonable timeframe.

Manager Hint 2.7:

Notice that the same family of requirements can be solved *very* differently. This guide explains ways for both **cost reduction** and **risk reduction** at the same time, by *combining* HA requirements with geo-redundancy requirements in a clever way, such that the combined solution will meet both at the same time. A resulting combined solution is called **Football on top of MARS**. It provides additional operational value, such as load balancing via the **ability for butterfly**, see section Flexibility of Handover / Failover Granularities.

There are some ongoing political discussions about detail requirements for geo-redundancy. The mimimum distance requirement between suitable geo-locations is seen differently by different interest groups, and even differently in different countries. Background are the **enormous cost** for setup of a datacenter.

While some NGOs = Non-Governmental Organizations are fighting for a minimum distance of only 5 km, the German government authority BSI recommends a minimum distance of 200 km between datacenters for **critical infrastructures.** See https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Standort-Kriterien_HV-RZ/Standort-Kriterien_HV-RZ.pdf?__blob=publicationFile&v=5. Although this is only a "recommendation" officially, certain sectors like **banking** are actually forced to treat this more or less like a requirement.

For an observer, it could be interesting how *international requirements* will evolve, and how rating agencies will change their rules during the course of the next decades.

2.7. What is *Cloud Storage*

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

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

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

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

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

A detailed analysis of consequences from this definition is in secction Architectural Properties of Cloud Storage.

2.8. What is SDS = Software Defined Storage

As explained in https://en.wikipedia.org/wiki/Software-defined_storage, SDS is a **marketing term**, subsuming a wide variety of offerings from several *vendors*.

In essence, it can be *almost anything* from the storage area, where hardware can be treated independently from software, or at least some software configuration is available.

Example 2.2:

Even a "simple" HDD = Hard Disk Drive device has not only some **network interface** (typically SATA or SAS in place of Ethernet), but also contains some software called

firmware, which *could* (at least potentially) be exchanged independently. Believe it or not: even such a "simple hardware" device is providing **storage virtualization**, although a rather primitive one. For example, it maps logical sector numbers (LBNs) to physical coordinates like CHS = Cylinder / Head / Sector, or similar. Newer 4k sector disks can emulate old 512 byte sector formats, etc. Thus such devices would match the fuzzy Wikipedia description of SDS.

 In practice, the term SDS is a **tautology** because it can mean almost anything from the storage area, thus the term is not really useful.

In order to talk about SDS in technical terms of architecture, here is an *attempt* to somehow narrow it down, and to somehow relate it to Cloud Storage:

SDS (in the sense of this guide) is a Cloud Storage system.

 Treating SDS as equivalent to Cloud Storage makes it more useful, but neglects the opportunity for defining something useful inbetween of Cloud Storage and "anything".

Notice that a Wikipedia search "storage as a service" (which could be abbreviated StaaS) is delivering a redirection to "Cloud Storage". Another missed opportunity for getting some useful structure into the **wild-growing jungle**, and for clearly explaining differences, and for a fruitful discussion of pro and cons.

Details 2.2: **Remark**

This is an indicator that the storage area is not really mature. There are more short-sighted hypes than fundamental concepts. This architecture guide is an attempt to guide[a] you through the hype jungle in a structured way.

[a] German saying, semantically translated to English: "You cannot see the forest because there are too many trees in front of it."

Manager Hint 2.8: **Indirect cost of hypes**

 Beware of hyped buzzwords like "storage as a service". It narrows your attention to network-centric architectures, and distracts your attention from major cost saving opportunities like `LocalSharding` (see section Variants of Sharding).

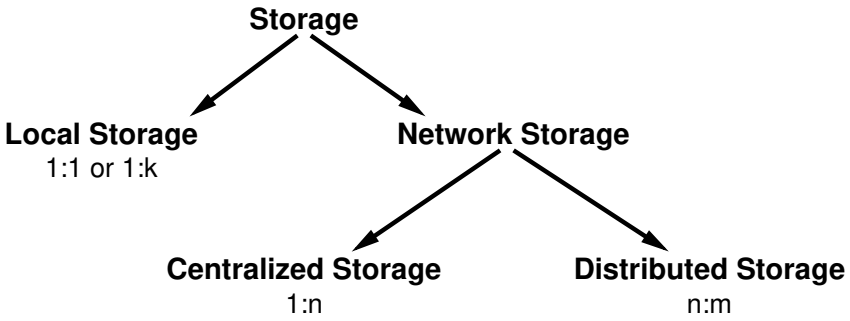
3. Architectural Principles and Properties

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

Manager Hint 3.1:

As a responsible manager, you will make architectural decisions, even if you are *not aware* of them. Bad decisions, even if you are not aware of its consequences, can endanger major products, and increase cost by *factors*. Once you have commited to a certain architecture, it will be *extremely cumbersome* to modify it later. Thus you need to get an architecture right from start. Typically, you will have **only one shot**.

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



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

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

However, this picture has changed with the advent of a new **load balancing** method called **LV Football**, see `football-user-manual.pdf`.

Manager Hint 3.2:

When Football is combined with a `FlexibleSharding` architecture (see section Flexi-bleSharding), practically the same flexibility as promised by `BigCluster` is possible.

3.1. Architectural Properties of Cloud Storage

Brief recall from section What is *Cloud Storage*. Cloud storage is

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

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

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

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


The requirement (1) "act as one" implies some appropriate type of **location transparency** (see section What is Location Transparency).


The definition says nothing about the **granularity** / sizes of the distributed resources. See section Granularity at Architecture for a more detailed discussion of opportunities arising from better informed decisions about this.


Notice that the term "network" does not occur in this definition. However, the term "distributed resources" is implying *some(!)* kind of network.


The definition does *not* imply some *specific* type of network, such as a costly **storage network** which must be capable of transporting masses of IO operations in **realtime**. In general, we are free to use other types of networks, such as cheaper **replication networks**, which need not be dimensioned for realtime IO traffic, but are sufficient for **background data migration**, and even over long distances, where *any* network has some bottlenecks.


Often, there are **restrictions from technology**. Not every architecture as discussed in this guide can be easily implemented via a certain technology. Example: when a so-called **Vendor Lock-In** is binding you to to a certain brand of commercial storage boxes, certain opportunities will be missed. By going to self-built and self-administered RAID storage, typically an invest factor between 3 and 10 can be saved (see section Cost Arguments from Technology). On top of this, about another factor of 2 is possible, about *halving your total hardware invest*, by use of Linux-based local storage + Football in place of network-based commercial storage, provided it is possible for your use case. See sections Proprietary vs OpenSource and Local vs Centralized Storage.


Notice that the definition says nothing about the **time scale** of operations[1]. In general, we are free to implement certain operations, such as **background data migration**, in a rather long timescale (from a human point of view). This bears an opportunity for **major cost reduction**, as well as **improving reliability** by decreasing dependencies from (hidden) SPOFs[2] = Single Points Of Failure.

Example 3.1: Replication network failures

Football on top of MARS for background LV migration over both short and geo-distances. When the replication network is down, it will just pause for a while, and MARS will automatically resume once the network is up again. Football can be configured to also resume the higher-level migration process, when necessary.

Example 3.2: Storage network failures

It is clear that a failure of a classical storage network will halt all services depending on it. Some people believe that realtime storage networks cannot be avoided, in order to react at varying load situations, and are running much faster due to load distribution. This is not the full picture:

[1] Go down to a time scale of microseconds. You will then notice that typical IO operations will require several hundreds of machine instructions between IO request *submission* and the corresponding IO request *completion*. This is not only true for local IO. In network clusters like Ceph, it will involve much more work, like creation of network packets, and lead to additional IO latencies implied by the network packet transfer latencies.

[2] Several people appear to work with the *assumption* that networks are available all the time. Although minor network outages can be compensated very well, there remains a **residual risk** for a major outage, similar to what happened in Fukushima. Thus such an attitude can endanger both companies and carreers.

1. Football plus FlexibleSharding can achieve a similar level of elasticity.
2. Load distribution is essentially nothing else but a variant of **data striping**. If you really need it for performance reasons, you can often do similarly with certain types of local RAID, such as RAID-10 or RAID-60, and with a variety of RAID parameters. Notice that *any* kind of data striping, whether at block level or at object level, is coming with some cost[a].
3. LocalStorage is even faster (when using a comparable technology yielding the same size), because IO does not involve *any dedicated storage network* at all. Therefore, it is also more reliable (when using comparable technology).
4. Reorg tasks: these can occur in all top-level architectures. In general, not all operations can run in realtime, by construction. For example, increasing the number of replicas in an operational Ceph cluster, already containing a few hundreds of terabytes of data, will not only require additional storage hardware, but will also take a rather long time, implied by the very nature of bigger reorganisational tasks.

[a] For a given redundancy degree k, **reliability is reduced** by striping. In case of RAID, this is well-known since decades. Unfortunately, in case of BigCluster some misleading "propaganda" was blurring the public opinion for many years. Notice that the BigCluster analysis in section Detailed Explanation of `BigCluster` Reliability is showing up some parallels to the well-known reliability loss caused by RAID striping, when some granularity differences (block vs object level etc) are ignored.

When **geo-redundancy** = some minimum distance between datacenters for **survial of geo-disasters** like earthquakes or floods is added to (2) as an additional requirement (see also section What is Geo-Redundancy), some *further consequences* will arise. For example, the German government authority BSI recommends a minimum distance of 200 km between datacenters for **critical infrastructures**[3]. Over suchalike distances, realtime storage networks cannot be used anymore in general. Thus some kind of "migration" of data over long distances will be needed anyway.

Manager Hint 3.3:

Since data migration is needed *anyway* over long distances, there is an opportunity for **saving cost *and* increasing reliabilty + flexibility** all at the same time.

Manager Hint 3.4:

Basic idea behind Football on top of a Sharding model: **minimize the *distances*** between your storage spindles and the corresponding data processing. When background data migration is automated properly, real-time storage networks can become superfluous, or at least the corresponding realtime IO traffic can be drastically reduced. When minimization is well dimensioned, a pair of storage + application server residing in the same geo-location can be **collapsed into a single box**. This is not only a **major cost reducer**, it also **improves reliability** because there are less components which can fail.

[3] See https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Sicherheitsberatung/Standort-Kriterien_HV-RZ/Standort-Kriterien_HV-RZ.pdf?__blob=publicationFile&v=5
Some press comments on this: https://www.it-finanzmagazin.de/bsi-rechenzentren-entfernung-bafin-84078/

Manager Hint 3.5:

Unfortunately, this opportunity is easily *missed* if both system architects and responsible managers are just requiring only DR = Disaster Recovery over long distances, instead of requiring the ability for butterfly (see section 3.5).

Details 3.1:

Essentially, suchalike minimum requirements can be easily interpreted as "everything has to be doubled" in order to survive any geo-disaster[a]. This would double cost in comparison to certain kinds of fully locally redundant architectures, missing the opportunity for ***splitting*** much of the overall redundancy into two geo-locations, **instead of doubling** virtually everything.

Some people are arguing that doubling were unavoidable, which is *incorrect in general*, as Football can demonstrate as a positive counter-example. See section Cost Arguments from Architecture.

Counter-productive cost arguments are sometimes heard when geo-redundancy is discussed about, without considering newer possibilities such as Football. As explained in section Flexibility of Handover / Failover Granularities, the **granularity of failover** should not be required as a **coarse failover of a full datacenter**, but explicitly be required as **fine-grained cross-geo failover + handover at VM level**, or at a similar granularity (c.f. section Granularity at Architecture). This will force people to think about **wide-area distribution** of resources instead of plainly doubling them (once again[b]).

[a] A geo-disaster like an earthquake will typically last for weeks, if not months, until it is fully repaired. During such a period, a single surviving datacenter must be capable of providing "good enough" SLAs. These disaster-SLAs can be lower than usual. For example, in place of an ordinary 99.98% availability, 98% may be a sufficient target *during* such a geo-disaster. By unnecessarily requiring much more during a very rare corner case, you can easily explode the cost, even beyond doubling, without reasonable benefit during ordinary operations.

[b] Example: commercial storage boxes from NetApp, IBM, etc already have some *local redundancy*, typically doubling the amount of physical disks you are actually buying when you buy a single storage box. Typically, the amount of *physical* disks is not directly reported as a KPI, although it is major cost producer. When introducing geo-replication, you will likely need to buy double the number of boxes, resulting in a total of about 4x the capacity at the physical layer. In contrast, MARS + Football can often be built on top of local RAID-6. As pointed out in section Cost Arguments from Architecture, this leads to only about 2.2x the physical capacity you will need to buy. In addition, the rackspace is much lower when using local storage, reducing the number of servers to deploy and administer, and reducing networking cost by omission of dedicated storage networks.

Manager Hint 3.6:

Important keyword for flexible cross-geo distribution: **ability for butterfly**, see section Flexibility of Handover / Failover Granularities.

There is a **tradeoff** between the effort for implementation of per-VM flexibility, and hardware cost savings. Sometimes arguments are heard that a high level of flexibility would be too costly. Although this might be true in some relatively small corner cases, the picture can rapidly change when thousands of servers and/or petabytes or storage are involved.

Manager Hint 3.7:

Doubling the overall cost for big datacenters instead of intelligently geo-distributing resources, is likely much more cost intensive in the long term than investing once into **intelligent abilities** of the company like Football, which can then **scale up** (more details see section Scalability Arguments from Architecture).

As a consequence from sufficiently fine-grained handover + failover, the above definition of cloud storage can be **met at geo-datacenter level**, i.e. the distributed resources according to (1) will be distributed over *multiple geo-redundant* locations / datacenters. As pointed out in section Cost Arguments, sometimes this may be even cheaper than building certain types of local redundancy inside the same datacenter.

Details 3.2:

The famous CAP theorem (see section 5.3 on page 93) is one of the motivations behind requirement (4) "eventually consistent". This is not an accident. There is a *reason* for it, although it is not a *hard* requirement. Strict consistency is not needed for many applications running on top of cloud storage. In addition, the CAP theorem and some other theorems cited at `https://en.wikipedia.org/wiki/CAP_theorem` are telling us that Strict Consistency would be **difficult and expensive** to achieve at global level in a bigger Distributed System, and at the cost of other properties. More detailed explanations are in section Explanation via CAP Theorem.

3.2. Suitability of Architectures for Cloud Storage

Details 3.3:

There are some consequences from the above definition of Cloud Storage (see section Architectural Properties of Cloud Storage), for each of our high-level storage architectures:

Distributed Storage, in particular `BigCluster` architectures (see section Distributed vs Local: Scalability Arguments from Architecture): many of them (with few exceptions) are conforming to all of these requirements. Typical granularity are objects, or chunks, or other relatively small units of data.

Distributed Storage is the growground where Cloud Storage was invented.

Many contemporary `BigCluster` implementations are *not really* supporting **geo-distribution** of masses of objects over long distances, in the sense of well-proven use cases (maturity). Small object granularity and/or strict consistency on top of unreliable objects are worsening the effects of the CAP theorem and its sister theorems. Thus object-based architectures are typically only suited for local (non-geo) operations.
Example: at the moment (mid 2019), Amazon AWS is offering object replication only over campus distances, which cannot meet the requirements from BSI.

Centralized Storage: does not conform to (1) and to (4) by definition[a]. By introduction of synchronous or asynchronous replication, it can be made to *almost* conform, except for (1) where some concept mismatches remain (probably resolvable by going to a RemoteSharding model on top of CentralStorage, where CentralStorage is

only a *sub-component*). Typical granularity is replication of whole internal storage pools, or of filesystem instances.

 In general, `CentralStorage` architectures are a **mismatch** to Cloud Storage, by their very nature. Healing suchalike **concept** mismatches may be close to impossible, or at least very tricky and costly.

 Adding asynchronous replication to commercial storage boxes will not only double the cost, which are anyway at a very high starting level (see section Cost Arguments from Technology). In addition, the **handover granularity** (see section Flexibility of Handover / Failover Granularities) may not meet the optimum.

LocalStorage, and some further models like `RemoteSharding` (see section Variants of Sharding)
There is some historical belief that cloud storage cannot be reasonably built on top of them. When newer developments and opportunities are taken into account, this has changed. Here are some examples, mentioning some example components:

(1) can be achieved at LV granularity with Football (see `football-user-manual.pdf`), which creates a **Big Virtual LVM Pool**. Football is in mass production at 1&1 Ionos since August 2018.

(2) can be achieved at disk granularity with local RAID, and at LV granularity with DRBD or MARS. Both are in mass production since several years.

(3) can be achieved at LV granularity with LVM snapshots, and/or ZFS (or other filesystem) snapshots, and/or above filesystem layer by addition of classical backup.

(4) at least `Eventually Consistent` or better can be alternatively achieved by one of the components

(4a) DRBD, which provides `Strict Consistency` during `connected` state, but works only reliably with passive crossover cables over **short distances** (see CAP theorem in section Explanation via CAP Theorem).

 DRBD violates any type of consistency within your *replicas* during (automatic) re-sync, and thus does not *fully* comply with the above definition of cloud storage in a *strong* sense. You may argue at a coarse time granularity scale in order to "fix" this.

(4b) MARS, which works over **long distances** and provides two different consistency guarantees at different levels, *both at the same time*:

locally: `Strict Consistency` at local LV granularity, also *within* each of the LV replicas.

globally: `Eventually Consistent` *between* different LV replicas (global level).
The CAP theorem (see section 5.3) says that `Strict Consistency` is **not possible** in general at *unplanned failover* during long-distance network outages (P = Partitioning Tolerance), when A = Availability is also a requirement.

 However, in case of a *planned handover*, MARS is also `Strictly Consistent` at a global level, but may need some extra time for catching up.
Notice: global `Strict Consistency` is also possible at a *coarse timescale*, in accordance with the CAP theorem, if you decide to sacrifice A = Availability during such a network incident by simply *not* doing a failover action. Just wait until the network outage is

gone, and MARS will automatically resume[b] everything ASAP, and thus you are using MARS *only* as a protection against **fatal** storage failures / unplanned **disasters**.
Notice: A = Availability is *not generally* required by the above definition of cloud storage, because from a user's perspective it would not generally make sense in the global internet where connection loss may anyway occur at any time. Thus it is a valid operational strategy to *not* fail-over your LVs during certain minor, or even during certain types of major network outages (e.g. when failover would not improve much).
Notice: long-term **disaster tolerance** (e.g. perpetual loss of some storage nodes during an earthquake) is *not* modeled by the CAP theorem, but is more or less required by (2) and (3) from the above definition of cloud storage.

`BigCluster` architectures are creating *virtual* storage pools out of physically distributed storage servers. For fairness reasons, creation of a big virtual LVM pool, must be considered as *another* valid Cloud Storage *model*, matching the above definition of Cloud Storage. The main architectural difference is granularity, as explained in section Granularity at Architecture, and the stacking order of sub-components.

[a] Notice that sharding on top of CentralStorage is no longer a CentralStorage model by definition, but a RemoteSharding model according to section Variants of Sharding.
[b] This automatic MARS behaviour is similar to the behaviour of DRBD in such situations, when DBRD can automatically go to `disconnected`-like state, and you are later manually or automatically resuming the DRBD connection for an incremental re-sync. MARS does everything automatically because it has no firmly built-in assumptions about the actual duration of any network communication.

Football is creating **location transparency** inside of the distributed virtual LVM pool. This is an important (though not always required) basic property of *any* type of clusters and/or grids.

3.3. Layering Rules and their Importance

Complex systems are composed of several layers. In this section, we will learn how to organize them (close to) **optimally**.

Manager Hint 3.8:

Non-optimal layering is a major cause of **financial losses**, decreased reliability / **increased risk**, **worse scalability**, etc.

Well-designed systems can be recognized as roughly following Dijkstra's famous **layering rules,** originating from his pioneer THE project. Wikipedia article https://en.wikipedia.org/wiki/THE_multiprogramming_system is mentioning an important principle behind Dijkstra's layers, in section "Design":

> **higher layers only depend on lower layers**

The original article http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD196.PDF resp https://dl.acm.org/citation.cfm?doid=363095.363143 contains very interesting information, and is a highly recommended reading. The introduction and the progress report is relevant for today's managers, optionally the "design experience", and certainly the conclusions. The section "System hierarchy" is relevant for today's system architects, while the rest is mostly of historical interest for OS and kernel specialists. Reading the relevant parts after more than

50 years is extremely well-invested time. Dijkstra provides solutions for **invariant problems** which are facing us today with the same boring ignorance, even after 50 years. The heart of his conclusions is **timeless**.

Dijkstra's methodology has been intensively discussed[4] by the scientific OS community, and has been generalized in various ways to what folklore calls "Dijkstra's layering rules". Here is a condensed summary of its essence:

- Layers should be viewed as **abstractions**.
- Higher layers should only depend on lower layers.
- Each layer should **add** some **new** functionality.
- Trivial conclusion by reversing this: **Regressions** should be avoided. A regression is when some functionality is *lost* at a higher layer, although it was present at a lower layer.

 This sounds very simple. However, on a closer look, there are numerous violations of these rules in modern system designs. Some examples will follow.

 The term "**functionality**" is very abstract, and deliberately not very specific[5]. It is **independent** from any implementations, programming languages, or programming / user interfaces, or other matters of **representation**.

 The same functionality may be accessible via *multiple* different **interfaces**. Thus a different interface does *not* imply that functionality is (fundamentally) different.

 Nevertheless, people are often confusing functionality with interfaces. They think that a different interface must provide a different functionality. As explained, this is not correct in general.

Manager Hint 3.9:

 Confusion of interfaces with functionality is exploited by so-called marketing drones and other types of advertising (e.g. aquisition of **venture capital**), in order to **open your money pocket**. As a responsible manager, you should always check the *functionality* behind a certain product and its interfaces: what is *really* behind the scenes?

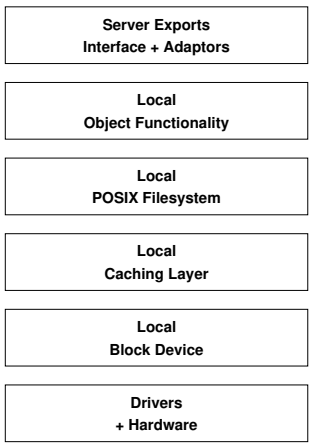
3.3.1. Negative Example: object store implementations mis-used as backend for block devices / POSIX filesystems

Several object store implementations are following the client-server paradigm, where servers and clients are interconnected via some $O(n^2)$ storage network (see section Distributed vs Local: Scalability Arguments from Architecture).

We start by looking at the *internal* architecture of certain OSD = Object Storage Device (see https://en.wikipedia.org/wiki/Object_storage) implementations. Some publications are treating them more or less as black boxes (e.g. as abstract interfaces). Certain people are selling this as an advantage.

However, we will check this here. Thus we need to take a closer look at the *internal* sub-architecture of certain OSD implementations:

[4] An important contribution is from Haberman, by clarifying that there exist serveral types of hierarchies.
[5] Elder schools of software engineering know that **design processes** must *necessarily* start with unspecific terms, in order to start to bridge the so-called **semantic gap**.



The crucial point is: several OSD implementations are internally using **filesystems** for creating the object abstraction.

Details 3.4: **OSD implementation strategies**

For implementors, this seems to be a very tempting[a] shortcut strategy. Implementing their own object store functionality on top of block devices, which could easily take some years or decades until mature enough for production use. Linus Torvalds, for example, is measuring the maturity cycles of filesystem implementations in units of *decades*, not in years. Pure object stores would need to solve similar *fundamental problems*, like **fragmentation problems**, which is a science in itself. Thus existing kernel-level filesystem implementations are often just re-used for OSDs. They seem to be already there, "for free".
However, at architectural level, they are *not* for free. They are violating Dijkstra's layering rules by causing *regressions*.
At abstract functionality level: passive objects, and even some associated *rich metadata*, are more or less nothing else but **restricted files**, optionally augmented with POSIX EAs = Extended Attributes[b].

- Object IDs can be **trivially mapped**[c] to filenames / pathnames. At *abstract functionality* level, there is almost no difference between pathnames and object IDs, with the exception that pathnames are *more general*, e.g. by allowing deep nesting into subfolders.
- Newer versions of certain Linux-based filesystems can even automatically generate random object keys, and even atomically (= free of race conditions when executed concurrently). Example: supply the option `O_TMPFILE` to `open()`, followed by `linkat()`.
- While filesystems are translating file IDs = pathnames into **file handles** before further operations can be carried out, object stores are typically skipping this intermediate step from a user's viewpoint. The user needs to supply the object ID for *any* operation.

  
  In the implementation, this can lead to considerable **runtime overhead**, because ID lookup functionality similar to `open()` has to be re-executed for each operation. In contrast, valid file handles are *directly* referring to the relevant kernel objects, without need to search for a filename again. Extreme example: consider the total runtime overhead by repeatedly appending 1 byte to an object in a loop.
- Consequently, certain file operations associated with file handles are missing in pure object stores, such as `lseek()`, as well as many other operations.
- **Concurrency** functionality of a POSIX-compliant[d] filesystem is much more elaborated than actually needed by an object store. Examples: fine-grained locking operations like `flock()` are typically not needed in pure object stores. The `rename()`

operation, including its side effects onto concurrency, would even *contradict* to the fundamental idea of immutable object IDs.

- **Shared memory** functionality. Filesystems need to support `mmap()` and relatives. This is *inevitable* in modern kernels like Linux, for hardware MMU-supported **execution of processes**, employing the COW = Copy On Write strategy. See `fork()` and `execve()` syscalls, and their relatives. In general, shared memory can be used by several processes concurrently, and on **sparse files**. Filesystem implementors need to spend a considerable fraction of their total effort on this. Concurrency on shared memory, together with SMP scalability to a contemporary degree, is what makes implementation really hard, and why there are only relatively few people in the world mastering this art. As a manager, compare with Dijkstra's remarks on required **skill levels** for serious OS work...

   Object stores are typically lacking shared memory functionalities completely. Thus they are not suited as a *core component*[e] of a modern OS.

   In comparison, creating a different interface for an already existing subfunctionality, and optionally adding some metadata harvesters and filters, is requiring much lower[f] skills and effort.

- Several less-used functionalities, like **hardlinks** etc.

[a] Linux kernel implementations of filesystems need typically at least 10 years, if not 20 years to be considered "mature" enough for mass production on billions of inodes.
[b] Posix EAs = Extended Attributes implementations as provided by classical filesystems are providing roughly the same functionalities as *passive* augmented object metadata. Even active metadata is possible, e.g. by separate processes present in `Akonadi` or `miner`. With such a standard addendum, classical filesystems can also be used for providing active functionality.
[c] Example: random hex key `0123456789ABCDEF` can be trivially mapped to a path `/objectstore/0123/4567/89ABCDEF` in an easily reversible way (bijective mapping)
[d] POSIX requires **strict consistency** for many operations, while weaker consistency models are often *sufficient* (but not required) for object stores.
[e] Years ago, certain advocates of object stores have claimed that filesystems would be superseded by object stores / OSDs in future. This is unrealistic, due to the lack of mentioned basic functionalities. When missing functionality would be added to object stores, they would turn into filesystems, or into so-called "hybrid systems". Consequently, there is no clue in claiming that object stores are forming a fundamental base for operating systems. They are essentially just a special case, optionally augmented with some active functionality, which in turn should be attributed to a *separate* layer, independently from filesystems or object stores.
[f] Roughly, computer science students should be able to do that after a 1 semester OS course.

Obviously, these functionalities are *lost* at the object layer and/or latest at the exports interface. Thus we have identified a Dijkstra regression.

 As explained in the detail box: **trivial differences** in an interface, such as usage of intermediate file handles / or not, or near-trivial **representation** variants like pathnames vs object IDs, are no valid[6] arguments for claiming differences in the *abstract functionality* in the sense of Dijkstra.

Manager Hint 3.10: *Real* functionality behind object stores

 Conclusion: *passive* object stores are approximately nothing else but a **special case** of fileystems.

[6] Arguing with (trivial) syscall combinations or trivial parameter passing can be observed sometimes. As a responsible manager, you should draw another conclusion: someone arguing this way is either fighting for a particular **political interest** in an **unfair** manner, and/or in reality he demonstrates nothing but an extremely **poor skill level**.

Now let us look at some *active* functionality of some object stores, such as automatic collection of **rich metadata**, or filtering functionality on top of them: are suchalike functionalities really specific for object stores?

There is a clear answer: NO.

Example 3.3:

For example, `Akonadi`, `miner`, and similar standard Linux tools are indexing the EXIF metadata of images, or metadata of mp3 songs, videos, etc, residing in a classical filesystem.

Do not draw wrong conclusions from the fact that the classical Unix Philosophy (see https://en.wikipedia.org/wiki/Unix_philosophy) has a long tradition of **decomposing** functionality into **separate layers**, such as the distinction between passive filesystems and active metadata indexing. When some object advocates are merging these separate layers into one, this is *not* an advantage. In contrary, there are disadvantages like *hidden cartesian products* occurring at architecture level, and possibly also in implementations.

Manager Hint 3.11: ***Real* implementation value of OSDs $\Longrightarrow$ business value**

For responsibles: when certain advocates are claiming that functionality mergers, such as more or less **trivial combinations** of filesystem sub-functionality with some metadata harvesters, are constituting some new product, be **cautious**. It is about ***your* money**, or about your company's money.
While it might be a "new" product from the perspective of end customers, you should **check** the **technical effort** for "implementing" the "new" functionality. There are cases where more than 90% functionality is already there. When it is from OpenSource, do not pay a lot of money for some more or less trivial adaptors.

When more than 95% of functionality is already there *for free*, beware of costly blown-up architectural ill-designs, such as $O(n^2)$ client-server BigCluster architectures.

Dijkstra's layering rules can be used as tools for analyzing this, and for discovery of **technical debt** by unfortunate layering, causing further cost and trouble in the long term.

When augmented metadata functionality is present (whether actively or passively), it should *not* be viewed as an integral part of object stores, but as an *optional addendum*.

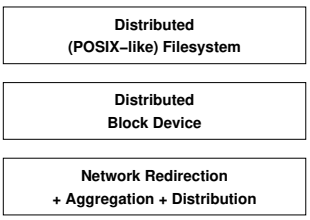
Reason: **rich metadata is *conceptually independent*** from both filesystems and object stores.

You may wonder what is the *damage* caused by Dijkstra regressions at object stores.

We now look at a *mis-use* of object stores, which has been unfortunately advocated by object store advocates several years ago. Some advocates appear to have learned from bad experiences with suchalike setups (see examples in section 4.3.5), no longer propagating suchalike mis-uses anymore, but to focus on more appropriate use cases for *native* object stores instead.

We continue by looking at the client part of distributed block devices / distributed filesystems on top of OSDs. The following example requires POSIX compliance[7] for toplevel application Apache webhosting with `ssh` access:

[7] 1&1 Ionos has made the experience that a near POSIX-compliant filesystem called `nfs` did not work correctly, causing customer complaints, because it is *not fully* POSIX-compliant.



It should catch your eyes that both block-device and filesystem functionality is re-appearing once again, although it had been already implemented at OSD level. Obviously, there are two more Dijkstra regressions.


Do not over-stress the fact that now we are creating *distributed* block-devices, or *distributed* filesystems in place of local ones. This does *not* imply that a `BigCluster` architecture is needed on top an $O(n^2)$ storage network, or that **random replication** inducing further problems and serious reliability problems (see section Reliability Arguments from Architecture) is needed. There are near-trivial alternatives at architecture level, see Variants of Sharding.


There is another (fourth) Dijkstra regression. Distributed block devices are typically storing 4k sectors or similar[8] **fixed-size** entities in the object store, although objects are capable of **varying sizes**. Thus objects and their *dynamic key indirection mechanisms* are "misused" for a restricted use case where array-like virtual data structures would be sufficient. When some petabytes of block device data are created in such a way, a **massive overhead**[9] is induced.

Manager Hint 3.12:

Some damages caused (or at least *supported*) by Dijkstra regressions:

- **Increased invest**. Further reasons like doubled effort are explained in section Cost Arguments from Architecture.
- **Increased operational cost**, both manpower and electrical power. Example: certain Ceph OSD implementations have been estimated as roughly consuming 1 GHz CPU power and 1 GB RAM per spindle. Even when newer versions are implemented somewhat more efficiently, there remains architectural Dijkstra overhead as explained above.
- **Decreased reliability** / **increased risk**, simply caused by **additional complexity** introduced by Dijkstra regressions. Further reasons are explained in section Reliability Arguments from Architecture.
- **Decreased total performance**, simply induced by regression overhead. Some more reasons can be found in sections Explanations from DSM and WorkingSet Theory and Performance Arguments from Architecture.
- **Limited scalability** as explained in sections Scalability Arguments from Architecture and Explanations from DSM and WorkingSet Theory is further worsened by Dijkstra regressions.

3.3.2. Positive Example: ShaHoLin storage + application stack

ShaHoLin = Shared Hosting Linux at 1&1 Ionos. It is a **managed product**, i.e. the sysadmins can login anywhere as `root`. Notice that this has some influence at the architecture. In general, unmanaged products need to be constructed somewhat differently.

[8] Mapping of multiple 4k sectors onto a smaller number of bigger objects (e.g. 128k) opens up another **tradeoff**, called **false sharing**. This can lead to serious performance degradation of highly random workloads.

[9] For example, an `xfs` inode has a typical size of 256 bytes. When each 4k sector of a distributed block device is stored as 1 object in an `xfs` filesystem consuming 1 inode, there is not only noticable space overhead. In addition, random access by large application workingsets will need at least two seeks in total (inode + sector content). Without caching, this just doubles the needed worst-case IOPS. When taking the lookup fuctionality into account, the picture will worsen once again.

ShaHoLin's architecture does not suffer from Dijkstra regressions, since each layer is adding new functionality, which is also available at, or at least functionally influences, any of the higher layers.

Because of this, and by using a scalability principle called Sharding (see sections Definition of Sharding and Variants of Sharding), architectural properties are **close to optimal**.

Details 3.5: ShaHoLin Layering

The following bottom-up description explains some granularity considerations at each layer:

1. Hardware-based RAID-6, with an internal sub-architecture based on SAS networking[a]. The newest LSI-based chip generation supports 8 GB fast BBU cache, which has RAM speed. Depending on the number of disks, this creates one big block device per RAID set. Current dimensioning (2019) is between ≈15 TB on 10 fast spindles in a small pizza box, and 48 large-capacity slower spindles with a total capacity of ≈300 TB, spread over 3 RAID sets. This is somewhat conservative; with current technology higher capacity would be possible, at the cost of lower IOPS.

2. LVM = Logical Volume Management. This is provided by the dm = device mapper infrastructure of the Linux kernel, and by the standard LVM2 userspace tools. It is sub-divided into the following sub-layers:
   a) PV = Physical Volumes, one per RAID set, with practically the same size / granularity.
   b) VG = Volume Group. All PVs ≅ RAID sets are merged into one local storage pool. Typical sizes are between 15 and 300 TB, depending on hardware class. Very old hardware may have only ≈3 TB, but these machines should go EOL soon.
   c) LV = Logical Volumes, one per VM ≅ LXC container instance. Typical sizes are between ≈300 GB and ≈40 TB. When necessary, the size can be dynamically increased during runtime. Typical number of LVs per physical machine (also called **hypervisor**) is between 3 and 14 (or exceptionally only 1 on very small old hardware).

   The number of LVs per hypervisor can change during operations by moving around some LVs ≅ VMs ≅ LXC containers via Football (see `football-user-manual.pdf`). This is used for multiple purposes, such as decommissioning of old hardware, or load balancing, or for physical reorganizations, e.g. defragmentation of racks in some of the datacenters.

3. Replication layer, using MARS. Each LV can be switched over individually (ability for butterfly, see Flexibility of Handover / Failover Granularities). In addition to geo-redundancy, MARS provides the base for Football. LV sizes / granularities are not modified by MARS.

4. Filesystem layer, typically `xfs` mounted locally[b]. This layer is extremely important for getting the granularities right: typically, each xfs instance contains several millions of customer inodes and/or files. In some cases, the number can climb up to several tenths of millions. Reason: shared webhosting has to deal with myriads of extremely small customer files, intermixed with a lower number of bigger files, up to terabytes in a handful of scarce corner cases.

5. LXC containers ≅ VMs. Each of them has a publicly visible customer IP address, which is shared by all of its customers (typically a few hundrets up to several tenthousands per container). Upon primary handover / failover, this IP is handed over to the sister datacenter via BGP = Border Gateway Protocol. Upon Football migrations, this IP is also retained, but just automatically routed to a different physical network segment.

6. Application layer. Here are only some important highlights:
   a) Apache, spawning PHP via suexec. One Apache instance per LXC container is typically sufficient for serving thousands or tenthousands of customers.

      Some surprising detail: `fastcgi` is deliberately *not* used at the moment, because security / **user isolation** is considered much more important than a few *permille(!)* of performance gain by saving a few `fork()` + `execve()` system calls. While the Linux kernel is highly optimized for them, typical PHP applications like Wordpress are poorly optimized, for example by clueless runtime inclusion of ≈120 PHP include files, cluelessly repeated for each and every PHP request. Even when `OpCache` is enabled, this costs much more than any potential savings by `fastcgi`.
   b) EhB = Enhanced Backup. This is a 1&1-specific proprietary solution, supporting a grand total of ≈10 billions of inodes. It is also organized via the Sharding principle, but based on a different granularity. In order to parallelize daily incremental-forever backups, several measures are taken. Among others, customer homedirectories are grouped into 49 subdirectories called *hashes* in 1&1-slang. Both backups and restores may run in parallel, independently for each hash, and distributed over multiple shards. Hashes are thus forming an **intermediate granularity** between xfs instances, and a grand total of ≈9 millions of customer home directories.

[a] Certain advocates are overlooking the fact that SAS busses are a small network, just using the SAS protocol in place of TCP/IP. When necessary, the SAS network can be dynamically extended, e.g. by addition of external enclosures.

[b] Only on a few old machines, which are shortly before EOL, `/dev/mars/vm_name` is exported via iSCSI and imported into some near-diskless clients. This is an old architectural model, showing worse reliability (more components which can fail), and higher cost (more hardware, more power, more rackspace, etc). Due to iSCSI, IOPS are much worse than with pure `LocalStorage`. Contrary to some old belief, it is *not* much more flexible. The ability for butterfly is already sufficient for rare exceptional overload situations, or for sporadic hardware failures. Since Football also works on the old iSCSI-based architecture, load balancing etc does not need to be done via iSCSI.

3.4. Granularity at Architecture

There are several alternative implementation technologies for (cloud) storage systems. They can be classified according to the granularity of their basic transfer units.

3.4.1. Granularities for Achieving Strict Consistency

End users are *always* expecting **strict consistency**[10] from a storage system. Whenever they are "saving" several "things" to a (cloud) storage system in a particular order, they are expecting to always retrieve the *newest* version of each of them, afterwards.

Here are the most important architectural differences between object-based storages and LV-based (Logical Volume) storages, provided that you *want to cover comparable use cases*:

[10] For an overview of consisteny models, see https://en.wikipedia.org/wiki/Consistency_model. While strict consistency is the most "natural" one as expected by humans, most other models are only of academic interest.

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

As indicated in sections Reliability Arguments from Architecture and Explanations from DSM and WorkingSet Theory, there are problems with object storage's **consistency model** when higher aggregates like LVs or filesystems are *requiring* **strict consistency**, but are built on top of objects which are only *eventually consistent* due to their inherent nature.

3.4.2. Granularity for Achieving Eventually Consistent

This section is *not* about expectations from users. It is about implementation-specific **weak consistency models**, such as **eventually consistent**., see https://en.wikipedia.org/wiki/Consistency_model#Eventual_consistency, or several other weak consistency models and their variants.

The following table reflects use cases for "native" object storage, where eventually consistent (or similar) is sufficient, or at least claimed to be sufficient:

Eventually Consistent sufficient	Objects	LVs
Granularity	medium (1 object = 1 file)	huge (several TiB)
Number of instances	medium	low to medium
Typical access	random keys	named + random
Update in place	no	yes
Resize during operation	no	yes
Object support	native	on top of
Scalable	at cluster	both cluster and grid
Location distances	per datacenter / on campus	long distances possible
Centralized pool management	per cluster	Football uniting clusters
Easy sharding support	possible but expensive	yes

3.5. Flexibility of Handover / Failover Granularities

This section is also relevant for **networking departments** and their **management** in a bigger enterprise.

There are two important properties of replication handover / failover:

1. **Timely behaviour**: how fast can it be done?
2. What is the **granularity**: which are the items that can be switched?
3. Physical **distance**: both geo-redundancy (see section What is Geo-Redundancy) and cross-datacenter replication, even when the latter is only over short distances, are requiring different **network support** than simple handover / failover in the same rack.

All of these aspects are only reasonable to implement via Location Transparency. Location Transparency has been introduced in section What is Location Transparency. Here we look into some details how to implement it.

3.5.1. Where to implement Location Transparency

Details 3.6: **Where location transparency makes sense or not**

In general, it is not necessary to implement location transparency *everywhere*, for each and every single component / subsystem. The art of system architecture consists of knowing

1. where it is *needed*,
2. where it is *beneficial* for future growth / future reqirements in multiple dimensions,
3. where it is (or will be) too expensive to pay off in the mid-term future, using current technology, but nevertheless *cheap provisions for its later introduction* can be prepared, and
4. where its lack can be easily (or even *trivially*) compensated by location transparency at another layer, such that a particular component does not need to be constructed with location transparency, but nevertheless the *overall system* is sufficiently location transparent, and
5. when there are multiple choices *where* to implement it, knowing which will be the best one for a familiy of use cases, and finally
6. *how* to implement it. For example, a common misconception is to believe that storage must always reside at a storage network. Football (see `football-user-manual.pdf`) demonstrates that sufficient[a] location transparency can be achieved on top of local storage, while expensive and performance-eating dedicated storage networks[b] are not generally necessary for achieving location transparency.

[a] There could be arguments that Football's background migrations might be too slow or might take too long for certain use cases. Notice that `BigCluster` also needs data migration during operations, e.g. upon replacement of physical disks. When the `FlexibleSharding` model (see section Flexible-Sharding) is combined with Football, it provides practically the same timescale and flexibility than `BigCluster`.

[b] Anyway, realtime storage networks cannot span long distances. Thus they are not suitable for achieving location transparency in a geo-redundant setup.

In the definition of Cloud Storage in section Architectural Properties of Cloud Storage, the requirement "act as one" is *implying* some appropriate type of location transparency of the resources.

Manager Hint 3.13:

Consequence: any system not sufficiently implementing location transparency of the customer's resources (visible layer from outside) should not be called "Cloud Storage" or a "Cloud Product" when location transparency is not sufficient from the viewpoint of customers.

In the rest of this section, we concetrate on cross-datacenter replication scenarios, including geo-redundancy.

3.5.2. Granularity of Cross-Datacenter and Geo-Redundant Handover / Failover

Typical management buzzwords like DR = Disaster Recovery or CDP = Continuous Data Protection are neglecting the *granularity* of the data units to be protected by replication, and the

ability for quick service[11] handover due to **maintenance** reasons such as power supply maintenance. The following table explains some differences when granularity aspects like replication at physical volume (PV) aka physical disk level versus logical volume (LV) resp filesystem level are taken into account:

Method	Disadvantages	Advantages
Backup at FS level	no real data consistency	logical copy
	no handover / failover	
	no load balancing	
	no CDP / high MTTR	
Backup via FS snapshots	handover cumbersome	some point-in-time consistency
	no real load balancing	
	medium to high MTTR	logical copy
	delayed consistency	
Replication at PV granularity	whole clusters switch	easier to setup
	no load balancing	
	physical copy	
	medium MTTR	
Replication at LV granularity	physical copy	load balancing between LVs
		easy migration / Football
		full handover consistency
		low MTTR

In order to implement good flexibility of handover / failover, the network infrastructure (as well as other infrastructures) must support it. Here are **levels of flexibility**, in ascending order:

0. (completely inflexible) Statically assigned IP addresses at *each* server and at *both* of 2 datacenters, and in particular for **customer traffic**. This is typical for contemporary backup solutions. As a consequence, any handover / failover attempt would need massive sysadmin work, even if there were enough CPU and RAM power at the target datacenter. Switching whole datacenters or bigger server farms would take days, if not weeks, to manually reconfigure. Consequence: sysadmins will heavily dislike such type of work (acceptance problem of geo-redundancy).

Some people think this can be easily done at DNS level. Just update all of your publicly visible DNS records to point to the new IP addresses. However, DNS updates have serious drawbacks for public internet traffic. Although there exists a field TTL = Time To Live for limiting the caching period of DNS clients, this field is *ignored* by many clients / DNS caches throughout the world. In practice it will take days, if not weeks, until the last client has got the new IP address, even if you try to speed this up by setting a TTL of 1 minute. It simply does not work as expected.

Dynamic routing protocols at AS = Autonomous Systems level are your friend, such as **BGP = Border Gateway Protocol**. For any *serious* cross-datacenter scenarios and/or geo-redundancy, it is a **must**. If you don't have the ability for **dynamic routing at the appropriate granularity**, you should better not claim that you are geo-redundant. If handover / failover takes far longer than acceptable by customer expectations / SLAs (typically minutes), you are *not really* geo-redundant from the viewpoint of your customers.

1. (inflexible) Manual or semi-automated routing at datacenter uplink level. Here the customer traffic is always routed to the *same* IP visible from outside, while there is a *separate* static IP per server for sysadmin `ssh` access. The customer traffic routing needs to be changed *globally* for the *complete* traffic to *any* of two datacenters, and thus is very inflexible. This model protects *only* against a full datacenter loss, but almost nothing else.

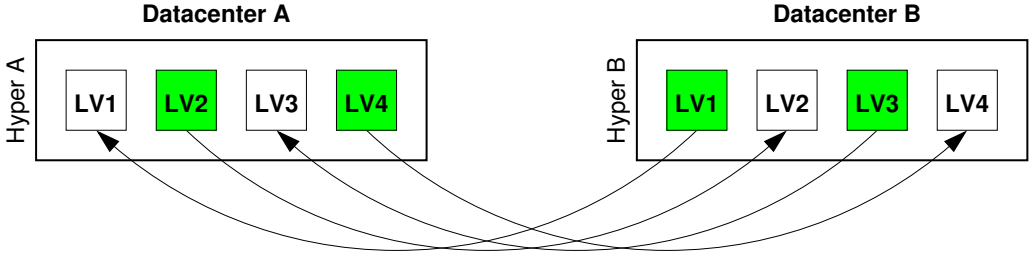
[11] In the table, "Backup" means that only the data is replicated into a different datacenter. In difference, "Replication" means that both the data and the necessary compute resources are available in two datacenters. See also sections What is Backup and What is Replication.

Unfortunately, this model appears very simple to implement, so both staff and chief executive managers are sometimes preferring this "simple" model, although it causes headaches at operational level when really needed.

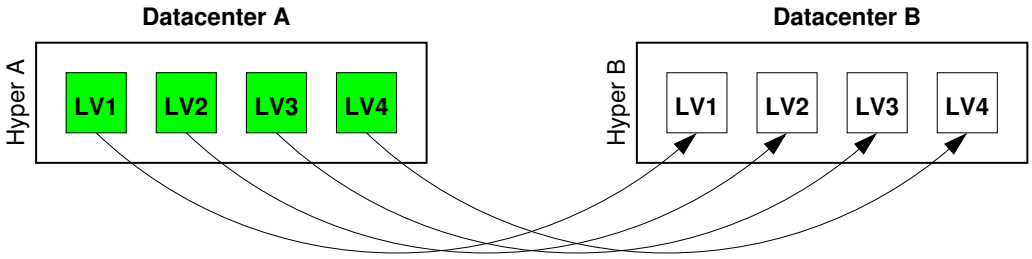
2. (medium flexibility) Dynamic routing of customer traffic at the granularity of building blocks, or even per hypervisor / physical server. When automated appropriately, switchover is a matter of minutes, or even seconds.

> Manager Hint 3.14: **Requirements for networking**
>
> Starting with this level of flexibility, **BGP** = Border Gateway Protocol or similar network protocols are a **must**.
>
> 
>
> Anyway, when you have the effort of BGP implementation for this level, consider to **do it right from scratch**. Also support the following better levels from the network side of the company.

3. (flexible) Dynamic routing of each VM / LV / resource, individually. This has massive advantages: in case of overload, DDOS attacks, etc, you can quickly load-balance into a so-called **butterfly runtime configuration**: half of your VMs belonging to the same hypervisor is running in datacenter A, while the other half is running in datacenter B. In the following illustration showing one hypervisor per datacenter, green color denotes the active (primary) side, while white means passive (secondary):



During butterfly, each of your hypervisor iron has to carry only *half* of the ordinary workload. For comparison, here is the normal situation where only datacenter A would be active:



In the above butterfly configuration, you have essentially **doubled the available CPU and RAM power**, when compared to the ordinary situation where side B does not carry any application workload. This is a *tremendous* aid for **survival** of certain types of incidents, such as (unhandled[12]) DDOS attacks.

4. (most flexible) In addition to dynamic routing at VM level, the VMs *themselves* are **location transparent** (see section What is Location Transparency). They may transparently migrate to another hypervisor, possibly residing in another building block, or even residing in a different datacenter. In its most general form, the number of replicas may be different for each VM, and may change dynamically, adapting to any needs.

[12] There is no 100% DDOS protection. Attackers are continuosly improving their methods. Catching all types of novel patterns is not possible in general.

Manager Hint 3.15: **Recommended flexibility**

The **ability for butterfly** is relevant at CTO level. It is a massive **risk reducer**, even at company and at stock exchange value level.
In order to really get it implemented in its best form, CTOs should clearly require

Location Transparency at Application Level

It means that not only your servers, but also your **services** can run in any of more than 1 datacenter, without notice by your customers.

The location of your services is no longer a primary key, but a dependent runtime attribute which may change at runtime. Of course, your databases, your dashboards, your monitoring, and other surrounding tools, must also be able to properly deal with location transparency.

Example 3.4: **Ability for butterfly**

1&1 Ionos ShaHoLin = Shared Hosting Linux has implemented the ability for butterfly via BGP location transparency on thousands of servers, and on several petabytes of data. See Positive Example: ShaHoLin storage + application stack.

4. Architectures of Cloud Storage / Software Defined Storage

This chapter compares several *architectural* alternatives with each other. In order to not get lost in the jungle of numerous implementations and their features, the description focuses on *architecture* (see section What is Architecture) wherever possible. Nevertheless, principal behaviour of implementations are also discussed.

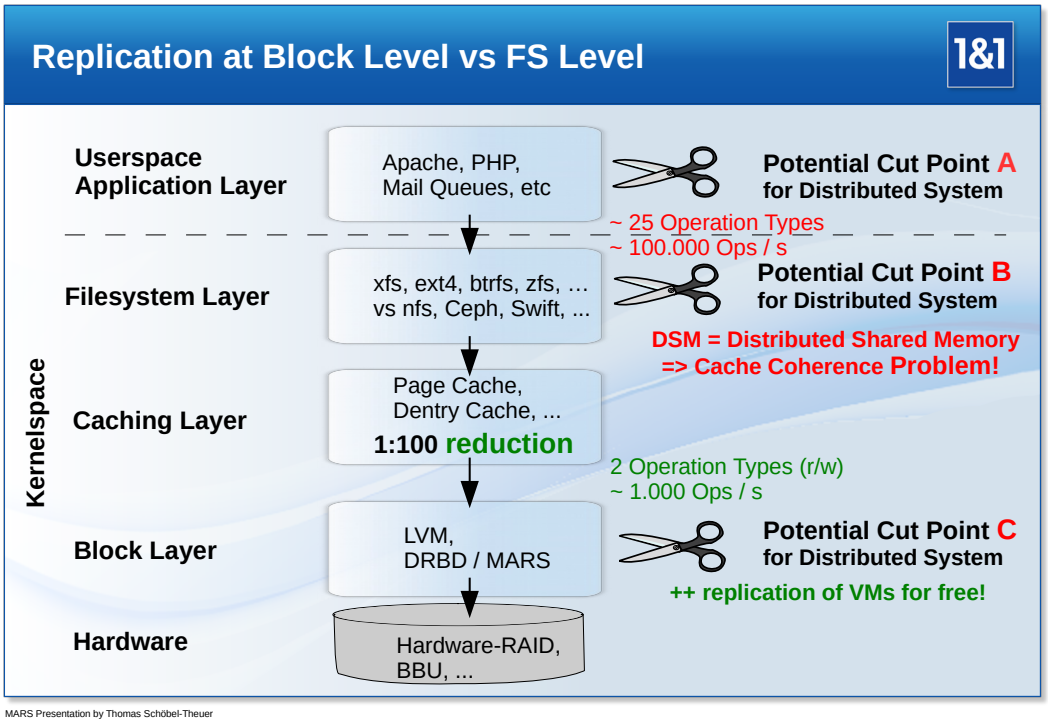
4.1. Performance Arguments from Architecture

4.1.1. Performance Penalties by Choice of Replication Layer

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

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

Choosing the wrong **layer** (see section Layering Rules and their Importance) for **mass data replication** may get you into trouble. Layer selection is much more important than any load distribution argument as frequently heard from certain advocates. Here is an architectural-level (cf section What is Architecture) explanation why replication at the block layer is more easy and less error prone:



The picture shows the main components of a standalone Unix / Linux system. It conforms to Dijkstra's layering rules explained in section Layering Rules and their Importance.

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

All these caches serve one main purpose[1]: they are reducing the load onto the storage by exploitation of fast RAM. A well-tuned cache can yield high cache hit ratios, typically 99%. In some cases (as observed in practice) even more than 99.9%.

Now start distributing the system over long distances. There are potential cut points A and B and C[2].

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

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

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

Manager Hint 4.1:

Caching can yield several *orders of magnitude* of performance.

In contrast, frequently heard load distribution arguments can only re-distribute the already existing performance of your spindles, but cannot magically "create" new sources of performance out of thin air.

In contrary, load distribution over a storage network is *costing* some performance, by introduction of additional latencies and potential bottlenecks.

When replicating at C, the Linux caches are *above* your cut point. Thus you will receive much less traffic at C, typically already reduced by a factor of 100, or even more. This is much more easy to cope with. *Local* caches and their SMP scaling properties can be implemented much more efficiently than distributed ones. You will also profit from **journalling filesystems** like `ext4` or `xfs`. In contrast, *truly distributed*[3] journalling is typically not available with distributed cluster filesystems.

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

This limitation isn't necessarily caused by the choice of layer. It is simply caused by the **laws of physics**: communication is always limited by the speed of light. A distributed filesystem is essentially nothing else but a persistent **DSM = Distributed Shared Memory**.

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

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

Manager Hint 4.2:

If you want mixed operations at different locations in parallel: split your data set into disjoint filesystem instances (or database / VM instances, etc). Then you should achieve the **ability for butterfly**, see section Flexibility of Handover / Failover Granularities.

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

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

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

All you need is careful thought about the *appropriate granularity* of your data sets (such as well-chosen *sets* of user homedirectory subtrees, or database sets logically belonging together, etc). An example hierarchy of granularities is described in section Positive Example: ShaHoLin storage + application stack. Further hints can be found in sections Granularity at Architecture and Variants of Sharding.

Sharding (see section Definition of Sharding) implementations like ShaHoLin (see section Positive Example: ShaHoLin storage + application stack) are essentially exploiting the scalability of SMP = Symmetric MultiProcessing, nowadays typically going into saturation around ≈ 100 hardware CPU threads for typical workloads, which is executed by *hardware* inside of your server enclosure. In contrast, DSM-like solutions are trying to distribute your application workload over longer distances, involving relatively slow system software instead of **hardware acceleration**. Therefore, SMP is preferable over DSM wherever possible.

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

Manager Hint 4.3: **Conclusion**

Active-passive operation over long distances (such as between continents) at **block layer** is an ***advantage***. It keeps your staff from trying bad / almost impossible things, like DSM = Distributed Shared Memory over long distances.

4.1.2. Performance Tradeoffs from Load Distribution

A frequent argument from BigCluster advocates is that random repliction would provide better performance. This argument isn't wrong, but it does not hit the point.

As analysed in section Similarities and Differences to Copysets, load distribution isn't a unique concept bound to BigCluster / random replication. Load distribution has been used since decades at a variety of **RAID striping** methods.

RAID striping levels like RAID-0 or RAID-10 or RAID-60 are known since decades, forming a mature technology. Also known since the 1980s is that the size of a single striped RAID set must not grow too big, otherwise reliability will suffer too much. Larger RAID systems are therefore **split** into multiple **RAID sets**.

This has some intresting parallels to the BigCluster reliability problems analyzed in section Detailed Explanation of `BigCluster` Reliability, and some workarounds, e.g. as discussed in section Similarities and Differences to Copysets.

Summary: both RAID striping and random replication methods are **limited** by the fundamental law of storage systems, see section Optimum Reliability from Architecture, in a similar way.

A detailed performane comparison at architcture level between random replication of variable-sized objects and striping of block-level sectors is beyond the scope of this architecture guide. However, the following should be be intuitively clear from section Layering Rules and their Importance and from Einstein's laws of the speed of light:

> Fine-grained load distribution over **short distances** and/or at **lower layers** has a **bigger performance potential** than over longer distances and/or at higher layers.

In other words: local SAS busses are capable of realtime IO transfers over very short distances (enclosure-to-enclosure), while an expensive IP storage network isn't realtime (due to packet loss). SAS busses are *constructed* for dealing with requirements arising from RAID, and have been optimized for years / decades.

Manager Hint 4.4: **Advice for performance-critical workloads**

Besides *local* SSDs, also consider some appropriate RAID striping at your (Lo-

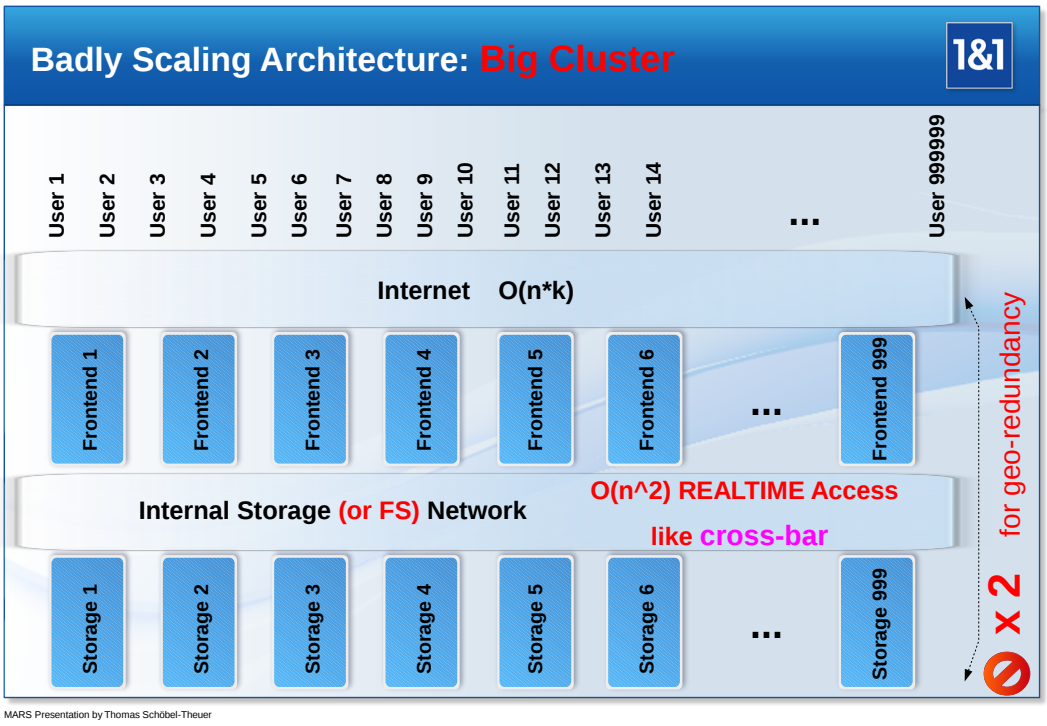
cal)Sharding storage boxes for performance-critical workloads. It is not only cheaper than BigCluster load distribution methods, but typically also more performant (on top of comparable technology and comparable dimensioning). Tradeoffs of various parameters and measurement methods for system architects are described at http://blkreplay.org.

RAID-6 is much cheaper[4] than RAID-10, and can also provide some striping with respect to (random) reads. However, random writes are much slower. For read-intensive workloads, the striping behaviour of RAID-6 is often sufficient. A tool for comparsion of different RAID setup alternatives can be found at http://www.blkreplay.org.

4.2. Distributed vs Local: Scalability Arguments from Architecture

Datacenters aren't usually operated for fun or for hobby. Scalability of an *architecture* (cf section 2.1) is very important, because it can seriously limit your business. Overcoming architectural ill-designs can grow extremely cumbersome and costly.

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



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

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

[4] Several OSDs are also using SAS or similar local IO busses, in order to drive a high number of spindles. Essentially, random replication is involving *two* different types of networks at the same time. This also explains why such a combination must necessarily induce some performance loss.

Manager Hint 4.5:

The $O(n^2)$ **cross-bar functionality** in **realtime** makes the storage network complicated and **expensive**, while decreasing grand-total reliability and thus **increasing risk**.

Details 4.1:

Some further factors are increasing the cost of storage networks:

- In order to limit error propagation from other networks, the storage network is often built as a *physically separate = dedicated* network.
- Because storage networks are heavily reacting to high latencies and packet loss, they often need to be dimensioned for the **worst case** (load peaks, packet storms, etc), needing one of the best = typically most expensive components for reducing latency and increasing throughput. Dimensioning to the worst case instead of an average case plus some safety margins is nothing but an expensive **overdimensioning / over-engineering**.
- When **multipathing** is required for improving fault tolerance of the storage network itself, (parts of) these efforts may easily *double*.
- When **geo-redundancy** is required (see section What is Geo-Redundancy), the total effort may easily double another time because in cases of disasters like terrorist attacks the backup datacenter must be prepared for taking over for multiple days or weeks.


In general, storage networks won't work over long distances. Even it would be possible, **asymmetry problems** would be introduced into an architecture which is *conceptually symmetric* by its very nature. Thus, and generally in $n : m$ relationships, failover granularities are tending to **stick to coarse**. Finer granularites as discussed in section Flexibility of Handover / Failover Granularities are much more difficult to achieve.

Fortunately, there is an alternative called "**Sharding Architecture**" or "**Shared-nothing Architecture**".

Definition of Sharding Notice that the term "Sharding" originates from database architecture https://en.wikipedia.org/wiki/Shard_(database_architecture) where it has a slightly different meaning than used here. Our usage of the term "sharding" reflects slightly different situations in some webhosting companies[5], and can be certainly transferred to some more application areas. Our more specific use of the term "sharding" has the following properties, *all at the same time:*

1. User / customer data is **partitioned**. This is very similar to database sharding. However, the original database term also allows *some* data to remain unpartitioned. In webhosting, suchalike may exists also, but typically only for *system data,* like OS images, including large parts of their configuration data. Suchalike system data is typically *replicated* from a central "golden image" in an *offline* fashion, e.g. via regular `rsync` cron jobs, etc. Typically, it comprises only of few gigabytes per instance and is mostly read-only with a slow change rate, while total customer data is typically in the range of some petabytes with a higher total change rate.

2. The system has (almost[6]) **no single point of contention**, and thus the partitions are **completely independent** from each other, like in **shared-nothing** architectures

[5] According to https://en.wikipedia.org/wiki/Shared-nothing_architecture, Google also uses the term "sharding" for a particular "shared-nothing architecture". Although our above definition of "sharding" does not fully comply with its original meaning, a similar usage by Google probably means that our usage of the term is not completely uncommon.

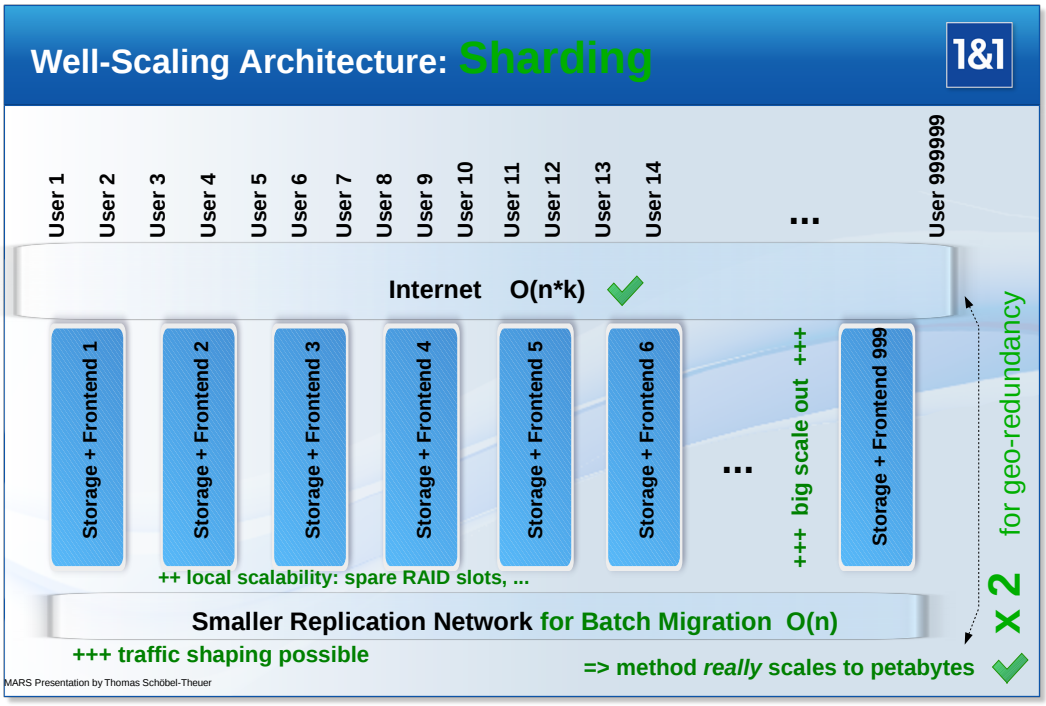
[6] In general, there are some more natural single points of contention, such as the physical space of a datacenter, which might be destroyed by an explosion, for example.

https://en.wikipedia.org/wiki/Shared-nothing_architecture. However, the original term "shared-nothing" has also been used for describing *replicas*, e.g. DRBD mirrors. In our context of "sharding", the shared-nothing principle *only* refers to the "**no single point of contention**" principle at *partitioning* level, which means it *only* refers to to the *partitioning* of the user data, but *not* to their replicas.

3. Shared-nothing replicas (e.g. in the sense of some DRBD descriptions) may be also present (and in fact they are at 1&1 Shared Hosting Linux), but these **replicas** are considered **orthogonal to sharding**. Customer data replicas form an *independent* dimension called "replication layer". The replication layer also obeys the shared-nothing principle in original sense, but it is *not* meant by our term "sharding" in order to avoid confusion[7] between these two independent dimensions.

Conceptual separation of replication from sharding has some advantages. For example, control over the replication degree k can be more fine-grained than at physical shard level. For example, both DRBD and MARS are supporting this, by allowing a different number of replicas for each logical resource.

Our sharding model does not need a dedicated storage network in general, at least when built and dimensioned properly. Instead, it *should have* (but not always needs) a so-called **replication network** which can, when present, be dimensioned much smaller because it does neither need realtime operations nor scalabiliy to $O(n^2)$:



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

Even in cases when any customer may potentially access any of the data items residing in the whole storage pool (e.g. like in a search engine), sharding can be often applied. The trick

[7] Notice that typically `BigCluster` architectures are also abstracting away their replicas when talking about their architecture.

is to create some relatively simple content-based dynamic switching or redirect mechanism in the input network traffic, similar to HTTP load balancers or redirectors.

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

Manager Hint 4.6:

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

Another good explanation can be found at http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/.

4.2.1. Variants of Sharding

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

Example 4.1: **Dimensioning of 1&1 Shared Hosting Linux (ShaHoLin)**

We have dimensioned several variants of this.

1. We are using 1U pizza boxes with local hardware RAID controllers with fast hardware BBU cache and ~ 10 local disks for the majority of LXC container instances where the "small-sized" customers (up to ~100 GB webspace per customer) are residing. Since most customers have very small home directories with extremely many but small files, this is a very cost-efficient model.
2. Less that 1 permille of all customers have > 250 GB (up to 2TB) per home directory. For these few customers we are using another dimensioning variant of the same architecture: 4U servers with 48 high-capacity spindles on 3 RAID sets, delivering a total PV capacity of ~300 TB, which are then cut down to ~10 LXC containers of ~30 TB each.
3. (currently in planning stage) An intermediate dimensioning between both extremes could save some more cost, and hopefully improve reliability even more, due to better pre-distribution of customer behaviour. The so-called midclass could be dimensioned as 90 TB per 2U pizza box, roughly on 12 spindles. It would carry the customers between ~50 and ~250 GB webspace each.

In order to operate this model at a bigger scale, you should consider the "container football" method as described in section 4.2.3 and in `football-user-manual.pdf`.

RemoteSharding This variant needs a (possibly dedicated) storage network, which is however only $O(n)$ in total. Each storage server exports a block device over iSCSI (or over another transport like MARS' prosumer device) to at most $O(k)$ dedicated compute nodes where k is some **constant**.

Details 4.2: **Hint 1**

It is advisable to build this type of storage network with **local switches** and no routers inbetween, in order to avoid $O(n^2)$-style network architectures and traffic. This reduces error propagation upon network failures. Keep the storage and the compute nodes locally close to each other, e.g. in the same datacenter room, or even in the same rack.

Details 4.3: **Hint 2**

Additionally, you can provide some (low-dimensioned) backbone for **exceptional(!)** cross-traffic between the local storage switches. Don't plan to use any realtime cross-traffic *regularly*, but only for clear cases of emergency!

In this model, a shard typically consists of one storage node plus $k+1$ or $k+2$ compute servers, introducing some additional failure redundancy *within* such a shard, while retaining the "no single point of contention" property *between* the shards (according to section Definition of Sharding).

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

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

Details 4.4: **Some use cases for BigClusterSharding**

This could make sense in the following use cases:

- When you **already have** invested into a big cluster, e.g. Ceph or Swift, which does not really scale and/or does not really deliver the expected reliability. Some possible reasons for this are explained in section Reliability Arguments from Architecture and subsection Explanations from DSM and WorkingSet Theory.
- When you really need a *single* LV which is necessarily **bigger** than can be reasonably built on top of local LVM. This means, you are likely claiming that you really need **strict consistency** as provided by a block device on more than 1 PB with current technology (2018). Examples are very **big enterprise databases** like classical SAP (c.f. section 4.6), or if you really need **POSIX-compliance** on a single big filesystem instance. Be conscious when you think this is the only solution to your problem. Double-check or triple-check whether there is *really* no other solution than creating such a huge block device and/or such a huge filesystem instance. Such huge SPOFs are tending to create similar problems[a] as described in section 4.3 for similar reasons.

[a] Running `fsck` or its Windows equivalents on huge filesystems is certainly no fun.

Details 4.5:

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

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

data is. Typically, this is all else but easy on a BigCluster model, even when stripped down to SmallCluster size.

4.2.2. FlexibleSharding

Notice that MARS' new prosumer device feature (formerly called *remote device*, like a kind of replacement for iSCSI) can not only be used for a `RemoteSharding` model, but *could* also be used for implementing some sort of "big cluster" model at block layer. However, consider the warnings for certain use cases from section Explanations from DSM and WorkingSet Theory. If you deserve a very similar level of flexibility as promised by `BigCluster`, read on.

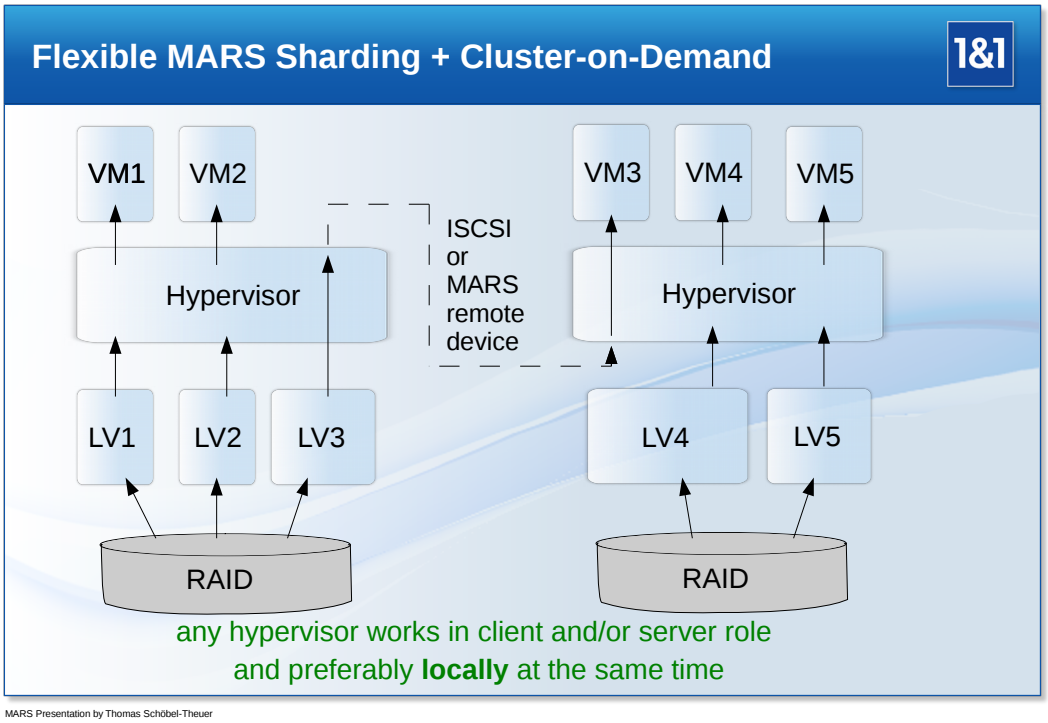
Models re-introducing some kind of $O(n^2)$ "big dedicated storage network", considering the *potential* connections, and $O(n)$ considering the *actual* realtime connections during runtime, are **not** the preferred model for MARS operations in large scale. Following is a compromize.

The basic idea is that each server *can* (as far as necessary) operate *both* in server *and* in client role, both at the same time, and individually for each resource.

Manager Hint 4.7:

Following is a **super-model** which combines both the "big cluster" and sharding models at block layer in a very flexible way, without fully depending on $O(n)$ realtime network connections. The result is a similar flexibility than promised by BigCluster.

The following example shows only two servers from a pool consisting of hundreds or thousands of servers:



The main difference to `BigCluster` is to use iSCSI or the MARS prosumer device *only where necessary*. Preferably, local storage is divided into multiple Logical Volumes (LVs) via LVM, which should be *directly* used *locally* by Virtual Machines (VMs), whenever possible. At abstract

architectual level, detail technologies KVM/qemu vs filesystem-based local LXC containers make no real difference[8].

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

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

Only in cases of resource imbalance, such as (transient) CPU or RAM peaks (e.g. caused by DDOS attacks), and only when the **ability for butterfly** (see section Flexibility of Handover / Failover Granularities) is not available[9] or is not sufficienct, only then the following **fallback strategy** is used: *Some* VMs or containers may then be run somewhere else over the network. In a well-balanced and well-dimensioned system, this will be the **vast minority**, and should be only used for dealing with timely load peaks, unforeseeable customer behaviour, etc.

Manager Hint 4.8:

Running (geo-)redundant VMs directly on the same servers as their storage devices is a major cost reducer.

You simply don't need to buy and operate $2 \cdot (n+m)$ servers, but only about $2 \cdot (\max(n,m) + m \cdot \epsilon)$ servers, where ϵ corresponds to some relative small extra resources needed by MARS.

In addition, **shared memory** can be exploited more efficiently.

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

4.2.3. Principle of Background Migration

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

- Moving per-customer data, typically at filesystem or database level via `rsync` or `mysqldump` or similar.

  Example 4.2: **Fine-grained migration of customer home directories**

  At 1&1 Shared Hosting Linux, we have about 9 millions of customer home directories. We also have a script `movespace.pl` using incremental `tar` or `rsync` for their moves. Now, if we would try to move around *all* of them this way, it could easily take years or even decades for millions of extremely small home directories, due to overhead like DNS updates etc. However, there exist a small handful of large customer home directories in the terabyte range. For these, and only for these, it is a clever idea to use `movespace.pl` because thereby the size of a LV can be regulated more fine grained than at LV level.

- Dynamically growing the sizes of LVs during operations.

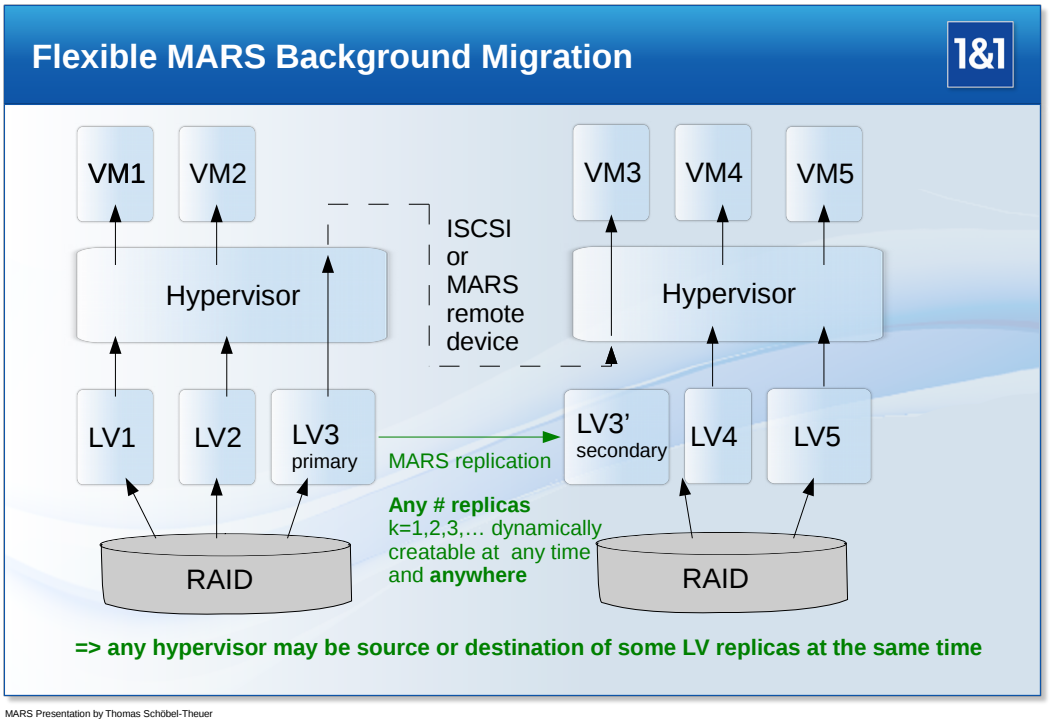
[8] A way for abstracting many details between KVM and LXC is for example provided by `libvirt`.

[9] This may happen when a disaster has already destroyed one of your datacenters, and thus you are forced to run in the surviving datacenter.

Example 4.3: Medium-grained extension of LVs

Football's `expand` operation roughly does the following: `lvresize` followed by `marsadm resize` followed by `xfs_growfs` or some equivalent filesystem-specific operation.

- Moving whole LVs via MARS + Football, as shown in the following example:



The idea of Football's `migrate` operation is to dynamically create *additional* LV replicas for the sake of **background migration**.

Example 4.4: using MARS as replication engine

- If not yet done, you should transparently introduce MARS[a] into your LVM-based stack. If you don't want more than $k = 1$ replicas in general, you can use the so-called "standalone mode" of MARS.
- Optionally: once you have MARS in place, you may use iSCSI or the MARS prosumer device or another means for exporting `/dev/mars/lv3` to another hypervisor. This might be the same hypervisor you want to migrate the data to, or it could be another machine. This is not generally needed, but it help for achieving a similar elasticity than promised by `BigCluster`.
- Now, for the sake of migration, you just create an additional replica at your target server via `marsadm join-resource`. Optionally, this may be the same server where the remote VM is already running at the moment. Wait until the additional mirror has been fully **synced** in background, while your application is continuously running and while the content of the LV is modified *in parallel* by your ordinary applications running inside the VM.
- Then you do a **primary handover** to your mirror (or to *any* of multiple mirrors). This is usually a matter of seconds. Newer versions of the prosumer device will allow this without shutdown of your VM. With standard[b] iSCSI, you will typically have to shortly shutdown the VM and to restart it a few seconds later.

- Once the application is running again at the old location or at another location, you may delete the old replica via `marsadm leave-resource` and `lvremove`.
- Finally, you may re-use the freed-up space for something else (e.g. `lvresize` of *another* LV followed by `marsadm resize` followed by `xfs_growfs` or similar). Or, you may later migrate *another* (smaller) LV to this server, in order to re-use of the free space, or similar.
- For the sake of **hardware lifecycle**, you may run a slightly different strategy: evacuate the original source server completely via Football, and eventually decommission it.
- In case you already have a redundant LV copy somewhere else, you may run a similar procedure, but starting with $k = 2$ replicas, and temporarily increasing the number of replicas to either $k' = 3$ when moving each replica step-by-step, or you may even directly go up to $k' = 4$ in one step, thereby moving *pairs* at once. Example: the latter variant is the default in the ShaHoLin configuration variant of Football, internally called Tetris.
  Technical details: see `football.sh` in the `football/` directory of MARS, which is a checkout of the Football sub-project, and `football-user-manual.pdf`.
- When already starting with $k \geq 3$ LV replicas in the starting position, you may have the luxury of using a lesser variant. For example, we have some mission-critical servers at 1&1 Ionos which are running $k = 4$ replicas all the time on relatively small but important LVs for extremely increased safety. Only in such a case, you may have the freedom to temporarily decrease from $k = 4$ to $k' = 3$ and then going up to $k'' = 4$ again, before starting primary handover. This has the advantage of requiring less temporary storage space for *swapping* some LV replicas.

[a] When necessary, create the first MARS replica with `marsadm create-resource` on your already-existing LV data, which will be retained unmodified, and restart your application again.

[b] There are some iSCSI features like ALUA which *should* be able to handover an active session to another storage box without interruption. However, the corresponding Linux documentation looks very sparse, and the maturity status for Linux initiators / targets is unclear at the moment.

4.3. Reliability Arguments from Architecture

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

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

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

Stimulated by practical experiences from truly less disastrous scenarios than mass power outage, theoretical explanations were sought. Surprisingly, they clearly show by mathematical arguments that `LocalSharding` is superior to `BigCluster` under practically important preconditions.

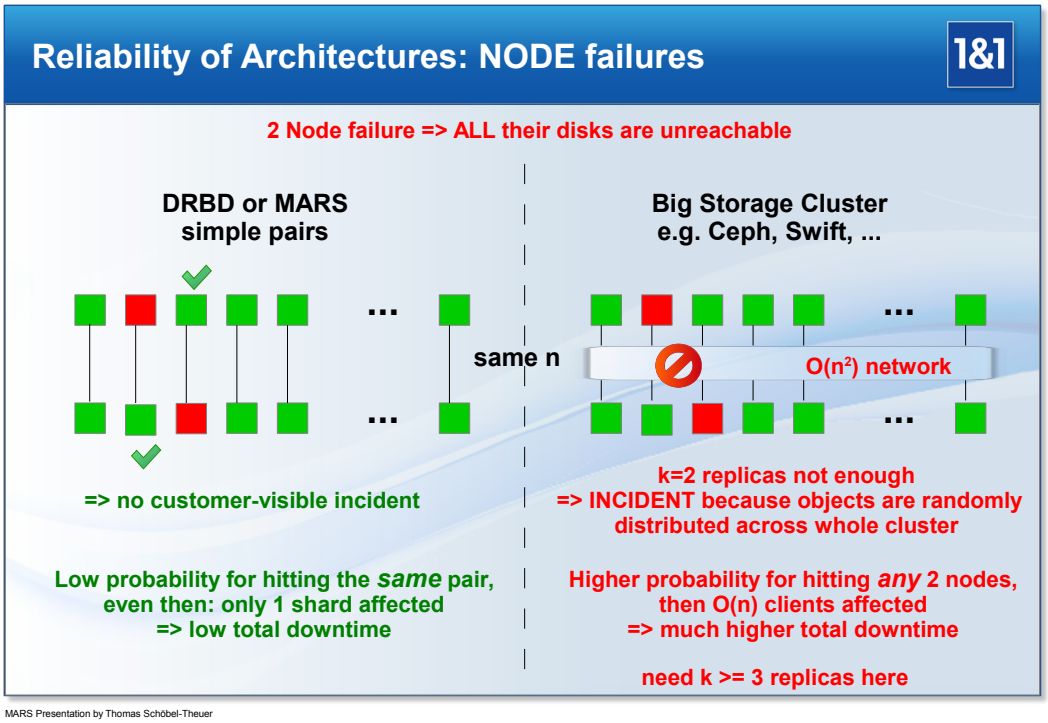
We start with an intutitive explanation. A detailed mathematical description of the model can be found in appendix A on page 130.

4.3.1. Storage Server Node Failures

4.3.1.1. Simple Intuitive Explanation in a Nutshell

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

In contrast, the **random replication** concept of big clusters is spreading huge masses of objects over a huge number of nodes $O(n)$, with some redundancy degree k denoting the number of object replicas. As a consequence, *any* k node failures out of $O(n)$ will make *some* objects inaccessible, and thus produce an incident. For example, when $k = 2$ and n is equal for both models, then *any* combination to two node failures occurring at the same time will lead to an incident:



Manager Hint 4.9:

Intuitively, it is easy to see that hitting both members of the *same* sharding pair at the same time is less likely than hitting *any* two nodes of a big cluster. Therefore, **sharding provides better reliability**, when built on top of comparable technology.

In addition: even when 1 shard out of n shards has an incident, the other $n - 1$ shards will continue to run. In contrast, when a `BigCluster` has an incident, *all* application instances are affected, due to *uniform* object distribution.

Manager Hint 4.10:

Another advantage of sharded pairs is **smaller incident size**.

If you are curious about some more details and more concrete behaviour, read on.

4.3.1.2. Detailed Explanation of `BigCluster` Reliability

The following analysis shows up some parallels to the well-known reliability loss caused by RAID striping. The main difference is granularity: variable-sized objects are used in place of fixed-size blocks. Therefore, this section is in reality about a **fundamental property of data distribution / striping**.

It is only formulated in terms of `BigCluster` and random replication for didactic reasons, because in the context of this architecture guide we need to compare with `LocalSharding`.

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

- We are looking at **storage node** failures only. As observed from practice, this is the most important failure granularity for causing incidents.
- Disk failures are regarded as already solved (e.g. by local RAID-6 or by the well-known compensation mechanisms of big clusters). Only in case they don't work, they are mapped to node failures, and are already included in the probability of storage node failures.
- We only look at **data replication** with a redundancy degree of a relatively small k. CRC methods are not modeled across storage nodes, but may be present *internally* at some storage nodes, e.g. RAID-5 or RAID-6 or similar methods, or may be present internally in some hardware devices, like SSDs or HDDs. Notice that *distributed* CRC methods generally involve very high overhead, and won't work in realtime across long distances (geo-redundancy).
- We restrict ourselves to temporary / **transient** failures, without regarding permanent data loss. Otherwise, the following differences between local-storage sharding architectures and big clusters would become even worse. When loosing some physical storage nodes forever in a big cluster, it is typically all else but easy to determine which data of which application instances / customers have been affected, and which will need a restore from backup.
- Storage network failures (parts, or as a whole) are ignored. Otherwise a fair comparison between the architectures would become difficult. If they were taken into account, the advantages of `LocalSharding` would become even bigger.
- We assume that the storage network (when present) forms no bottleneck. Network implementations like TCP/IP versus Infiniband or similar are thus ignored.
- Software failures / bugs are also ignored[10]. We are only comparing *architectures* here, not their various implementations (see What is Architecture).
- The x axis shows the number of basic storage units $n = x$ from an *application* perspective, meaning "usable storage" or "net amount of storage". For simplicitiy of the model, one basic application storage unit equals to the total disk space provided by one physical storage node in the special case of $k = 1$ replicas.

  Stated simply, this means that there is exactly 1 LV = 1 PV per each application unit present at the x axis. So we have a total of exactly x LVs. Of course, you might create a more elaborate model by introduction of some constant $l \geq 1$ for a grand total of $l \cdot x$ LVs on top of $x = n$ PVs, but we don't want to complexify our model unnecessarily.

  Attention! when increasing the number of replicas k, the total number of storage nodes needs to be **increased accordingly**. Typically, you will need to deploy $k \cdot n$ physical storage nodes in order to get n net storage units from a user's perspective.

[10] When assuming that the probability of bugs is increased by increased architectural complexity, a `LocalSharding` model would likely win here also. However, such an assumption is difficult to justify, and might be wrong, depending on many (unknown) factors.

 Attention! k has a strong influence at the **price tag** of any of the competing architectures. You cannot assume an "infinite amount of money". Therefore, only relatively small k are bearable for business cases.

- As already stated, we assume that the number of application instances is linearly scaling with n. For simplicity, we assume that the number of applications running on the whole pool is *exactly* n. Of course, you might also introduce some *coupling constant* here, but don't complexify the model unnecessarily.

- We assume that the storage nodes are (almost completely) filled with data (sectors with RAID, and/or objects with `BigCluster`). Otherwise, the game would be pointless on empty clusters / shards.

- We assume that the number of sectors / objects per storage node is "very large". Some examples: a logical volume of 4 TB has 1,000,000,000 sectors or object, each 4 KB in size. A physical storage node providing 40 TB of storage will then provide 10 billions of sectors / objects.

- For the `BigCluster` architecture, we assume that all objects are always distributed to $O(n)$ nodes. We will later discuss some variants where it is distributed to *less* nodes. This assumption is only for explaining the **principal behaviour of data distribution / striping**, and also for one of its variants called **random replication**. For simplicity of the model, we assume a distribution via a *uniform* hash function. In general, the principal behaviour would also work for many other distribution functions, such as RAID striping, or even certain non-uniform hash functions over $O(n)$ nodes. As discussed later, totally different hash functions (e.g. distributing only to a constant number of nodes) would no longer model a `BigCluster` architecture in our sense.
  In the below example, we assume a uniform object distribution to *exactly* n nodes. Notice that any other $n' = O(n)$ with $n' < n$ will produce similar results for $n' \to \infty$, but may be better in detail for smaller n'.

- When random distribution / random replication methods are used at BigCluster object stores, we assume that for any pair (or k-tuple) of storage nodes, the total number of objects is so high that there always *exists* some objects which are present at *all* of the nodes of any pair / k-tuple for any reasonable (small) k. This means, we assume not only uniformity in random replication, but also that the total number of objects is practically "infinite" compared to relatively small practical values of k.

   For mathematically interested readers: be careful when trying to argue with the probability to hit some object intersection for some given k-tuple of storage nodes while n is a growing parameter. Even when such a *single* probability is declining with growing both k and n, and even when the *single* probability for the existence of an intersection somewhen gets lower than 1, this has an impact onto the *total*[11] incident probability of the *whole* BigCluster. In *general*, the *number* of such tuples is growing with $O(\binom{k \cdot n}{k}) = O((k \cdot n)!)$, which is even worse than an exponential growth. So, don't forget to sum up *all* probabilities even if a single one appears to be "neglectible".

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

The following assumptions are not part of the model, but are simplifying the below *example* graphics. You may choose other parameter values than the following ones, without changing the principal behaviour of the model, but then the *example* would become less intuitive for **humans**.

[11] Mathematical probabilties are always about a huge number of *repetitions* of a certain experiment. Even when a single "failure experiment" does *not always* lead to an incident from a customer's perspective, it can contribute to the overall incident probability, when there is a *chance*, even when the chance is very low.

- For simplicity of the *example*, we assume that any single storage server node used in either architecture, including all of its local disks, has a reliability of 99.99% (four nines). This means, the probability of a storage node failure is uniformly assumed as $p = 0.0001$.
- This means, during an observation period of $T = 10,000$ operation hours, we will have a total downtime of 1 hour per server in statistical average. For simplicity, we assume that the failure probability of a single server does neither depend on previous[12] failures nor on the operating conditions of any other server. It is known that this is not true in general, but otherwise our model would become extremely complex.
- More intuitively, our observation period of $T = 10,000$ operation hours corresponds to about 13 months, or slightly more than a year.
- Consequence: when operating a pool of 10,000 storage servers, then in statistical *average* there will be *almost always* one node which is failed at the moment. The overall behaviour is like a "permanent incident" which has to be solved by the competing storage architectures.
- Hint: the term "statistical average" is somewhat vague here, in order to not confuse readers[13]. A more elaborate statistical model can be found in appendix A on page 130.

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

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

Under the precondition of $k = 1$ replicas, a failure at *any one* of the n servers has two possible ways to influence the downtime from an application's perspective:

1. LocalSharding (DRBDorMARS): downtime of 1 storage node only influences 1 application unit depending on 1 basic storage unit. This is the case with the DRBDorMARS model, because there is no communication between shards, and we assumed that 1 storage server unit also carries exactly 1 application unit.
2. BigCluster: here the downtime of 1 storage node will **tear down more** than 1 application unit, because any of the application units have spread their storage to more than 1 storage node via uniform hashing (see assumptions above).

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

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

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

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

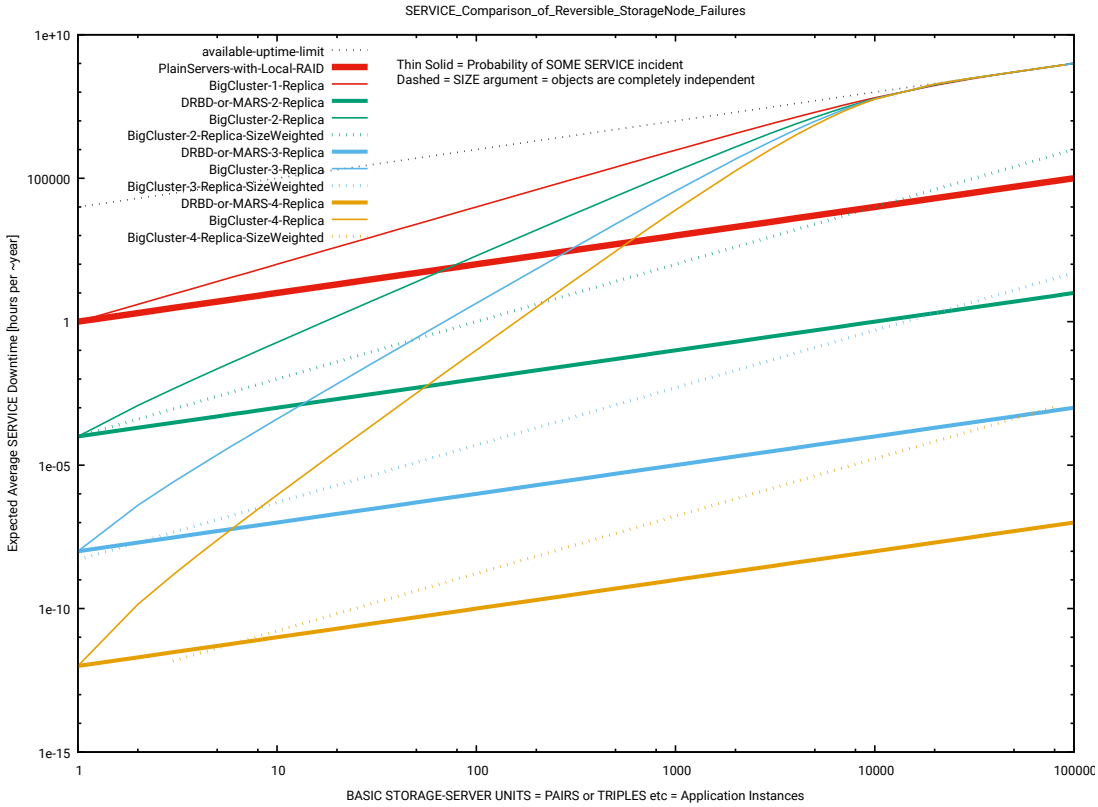
Would it help to increase both n and k to larger values?

Let us first stay at $k = 1$, looking at the behaviour when $n \to \infty$. The generalization to bigger redundancy degrees k will follow later.

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

[12] Mathematically, we are using some Poisson process model here. Of course, it would be possible to use more sophisticated models, but this might turn out as a *major* research undertakement.

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



First, we look at the red lines, corresponding to $k = 1$. The behaviour of the thick red line should be rather clear in double logscale: with increasing number of servers at the x axis, the total downtime y is also increasing. This forms a straight line in double logscale, where the slope is 1 (proportional to n), and the distances between the start of the other colored lines are multiples of $1/p$ for the given incident probability p.

Next, we are looking at the thin solid red line for `BigCluster` $k = 1$. Why is it converging against the dotted grey line around $n = 10000$?

At $n \geq 10000$ servers, there is a "permanent incident". In statistical average, there is approximately *always* some server down. Due to $k = 1$ replica, the whole cluster will then be down from a user's perspective. The thin dotted grey line denotes the total number of operation hours to be executed for each n, so this is the limes line we are converging against for big enough n.

This does not look nice from a user's perspective. Can we heal the problem by deploying more replicas k?

Let us look at the green solid lines, correponding to $k = 2$ replicas. Why is the thin green BigCluster line also converging against the same dotted limes? And why is this happening around the same point, around $n \approx 10000$?

When you want to operate $n = 10000$ application instances with a replication degree of $k = 2$ replicas, then you will need to deploy $k \cdot n = 20000$ storage servers. When you have 20000 storage servers, in statistical average about 2 of them will be down at the same time. When $k = 2$ servers are down at the same time, again the whole cluster will be down from a user's perspective. Thus the green line is also converging against the grey dotted limes line, roughly also around $n \approx 10000$.

Why is the green thicker DRBDorMARS line much better?

In double logscale plot, it forms a *parallel* line to the corresponding red line. The distance is conforming to $1/p$. This means that the incident probability for hitting *both* members of the *same* shard is *improved* by a factor of 10,000.

Finally, we look at all the other solid lines in any color. All the thin solid `BigCluster` lines are converging against the same limes line, regardless of replication degree k, and around the

same $n \approx 10000$. Why is this the case?

Because our BigCluster model as defined above will distribute *all* objects to *all* servers uniformly, there will almost always *exist* some objects for which no replica is available at almost any given point in time. This means, you will almost always have a **permanent incident** involving the same number of nodes as your replication degree k, and in turn *some* of your objects will not be accessible at all. This means, at around $x = 10,000$ application units you will loose almost any advantage from increasing the number of replicas. Adding more replicas will no longer help at $x \geq 10,000$ application units.

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

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

 Under some further preconditions, it would be possible to argue with the *size* of incidents. However, now a big fat warning.

Manager Hint 4.11: Size-weighted incident probabilities

When you are **responsible** for operations of **thousands of servers**, you should be very conscious about preconditions for size-weighted downtime probabilities (dashed lines). Otherwise you could risk both the health of your business, and your career.

Details 4.6: Some preconditions for size-weighted incident probabilities

In short:

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

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

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

Conversely, the assumption that *any* unaccessible object will halt your application, might be too strong for *some* use cases. Therefore, some practical behaviour may be inbetween the solid thin lines and the dashed lines of some given color. Be extremely careful when constructing such an intermediate case. Remember that the plot is in logscale, where constant factors will not make a huge difference. The above example of a loss rate of 1/1,000,000 of sectors in a classical filesystem should not be extended to lower values like 1/1,000,000,000 without knowing exactly how the filesystem works, and how it will react *in detail*. The grey zone between the extreme cases thin solid vs dashed is a **dangerous zone**!

[a] Several distributed filesystems are separating their metadata from application data. Advocates are selling this as an advantage. However, in terms of **reliability** this is clearly a **disadvantage**. It increases the *breakdown surface*. Some distributed filesystems are even *centralizing* their metadata, sometimes via an ordinary database system, creating a SPOF = Single Point Of Failure. In case of inconsistencies between data and metadata, e.g. resulting from an incident or from a software bug, you will need the equivalent of a **distributed `fsck`**. Suchalike can easily turn into **data loss** and other nightmares, such as node failures during the consistency check, for example when your hardware is flaky and produces intermitting errors.

Manager Hint 4.12:

As a manager, if you want to stay at the **safe side**, simply obey the fundamental law as explained in the next section:

4.3.2. Optimum Reliability from Architecture

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

Actually, several BigCluster implementation are doing similar measures, in order to workaround the problems analyzed here. There are various terms for suchalike measures, like copysets, spread factors, buckets, etc.

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

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

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

There exists no better model because shards consisting of exactly k nodes where k is the redundancy degree are already the *smallest possible shards* under the assumptions of section 4.3.1.2. Any other model truly involving $k' > k$ nodes for distribution of objects at any shard is **always** worse in the dimension of reliability. Thus the above sentence follows by induction.

Manager Hint 4.13:

The above sentence is formulating a **fundamental law of storage systems**. An intuitive formulation for humans:

> **Spread your per-application data to as less nodes as possible.**

This includes unnecessary spreading between dedicated client and server machines, in

place of local storage. Thus `LocalSharding` is the best architectural model.

This is intuitive: the more nodes are involved for storing the *same* data belonging to the *same* application instance (i.e. belonging to the same LV), the higher the **risk** that *any* of them can fail.

 Consequence: the ***concept* of random replication**[14] tries to do the *opposite* of this, by its very nature.

Manager Hint 4.14:

Thus the *concept* of **random replication does not work as expected**.

 This does not imply that random replication does not generally work at all. Section Explanations from DSM and WorkingSet Theory mentions a few use cases where it appears to work in practice. However, after **investing a lot** of effort / energy / money into a very complicated architecture and several implementations, the outcome is **worse** = **non-optimal** in the dimension of reliability.

 There exist some *workarounds* as discussed in section Similarities and Differences to Copysets. These can only patch the most urgent architectural problems, such that operation remains *bearable* in practice. They cannot fix the **Dijkstra regression overhead** explained in section Negative Example: object store implementations mis-used as backend for block devices / POSIX filesystems. The above plot explains why even the workarounds are **far from optimal** for a given fixed[15] redundancy degree k.

Manager Hint 4.15: **Summary from a management viewpoint**

 Under comparable conditions for big installations, random replication is requiring **more invest** than Sharding (e.g. more client/server hardware and an $O(n^2)$ realtime storage network), in order to get a ***worse result*** in the **risk dimension**.

4.3.3. Error Propagation to Client Mountpoints

This section deals with a *pathological* setup. Best practice is to avoid such pathologies.

The following is only applicable when **filesystems** or whole **object pools** (buckets) are exported over a storage network, in order to be **mounted** in parallel at $O(n)$ mountpoints *each*.

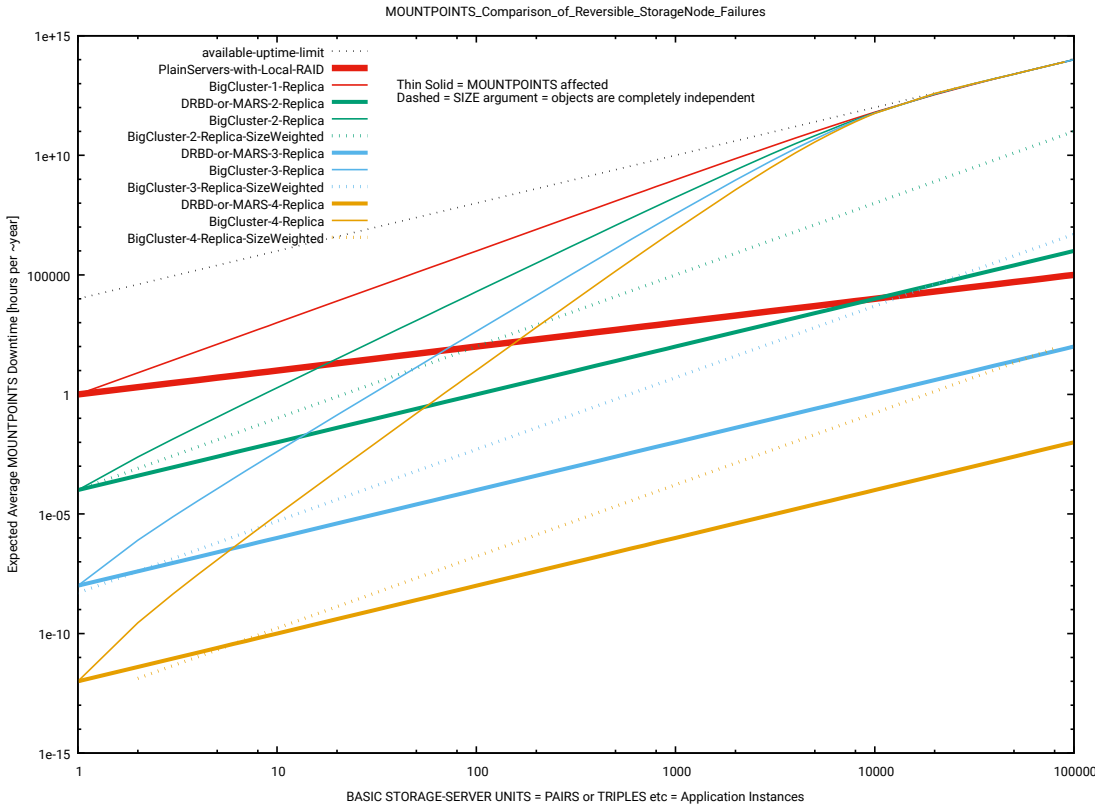
In other words: somebody is trying to make *all* server data available at *all* clients. In spirit, this is also some BigCluster-like **way of thinking**. It just relates to the filesystem layer, c.f. section Performance Arguments from Architecture.

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

[14] A very picky argument might be: random distribution could be viewed as *orthogonal* to random replication, by separating the concept "distribution" from the concept "replication". Then the above sentence should be re-formulated, using "random distribution" instead. However notice than *random* replication + distribution on exactly $n \cdot k$ nodes would degenerate, since it no longer is really "random", but only has the freedom degree of a "permutation".

[15] As explained in section Cost Arguments from Architecture, several `BigCluster` best practices are typically requiring $k = 3$ replicas. Some advocates have taken this as granted. For a **fair comparison** with Sharding, they will need to compare with $k = 3$ LV replicas.

Notice the **slopes** in the following plot. Some are correponding to n^2, and thus are even worse than in the previous plot:



As a result, we now have a total of $O(n^2)$ mountpoints = our new basic application units[16].

The problem is much worse than explained in section Explanations from DSM and WorkingSet Theory, or in Example Failures of Scalability where a disaster already occurred at $n = 6$. Suchalike $O(n^2)$ architectures are simply **hazardous**. Thus a clear warning: don't try to build systems in such a way.

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

Manager Hint 4.16: Clear advice

Don't use $O(n^2)$ mountpoints in total. It's a very bad idea.

4.3.4. Similarities and Differences to Copysets

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

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

[16] If you like, please create another mathematical model in terms of number of clients, instead of the number of mountpoints. Though the plot curves will be different, and certainly will explain an interesting behaviour, the management conclusions will not change too much.

Similarities Both are concluding: the concept of Random Replication of the storage data to large number of machines will reduce reliability. When chosing too big sets of storage machines, then the storage system as a whole will become practically unusable. This is common sense between the USENIX paper and the analysis from section Detailed Explanation of `BigCluster` Reliability.

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

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

This changes the *timely granularity* of data access: while BigCluster is transferring *each* IO request over the storage network in *realtime*, nothing is transferred over an external network at LocalSharding, provided that no migration is necessary. Typically, migrations are a **rare exception**. Normally, the data is already **close to the consumer**. Only in rare situations when migration is needed, local IO transfers are *shifted over* to external migration processes. The outcome of a successful migration is that local IO is then sufficient again.

In essence, Football is an **optimizer for data proximity**: always try to keep the data as close[17] to the consumers as possible.

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

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

  In contrast, MARS uses the client-server paradigm at a different granularity: each machine can act in client role and/or in server role *at the same time*, and *individually* for each LV. Thus it is possible to use local storage.

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

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

- Notice: we do not preclude further fine-grained distribution of LV data at lower levels, such at LVM level and/or below, but this is something which should be *avoided* if not absolutely necessary (see Optimum Reliability from Architecture). Preferred method in typical practical use cases: some storage servers may have some spare RAID slots to be populated later, by resizing the PVs = Physical Volumes before resizing LVs. Another alternative is dynamic runtime extension of SAS busses, by addition of external enclosures.

- Notice that a typical local RAID system *is also* a Distributed System, according to some reasonable definition. Typical RAID implementations just involve SAS cables instead of Ethernet cables or Infiniband cables. Notice that this also applies to many "Commodity

[17] When the many local SAS busses are also viewed as a network, and when these are logically united with the replication network to a bigger *logical* network which is *heterogenous* at physical level: Football does nothing else but trying to **offload** all IO requests to the local SAS networks, instead of overloading the wide-area IP network. In essence, this is a specialized traffic scheduling strategy for a two-level network.

Hardware" approaches, like Ceph storage nodes driving dozens of local HDDs connected over SAS or SATA. The main difference is just that instead of a hardware RAID controller, a hardware HBA = Host Bus Adapter is used instead. Instead of Ethernet switches, SAS multiplexers in backplanes are used. Anyway, this forms a locally distributed sub-system.

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

- The USENIX paper defines its copysets on a per-chunk basis. Similarly to before, we might transfer this definition to a Sharding Approach by relating it to a per-LV basis. As a side effect, a copyset can then trivially become identical to S when the definition is S is also changed to a per-LV basis, analogously. In the Sharding Approach, a distinction is not absolutely necessary, while the USENIX paper has to invest some effort into clarifying the relationship between S and copysets as defined on a BigCluster model.

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

- This means: LocalSharding tries to *minimize* the *size* of S for any given per-LV k, which will lead to the best possible reliability (under the conditions described in section Detailed Explanation of `BigCluster` Reliability) as has been shown in section Optimum Reliability from Architecture.

Another parallel comes to mind: classical RAID striping has introduced the concept of **RAID sets** since decades. Similarly to random replication, RAID striping is motivated by *load distribution*. Similarly to our previous discussion, this induces some **cost**. This is not only about RAID-0 vs RAID-10 by introduction of some more replicas[18]. It is a general problem caused by too high stripe spreading. When a single striped RAID set would grow too big, reliability would suffer too much. Thus multiple smaller RAID sets are traditionally used in place of a single big one[19]. This is somewhat similar to copysets, when taking the spread factor S as analog to the RAID set size, by using objects in place of sector stripes, and a few other differences like using some well-known *stripe distribution function* in place of random replication. Compare with section Optimum Reliability from Architecture: RAID sets are just another example workaround for consequences from the fundamental law of storage systems.

4.3.5. Explanations from DSM and WorkingSet Theory

When looking for practical advice, just read the below example use cases, and skip the rest, which is mostly of academic interest.

This section tries to explain the BigCluster incidents observed at some 1&1 Ionos doughter from a different perspective. In the OS literature and community, DSM = Distributed Shared Memory and Denning's workingset theory from the 1960s are typically attributed to a different research area.

[18] Random replication is be more like RAID-01: first *all* the physical disks are striped, then replicas are created *on top* of it. Reversing this order would be more similar to RAID-10, and could lead to an improvement of random replication. However, this would contradict to a basic idea of BigCluster, that you can add *any* number of storage nodes at any time. Instead of adding an *odd* number of OSDs, each potentially of different size, now an *even* number needs to be added for $k = 2$ replicas, or equal-sized triples for $k = 3$,etc.

[19] Practical example from experience: for RAID-60, a typical RAID-6 sub-set should not exceed 12 to 15 spindles.

Example 4.5: **Example use cases for `BigCluster`**

Personal discussions with some prominent promoters of Ceph found some informal agreements about some use cases where BigCluster appears to be well suited:

- Large collections of audio / video files. These are never modified in place, but written once, and then ***streamed***. Thus it is possible to use relatively large object sizes, or even 1 video file = 1 object. Then streaming involves only a low number of objects at the same time, down to a per-application parallelism degree of typically only 1.
- Measurement data like in CERN physics experiments, where often some *streaming model* is predominant.
- Backups and long-term archives, when also accomplished via *streaming*.

Example 4.6: **Example problems for `BigCluster`**

In contrast to this, here are some other use cases where BigCluster did not meet expectations of some people at 1&1 Ionos:

- Virtual block devices involving **strict consistency** on top of a very high number of small "unreliable" / eventually consistent objects.
- CephFS with **highly parallel random updates** to a huge number of files / inodes, also involving strict consistency in some places (e.g. concurrent metadata updates belonging to the same directory).

Here is a *first attempt* to explain these behavioural observations from a more generalized viewpoint. The author is open for discussion, and will modify this part upon better understanding.

For the following, you will need profound[20] knowledge in Operating System Principles (aka Theory of Operating Systems).

Ceph & co are apparently shining at use cases where the *object paradigm* is naturally well-suited for the *application behaviour*.

Application behaviour has been studied in the 1970s. Theorists know that in general it is *unpredictable* due to Turing Completeness, but practical obervations are revealing some frequent *behavioural patterns*. Otherwise, caching would not be beneficial in practice.

While Denning had studied and modelled application behaviour for typical drum storage devices of his era, later DSM people stumbled over similar problems: the *frequency of access to needed data* can grow much higher than the channel / transport capacities can[21] provide. Denning and Saltzer coined a term for this: **thrashing**.

Thrashing means that more time is spent by *fetching* data than by *working* with it, because the transports are *overloaded*. As Denning observed, thrashing essentially means that the system becomes *unusable by customers*. Thrashing is a highly non-linear **self-amplifying effect**, similar to traffic jams at highways: one it has started, it will worsen itself.

Hint for research 4.1:

Although some historic descriptions of thrashing are mentioning contemporary hardware devices like drum storage, the *concept* is very universal. Thrashing can be transferred and **generalized** to modern instances of **storage pyramids**, and/or also to remote access over **network bottlenecks**.

Saltzer found a workaround for his contemporary batch operating systems: limit the parallelism degree of concurrently running batch jobs. In his Multics project, this was also transferred to interactive systems, by limiting the swap-in parallelism degree of his contemporary segment

[20] In addition to standard Operating System text books like Silberschatz or Tanenbaum, you may need to consult some of the original work of further authors mentioned above.

[21] In general, this is unavoidable. In a **storage pyramid**, the CPU is always able to access RAM pages with a much higher frequency than any (R)DMA transport can supply.

swapping methods. Although this may sound counter-intuitive for modern readers: by introduction of a certain type of **artificial limitation** at or around the non-linear regression point, the **user experience was *improved***.

Now comes a conclusion: when thrashing occurs in a modern BigCluster model for whatever reason, the self-amplification will be likely worse than in a LocalSharding model, due to the following reasons:

- **Overload propagation**: when some parts of the $O(n^2)$ storage network are overloaded, other parts may also become affected in turn, due to sharing of network resources, such as cross-traffic lines. Once queueing has started somewhere, it is likely to worsen, and likely to induce further queueing at other parts of the shared network. The more other parts are affected transitively, the more parts will get overloaded. So the overload, once it has started somewhere, has a higher probabilty for *spreading out* even to parts which were not overloaded before (self-amplification at BigCluster level).

- Random replication of objects adds *artificial randomness* to the ***locality of reference***, as described by Denning.

- Original DSM was trying to provide a strict or near-strict consistency model for application programmers. Later research then tried some weaker consistency models, without getting a final breakthrough for general use cases. BigCluster is similarly organized to DSM, but on slow *remote storage* instead of logically shared remote RAM over fast RDMA. Thus we can expect similar problems as observed by the DSM community, like **single points of contention**, etc. These might become even worse once they have appeared.

In a nutshell: **system stability** under overload conditions, once they have started somewhere, is highly non-linear, and tends to spread[22], and to self-amplify.

In contrast, sharding models are not spreading any overload to other shards by definition. So the total availability from the viewpoint of the *total* set of customers is less vulnerable to impacts.

Manager Hint 4.17: **Risk characterization in a nutshell**

While BigCluster increases the risk of spread-out of overload and other stability problems similarly to a **domino effect**, Sharding is restricting those risks by **fencing**.

In the above use cases where BigCluster is shining, overload is unlikely, since the *parallelism of object access* is limited. This is somewhat similar to Saltzer's historic workaround for trashing. *Streaming* at application behaviour level will translate into streaming at the network layer. Classical TCP networks dealing with a relatively low number of high-throuhput streaming connections are just *constructed* for dealing with packet loss, such as caused by overload, e.g. by their **congestion control**[23] algorithms.

In contrast, an extremely high number of parallel short connections would be similar to a "SYN flood attack", or similar to a classical UDP packet storm. It would allow for a much higher parallelism degree, but will be more vulnerable to packet loss / packet storm effects / etc, and more vulnerable to self-amplification. These application behaviour types are avoided in the above use case examples for BigCluster.

[22] In the past, advocates of BigCluster have placed the argument that BigCluster can *equallay distribute* the total application load onto $O(n)$ storage servers, so a single overloaded client will get better performance than in a sharding model. This argument contains the *implicit assumption* that load distribution is behaving **linearly**, or close to that. However, Denning and Saltzer found that system reaction due to overload by workingset behaviour is *extremely* non-linear, and may *completely* tear down systems even when only *slightly* overloaded. Although there may exist some areas where the assumption of linearity is correct and may lead to improvements by better load distribution, "unpredictable" behaviour due to self-amplification of overload at BigCluster level may result in the *opposite*. Denning has provided a mathematical model for this, which could probably be transferred to modern application behaviour.

[23] Recommended reading: the papers from Sally Floyd.


In addition, storing video files as immutable BLOBs will limit the **randomness** of *locality of references*, while splitting into millions of very small objects may easily lead to an explosion of randomness by some orders of magnitude.

4.4. Scalability Arguments from Architecture

Manager Hint 4.18: Importance of scalability

Scalability is important for **mass data** / mass production. It determines the technical limits of **scaling effects**. Bad scalability can seriously **limit the business**, and its resolvement can produce high **cost**.

Unfortunately, in some circles, seriously wrong habits have established. I know of examples causing unnecessary problems and cost in the range of millions of €.

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

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

This is not only because of confusion of "architecture" with "implementation", cf section What is Architecture. It is also **fundamentally broken** because it assumes some "linearity" in a field which is **non-linear *by definition***.

If scalability would be linear, the term would not be useful at all, because there would be *no limit*. However, limits exist in practice, and the term "scalability" is a ***means* for describing the behaviour at or around the limit**.

Another *incorrect* way of ill-defining / ill-using the term "scalability" is looking at some relatively big ***example*** cluster, which is working in practice for some **particular use case**, and then concluding that it will also work in **another use case**. Arguing with an *example* of a working system is wrong by construction. In general, examples can only be used for ***disproving*** something, but never as a positive proof of concept[24].

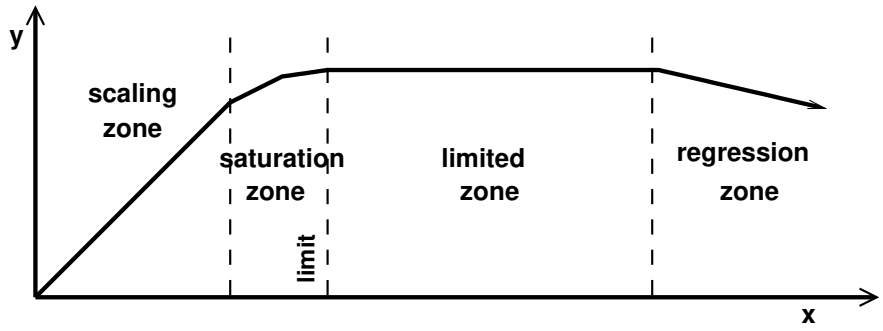
Examples for suchalike examples: section Explanations from DSM and WorkingSet Theory mentions some use cases where `BigCluster` architecture implementations via Ceph are shining. These example use cases are showing some commonalities, like relatively low performance demands at the storage, and relatively low IO parallelism degree[25], and streaming-like **access patterns**. However, it also mentions some *other* use cases, where it did *not* work as expected.


Humans are **selective** in their perception. Evolution has created this, for our protection against overload in the information flood. Unfortunately, looking only at some positive use case examples, while either not knowing or ignoring other counter-examples, can be dangerous.


Every storage system on this globe has ***always* some scalability limit**, somewhere. Even the internet has some limit. Scalability is *always* a **non-linear** behaviour. In order to find the practical limit, you must *reach* it.

[24] Unfortunately, the term PoC = Proof Of Concept is used wrongly in large parts of the industry. It should be termed PoI = Proof of Implementation, or VoI = Validation of Implementation or CoI = Check of Implementation instead. A concept can have multiple implementations, but only *one* of them has been actually checked in most cases.

[25] Example: many people are not aware that Apache is acting like a **fork bomb**. When thousands of Apache processes are running in parallel, a parallelism of several thousands of IO requests *may* occasionally occur during *peaks*, although caches will serve them *most* of the time. Certain storage systems are reacting with **instability**, sometimes even when "hammered" only once with a very short but massive overload peak.



Typically, the overall scalability behaviour can be divided into several *zones*. Only in the scaling zone, some near-linear behaviour can be expected. Next comes the saturation zone, where the effects of inherent system limits are already retarding progress. After exeeding the scalability limit, no further progress is happening. Upon overload, many systems are even reacting with a *regression*.

Any serious study should consciously deal with *all* of these zones, possibly except the regression[26] zone.

 There exists no excuse for omission[27] of the limit. When the limit is **unknown**, then you simply **must not claim a certain scalability**!

 Example use cases are principally insufficient for proving general scalability behaviour, as well as for comparing the scalability of architectures and/or of certain implementations. Examples can only be used for *disproving* scalability.

 Caution: when a particular *implementation* does not work as expected, this does not generally prove that the corresponding concept / architecture does not work at all. There may be **bugs** and other **sources of error** in the particular implementation, which just need to be ***fixed***.

Manager Hint 4.19: Required skill level for architects

 The **suitability of architectures for certain use cases** needs to be checked separately. This is an expert task, requiring high levels of skills and experience.

 Cross-checking by several experts may lead into systematical ill-designs by **information bubbles**. Well-foundation of arguments, well-founded measurements on basis of solid methodology, etc, are much more important than number of votes!

[26] Entering the regression zone might possibly lead to destruction of certain systems, or to other damages. Then it is acceptable to not enter it. It would *honorable* to mention any **risks** associated with suchlike overload behaviour.

[27] Several years ago, I attended a presentation at an OpenSource conference. It was about scalability of a Java programming environment for some SMP machines. The presenter showed some nice graphics, however showing only the scaling zone. He publicly claimed "arbitrary scalability". After the talk, I tried to catch him, in order to ask him under 4 eyes whether he encountered some limit. His very short answer was that there is no limit, completely unwilling to talk with me at all, and quickly leaving the room.

Unfortunately, this "methodology" seemed to have been copied by more and more presenters. The problem is not only wrong claims. The problem is that managers and other decision makers can lose a lot of money when believing suchalike **fake results**.

4.4.1. Example Failures of Scalability

Manager Hint 4.20: **Recommended reading**

The following example is a **must read** not only for **responsibles**, but also for system architects, and also for sysadmins.

The numbers and some details are from my memory, thus it need not be 100% accurate in all places.

It is about an operation environment for a *new* product, which was a proprietary web page editor running under a very complicated variant of a LAMP[28] stack.

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

Setup 1 (NFS) The first setup consisted of $n = 6$ storage servers, each replicated to another datacenter via DRBD. Each server was exporting its filesystems via NFS to about the same number of client servers, where Apache/PHP was supposed to serve the HTTP requests from the customers, which were entering the client cluster via a HTTP load balancer. The load balancer was supposed to spread the HTTP load to the client servers in a **round-robin** fashion.

Details 4.7:

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

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

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

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

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

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

Result: the problem with the "hangs" disappeared.

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

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

[28] LAMP = Linux Apache Mysql PHP

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

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

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

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

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

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

Details 4.8:

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

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

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

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

Details 4.9:

Retrospective explanation: DRBD was definitely *not* the real reason for the critical incident. The replication traffic per shard is so low in average that until today, no replacement by MARS was absolutely necessary[a], although the distance is over 50 km. If you wonder why such low write traffic demands can cause such a big incident: look at the **cache reduction** graphics in section Performance Arguments from Architecture. Today, the "save" buttons of the customers are just triggering some *extra* **writebacks** from the Page Cache of the kernel into the block layer, after some *delay*. These writebacks are not performance critical in reality, because the Page Cache is running them ***asynchronously in background***.

In contrast, distributed filesystems like `NFS` or `ocfs2` or `glusterfs` are not work-

ing asynchronously in many places, but will often schedule their requests *synchronously* into ordinary network queues, which form a **sequential bottleneck**, competing with other high-frequent filesystem operations. In addition, the "save" button triggers masses of metadata / inode updates in a short time, often residing in the same directory. Such a directory may thus form a "global" bottleneck. When suchalike competing **metadata updates** are distributed via a round-robin load balancer, the problem can easily become critical by the **cache coherence problem**. While local filesystems can smoothen such application behaviour via the Dentry Cache plus Inode Cache, which also show some asynchronous writeback behaviour, network filesystems are often unable to deal with this performantly.

Although DRBD has a similar sequential bottleneck at the low-frequency block layer by its write-through strategy into its replica, this does not really matter: all other writebacks from the Page Cache are *also* started asynchronously, and triggered low-frequently, and are occurring after some *delay* (which in turn will smoothen the **spikes** caused by **mass dirtification** of many small files and inodes in a short time as caused by the "save" button), and thus are not really performance critical for this particular use case.

[a] Many sysadmins are running a conservative strategy: never touch a running system...

This is a striking example why careful **selection of granularity level** (filesystem vs block layer, see section Performance Arguments from Architecture) is essential.

This is also a striking example why asynchronous operations can form a huge advantage in certain use cases.

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

4.4.2. Properties of Storage Scalability

4.4.2.1. Influence Factors at Scalability

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

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

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

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

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

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

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

4. The **transformation** of the application workingset behaviour at architectural level, sometimes caused by certain components resp their specific implementation or parameterization. Examples are intermediate virtualization layers, e.g. vmware `*.vmdk` or KVM `*.qcow2` container formats which can completely change the game, not only in extreme cases. Another example is **random distribution** to (or **random replication** inside of) object stores, which can turn some uncomplicated sequential workloads into highly problematic *random IO* workloads. See also section Similarities and Differences to Copysets. Don't overlook such potential pitfalls!

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

6. The **number** of storage **nodes** n. In some architectures, addition of more nodes can make the system *worse* instead of better, c.f. section Reliability Arguments from Architecture.

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

8. The chosen **implementation** of the architecture. Be sure to understand the difference between an *architecture* and an *implementation* of that architecture (section What is Architecture).

9. The size and types / properties of various **caches** at various layers. You need to know the general properties of **inclusive** vs **exclusive** cache architecture. You absolutely need to know what **thrashing** is, and under which conditions it can occur.
   It is advantagous for system architects to know[29] pre-loading strategies, as well as replacement strategies. It is advantageous to know what `LRU` or `MFU` means, what their induced *overhead* is, and how they *really* work on *actual* data, not just on some artificial lab data. You also should know what an **anomaly** is, and how it can be produced not only by `FIFO` strategies, but also by certain types of ill-designed multi-layer caching. Beware: there are places where `FIFO`-like behaviour is almost impossible to avoid, such as networks. All of these is outside the scope of this MARS manual. You should *measure*, when possible, the **overhead** of cache implementations. I know of *examples* where caching is c*ounter-productive*. For example, certain types and implementations of SSD

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

caches are over-hyped. Removing a certain cache will then *improve* the situation. Notice: caches are conceptually based on some type of **associative memory**, which is either very fast but costly when directly implemented in hardware, or it can suffer from tremendous performance penalties when implemented inappropriately in software.

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

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

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

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

4.4.3. Case Study: Example Scalability Scenario

To get an impression what "enterprise critical data" can mean in a concrete example, here are some characteristic numbers from 1&1 Ionos ShaHoLin (Shared Hosting Linux) around spring 2018.

When the whole system would have to be re-constructed from scratch at a green field, the following number from the current implemenation would be *requirered input parameters* for *any* potential solution architecture, such as `CentralStorage` vs `BigCluster` vs `Sharding`:

- Webhosting very close to 24/7/365.
- Overall customer-visible HA target of 99.98%, including WAN outages. Technically, a much better system-only HA target would be possible, but there are also some *external* incident sources like frequent updates of userspace software and a varity of application software libraries, frequent security updates, etc. Although managed by ITIL processes, these sources are outside of the scope of this *system architecture* guide.
- About 9 millions of customer home directories.
- About 10 billions of inodes, with daily incremental backup.
- More than 4 petabytes of *net* data (total `df` filling level) in spring 2018, with a growth rate of 21% per year.
- All of this permanently replicated into a second datacenter.
- In catastrophic failure scenarios, *all* resources must be switchable within a short time.

In order to not bail out too many competing solutions via preconditions, the following is treated as a nice-to-have feature (only for the sake of the following sandbox game, while in reality the sysadmins would vote for a *hard requirement* instead):

- Ability for butterfly, cf section Flexibility of Handover / Failover Granularities.

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

4.4.3.1. Theoretical Solution: `CentralStorage`

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

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

The current number of client LXC containers is about 2600, independently from location. You will have to support growth in number of them. For maintenance, these need to be switchable to a different geo-datacenter at any time (e.g. risk mitigation of power supply maintenance in a datacenter), at least at hypervisor granularity. As explained in sections Flexibility of Handover / Failover Granularities and What is Location Transparency, handover *should be* at per-VM granularity, otherwise you would cause a regression in operability. The number of bare metal servers running the total workload can vary with hardware architecture / hardware lifecycle, and with growth, such as already demonstrated during the course of internal "Efficiency projects". You will need to dimension a dedicated storage network for all of this.

If you find a solution which can do this with current `CentralStorage` technology for the next 5 years, then you will have to ensure that restore from backup[30] can be done in less than 1 day in case of a fatal disaster, see also treatment of `CentralStorage` reliability in section Reliability Differences CentralStorage vs Sharding. Notice that the current self-built backup solution for a total of 15 billions of inodes is based on a sharding model; converting this to some more or less centralized solution would turn out as another challenge.

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

 In case you actually would want to build a RemoteSharding model on top of commercial storage, you need to consider Cost Arguments from Technology.

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

Notice that classical argumentations with ***manpower*** will not work. The current operating team is about 10 persons, with no dedicated storage admin. This relatively small team is not only operating a total of more than 6,000 shared boxes in all datacenters, but also some tenthousands of managed dedicated servers, running essentially the same software stack, with practically fully automated mass deployment. Most of their tasks are related to central software installation, which is then automatically distributed, and to operation / monitoring / troubleshooting of masses of client servers. Storage administration tasks in isolation are costing only a *fraction* of this. Typical claims that `CentralStorage` would require less manpower will not work here. Almost everything which is needed for *mass automation* is already automated.

 Neglecting the tenthousands of managed dedicated servers would be a catastrophic ill-design. Their hardware is already given, by existing customer contracts, some of them decades old. Although it may be possible to modify *some* of these contracts, you simply cannot funda-

[30] Local snapshots, whether LVM or via some COW filesystem, do not count as backups (see section What is Replication). You need a *logical* copy, not a *physical* one, in case your production filesystem instance gets fatally damaged, such that `fsck` won't help anymore.

mentally change *all* the hardware of these customers including their *dedicated* local disks, which was / is their *main selling point*. You cannot simply convert them to a shared `CentralStorage`, even if it would be technically possible, and if it would deliver similar IOPS rates than tenthousands of local spindles (and if you could reach the bundled performance of local SSDs from newer contracts), and even if you would introduce some interesting **storage classes** for all of this. A dedicated server on top of a shared storage is no longer a dedicated one. You would have to migrate these customers to another product, with all of its consequences. Alone for these machines, *most*[31] of the current automation of `LocalStorage` is needed *anyway*, although they are not geo-redundant at current stage.

Conclusion: `CentralStorage` is simply *unrealistic*.

4.4.3.2. Theoretical Solution: `BigCluster`

The main problem of `BigCluster` is **reliability**, as explained intuitively in section Reliability Arguments from Architecture, and graphically in section Detailed Explanation of `BigCluster` Reliability, and mathematically in appendix A on page 130, and as observed in several installations not working as expected. It would be a bad idea to ignore the explanations from section Explanations from DSM and WorkingSet Theory.

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

The total number of servers would need to be roughly *doubled*[32]. Not only their CAPEX, but also the corresponding OPEX (electrical power, rackspace, manpower) would increase. Alone their current electrical power cost, including cooling, is more than the current sysadmin manpower cost. Datacenter operations would also increase. On top, a dedicated storage network and its administration cost would also be needed.

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

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

4.4.3.3. Current Solution: `LocalSharding`, sometimes `RemoteSharding`

Short story: the architecture as well its current implementation works since decades, and is both cheap and robust since geo-redundancy had been added around 2010.

With the advent of Football (see `football-user-manual.pdf`), the `LocalSharding` architecture is raising up on par with the most important management abilities of `CentralStorage` and `BigCluster` / Software Defined Storage.

Pre-configured `RemoteSharding` on top of dedicated Linux-based storage boxes is currently being reduced in favour of the cheaper and more reliable `LocalSharding` combined with Football. The dedicated storage boxes are almost EOL due to their age, and should vanish some day.

There is another story about tenthousands of managed dedicated servers: without the traditional ShaHoLin sharding architecture and all of its automation, including the newest addition called Football, the product "managed dedicated servers" would not be possible in this scale. By definition, the dedicated server product *is* a sharding implementation. Thanks to football, further business opportunities like migration onto virtualized shared hardware (with optional **resource partitioning**) are possible.

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

4.4.4. Scalability of Filesystem Layer vs Block Layer

Following factors are responsible for better architectural (cf section 2.1) scalability of the block layer vs the filesystem layer, with a few exceptions (list may be incomplete):

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

[32] One of the problems of the current Ceph *implementation* is its massive consumption of CPU power and RAM. Even if this would be improved in future, the *architectural* drawbacks will remain.

1. **Granularity** of access: **metadata** is often smaller than the content data it refers to, but access to data is typically not possible without accessing corresponding metadata *first*. When *masses* of metadata are present (e.g. some billions of inodes like in section Case Study: Example Scalability Scenario), and/or when metadata is accessed **more frequently** than the corresponding data (e.g. in stateless designs like Apache), it is likely to become the bottleneck.

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

    By default, the block layer itself has almost[33] no metadata at all. Therefore it has an *inherent advantage* over the filesystem layer in such use cases.

2. **Caching**: shared memory caches in kernelspace (page cache + dentry cache) vs distributed caches over network. See the picture in section Performance Arguments from Architecture.

    There exist *examples* where shared distributed caches do not work at all. Frequently, this has to with strict consistency requirements, and with runtime access patterns. I know of *several* projects which have failed. Another project than mentioned in section Example Failures of Scalability has failed because of violations of POSIX filesystem semantics.

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

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

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

- Databases: these are typically operating on specific container formats, where no frequent *external* metadata access is necessary, and where no sharing of the *container as such* is necessary. Typically, there is no big performance difference between storing them in block devices vs local filesystems (although it could be viewed as a minor Dijkstra regression).

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

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

   Filesystems on top of object stores (see section Granularity at Architecture) are no true intermediate filesystems. They are violating Dijkstra's important layering rules (see section Layering Rules and their Importance) at *several* places. A similar argument holds for block devices on top of object stores. Another layering violation may result from VM container formats like `*.vmdk` or `*.qcow2`, which cannot always be avoided. Be warned that such container formats *themselves* can act as game changers with respect to performance, parallelism degree, reliability, etc. This does not mean that you have to avoid them generally. Layering violations just create an additional *risk*, which need not always materialize, and need not always be fatal. However, be sure to **check their influence**, and don't forget to measure their *workingset* and their *caching behaviour* (which can go both into positive and into negative direction), in order to really *know what you are doing.*

[33] There may be tiny metadata, such as describing the size of the whole block device.

There exist a few cases where a distributed filesystem, sometimes even actually with $O(n^2)$ behaviour according to section Error Propagation to Client Mountpoints, *must* be used, because there exists a *hard requirement* for it. Some examples (list is certainly incomplete):

- HPC = **High Performance Computing** on modern supercomputers, consisting of a high number of n compute nodes, are often requiring access to a shared persistent data pool, where each of the n nodes must be sometimes able to access the same persistent data, sometimes both for reading and writing. Therefore, several supercomputers are using cluster filesystems like Lustre.

   Care must be taken that high-frequency / fine granularity communication over the distributed filesystem and its dedicated storage network does not take place, but instead occurs over the ordinary low-latency communication fabrics each modern supercomputer is relying on. True $O(n^2)$ storage access behaviour should be avoided as far as possible (given by the problem to be solved). When absolutely necessary, location transparency (as possible with cluster filesystems like Lustre) as well as its DSM = Distributed Shared Memory model must be given up, and an **explicit communication model** must be used instead, which allows explicit control over replicas and their communication paths (e.g. propagation in a binary tree fashion), although it results in much more work for the programmers. Only low frequency / coarse granularity transfers of *bulk data* with *high locality* should run over distributed filesystems, preferably in *streaming* mode (c.f. section Explanations from DSM and WorkingSet Theory). The total frequency of metadata access should be low, because metadata consistency may form a bottleneck when updated too frequently. The programmers of the distributed application software need to take care for this.

   Notice that certain supercomputer workloads may be crying for a RemoteSharding or FlexibleSharding storage architecture in place of a BigCluster architecture. However, this is very application specific.

- Student pools at universities, or location-independent workplaces at companies. This is just the usecase where NFS was originally constructed for. Typically, **workstation workloads** are neither performance critical, nor prone to actual $O(n^2)$ behaviour (although the network infrastructure would *allow* for it), because each user has her own home directory which is typically *not shared* with others, and she cannot split herself and sit in front of multiple workstations at the same time. Thus the *local per-workstation* NFS caching strategies have a good chance to hide much of the network latencies, and thus the actual total network workload is typically only $O(n)$.

   This can lead to a dangerous misinterpretation: because it apparently works even for a few thousands of workstations, people conclude *wrongly* that the network filesystem "must be scalable". Some people are then applying their experience to completely different usecases, where much higher metadata traffic by several orders of magnitudes is occurring (such as in webhosting), or even where true $O(n^2)$ runtime behaviour is occuring (see section Example Failures of Scalability).

 In general: when something works for usecase A, this **does *not* prove** that it will also work for another usecase B. See explanations from start of section Scalability Arguments from Architecture.

4.5. Point-in-time Replication via ZFS Snapshots

Some ZFS advocates believe that ZFS snapshots, which were originally designed for backup-like use cases, are also appropriate solutions for achieving geo-redundancy (cf section What is Geo-Redundancy). The basic idea is to run incremental ZFS snapshots in an endless loop, e.g. via some simple scripts, and expediting to another host where the snapshots are then applied

to another ZFS instance. When there is less data to be expedited, loop cycle times can go down to a few seconds. When much data is written at the primary site, loop cycle times will rise up.

The following table tries to explain why geo-redundancy is not as simple to achieve as believed, at least without addition of sophisticated additional means[34]:

OpenSource Component	DRBD	MARS	ZFS
Synchronity (in average)	yes	delay	delay * 1.5
Generic solution	yes	yes	FS-specific
Granularity	LVs	LVs	subvolumes
Built-in snapshots	no	no	yes
Long distances	no	yes	yes
Replication parallelism (per gran.)	1	≥ 2	1
Built-in primary/secondary roles	yes	yes	no
Built-in handover (planned)	mostly	yes	no
Built-in failover (unplanned)	yes	yes	no
Built-in data overflow handling	unnecessary	yes	no, missing
Unnoticed data loss due to overflow	no	no	possible
Split-brain awareness	yes	yes	no
Execute split-brain resolution	yes	yes	no
Protect against illegal data modification	yes	yes	no

The last item means that ZFS by itself does not protect against amok-running applications modifiying the secondary (backup) side in parallel to the replication process (at least not by default). Workarounds may be possible, but are not easy to create and to test for enterprise-critical applications.

 In simple words: the **ability for butterfly** is non-trivial to achieve. It can easily turn into a nightmare, if you would try to establish it on top of larger `zfs` installations. Although termed "replication", it is more similar to "backup".

 Known `zfs` replication setups at sisters of 1&1 Ionos are lacking the butterfly ability, likely due to these difficulties.

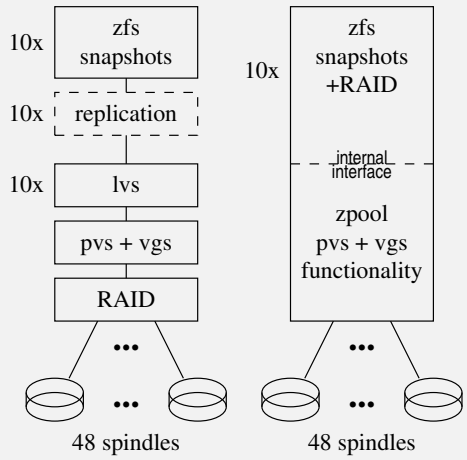
 Notice that `zfs` **snapshots** (without adding replication on top of it) can be **easily combined** with DRBD or MARS replication, because `zfs` snapshots are residing at *filesystem* layer, while DRBD / MARS replicas are located at the lower *block* layer.

Details 4.10: Combination of zfs with MARS

Just create your zpools at the *top* of DRBD or MARS virtual devices, and use `zpool import` / `export` *individually* at handover / failover of each LV. A relatively easy way for implemention is the `systemd` interface of MARS (see the according section in `mars-user-manual.pdf`). You just need to write *once* a small unit template file, containing a few `zpool` commands. This small template will then be automatically instantiated for each resource by the `marsadm` macro processor, as often as needed.

 There is a ***fundamental* difference** between zpools and classical RAID / LVM stacked architectures. Some zfs advocates are propagating zpools as a replacement for both RAID and LVM. However, there is a massive difference in architecture, as illustrated in the following example (10 logical resources over 48 physical spindles), achieving practically the ***same*** *zfs snapshot functionality* from a user's perspective, but in a different way:

[34] ZFS advocates often argue with many features which aren't present at other filesystem types. The above table shows some dimensions not dealing with properties of local filesystems, but with *problems / tasks* arising in long-distance distributed systems involving masses of enterprise-critical storage.



When RAID functionality is executed by zfs, it will be located at the *top* of the hierarchy. On one hand, this easily allows for different RAID levels for each of the 10 different logical resources. On the other hand, this *exposes* the **physical spindle configuration** to the topmost filesystem layer (48 spindles in this example). There is no easy way for replication of these *physical properties* in a larger / heterogenous distributed system, e.g. when some hardware components are replaced over a longer period of time (hardware lifecycle, or LV Football as explained in `football-user-guide.pdf`). Essentially, only replication of *logical* structures like snapshots remains as the only reasonable option, with its drawbacks as explained above.


There is another argument: zfs tries to *hide* its internal structures and interfaces from the sysadmins, forming a more or less **monolithic**[a] **architecture** as seen from outside.


This violates the classical *layering rules* from Dijkstra (see section Layering Rules and their Importance). In contrast, classical LVM-based configurations (see section Positive Example: ShaHoLin storage + application stack or the example setup in `mars-user-manual.pdf`) are **component oriented**, according to the **Unix Philosophy**.

[a] Some sysadmins acting as `zfs` advocates are reclaiming this as an advantage, because they need to understand only a single tool for managing "everything". However, this is a short-sighted argument when it comes to *true* flexibility as offered by a component-based system, where multiple types of hardware / software RAID, multiple types of LVM functionality, and much more can be almost orthogonally combined in a very flexible way.

4.6. Local vs Centralized Storage

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

4.6.1. Internal Redundancy Degree

Details 4.11:

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

1. Redundant power supplies with UPS.

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

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

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

Manager Hint 4.21: **Redunduncy degree of RAID vs commercial appliances**

When dimensioned appropriately, real architectual and functional differences at block layer are smaller than certain people are claiming. For many block layer use cases, redundancy is **roughly comparable**.
If you compare typical prices for both competing systems, you will notice a *huge* difference in favour of RAID. See also section Cost Arguments from Technology.

4.6.2. Capacity Differences

There is another hard-to-die myth: commercial storage would provide higher capacity. Please read the data sheets. It is *possible* (but not generally recommended) to put several hundreds of spindles into several external HDD enclosures, and then connect them to a redundant cross-coupled pair of RAID controllers via several types of SAS busses.

Manager Hint 4.22: **Maximum possible RAID capacity**

By filling a rack this way, RAID can easily reach similar, if not higher capacities than commercial storage boxes, for a *fraction* of the price.

However, this is not the recommended way for *general* use cases, but could be an option for low demands like archiving. The big advantage of RAID-based local storage is **massive scale-**

out by sharding, as explained in section Distributed vs Local: Scalability Arguments from Architecture.

4.6.3. Caching Differences

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

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

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

Reason: the standard Linux kernel has two main caches, the Page Cache for file content, and the Dentry Cache (plus Inode slave cache) for metadata. Both caches are residing in **RAM**, which is the *fastest* type of cache you can get. Some more details are in section Performance Arguments from Architecture.

Nowadays, typical servers have several hundreds of gigabytes of RAM, sometimes even up to terabytes, resulting in an incredible caching behaviour which can be measured[35].

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

Manager Hint 4.23: **IOPS over-engineering**

IOPS over-engineering by some orders of magnitudes can cause *considerable* unnecessary expenses. Be sure to carefully **check real demands**!

Details 4.12:

Some (but not all) commercial storage systems can deliver similar IOPS rates, because they have **internal RAM caches** in the same order of magnitude. Notice that persistent RAM is the **most expensive** type of scalable storage you can buy.
People who are demanding such systems are typically falling into some of the following classes (list is probably incomplete):

- some people know this, but price does not matter - the more caches, the better. Wasted money for doubled caches does not count for them, or is even viewed as an advantage to them (personally). Original citation of an anonymous person: "only the best and the most expensive storage is good enough for us".
- using NFS, which has extremely poor filesystem caching behaviour because the Linux nfs client implementation does not take full advantage of the dentry cache. Sometimes people know this, sometimes not. Please read an important paper on the Linux implementation of nfs. Please search the internet for "Why nfs sucks" from Olaf Kirch (who is one of the original Linux nfs implementors), and *read* it. Your opinion about nfs might change.
- have transactional databases, where high IOPS may be *really* needed, but ***exceptionally***(!) for this class of application. For very big enterprise databases like big SAP installations, there may be a very valid justification for big RAM caches at storage layers. However: smaller transactional loads, as in webhosting, are *often* (not always) hammering a *low* number of **hot spots**, where *big* caches are not really needed. Relatively small BBU caches of RAID cards will do it also. Often people don't notice this because they don't measure the **workingset be-**

[35] Caution: this requires *extremely solid* expert knowledge and experience. It can be easily done wrongly. When managers are believing **fake results**, whether produced by accident from people stuck to **second-order ignorance**, or whether produced for some **political reasons**: This can be **dangerous for companies**.

haviour of their application, as could be done for example with `blkreplay` (see https://blkreplay.org).

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

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

Manager Hint 4.24:

There is no point in accessing the fastest possible type of RAM cache remotely over a network. RAM is best **invested money** when installed **locally**, *directly* for your applications / services / compute nodes.

Even expensive hardware-based RDMA (e.g. over Infiniband) cannot deliver the same performance as **directly caching** your data in the ***same* RAM** where your application is running. The Dentry Cache in the Linux kernel provides highly optimized **shared metadata** in SMP and NUMA systems (nowadays scaling to more than 100 processor cores), while the Page Cache provides **shared memory** via hardware MMU. This is crucial for the performance of classical local filesystems.

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

4.6.4. Latencies and Throughput

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

In the meantime, many people agree to a Google paper that the *ratio* of market prices (price per terabyte) between HDDs and SSDs are unlikely to change in a fundamental[36] way during the next 10 years. Thus, most large-capacity enterprise storage systems are built on top of HDDs.

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

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

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

The laws of information transfer are telling us:

Manager Hint 4.25:

With **increasing distance**, both latencies (laws of Einstein) and throughput (laws of energy needed for compensation of SNR = signal to noise ratio) are becoming worse. Distance matters.

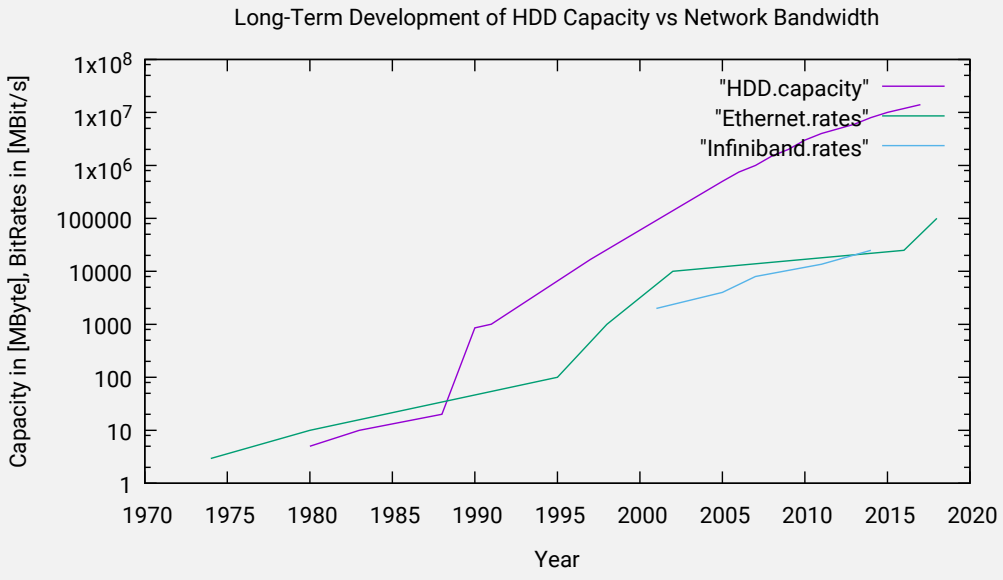
Because of this fundamental law, Football+MARS is **minimizing IO distances**.

The number of intermediate components, like routers / switches and their **queuing**, matters too.

Consequently, local storage has *always* an architectural[37] advantage in front of any attachment via network. Centralized storages are bound to some network, and thus suffer from disadvantages in terms of latencies and throughput.

Details 4.13:

What is the expected long-term future? Will additional latencies and throughput of centralized storages become better over time?
It is difficult to predict the future. Let us first look at the past evolution. The following graphics has taken its numbers from Wikipedia articles https://en.wikipedia.org/wiki/List_of_device_bit_rates and https://en.wikipedia.org/wiki/History_of_hard_disk_drives, showing that HDD capacities have grown **over-proportionally** by about 2 orders of magnitude over about 30 years, when compared to the relative growth of network bandwidth.
In the following graphics, effects caused by decreasing form factors have been neglected, which would even *amplify* the trend. For fairness, bundling of parallel disks or parallel communication channels[a] have been ignored. All comparisons are in logarithmic y axis scale:



What does this mean when extrapolated into the future?
It means that concentrating more and more capacity into a single rack due to increasing data density will likely lead to more problems in future. Accessing more and more data over the network will become increasingly more difficult when concentrating high-capacity HDDs or SSDs[b] into the same space volume as before.
In other words: centralized storages are no good idea yet, and will likely become an even

[37] In order to be fair, an architectural comparison must be made under the assumption of comparable low-level technologies.

worse idea in the future.

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

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

Example 4.7: Risky central storage architecture

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

Example 4.8: Non-competing scalabilty of central storage

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

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

Manager Hint 4.26: **Strategic advice**

Today's **future** is likely dominated by **scaling-out architectures** like **sharding**, as explained in section Distributed vs Local: Scalability Arguments from Architecture.

4.6.5. Reliability Differences CentralStorage vs Sharding

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

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

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

In contrast, sharded storage (for example the LocalSharding model, see also section Variants of Sharding) has MPOF = Multiple Points Of Failure. It is unlikely that many shards are failing fatally at the same time, because shards are *independent*[38] from each other by definition (see section Definition of Sharding for disambiguation of terms "sharding" and "shared-nothing").

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

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

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

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

1. There are much **less customers affected** (typically only 1 shard out of n shards).

2. **MTTR** = Mean Time To Repair is typically much better because there is much less data to be restored.

3. **Residual risk** plus resulting fatal damage by **un-repairable problems** is thus lower.

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

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

In contrast, the (Local)Sharding model is *distributing* the **indispensible incidents** (because **perfect systems do not exist**, and **perfect humans do not exist**) to a lower number of customers with higher frequency, such that the **total impact onto the business** becomes bearable.

Manager Hint 4.27: **Risk analysis of CentralStorage**

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

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

Conclusions: CentralStorage is something for

- Small to medium-sized companies which don't have the **manpower** and the **skills** for professionally building and operating a (Local)Sharding (or similar) system for their enterprise-critical mass data their business is relying upon.

- ***Monolithic* enterprise applications** like classical SAP which are anyway bound to a specific vendor, where you cannot select a different solution (so-called **Vendor Lock-In**).

- When your application **is neither shardable** by construction (c.f. section 4.2), or when doing so would be a too high effort, **nor going to BigCluster**[39] (e.g. Ceph / Swift / etc, see section Reliability Arguments from Architecture) is an option.

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

Manager Hint 4.28:

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

Manager Hint 4.29:

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

4.6.6. Proprietary vs OpenSource

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

Architecture: LocalStorage vs CentralStorage vs DistributedStorage

Licensing: Proprietary vs OpenSource

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

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

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

Therefore **interoperability** is a *must* in the opensource business. For example, you can relatively easily migrate between DRBD and MARS, forth and backwards, see `mars-user-manual.pdf`. The *generic* block devices provided by both DRBD and MARS (and by the kernel LVM2 implementation, and many others ...) can interact with zillions of filesystems, VMs, applications, and so forth.

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

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

This can be a serious threat to your business relying on the value of your data. In particular, buying storage systems from *small* vendors may increase this risk[40].

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

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

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

Manager Hint 4.30: **Long-term strategy**

When some appropriate OpenSource solution, or when some OpenSource components are availabe, its long-term TCO will be typically better than from proprietary vendors.

4.7. Cost Arguments

A common pre-jugdement is that "big cluster" is the cheapest scaling storage technology when built on so-called "commodity hardware".

While this is very often true for the "commodity hardware" part, it is often *not* true for the "big cluster" part. Let us first look at the "commodity" part.

4.7.1. Cost Arguments from Technology

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

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

You can see that any self-built and self-administered storage (whose price varies with slower high-capacity disks versus faster low-capacity disks) is much cheaper than any commercial offering by about a factor of 10 or even more.

Manager Hint 4.31:

If you need to operate several petabytes of data, self-built storage is *always* cheaper than commercial one, even if some more manpower is needed for commissioning and operating, than for communications with the storage provider. You don't have to pay the shareholders of the storage provider. Instead, the savings will benefit your *own* shareholders.

Here we just assume that the storage is needed permanently for at least 5 years, as is the case in web hosting, databases, backup / archival systems, and many other application areas.

Commercial offerings of cloud storage are way too much hyped. Apparently some people don't seem to know that the generic term "Cloud Storage" refers to a *storage class* (see section Architectural Properties of Cloud Storage), not to a particular *instance* like original Amazon S3, and that it is possible to build and operate almost any instance of any storage class yourself.

From a commercial perspective, **outsourcing** of *huge masses* of enterprise-critical storage (to whatever class of storage) usually pays off **only when** your storage demands are either *relatively moderate*, or are *extremely* varying over time, and/or when you need some *extra* capacity only *temporarily* for a *very* short time.

4.7.2. Cost Arguments from Architecture

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

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

As shown in section Reliability Arguments from Architecture, and as recommended by several advocates, two replicas are typically not sufficient for BigCluster. Even addicts of BigCluster are typically recommending 3 replicas in so-called "best practices", leading to the following more realistic table:

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

The crucial point is not only the number of extra servers needed for dedicated storage boxes, but also the total number of HDDs. While big cluster implementations like Ceph or Swift can *theoretically* use some erasure encoding[41] for avoiding full object replicas, their *practice* as seen in internal 1&1 Ceph clusters is similar to RAID-10, but just on objects instead of block-based sectors.

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

There is a surprising consequence from this: geo-redundancy is not as expensive as many people are believing. It just needs to be built with the proper architecture. A sharded geo-redundant pool based on hardware RAID-60 (last column "SHA×2") costs typically *less* than a non-georedundant big cluster with typically needed / recommended number of replicas (column "BC"). A geo-redundant sharded pool provides even better failure compensation (see sections Reliability Arguments from Architecture and Flexibility of Handover / Failover Granularities), and comparable flexibility when combined with Football (see section Principle of Background Migration).

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

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

[41] There is a reason why erasure encoding is not practical for many `BigCluster` use cases. The number of total IO requests sent to the internal disks is much higher than the number of IO requests sent to the storage by your application, in order to update additional redundancy information. Like RAID-6, this is typically by *factors*. While RAID-6 is **offloading** this additional workload to a small *specialized* and realtime-capable network called SAS bus, `BigCluster` is typically spreading this workload over an unreliable IP network with packet loss, spanning much larger distances, and involving more switches / routers.

5. Use Cases for MARS

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

Manager Hint 5.1: Use cases MARS vs DRBD

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

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

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

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

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

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

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

[2] Active-active won't work over long distances at all because of high network latencies (cf chapter 4). Probably, for replication of whole clusters over long distances DRBD and MARS could be stacked: using DRBD on top for MARS for active-active clustering of `gfs2` or `ocfs2`, and a MARS instance *below* for failover of *one* of the DRBD replicas over long distances.

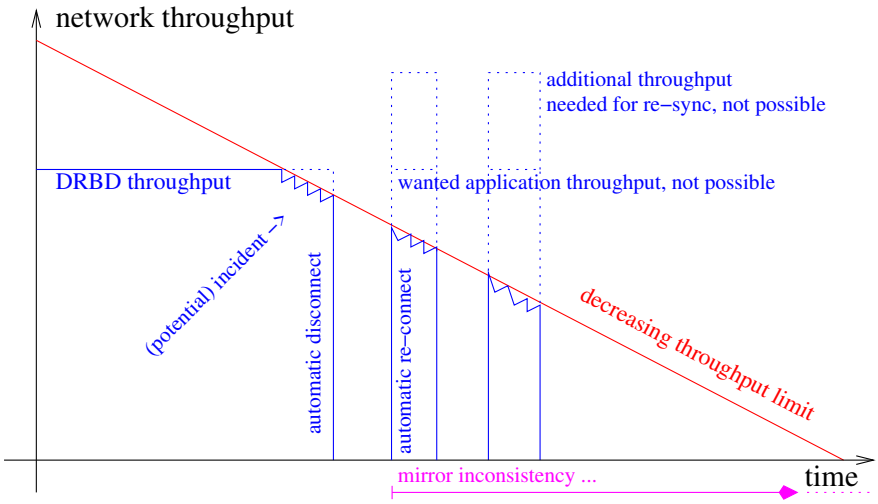
5.1. Network Bottlenecks

5.1.1. Behaviour of DRBD

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

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

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



The simplified scenario is the following:

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

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

What will happen in this scenario?

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

Once the maximum network throughput (red line) starts to fall short of the required application throughput (first blue dotted line), we get into trouble. By its very nature, DRBD works **synchronously**. Therefore, it *must* transfer all your application writes through the bottleneck, but now it is impossible[4] due to the bottleneck.

As a consequence, the application running on top of DRBD will see increasingly higher IO latencies and/or stalls / hangs. We found practical cases (at least with former versions of DRBD) where IO latencies exceeded practical monitoring limits such as 5 s by far, up to the range of *minutes*. Experienced sysadmins will know what happens next: your application will run into an **incident**, and your customers will be dissatisfied.

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

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

Details 5.1:

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

[a] By appropriately tuning various DRBD parameters, such as `timeout` and/or `ping-timeout`, you can keep the impact of the incident below some viable limit. However, the automatic disconnect will then happen earlier and more often in practice. Flaky or overloaded networks may easily lead to an enormous number of automatic disconnects.

What happens next in our scenario? During the `disconnect`, DRBD will record all positions of writes in its bitmap and/or in its activity log. As soon as the automatic re-connect succeeds after `connect-int` seconds, DRBD has to do a partial re-sync of those blocks which were marked dirty in the meantime. This leads to an *additional* bandwidth demand[5] as indicated by the upper dotted blue box.

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

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

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

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

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

Manager Hint 5.2: **Resilience of DRBD**

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

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

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

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

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

Manager Hint 5.3: **Risks from non-crossover DRBD**

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

 Notice that *classical* crossover cables usually do not show a behaviour like depicted by the red line. Traditional crossover cables are *passive components* which normally[6] either work, or not. The binary connect / disconnect behaviour of DRBD has no problems to cope with that.

 However, some newer Ethernet cable technologies like SFP+ and faster are no longer passive. They have some internal chips inside of their plugs. Thus they may **fail independently** from your storage nodes. Then you run at least the risks from the CAP theorem, see section Explanation via CAP Theorem. In addition to CAP effects, intermitting errors such as flaky electrical contacts may rise the above risk of permanent data loss.

Details 5.2:

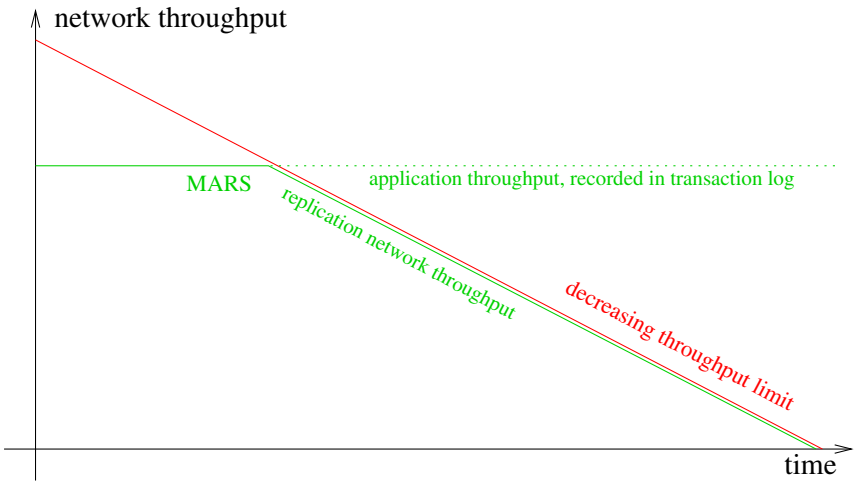
 or  Linbit recommends a **workaround** for the inconsistencies during re-sync: LVM snapshots. We tried it, but found a *performance penalty* which made it prohibitive for our concrete application. A problem seems to be the cost of destroying snapshots. LVM uses by default a BOW strategy (Backup On Write,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

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

5.1.2. Behaviour of MARS

The behaviour of MARS in the above scenario:

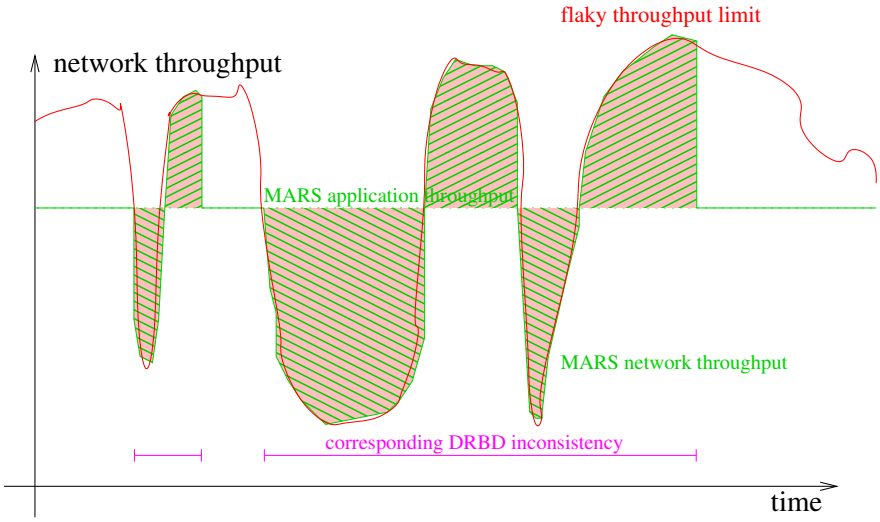
[6] Exceptions might be mechanical jiggling of plugs, or electro-magnetical interferences. We never noticed any of them.



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


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

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



This time, the network throughput (red line) is varying[7] in some unpredictable way. As before, the application throughput served by MARS is assumed to be constant (dotted green line, often superseded by the solid green line). The actual replication network throughput is depicted by the solid green line.

As you can see, a network dropdown undershooting the application demand has no impact onto the application throughput, but only onto the replication network throughput. Whenever the network throughput is held back due to the flaky network, it simply catches up as soon as

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

possible by overshooting the application throughput. The amount of lag-behind is visualized as shaded area: downward shading (below the application throughput) means an increase of the lag-behind, while the upwards shaded areas (beyond the application throughput) indicate a decrease of the lag-behind (catch-up). Once the lag-behind has been fully caught up, the network throughput suddenly jumps back to the application throughput (here visible in two cases).

 Note that the existence of lag-behind areas is roughly corresponding to DRBD disconnect states, and in turn to DRBD inconsistent states of the secondary as long as the lag-behind has not been fully cought up. The very rough[8] duration of the corresponding DRBD inconsistency phase is visualized as magenta line at the time scale.

Manager Hint 5.4: **Optimum throughput via MARS**

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

 Note that *in average* during a longer period of time, the network must have emough capacity for transporting *all* of your data. MARS cannot magically break through information-theoretic limits. It cannot magically transport terabytes of data in a few seconds over very slow modem[9] lines. Only *relatively short* network problems / packet loss can be compensated, depending on the capacity of the `/mars` filesystem.

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

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

Manager Hint 5.5: **Risk reduction via MARS**

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

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

[9] A certain colleague at 1&1 is using MARS for a private application: CDP = Continuous Data Protection of a critical Windows VM over his home DSL line.

Even when the capacity of `/mars/` is exhausted and thus emergency mode is entered, the replicas will *not* become inconsistent by themselves. However, when the emergency mode is later *cleaned up* for a replica via `marsadm invalidate`, it will become *temporarily* inconsistent during the fast full sync.

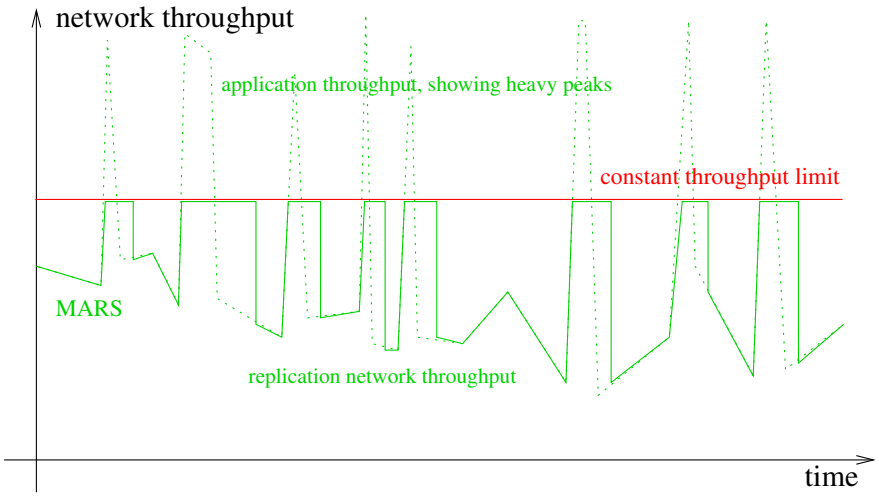
When you have a total of $k \geq 3$ replicas, you don't need to invalidate them *all in parallel*. By cascading the full syncs sequentially, you can retain some consistent, but outdated replica for the meantime, until all sync have finished.

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

Details 5.3: **Simple traffic shaping be default**

Newer versions of MARS are automatically setting the so-called TOS fields in standard TCP/IP packets for you, which is backwards compatible with the newer DSCP feature. You just need to properly configure your network equipment for this type of traffic shaping, unless it isn't already enabled by default from various network vendors. In the latter case, you don't need to do anything, in order to get some improvements by automatic traffic shaping for free. Details are in `mars-user-manual.pdf`.

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



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

When the peaks are exceeding the network capacities for some short time, the replication network throughput (solid green line) will be limited for a short time, stay a little bit longer at the limit, and finally drop down again to the normal workload.

Manager Hint 5.6: **Resilience against load peaks**

In other words, you get a flexible buffering behaviour, coping with application load peaks.

Similar scenarios (where both the application workload has peaks and the network is flaky to some degree) are rather common.

If you would use DRBD in place of MARS, you were likely to run into regular application performance problems and/or frequent automatic disconnect cycles, depending on the height and on the duration of the peaks, and on network resources. As observed at 1&1, even permanent data loss is possible, with some residual probability.

5.2. Long Distances / High Latencies

Details 5.4:

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

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

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

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

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

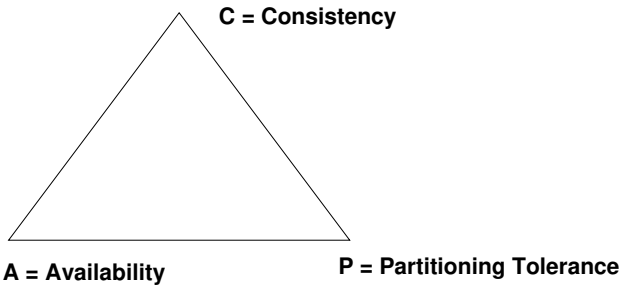
Manager Hint 5.7: **Safety rule for synchronous replication**

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

There may be some exceptions[10], e.g. when dealing with low-end workstation loads. But when you are **responsible** for a whole datacenter and/or for **enterprise-critical data**, don't waste your time by trying (almost) impossible things. We recommend to use MARS in such use cases.

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

5.3. Explanation via CAP Theorem



The famous CAP theorem, also called Brewer's theorem, is important for a deeper understanding of the differences between DRBD and MARS. A good explanation can be found at https://en.wikipedia.org/wiki/CAP_theorem (retrieved July 2018).

The CAP theorem states that only 2 out of 3 properties can be achieved at the same time, when a Distributed System is under pressure: C = Consistency means ***Strict*** Consistency at the level of the *distributed* system (which is *not* the same as strict consistency *inside* of one of the *local* systems), A = Availability = intuitively clear from a user's perspective, and P = Partitioning Tolerance = the network may have its own outages at any time (which is a negative criterion).

As explained in the Wikipedia article, the P = Partitioning Tolerance is a property which is imporant at least in *wide-distance* data replication scenarios, and possibly in some other scenarios. There the property P cannot be chosen at runtime, but is *given* by *setup* of the Distributed System.

5.3.1. CAP Differences between DRBD and MARS

If you are considering only short distances like passive crossover cables between racks, *then* (and *only then*) you may *assume(!)* that no effort for achieving property P is required, because it it is already given for free. Then, and only then, you can get both A and C at the same time, without sacrificing P, because P is already for free by *assumption*. In such a passive crossover cable scenario, getting all three properties C and A and P is possible, similarly to an explanation in the Wikipedia article.

 Newer types of network cables for 10 GBit and more (e.g. SFP+) may have some active chips internally in their plugs. Suchalike technologies are no longer passive. Consequently, the assumption "passive component which cannot fail" is no longer true by construction.

Relying on the assumption "P is for free = the network cannot fail" leads us to classical use cases for DRBD: when both DRBD replicas are always staying physically connected via a passive crossover cable (which is *assumed* to never break down), you *could potentially* get both strict global consistency and availability.

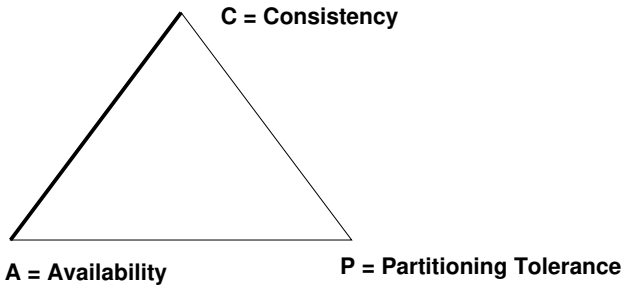
Whether this is real in practice for DRBD, is a different story. It depends on the *implementation* of DRBD. Some sysadmins at 1&1 Ionos have made the experience that there is no 100% CAP guarantee, regardless of DRBD protocol configuration, while they were testing only some cases where only *one* of the DRBD nodes was failing[11]. Both C and A are provided by DRBD during `connected` state, while P is *assumed* to be provided by a passive component.

By addition of iSCSI failover (e.g. ALUA and similar technologies), it *should* be possible to achieve A, even in case of single storage node failures, while retaining C from the viewpoint[12] of the application.

[11] In addition, you will need some further components like Pacemaker, iSCSI failover, etc. These might also be involved in the practically observed behaviour.

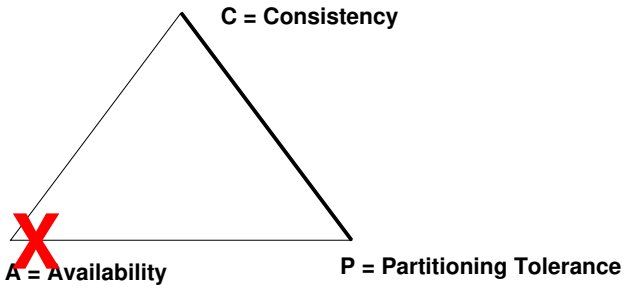
[12] Notice: the CAP theorem does not deal with node failures, only with *network* failures. Node failures would always violate C by some "strong" definition. By some "weaker" definition, the downtime plus recovery time (e.g. DRBD re-sync) can be taken out of the game. Notice: while a node can always "know" whether it has failed (at least after reboot), network failures cannot be distinguished from failures of remote nodes in general. Therefore node failures and network failures are fundamentally different by their nature.

This is explained by the thick line in the following variant of the graphics, which is only valid for passive crossover cables where P need not be guaranteed by the replication because it is already assumed for free:

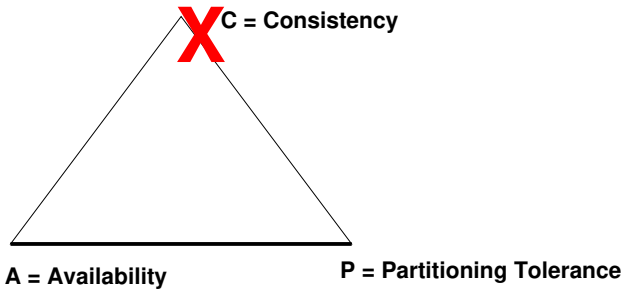


Now look at the case of a truly Distributed System, where P cannot be assumed as for free. For example, try to use DRBD in a long-distance replication scenario. There we cannot assume P as already given. We **must *tolerate*** replication network outages. DRBD is reacting to this differently in two different modes.

First we look at the (short) time interval *before* DRBD recognizes the replication network incident, and before it leaves the `connected` state. During this phase, the application IO will **hang** for some time, indicating the (temporary) sacrifice (from a user's perspective) by a red X:



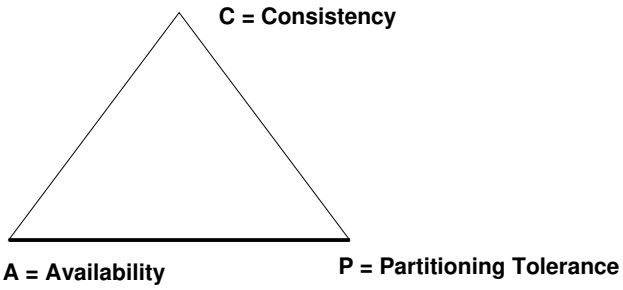
Because Availability is one of the highest goods of enterprise-critical IT operations, you will typically configure DRBD such that it automatically switches to some variant of a `disconnected` state after some timeout, thereby giving up consistency between both replicas. The red X indicates not only loss of global strict consistency in the sense of the CAP theorem, but also that your replica will become `Inconsistent` during the following re-sync:



You may wonder what the difference to MARS is. As explained in section Architectural Properties of Cloud Storage, MARS is not only intended for wide distances, but also for **Cloud Storage** where no strict consistency is required at global level by definition, but instead **Eventually Consistent** is the preferred model for the Distributed System. Therefore, *strict* consistency (in the sense of the CAP theorem) is *not required by definition.*

Consequently, the red X is not present in the following graphics, showing the state where MARS is remaining *locally consistent* all the time[13], even when a network outage occurs:

[13]Notice that the *initial* full sync is not considered here, neither for DRBD, nor for MARS. *Setup* of the Distributed System is its own scenario, not considered here. *Repair* of a *damaged* system is also a different scenario, also not considered here. Notice the MARS' emergency mode also belongs to the class of "damages", as well as DRBD' disk failure modes, where is has some additional functionality compared to the current version of MARS.



Notice: MARS does not guarantee strict consistency *between* LV replicas at the level of the Distributed System, but only Eventually Consistent. However, *at the same time* it *also* guarantees strict consistency *locally*, and even at *each* of the passive replicas, each by each. Don't confuse these different levels. There are two different consistency guarantees at different levels, both at the same time. This might be confusing if you are not looking at the system at different levels: (1) overall Distributed System versus (2) each of the local system instances.

Why does MARS this? Because a better way is not possible at all. The CAP theorem tells us that there exists no better way when both A has to be guaranteed (as almost everywhere in enterprise-critical IT operations except database systems), and P has to be ensured in geo-redundant datacenter disaster scenarios or some other scenarios. Similarly to natural laws like Einstein's laws of the speed of light, there *does not exist* a better way!

Manager Hint 5.8: **Solution classification of DRBD**

Conclusion from the CAP theorem: when P is a *hard requirement*, don't use DRBD (or any other *synchronous* replication implementation) for long-distance and/or true Cloud Storage scenarios. It is only well-suited for short-distance crossover cable scenarios.

The red X is in particular problematic during re-sync, after the network has become healthy again (cf section Behaviour of DRBD). MARS has no red X at C because of its **Anytime Consistency**, which refers to *local* consistency, and which is violated by DRBD during certain important phases of its regular operation.

Manager Hint 5.9: **Impossible requirements**

If you think that you require alle three properties C+A+P, but you don't have passive crossover cables over short distances, you are requiring something which is **impossible** in general. You need give up one of them, at least with a certain probability.

There exists no solution, with whatever component, or from whatever commercial storage vendor. Although some "marketing drones" are claiming the impossible, e.g. by citing *examples*, which are then incorrectly generalized. You might have luck, and there might be *exceptional examples* where all three C+A+P were ok, **by chance**. But there remains a **risk**. The CAP theorem is as hard as Einstein's natural laws are.

You need a conscious decision about **priorities**, which property to drop first. Rethink your complete concept, from end to end. Something is wrong, somewhere. Ignoring a fundamental law like CAP on enterprise-critical use cases can endanger a company and/or your career.

5.3.2. CAP Commonalities between DRBD and MARS

In this subsection, we look at the case that P is not for free, but has to be ensured by the Distributed Storage system.

You may have noticed that MARS' ordinary CAP behaviour is similar to DRBD's CAP picture in `disconnected` state, or during similar states when the replication network is interrupted.

Replication network interruption is also known as "Network Partitioning". This is where property P = Partitioning Tolerance comes into play.

When a network partition has *actually occurred*, both DRDB and MARS allow you to do the same: you may **forcefully switch** the `primary` role, which means activation of a former `secondary` node. In such a situation, you can issue commands like `drbdadm primary --force` or `marsadm primary --force`. It is no accident that both commands are looking similar to each other.

The outcome will be the same: you will most likely get a `SplitBrain` situation.

The possibility of getting a split brain is no specific property of neither DRBD nor MARS. It will also happen with any other replication system, whether synchronous or asynchronous.

It is one of the consequences from the CAP theorem when (1a) P has to be assured, and (1b) a network partition has *actually occurred*, and (2) when A = Availability is enforced at both sides of the network partition. The result is that C = *global* Consistency may be violated, by creation of two or more versions of the data.

Details 5.5:

Fortunately, there is a method for *dynamic* control of SplitBrain at *runtime*. The decision about forceful creation of SplitBrain can be made *dynamically dependent* on further external factors, like current customer demands, or forecasts, etc.

Careful: at least for some application classes, it is a bad idea to systematically create split brain via automatic cluster managers, e.g. Pacemaker or similar. As explained in section 6.1 on page 99, some cluster managers were originally constructed for truly shared disk scenarios, where no split brain can occur by construction. Using them in masses on versioned data in truly distributed systems can result in existential surprises, once a bigger network partition and/or a flaky replication networks triggers them in masses, and possibly at unexpected moments.

Manager Hint 5.10:

Split brain should not be provoked when not *absolutely* necessary.

Split brain resolution is all else but easy *in general*. When the data is in a generic block device, you typically will have no general means for *merging* both versions. This means, split brain resolution is typically only possible by **throwing away** some of the versions.

This kind of split brain resolution problem is not specific for DRBD or MARS. It is a fundamental property of Distributed Systems, and the difficulty of resolution is an inherent property of generic block devices.

DRBD and MARS have some commands like `drbdadm invalidate` or `marsadm invalidate` for this. Again, the similarity is no accident.

Notice that classical filesystems aren't typically better than raw block devices. There are even more possibilities for tricky types of **conflicts** (e.g. on path names in addition to file content). Anyway, long-distance replication should not be done at filesystem layer, see section Performance Penalties by Choice of Replication Layer.

Similary, BigCluster object stores are often suffering from similar (or even worse) problems, because higher application layers may have some hidden internal dependencies between object versions, while the object store itself is agnostic of version dependencies in general[14].

Cautious: when stacking block devices or filesystems, or any other complex *structured aggregates* on top of some BigCluster object store, you are creating another fundamental risk,

[14] There exists lots of types of potential dependencies between objects. Timely ones are easy to capture, but this is not sufficient in general for everything.

in addition to Dijkstra regressions explained in section Negative Example: object store implementations mis-used as backend for block devices / POSIX filesystems. Several types[15] of object stores will not magically resolve any split brain for you. Check whether your favorite object store implementation has some kind of equivalent of a `primary --force` command. If it doesn't have one, or only a restricted one, you should be ***alerted***. In case of a *long-lasting(!)* storage network partition, you might need suchalike *desperately* for ensuring A, even at the cost of C[16].

Check: whether you need this is heavily depending on the ***application class*** (see also the Cloud Storage definition in section Architectural Properties of Cloud Storage). If you *would* need it, but you are **not prepared for suchalike scenarios at your enterprise-critical data**, it could cost you a lot of money and/or reputation and/or even your existence.

Notice: the *concept* of `SplitBrain` is occurring almost everywhere in truly Distributed Systems when C can be violated in favour of A+P. It is a very general consequence[17] of the CAP theorem.

The only reliable way for avoiding split brain in truly distributed systems would be: don't insist on A = Availability. Notice that there exist only a few application classes, like certain types of banking, where C is typically a higher good than A.

Notice that both DRBD and MARS are supporting suchalike application classes also: just *don't* add the option `--force` to the `primary` switch command.

However: even in banking, some *extremely extraordinary* scenarios might occur, where sacrifice of C in favour of A could be necessary (e.g. when *manual cleanup* of C is cheaper than long-lasting violations of A).

Manager Hint 5.11: **Summary CAP decisions**

Both DRBD and MARS have some emergency measure for killing C in favour of A. It requires your **conscious decision** whether / where / when to use it, *or not*.

5.4. Higher Consistency Guarantees vs Actuality

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

Details 5.6:

Some people, often located at database operations, are obtrusively arguing that actuality is such a high good that it must not be sacrificed under any circumstances.
Anyone arguing this way has at least the following choices (list may be incomplete):

1. None of the above use cases for MARS apply. For instance, short distance replication over crossover cables is sufficient, and the network is reliable enough such that bottlenecks can never occur (e.g. because the total load is extremely low,

[15] Notice: BigCluster architectures are typically discriminating between between client servers and storage servers. This will typically introduce some more possibilities into the game, such as forced client failover, independently from forced storage failover.

[16] Notice that the C functionality is often not implemented by the object store itself (which typically provides only *eventually consistent* at object granularity), but implemented by the distributed block device or distributed filesystem, if it is implemented at all. There is a fundamental problem with at least 3 different granularities to be resolved: in order to guarantee strict consistency at (1) aggregate granularity, which is independent from the (2) network partition granularity, in general multiple versions of objects may be required at (3) object granularity. Does your object store have a means for this, similarly to multiversion databases, e.g. multiversion timestamp ordering?

[17] There exist only few opportunities for generic conflict resolution, even in classical databases where *some* knowledge about the structure of the data is available. Typically, there exist some more *hidden* dependencies than people are expecting. Lossless `SplitBrain` resolution will thus need to be implemented at application layer, if it is possible at all.

or conversely the network is extremely overengineered / expensive), or the occurrence of bottlenecks can *provably* be taken into account. In such cases, DRBD is clearly the better solution than MARS, because it provides better actuality than the current version of MARS, and it uses up less disk resources.

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

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

4. People do not know the CAP theorem (see section Explanation via CAP Theorem), and are trying to require something which simply is **impossible**.

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

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

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

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

A common misunderstanding is about the actuality guarantees provided by filesystems. The buffer cache / page cache uses by default a **writeback strategy** for performance reasons. Even modern journalling filesystems will (by default) provide only consistency guarantees, but no strong actuality guarantee. In case of power loss, some transactions may be even *rolled back* in order to restore consistency. According to POSIX[18] and other standards, the only *reliable* way to achieve actuality is usage of system calls like `sync()`, `fsync()`, `fdatasync()`, flags like `O_DIRECT`, or similar. For performance reasons, the *vast majority of applications* don't use them at all, or use them only sparingly!

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

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

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

6. Requirements of Long-Distance Replication

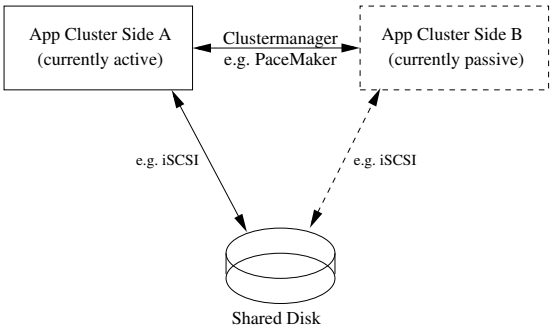
6.1. Avoiding Inappropriate Clustermanager Types for Medium and Long-Distance Replication

This section addresses some wide-spread misconceptions. Its main target audience is *userspace* developers, but others may profit from **detailed explanations of problems and pitfalls**. When the problems described in this section are solved somewhen in future, this section will be shortened and some relevant parts moved to the appendix.

Doing **HA = High Availability** (see section What is HA = High Availability) wrong at *concept level* may easily get you into trouble, and may cost you several millions of € or $ in larger installations, or even knock you out of business when disasters are badly dealt with at higher levels such as clustermanagers.

6.1.1. General Cluster Models

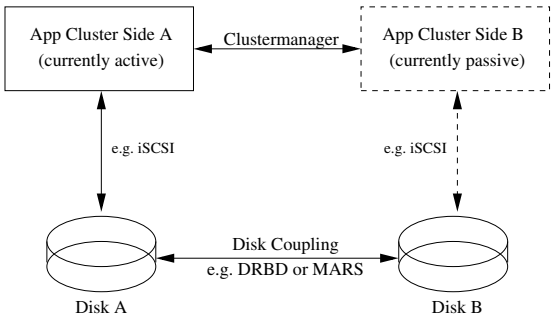
The most commonly known cluster model is called **shared-disk**, and typically controlled by clustermanagers like `PaceMaker`:



The most important property of shared-disk is that there exists only a single disk instance. Nowadays, this disk often has some *internal* redundancy such as RAID. At *system* architecure layer / network level, there exists no redundant disk at all. Only the application cluster is built redundantly.

It should be immediately clear that shared-disk clusters are only suitable for short-distance operations in the same datacenter, or better in the same room / rack. Although running one of the data access lines over short distances between very near-by datacenters (e.g. 1 km) would be theoretically possible, there would be no sufficient protection against failure of a whole datacenter.

Both DRBD and MARS belong to a different architectural model called **shared-nothing**:



The characteristic feature of a shared-nothing model is (additional) **data redundancy at network level**.

Shared-nothing "clusters[1]" could theoretically be built for *any* distances, from short to medium to long distances. However, concrete technologies of disk coupling such as synchronous operation may pose practical limits on the distances (see chapter Use Cases for MARS).

In general, clustermanagers must fit to the model. Some clustermanager can be configured to fit to multiple models. If so, this must be done properly, or you may get into serious trouble.

Manager Hint 6.1:

Some people don't know, or they don't believe even when told them, that different architectural models like shared-disk or shared-nothing will *require* an *appropriate* type of clustermanager and/or at least a different configuration. Failing to do so, by selection of an inappropriate clustermanager type and/or an inappropriate configuration may be **hazardous**.

Pitfall: suchalike problems are typically appearing **only during incidents**.

It is dangerous to conclude from "stable ordinary operation" that the system is reliable. The real **risk** is that **data inconsistencies** are showing up at the **wrong moment**, when the clustermanager has to execute the right actions for compensation of a certain component failure.

Selection of the right model alone is not sufficient. Some, if not many, clustermanagers have not been designed for long distances (see section What is Geo-Redundancy).

As explained in section Special Requirements for Long Distances, long distances have further **hard requirements**. Disregarding them may be also hazardous!

6.1.2. Handover / Failover Reasons and Scenarios

From a sysadmin perspective, there exist a number of different **reasons** why the application workload must be switched from the currently active side A to the currently passive side B:

1. Some **defect** has occurred at cluster side A or at some corresponding part of the network.
2. Some **maintenance** has to be done at side A which would cause a longer downtime (e.g. security kernel update or replacement of core network equipment or maintainance of UPS or of the BBU cache etc - hardware isn't 24/7/365 in practice, although some vendors *claim* it - it is either not really true, or it becomes *extremely* expensive).

Both reasons are valid and must be automatically *handled* (but not necessarily automatically *triggered*) in larger installations. In order to deal with all of these reasons, the following basic mechanisms can be used in either model:

[1] Notice that the term "cluster computing" usually refers to short-distance only. Long-distance coupling should be called "grid computing" in preference. As known from the scientific literature, grid computing requires different concepts and methods in general. Only for the sake of simplicity, we use "cluster" and "grid" interchangeably.

1. **Failover** (triggered either manually or automatically)
2. **Handover** (triggered manually[2])

It is important to not confuse handover with failover at concept level. Not only the reasons / preconditions are very different, but also the *requirements*.

Example 6.1:

Precondition for handover is that *both* cluster sides are healthy, while precondition for failover is that *some really relevant(!)* failure has been *detected* somewhere (whether this is *really* true is another matter). Typically, failover must be able to run in masses, while planned handover often has lower scaling requirements.

Not all existing clustermanagers are dealing with all of these cases (or their variants) equally well, and some are not even dealing with some of these cases / variants *at all*.

Some clustermanagers cannot easily express the concept of "automatic triggering" versus "manual triggering" of an action. There exists simply no cluster-global switch which selects either "manual mode" or "automatic mode" (except when you start to hack the code and/or write new plugins; then you might notice that there is no sufficient architectural layering / sufficient separation between mechanism and strategy).

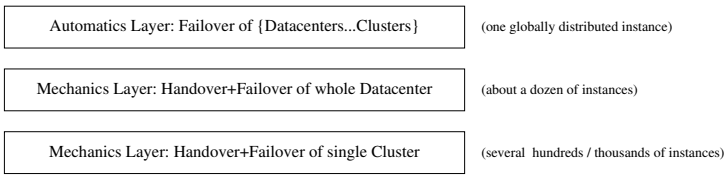
Manager Hint 6.2:

Being forced to permanently use an automatic mode for **triggering** several hundreds or even thousands of clusters is not only boring, but bears a **considerable risk** when automatics do a wrong decision at hundreds of instances in parallel.

6.1.3. Granularity and Layering Hierarchy for Long Distances

Many existing clustermanager solutions are dealing with a single cluster instance, as the term "*cluster*manager" suggests. However, when running several hundreds or thousands of cluster instances, you likely will not want to manage each of them individually. In addition, failover should *not only* be *triggered* (not to be confused with *executed*) individually at cluster level, but likely *also* at a higher granularity such as a room, or a whole datacenter. Otherwise, some chaos is likely to happen.

Here is what you probably will **need**, possibly in difference to what you may find on the market (whether OpenSource or not). For simplicity, the following diagram shows only two levels of granularity, but can be easily extended to multiple layers of granularity, or to some concept of various *subsets of clusters*:



Notice that many existing clustermanager solutions are not addressing the datacenter granularity at all. Typically, they use concepts like **quorums** for determining failures *at cluster level* solely, and then immediately executing failover of the cluster, sometimes without clean architectural distinction between trigger and execution (similar to the "separation of concerns" between **mechanism** and **strategy** in Operating Systems). Sometimes there is even no internal software layering / modularization according to this separation of concerns at all.

 When there is no distinction between different levels of granularity, you are hopelessly bound to a non-extensible and thus non-adaptable system when you need to operate masses of clusters.

[2] Automatic triggering could be feasible for prophylactic treatments.

Manager Hint 6.3: **Minimum requirements for larger installations**

 A lacking distinction between automatic mode and manual mode in a cluster management solution, and/or lack of corresponding **architectural software layers** is not only a blatant ignoration of well-established best practices of **software engineering**, but will bind you even more firmly to an **inflexible system**, producing direct and indirect **long-term follow-up cost**.

 Terminology: for practical reasons, we use the general term "clustermanager" also for speaking about layers dealing with higher granularity, such as datacenter layers, and also for long-distance replication scenarios, although some terminology from grid computing would be more appropriate in a scientific background.

Please consider the following: when it comes to long-distance HA, the above layering architecture is also motivated by vastly different numbers of instances for each layer. Ideally, the topmost automatics layer should be able to overview several datacenters in parallel, in order to cope with (almost) global network problems such as network partitions. Additionally, it should also detect single cluster failures, or intermediate problems like "rack failure" or "room failure", as well as various types of (partial / intermediate) (replication) network failures. Incompatible decisions at each of the different granularities would be a no-go in practice. Somewhere and somehow, you need one single[3] top-most *logical* problem detection / ranking instance, which should be *internally distributed* of course, typically using some **distributed consensus protocol**; but in difference to many published distributed consensus algorithms it should be able to work with *multiple* granularities at the same time.

6.1.4. Discussion of Handover / Failover Methods

6.1.4.1. Failover Methods

Manager Hint 6.4:

Failover methods are only needed in case of an incident. They should not be used for regular handover, because preconditions are different. Inappropriate merges of both method classes will cause unnecessary **indirekt cost**.

STONITH-like Methods STONITH = Shoot The Other Node In The Head

These methods are widely known, although they have several serious drawbacks. Some people even believe that *any* clustermanager must *always* have some STONITH-like functionality. This is wrong. There *exist* alternatives, as shown in the next paragraph.

 A historical motivation for STONITH was prevention of illegal modifications of the *shared disk* by amok-running defective clients. In those ancient times, disks were *passive* mechanical components, while their disk controller was often belongig to the server. In modern shared-nothing scenarios, this motivation does no longer exist. Anyway, you can achieve **disk fencing** by various software means nowadays.

 The most obvious drawback is that STONITH will always create a **damage**, by definition.

[3] If you have *logical pairs of datacenters* which are firmly bound together, you could also have several topmost automatics instances, e.g. for each *pair* of datacenters. However, that would be very **inflexible**, because then you cannot easily mix locations or migrate your servers between datacenters. Using $k > 2$ replicas with MARS would also become a nightmare. In your own interest, please don't create any concepts where masses of hardware are firmly bound to fixed constants at some software layers.

Example 6.2:

Typical contemporary STONITH implementations are using IPMI and relatives for automatically powering off your server, or at least pushing the (virtual) reset button. This will *always* create a certain type of damage: the affected systems will definitely not be available, at least for some time until it has (manually) rebooted.

The STONITH damage leads to a *conceptual* contradiction: the reason for starting failover is that you want to restore availability as soon as possible, but in order to do so you will first *destroy* the availability of a particular *component*. This may be counter-productive.

Example 6.3:

When your hot standby node B does not work as expected, or if it works even *worse* than A before, you will *at least* loose some time until you *can* become operational again at the old side A. In addition, pushing the reset button bears the **risk of unnecessary data loss** from RAM buffers not yet written to disk, and in turn to **risk of data inconsistencies**, like need for a filesystem check. When some of the hardware is defective, like for example the boot disk or the boot sector, the system may not come up at all after reset.

Example 6.4: **STONITH variant for shared-nothing**

Here is an example method for handling a failure scenario. The old active side A is assumed to be no longer healthy anymore. The method uses a sequential state transition chain with a STONITH-like step:

Phase1 Check whether the hot standby B is currently usable. If this is violated (which may happen during certain types of disasters), abort the failover for any affected resources.

Phase2 *Try* to shutdown the damaged side A (in the *hope* that there is no *serious* damage).

Phase3 In case phase2 did not work during a grace period / after a timeout, assume that A is badly damaged and therefore STONITH it.

Phase4 Start the application at the hot standby B.

Notice: any cleanup actions, such as **repair** of defective hard- or software etc, are outside the scope of failover processes. Typically, they are executed much later when restoring redundancy.
Also notice: this method is a *heavily* distributed one, in the sense that sequential actions are alternated multiple times on different hosts. This is known to be cumbersome in distributed systems, in particular in presence of network problems.
Phase4 in more detail for DRBD, augmented with some pseudo code for application control:

1. at side B: `drbdadm disconnect all`
2. at side B: `drbdadm primary --force all`
3. at side B: `applicationmanager start all`

The same phase4 using MARS:

1. at side B: `marsadm pause-fetch all`
2. at side B: `marsadm primary --force all`
3. at side B: `applicationmanager start all`

This sequential 4-phase method is far from optimal, for the following reasons:

- The method tries to handle both failover and handover scenarios with one single sequential receipe. In case of a true failover scenario where it is *already known for sure* that side A is badly damaged, this method will unnecessarily waste time for phase 2. This could be fixed by introduction of a conceptual distinction between handover and failover, but it would not fix the following problems.
- Before phase4 is started (which will re-establish the service from a user's perspective), a lot of time is wasted by *both* phases 2 *and* 3. Even if phase 2 would be skipped, phase 3 would unnecessarily cost some time. In the next paragraph, an alternative method is explained which eliminates any unnecessary waiting time at all.
- The above method is adapted from the shared-disk model. It does not take advantage of the shared-nothing model, where further possibilities for better solutions exist.
- In case of long-distance network partitions and/or sysadmin / system management subnetwork outages, you may not even be able to (remotely) execute STONITH at all. Thus the above method misses an important failure scenario.

Some people seem to have a *binary* view at the healthiness of a system: in their view, a system is either operational, or it is damaged. This kind of view is ignoring the fact that some systems may be half-alive, showing only *minor* problems, or occurring only from time to time.

It is obvious that damaging a healthy system is a bad idea by itself. Even *generally* damaging a half-alive system in order to "fix" problems is not generally a good idea, because it may increase the damage when you don't know the *real* reason[4].

Even worse: in a distributed system[5] you sometimes *cannot(!)* know whether a system is healthy, or to what degree it is healthy. Typical STONITH methods as used in some contemporary clustermanagers are **assuming a worst case**, even if that worst case is currently not for real.

Details 6.1: **Advice**

Avoid the following **fundamental flaws** in failover concepts and healthiness models, which apply to implementors / configurators of clustermanagers:

- Don't mix up knowledge with conclusions about a (sub)system, and also don't mix this up with the real state of that (sub)system. In reality, you don't have any knowledge about a complex distributed system. You only may have *some* knowledge about *some* parts of the system, but you cannot "see" a complex distributed system as a whole. What you think is your knowledge, isn't knowledge in reality: in many cases, it is *conclusion*, not knowledge. Don't mix this up!
- Some systems are more complex than your model of it. Don't neglect important parts (such as networks, routers, switches, cables, plugs) which may lead you to wrong conclusions!
- Don't restrict your mind to boolean models of healthyness. Doing so can easily create unnecessary damage by construction, and even at concept level. You should know from software engineering that defects in concepts or models are much more serious than simple bugs in implementations. Choosing the wrong model cannot be fixed as easily as a typical bug or a typo.
- Try to deduce the state of a system as **reliably** as possible. If you don't know something for sure, don't generally assume that it has gone wrong. Don't confuse missing knowledge with the conclusion that something is bad. Boolean algebra restricts your mind to either "good" or "bad". Use at least **tri-state algebra** which has a means for expressing **"unknown"**. Even better: attach a probability

[4] Example, occurring in masses: an incorrectly installed bootloader, or a wrong BIOS boot priority order which unexpectedly lead to hangs or infinite reboot cycles once the DHCP or BOOTP servers are not longer available / reachable.

[5] Notice: the STONITH concept is more or less associated with short-distance scenarios where **crossover cables** or similare equipment are used. The assumption is that crossover cables can't go defective, or at least it would be an extremely unlikely scenario. For long-distance replication, this assumption is simply not true.

to anything you (believe to) know. Errare humanum est: nothing is absolutely for sure.

- Oversimplification: don't report an "unknown" or even a "broken" state for a complex system whenever a smaller subsystem exists for which you have some knowledge (or you can conclude something about it with reasonable evidence). Otherwise, your users / sysadmins may draw wrong conclusions, and assume that the whole system is broken, while in reality only some minor part has some minor problem. Users could then likely make wrong decisions, which may then easily lead to bigger damages.

- Murphy's law: **never assume that something can't go wrong!** Doing so is a blatant misconception at topmost level: the *purpose* of a clustermanager is creating High Availablity (HA) out of more or less "unreliable" components. It is the damn duty of both a clustermanager and its configurator to try to compensate *any* failures, *regardless of their probability*[a], as best as possible.

- Never confuse **probability** with **expectancy value!** If you don't know the mathematical term "expectancy value", or if you don't know what this means *in practice*, don't take responsibility for millions of € or $.

- When operating masses of hard- and software: never assume that a particular failure can occur only at a low number of instances. There are ***unknown(!)* systematic errors** which may pop up at the wrong time and in huge masses when you don't expect them.

- Multiple layers of fallback: *any* action can fail. Be prepared to have a plan B, and even a plan C, and even better a plan D, wherever possible.

- Never increase any damage anywhere, unnecessarily! Always try to *miminize* any damage! It can be mathematically proven that in deterministic probabilistic systems having finite state, increases of a damage level *at the wrong place* will *introduce* an *additional risk* of getting into an **endless loop**. This is also true for nondeterministic systems, as known from formal language theory[b].

- Apply the **best effort principle**. You should be aware of the following fact: in general, it is impossible to create an *absolutely reliable system* out of unreliable components. You can *lower* the risk of failures to any $\epsilon > 0$ by investing a lot of resources and of money, but whatever you do: $\epsilon = 0$ is impossible. Therefore, be careful with boolean algebra. Prefer approximation methods / optimizing methods instead. Always do *your* best, instead of trying to reach a *global* optimum which likely does not exist at all (because the ϵ can only *converge* to an optimum, but will never actually reach it).
  The best effort principle means the following: if you discover a method for improving your operating state by reduction of a (potential) damage in a reasonable time and with reasonable effort, then **simply do it**. Don't argue that a particular step is no 100% solution for all of your problems. *Any improvement* is valuable. **Don't miss any valuable step** having reasonable cost with respect to your budget. Missing valuable measures which have low cost are certainly a violation of the best effort principle, because you are not doing *your* best. Keep that in mind.
  If you have *understood* this (e.g. deeply think at least one day about it), you will no longer advocate STONITH methods *in general*, when there are alternatives. STONITH methods are only valuable when you *know in advance* that the final outcome (after reboot) will most likely be better, and that waiting for reboot will most likely *pay off*. In general, this condition is *not true* if you have a healthy hot standby system. This should be easy to see. But there exist well-known clustermanager solutions / configurations blatantly ignoring[c] this. Only when the former standby system does not work as expected (this means that *all* of your redundant systems are not healthy enough for your application), *only then*[d] STONITH is

unevitable as a *last resort* option.
In short: blindly using STONITH without true need during failover is a violation of the best effort principle. You are simply not doing your best.

- When your budget is limited, carefully select those improvements which make your system **as reliable as possible**, given your fixed budget.
- Create statistics on the duration of your actions. Based on this, try to get a *balanced* optimum between time and cost.
- Whatever actions you can **start in parallel** for saving time, do it. Otherwise you are disregarding the best effort principle, and your solution will be sub-optimal. You will require deep knowledge of parallel systems, as well as experience with dealing with problems like (distributed) races. Notice that *any* distributed system is *inherently parallel*. Don't believe that sequential methods can deliver an optimum solution in such a difficult area.
- If you don't have the **necessary skills** for (a) recognizing already existing parallelism, (b) dealing with parallelism at concept level, (c) programming and/or configuring parallelism race-free and deadlock-free (or if you even don't know what a race condition is and where it may occur in practice), then don't take responsibility for millions of € or $.
- Avoid hard timeouts wherever possible. Use **adaptive timeouts** instead. Reason: depending on hardware or workload, the same action A may take a very short time on cluster 1, but take a very long time on cluster 2. If you need to guard action A from hanging (which is almost always the case because of Murphy's law), don't configure any fixed timeout for it. When having several hundreds of clusters, you would need to use the *worst case value*, which is the longest time occurring somewhere at the very slow clusters / slow parts of the network. This wastes a lot of time in case one of the fast clusters is hanging. Adaptive timeouts work differently: they use a kind of "progress bar" to monitor the *progress* of an action. They will abort only if there is *no progress* for a certain amount of time. Hint: among others, `marsadm view-*-rest` commands or macros are your friend.

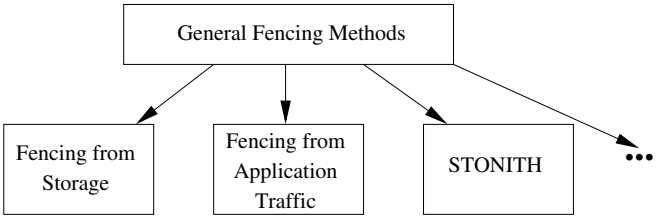
[a] Never claim that something has only low probability (and therefore it were not relevant). In the HA area, you simply **cannot know** that, because you typically have *sporadic* incidents. In extreme cases, the *purpose* of your HA solution is protection against 1 failure per 10 years. You simply don't have the time to wait for creating an incident statistics about that!

[b] Finite automatons are known to be transformable to deterministic ones, usually by an exponential increase in the number of states.

[c] For some *special(!)* cases of the shared-disk model, there exist some justifications for doing STONITH *before* starting the application at the hot standby. Under certain circumstances, it can happen that system A running amok could destroy the data on your single shared disk (example: a filesystem doubly mounted *in parallel*, which will certainly destroy your data, except you are using `ocfs2` or suchalike). This argument is only valid for *passive* disks which are *directly* attached to *both* systems A and B, such that there is no *external* means for fencing the disk. In case of iSCSI running over ordinary network equipment such as routers or switches, the argument "fencing the disk is otherwise not possible" does not apply. You can interrupt iSCSI connections at the network gear, or you can often do it at cluster A or at the iSCSI target. Even commercial storage appliances speaking iSCSI can be remotely controlled for forcefully aborting iSCSI sessions. In modern times, the STONITH method has no longer such a justification. The justification stems from ancient times when a disk was a purely passive mechanical device, and its disk controller was part of the server system.

[d] Notice that STONITH may be needed for (manual or partially automatic) *repair* in some cases, e.g. when you know that a system has a kernel crash. Don't mix up the repair phase with failover or handover phases. Typically, they are executed at different times. The repair phase is outside the scope of this section.

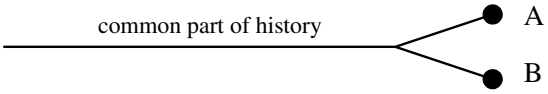
ITON = Ignore The Other Node This strategy means **fencing from application traffic**, and can be used as an alternative to STONITH when done properly.



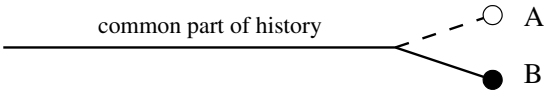
Fencing from application traffic is best suited for the shared-nothing model, but can also be adapted to the shared-disk model with some quirks.

The idea is simple: always route your application network traffic to the current (logically) active side, whether it is currently A or B. Just don't route any application requests to the current (logically) passive side at all.

For failover (and *only* for that), you *should not care about* any split brain occurring at the low-level generic block device:



Although having a split brain at the generic low-level block device, you now define the "logically active" and "logically passive" side by yourself by *logically ignoring* the "wrong" side as defined by yourself:



This is possible because the generic block devices provided by DRBD or MARS are completely **agnostic** of the "meaning" of either version A or B. Higher levels such as clustermanagers (or humans like sysadmins) can assign them a meaning like "relevant" or "not relevant", or "logically active" or "logically passive".

As a result of fencing from application traffic, the "logically passive" side will *logically* cease any actions such as updating user data, even if it is "physically active" during split-brain (when two primaries exist in DRBD or MARS sense[6]).

If you already have some load balancing at the network, or BGP, or another *mechanism* for dynamic routing, you already have an important part for the ITON method. Additionally, ensure by an appropriate *strategy* that your balancer status / BGP announcement etc does always coincide with the "logically active" side (recall that even during split-brain *you* must define "logically active" **uniquely**[7] by yourself).

Example 6.5: **Application fencing**

Phase1 Check whether the hot standby B is currently usable. If this is violated (which may happen during certain types of disasters), do not start failover for any affected resources.

Phase2 Do the following *in parallel*[a]*:*

- Start all affected applications at the hot standby B. This can be done with the same DRBD or MARS procedure as described in STONITH-like Methods.
- Fence A by fixedly routing all affected application traffic to B.

That's all which has to be done for a shared-nothing model. Of course, this will likely produce a split-brain (even when using DRBD in place of MARS), but that will not

[6] Hint: some clustermanagers and/or some people seem to define the term "split-brain" differently from DRBD or MARS. In the context of generic block devices, split brain means that the *history* of both versions has been split to a Y-like **fork** (for whatever reason), such that re-joining them *incrementally* by ordinary write operations is no longer guaranteed to be possible. As a slightly simplified definition, you might alternatively use the definition "two incompatible primaries are existing in parallel", which means almost the same in practice. Details of formal semantics are not the scope of this treatment.

[7] A possible strategy is to use a Lamport clock for route changes: the change with the most recent Lamport timestamp will always win over previous changes.

matter from a user's perspective, because the users will no longer "see" the "logically passive" side A through their network. Only during the relatively small time period where application traffic was going to the old side A while not replicated to B due to the incident, a very small number of updates *could* have gone lost. In fields like webhosting, this can be taken into account. Users will usually not complain when some (smaller amount of) data is lost due to split-brain. They will complain when the service is unavailable.

[a] For database applications where no transactions should get lost, you should slightly modify the order of operations: first fence the old side A, then start the application at standby side B. However, be warned that even this cannot guarantee that no transaction is lost. When the network between A and B is interrupted *before* the incident happens, DRBD will automatically disconnect, and MARS will show a lagbehind. In order to fully eliminate this possibility, you can either use DRBD and configure it to hang forever during network outages (such that users will be unable to commit any transactions at all), or you can use the shared-disk model instead. But in the latter case, you are introducing a SPOF at the single shared disk. The former case is logically almost equivalent to shared-disk, but avoiding some parts of the physical SPOF. In a truly distributed system, the famous CAP theorem is limiting your possibilities. Therefore, no general solution exists fulfilling all requirements at the same time.

This method is the **fastest** for restoring HA, because it doesn't try to execute any (remote) action at side A. Only from a sysadmin's perspective, there remain some cleanup tasks to be done during the following repair phase, such as split-brain resolution, which are outside the scope of this treatment.

By running the application fencing step *sequentially* (including wait for its partial successfulness such that the old side A can no longer be reached by any users) in front of the failover step, you may minimize the amount of lost data, but at the cost of total duration. Your service will take longer to be available again, while the amount of lost data could be *theoretically* somewhat smaller.

Details 6.2:

A few people might clamour when some data is lost. In long-distance replication scenarios with high update traffic, there is *simply no way at all* for guaranteeing that no data can be lost ever. According to the laws of Einstein and the laws of Distributed Systems like the famous CAP theorem (see section Explanation via CAP Theorem), this isn't the fault of DRBD+proxy or MARS, but simply the *consequence* of having long distances. If you want to protect against data loss as best as possible, and when you can afford it financially, then don't use $k = 2$ replicas. Use $k \geq 3$, and spread them over different distances, such as mixed small + medium + long distances. Future versions of MARS are planned to support adaptive pseudo-synchronous modes, which will allow individual adaptation to network latencies / distances.

The ITON method can be adapted to shared-disk by additionally fencing the common disk from the (presumably) failed cluster node A.

6.1.4.2. Handover Methods

Planned handover is conceptually simpler, because both sides must be (almost) healthy as a *precondition*. There are simply no pre-existing failures to deal with.

Here is an example using DRBD, some application commands denoted as pseudo code:

1. at side A: `applicationmanager stop all`
2. at side A: `drbdadm secondary all`
3. at side B: `drbdadm primary all`
4. at side B: `applicationmanager start all`

MARS already has a conceptual distinction between handover and failover. With MARS, it becomes even simpler, because a generic handover procedure is already built in:

1. at side A: `applicationmanager stop all`
2. at side B: `marsadm primary all`
3. at side B: `applicationmanager start all`

When using the `systemd` interface of `marsadm` (see `mars-user-mnaual.pdf`), this can be shortened into only one command:

1. at side B: `marsadm primary all`

6.1.4.3. Hybrid Methods

In general, a planned handover may fail at any stage. Notice that such a failure is also a failure, but (partially) caused by the planned handover. You have the following alternatives for automatically dealing with such cases:

1. In case of a failure, switch back to the old side A.
2. Instead, forcefully switch to the new side A, similar to the methods described in section 6.1.4.1.

Similar options exist for a failed failover (at least in theory), but chances are lower for actually recovering if you have only $k = 2$ replicas in total.

Whatever you decide to do in what case in whatever priority order, whether you decide it in advance or during the course of a failing action: it simply means that according to the best effort principle, you should **never leave your system in a broken state** when there exists a chance to recover availability with any method.

Therefore, you should *implement* neither handover nor failover in their pure forms. Always implement hybrid forms following the best effort principle.

6.1.5. Special Requirements for Long Distances

Most contemporary clustermanagers have been constructed for short distance shared-nothing clusters, or even for *local* shared-nothing clusters (c.f. DRBD over crossover cables), or even for shared-disk clusters (*originally*, when their *concepts* were developed). Blindly using them for long-distance replication without modification / adaptation bears some additional risks.

- Notice that long-distance replication always *requires* a **shared-nothing** model.
- As a consequence, **split brain** can appear *regularly* during failover. There is no way for preventing it! This is an *inherent property* of distributed systems, not limited to MARS (e.g. also ocurring with DRBD if you try to use it over long distances). Therefore, you *must* deal with occurences of split-brain as a *requirement*.
- The probability of **network partitions** is much higher: although you should have been required by Murphy's law to deal with network partitions already in short-distance scenarios, it now becomes *mandatory*.
- Be prepared that in case of certain types of (more or less global) internet partitions, you may not be able to trigger STONITH actions *at all*. Therefore, **fencing of application traffic** is *mandatory*.

7. Advice for Managers and Architects

7.1. Maturity Considerations for Managers

7.1.1. Maturity of Architectures

Manager Hint 7.1:

Instances of storage system *architectures* (see section What is Architecture) typically have a **lifetime** of **decades**.
While implementations / components / storage vendors etc can often be exchanged or updated more frequently (typically lifecycles of 3 to 5 years for CAPEX reasons), **fundamental architectures** are much less flexible to change, and thus are *forcing* you into a **long-term strategy**.

In contrast, certain hardware technologies have a much lower lifetime, typically between 1 and 2 years. New server hardware / new disks / SSDs etc are hitting their market all the time, like waves in the ocean.

System software technologies (OS layer) typically have a lifetime inbetween hardware and architecture lifetimes. Although their update cycles / minor release cycles are typically even faster than hardware releases, their *fundamental product appearance points* are rather stable[1]. For example, the Linux kernel is now more than 20 years old, while its *fundamental architecture* has been copied from Unix and is now almost 50 years old.

Certain advocates are arguing with the *current* status of maturity of *components*. In a long-term business operated by professionals, there is an observable long-term trend:

> **Maturity of components is (almost) always improving over the years.**

Of course, maturity is important. In sensible areas, so-called "banana software" may even kill you. In such a situation, the *current* maturity status is important. However, once an implementation is *mature enough*, and/or once only some nice-to-have features are deservable, the long-term maturity trend / forecast of implementations / components is more important than the current status. You can influence this with your **long-term investment decisions**.

There exists something which is even more important:

> **Maturity of fundamental architectures is most important, because they *cannot* improve. Architectures need to be right from scratch.**

This is similar to mathematics: Pythagoras' theorem or Einstein's laws cannot be improved. They will last forever. At most, they can get old-fashioned or otherwise **outdated** / obsoleted. However, there are other chances and **opportunities**:

- New / better architetures may appear (rarely).
- Implementations of architectures should evolve slowly over time.
- Implementations may slowly migrate to other architectures, or even support multiple architectures at the same time (convergence properties).

[1] Appearance of certain technologies may occur in **hype cycles**, caused by *social* effects. While there are founding wa ves for (sometimes similar) product classes, other solution appearancesare more evenly spread over the decades. For example, appearance of many Unix clones / descendants appears to rather smoothly distributed over half a century.

Manager Hint 7.2: **General advice**

Pay more attention to fundamental architectures. Develop a long-term strategy for maturity of components and implementations.

7.1.2. Maturity of MARS

Notice that MARS itself is just a component. For a fully functional system, you will need some more infrastructure at several layers.

- **MARS** itself is in production since 2013, and on mass data (several petabytes) since 2014. MARS itself is *generic*, and can be used for a multitude of Linux application stacks.
- A **cluster manager**[2] is typically also needed for mass installations. You can use the `systemd` template engine of `marsadm`, see `mars-user-manual.pdf`, which is easily configurable by Linux sysadmins.
- Tpically, **monitoring** is anyway specific for each application stack. Adding some simple Icinga scripts or similar should be no problem for professional Linux admins.
- Automatic **mass deployment**: this is anyway specific for the deployment system used for your system plus application stack. At the moment, plugins for generic solutions like OpenStack etc are missing. This is an opportunity for other OpenSource projects!
- The **Football framework** is in mass production at 1&1 Ionos ShaHoLin since 2018. It has some plugin for driving the `systemd` cluster manager. Its plugin architecture should allow easy adaptation to other system and application stacks.
- Another opportunity for OpenSource projects: some web-based point-and-click **dashboard** similar to the Ceph Dashboard, but displaying and controlling sharded LVM pools which are replicated via MARS, and also controlling Football, would be a highly appreciated addendum.

7.2. Recommendations for Hard- and Software Project Setup

Big enterprises are often binding their technical projects (whether developmental or operational ones) to *specific* products, or to *specific* platforms. In addition, inter-team organisational structures are tending to *fragmentation*. This can can easily produce lots of **missed opportunties** for **synergy effects**.

Manager Hint 7.3:

In the storage field, missed synergy effects from projects are often creating considerable ***direct* cost**. For a total of petabytes, this can easily sum up to some millions. ***Indirect* long-term cost**, including **insufficient flexibility** for the market, can be even higher.

This section hints you at some countermeasures.

7.2.1. Hardware Projects and Virtualization

This section hints you at several pitfalls, which may result from misconceptions.

[2] 1&1 Ionos ShaHoLin uses a self-built proprietray cluster manager called `cm3`. It works only with the internal 1&1 database infrastructure, and is not generic.

7.2.1.1. Physical Hardware vs Virtual Hardware

In theory, server hardware is independent from system software. For example, you may install both Windows and Linux onto the same server iron. In practice, however, each software application stack may have *different* requirements for ...

- CPU power
- RAM size
- IOPS demands

... independently from storage, whether it would be local one, or remote storage over network. In order to save cost, several companies are using **virtualization**.

Details 7.1: **Capabilities of virtualization**

Several people are believing that virtualization will *generally* improve things. While this is often true, there are *exceptions*.

For large applications requiring a lot of CPU and RAM, such as big databases, or masses of smaller databases, or webhosting with PHP as a primary consumer of resources, virtualization will *not* magically give you more resources. It can just *dynamically re-distribute* existing hardware resources across the same hypervisor iron, without magically creating new resources out of thin air.

Do not try to virtualize a system which is *already virtualized*. This can only be counter-productive. Many people do not know that **classical UNIX processes**[a] are also a form of virtualization. When your system is already at its limit when carrying masses of conventional processes (e.g. by dynamically scaling the number of daemons / server processes), an additional KVM layer or *masses* of docker instances (lesser with an LXC layer or a *low* number of docker instances allowing resource sharing in the kernel) will *not* speed up your existing processes, but in contrary, will likely lead to **density regressions**.

Do not neglect the **overhead of virtualization**. Running several dozens to hundrets of KVM instances on one iron will consume a lot of RAM overhead, while the same amount of LXC containers is typically cheaper. For CPU overhead, the picture is similar, but typically less stronger, provided that CPU overbooking is *very moderate*. When overbooking CPU too much with KVM / qemu (or commercial alternatives like vmware), so-called **steal overhead** can grow considerably, depending on various influences.

Do not expect linear behaviour: steal overhead can *amplify itself* in various situations, and hardware-based SMP systems can also go into **RAM thrashing** / multilayer **CPU cache thrashing** when overloaded with too big workingsets (cf. section Explanations from DSM and WorkingSet Theory).

The so-called **noisy neighbour problem** has been publicly advocated a few years ago, thus it is known by more people. However, it is only a special sub-problem of more general workingset problems.

[a] Originally, processes have been invented at the beginning of the 1960s for better exploitation of expensive physical resources, originally by providing multiple "virtual computers" to *different* users. Later, the concept of "communicating sequential processes" (Hoare) become popular as a structuring aid for the *same* user, which is now standard, and has been extended in various ways.

Manager Hint 7.4: **Capabilities of virtualization**

Avoid the above detail problems, which can lead to **serious cost increase** (both direct and indirect cost), by careful checking in advance.

Let the check done by skilled experts who know what a workingset is, and how to measure it, and how to workaround corresponding problems.

7.2.1.2. Storage Hardware

It is easy to miss opportunities for cost savings, or even to produce *massive regressions* by *factors*, by **unexpected side effects** of management decisions.

Manager Hint 7.5: **Missed: architecture had to follow organization**

A frequent mistake is to organize teams or departments by introduction of a border between "storage admins" and "sysadmins", and assigning them more or less complementary technical responsibilities. Typical arguments can be heard that each could then better *concentrate* at his speciality.

What looks like a "good idea" at first glance, will likely prevent several cost-saving models like `FlexibleSharding`, see sections Variants of Sharding and FlexibleSharding. As explained there, this can **increase cost** by factors, and **reduce reliability** considerably (see section Optimum Reliability from Architecture).

Similarly: creating a department (or a team) which is **responsible for the whole storage of a divison or of the company** is a very bad idea. It will **bind you for decades**, likely to either cost-intensitive commercial storage appliances (depending on the gusto of involved people, see section Local vs Centralized Storage), and/or to some `BigCluster` architecture. It simply means that only **network-centric storage hardware** can be used in practice, and that an expensive storage network becomes mandatory in practice (otherwise capacity planning etc could become difficult). Other types of storage will become almost impossible. Changing such an architecture for some petabytes of data will be very cumbersome and time-consuming.

Manager Hint 7.6: **Better: organization follows architecture**

Always consider alternatives, and determine / estimate their TCO for at least 5 years, better 10 years. You need to include **migration cost** when both EOL storage hardware and EOL server hardware has to be replaced by newer one (hardware lifecycle).

Notice: the `FlexibleSharding` model is naturally well-suited for VMs of various types. If you want to splice the overall IT responsibility, then the **VM layer** is *typically* a much better *candidate* than introduction of a dedicated network-centric storage layer.

7.2.2. Software Project Recommendations

On one hand, software *appears* to be easier exchangeable than masses of hardware. However, this only applies to *components* in practice. More complex software stacks or networks are

typically too complex, and are often containing lots of **hidden dependencies**.

In this section, we will look at various obstacles where software, and in particular the **fundamental architecture** of software, is **limiting flexibility** and producing **unnecessary cost**.

The scope of this section is exceeding the storage area. Most of the given advice will also apply to more general enterprise software.

7.2.2.1. Usefulness Scope of Software

Manager Hint 7.7:

A very important property of software: after it is **written once**, in general it can be **instantiated many times**.

While creation of copies of tangible goods typically costs a lot of effort and money, software copies are costing *almost nothing*. This is a major source of **cost saving potential**, while at the same time **improving quality** as explained below.

Observations from the whole industry, not very specific for a single company: in practice there exists *lots* of software which actually is installed only *once*. Most of it is constructed in such a way that it *cannot* be easily installed another time, or suchalike would not be useful, because it is **firmly bound** to a **singleton instance**.

Example 7.1: **Singletons**

So-called "enterprise databases" which often have their own enterprise-specific databasse schema, or even their own **product-specific schema**. Much of the software / scripts around them makes only sense for this particular schema.

Example 7.2: **Workarounds for incompatibilities**

So-called "middleware[a]" is often translating and adapting between multiple singletons. It makes no sense to instantiate this type of "middleware" somewhere else.

Another frequent ill-design is placement of **business logic** in so-called "middleware". According to Dijkstra's layering rules (see section Layering Rules and their Importance), business logic should get its own layer, independently from cross-platform concerns (aka **separation of concerns**).

[a] This usage of the term "middleware" is *incorrect* in strong sense. The original goal of middleware was providing **universally generic** marshalling and translation of data formats between "incompatible" "platforms" (where nowadays the latter term often is also used incorrectly, because a "platform" is a **stable interface** / foundation for a ***multitude* of application *classes***).

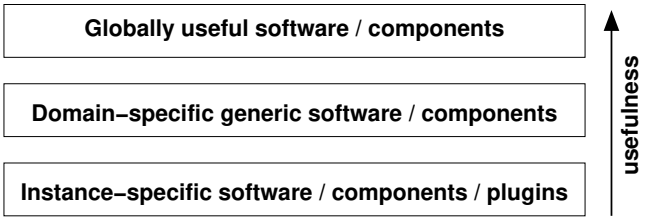
Manager Hint 7.8:

From the discipline of software engineering[a]: **non-instantiable singletons** are an **indicator** of ***poor software design*** *and practice.*

Likely, your competitors will have similar problems, often without noticing them. If you are the first to **overcome them in long term**, you will get an **advantage**.

[a] Explanation: software engineering as a discipline has the *opposite* goal of *maximizing* several important KPIs of software.

The usefulness of software and/or of its components can be roughly classified as follows:



Example 7.3: Globally useful software

The Linux kernel is installed at several *billions* of instances. From the biggest server, on supercomputers, down to *billions* of smartphones, and on tiny IoT gadgets. In order to support such a wide variety of hardware, it is **highly customizable** through thousands of compile-time config options, and lots of runtime options. Additionally, it has a high degree of automatic adaptation to hardware components, **automatic self-configuring**, etc. Its userspace API does not only support classical libc-based Unix software, but also the completely different execution engines of smartphones, and much more.

Example 7.4: Domain-specific generic software

Football (see `football-user-manual.pdf`) is domain-specific in the sense that it is only useful for sharded storage, but not for BigCluster storage. Its main part is generic, since it is **extensible via plugins**. For usage in other application areas than currently in production, some new plugins might be necessary.

Example 7.5: Instance-specific software

Tetris is the 1&1-internal name for the instance-specific customization plugin of Football. It is only useful at the 1&1 Ionos ShaHoLin software instance.


In general, most instance-specific software is *not* based on higher usability levels. Then the *whole* invest is practically not re-usable.


Tetris is an example how divergent requirements from broader usefulness desires can be combined with instance-specific requirements.

Manager Hint 7.9: Maximizing the usefulness KPI


In general, all three usefulness classes are needed for a healty enterprise. It is not possible to operate your business purely with "globally useful software".
You can **maximize the overall usefulness** by using *as much* from the upper classes *as reasonably possible*.


For example, you can make an inventory of all your software assets, including(!) free ones from OpenSource your people are just downloading and installing, and **evaluate the usefulness** according to the above classification, then determine the **number of instances** for each asset, and finally **create a weighted**[a] **KPI** out of it.


It is *critical* to **not forget *external* OpenSource** assets which cost *nothing*, but heavily contribute to your business value, and/or contribute by risk reduction, etc. Beware of SAP & relatives, typically there exists no inventory for them.


You might derive further sub-KPIs, such as per-asset TCO, or business value, or risk indicators, etc.


As a side effect, you will likely find much more opportunities for long-term improvements of your enterprise than you can implement in short term. Evaluate their **potential**, and **prioritize** accordingly.

[a] The "size" or "development effort" for software components needs to be taken into account. They can vary by some orders of magitudes. Treating them as "equal-sized bricks" would massively overemphasize tiny helper scripts. Since there is often some binary-only proprietary software, a possibly weighting method could be the *installed binary size* in bytes. This will also lead to distortions, but typically less significant than "uniform bricks". Theoretically, you could discriminate between code and data (e.g. images), but this might lead to a high effort for inventory. Simple solutions are better in practice. Exceptional corrections can be applied when distortion are getting too high in certain places.

As a manager, the big question is: *how* can you achieve better usefulness in **long term**? Just use a KPI, or are there further aspects not modeled by KPIs?

For a better background, have a brief look at the following classification of **architectural potentials**.

7.2.2.2. Architectural Levels of Genericity

Managers only interested in an overview may skip the rest after the first graphics, showing 3 different levels of genericity. Architects should *not* skip the examples.

Here is a classification of **genericity** according to its **re-use potential**.

Manager Hint 7.10: **Genericity and re-use**

Re-use means that each time something needs to be implemented, or each time some requirements are changing, some new software components need not be implemented from scratch, but already existing components / parts are just **recycled** and used in a different way or in a different context.


In general, components / parts **need to be constructed for re-use**. When not prepared for re-use, artefacts will be less useful, or even not useful for re-use at all.

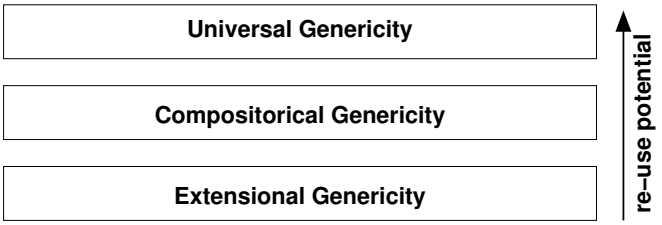

A good way for re-use preparation is **genericity**. It means that something is only "prepared for use", by providing some *concrete interface* for both use and re-use, such that any concrete usage is *relatively easy*.
In other words: although the *first use* is slightly more expensive because of intermediate introduction of genericity and its documented or *self-documenting(!)* interfaces, *any* later *re-use* will then be **cheaper** than making everything from scratch again. When re-use is executed frequently enough, **investments into genericity** will **pay off rapidly**.


If you are **unsatisfied** with **software development productivity** in your company, consider the following. You need to **explicitly request** a certain level of genericity as a **preparation for long-term re-use**. Otherwise, you likely won't get it.


Reason: people want to finish their current projects *as fast as possible*, typically *missing* important opportunities for preparation of re-use (provided they have the necessary skills). This behaviour is often heavily amplified by **deadlines**.

The following can be used to classify not only the genericity of software itself or of programming

styles, but also of **software architectures** (see section What is Architecture). The biggest potential of genericity is when applied at architectural level:



1. **Universal genericity** means that potentially an **infinite** number of re-usage variants (potential: ∞) can be derived **easily**, by **configuration** and/or by **convention**. A few examples:

Example 7.6: Unix files

Invented in the 1970s, Unix files are extremely universal. They can hold *anything*, from simple ASCII text to executables, and to complex database containers. This is possible by a **universally generic representation**: a file is nothing but a sequence of bytes[a], with an arbitrary length, which can change dynamically at runtime.

The genericity of Unix files is a striking example that sometimes **less code is more value**! Unix files are **simpler** than the **unnecessary complexity** of historical record-based predecessor file concepts.

The **only invest** for exploitation of fruitful generic simplification: **careful thinking** before starting an implementation, best from experienced software architects / experts. This can save you up to *factors*!

[a] Predecessor filesystems were typically more complex, e.g. a file was a sequence of *records*. There was a variety of variants, like fixed-length records, variable-length records, indexed records, etc. These had further problems, because the *byte* was not yet standardized as exactly 8 bit. There were 6-bit bytes, or 12-bit bytes, etc.

Example 7.7: Business process languages

Business process languages like BMPL and their execution engines are modern universally generic systems, but typically used for domain-specific purposes. There you can see that both concepts usefulness vs genericity are *orthogonal* to each other by some degree.

Example 7.8: Universal compilers / interpreters

LISP is one of the eldest programming languages in the world, invented 1959. It can be used to express **any mathematical problem**, while only a *subset* of them is actually **computable**, even by modern supercomputers. While the Unix file as such is just a *passive* item and thus not prone to computability problems, active items like LISP programs are **Turing Complete**, which is a two-sided sword in practice. Although *extremely capable*, it is not easy to understand and to control. Many modern **IT risks** (e.g. security risks) can be deduced from Turing Completeness.

There are universally generic compilers and interpreters, for example parser

generators, which are *not* Turing Complete by their basic configuration language.

Example 7.9: **Macro mechanisms and C++ templates**

Parameterization can be done via C preprocessor macros, or C++ templates, or other macro processors. Macro substitution can not only be applied to programming languages, but also to configuration data. An example is the `systemd` interface of `marsadm`, see `mars-user-manual.pdf`. It suffices to define a certain `systemd` unit template only once, and then let it automatically instantiate for hundrets or thousands of LVs and their application stacks.

Manager Hint 7.11: **Recommendation**

Universal genericity has the **highest potential**, and should be always considered for *passive* use cases. Several *active* systems, however, bear a relatively high risk when Turing Complete, when not developed and maintained and operated by **highly skilled staff** which can *really* deal with their complexity, and who are **really knowing what they are doing**.

2. **Compositorical genericity** is similar to the composibility of LEGO bricks: via a more or less **uniform standard interface**, numerous re-combinations / compositions can be easily created. Its potential is similar to **permutations**, thus factorial: $O(n!)$.

Example 7.10: **Pipe and filters style**

A good example is an architectural style called **pipe and filters style**, which is the heart of the Unix Philosophy. In the original Unix concept, a relatively *low* number of simple[a] basic operators were used for creation of an extremely wide variety of complex data processing pipelines.

There is a programming language which directly supports this style, called Bash Script in its modern version.

[a] Modern Unix-like systems including GNU/Linux have much more complex operators, some with hundreds of options. Nevertheless, they can also be used for compositorical genericity.

Example 7.11: **Stacked block devices**

Linux has inherited the concept from Unix. In Unix, "everything is a file", and thus Unix devices are also *represented* as a file. Block devices are a special case, where only certain access granularities like multiples of *sectors* are possible. Modern Linux has augmented the concept with several special operations, such as `BLKDISCARD` and other `ioctl()` syscalls. Nevertheless, block devices are stackable, for example for creation of software RAID. Stacks are very flexible, for example you may place MARS on top of LVM on top of software RAID, or in a different order, or you may insert SSD caches at various positions, etc. The number of *potential* combinations is very high.

For *usage* of stacked block devices, you don't need to be a programmer. Exploiting compositorical genericity is possible from sysadmin space.

However, *creation* of a new stackable component is a completely different story. Linux **kernel programming** requires completely different skills, and even

among kernel hackers a junior level is all else but sufficient[a]. As a manager, do not confuse these HR requirements!

[a] C programming is *one* of many *preconditions* for kernel hacking. It is however not sufficient. The Linux kernel is a technical universe in itself. While many userspace C programmers need not deal with **concurrency**, or only with harmless standard cases, kernel programmers need to know and have experiences with about a *dozen* of different concurrency models and their concrete implementations. This is required for SMP scalability, weak memory semantics / memory barrier hardware operations, RCU, and much more, in addition to classical interrupt-driven concurrency models.

Example 7.12: **Electrical engineering**

Electrical engineers have used compositorical genericity even before the digital computer had been invented. Their **wiring diagrams** are connecting basic **functional units**, for example transistors or resistors, or whole sub-circuits.

Manager Hint 7.12: **Established use case for compositorical genericity**

By using Linux, you automatically get it via `ssh` commandlines used by sysadmins. Experienced Linux seniors will confirm that its **automation potential** is beyond anything having a graphical point-and-click interface. System administration for several hundrets or thousands of servers would be an extreme effort, or almost impossible otherwise.

Manager Hint 7.13: **New use cases for compositorical genericity**


There are much more use cases where compositorical genericity would be extremely beneficial. Its potential is $O(n!)$ where n is not the number of developers, but the number of functor instances[a].


The biggest practical obstacle is that too few people know of its enormous potential, and even less people have practical experiences with it in larger scale systems, such as Distributed Systems. When you have few excellent people with the necessary skills, don't force them to use so-called "standard paradigms" like OO, but let them exploit the much higher potential of compositorical genericity. Often, they won't be able to do so unless you help them by creating a special friendly working environment.


Don't be surprised when a single developer shows a **productivity** roughly equivalent to 10 conventional OO developers, or even more.


Do not confuse the roles of sysadmins with the roles of developers. Just because sysadmins usually are more used to pipe and filters style, this does not magically convert them into developers. A developer for compositorical genericity at large scale needs to know much more, at least at a **master's level in computer science**, if not at a PhD level.


Do not populate a team with OO addicts or with people who don't have the necessary skills, if you want to exploit the potential of compositorical genericity. Ask the inevitable *experienced technical leader*, who else may have the necessary skills, in order to qualify as additional team member. There exists practically no standard hiring profile at the job market.

[a] In general, a functor of a certain type can be instantiated several times, even in the same pipeline.

3. **Extensional genericity** means that an existing component needs to be re-used by *extending* it. Its potential is only $O(k)$ where k is a constant depending on your development resources.

Example 7.13: **Classical OO = Object Orientation**

No detailed explanation necessary, because many people already know what **OO inheritance** is, and have some experiences with it.

 Typically, programmer skills are required for non-trivial large-scale systems. Pure sysadmin skills are often not sufficient.

 There are lots of programmers at the job market, qualifying for OO. However, many of them are often lacking some sysadmin skills when HA operations is required. Thus a *mixed team* with both skill sets is something you should consider for **enterprise-critical** application stacks. In addition, automated testing is highly recommendable.

Manager Hint 7.14: **Classical OO = Object Orientation**

Probably you are surprised that classical OO inheritance has the *least* potential, only $O(k)$, while alternatives are much better, e.g. $O(n!)$ or $O(\infty)$.
Reason: for any new OO functionality, some skilled programmer has to write some program code, which needs to be tested and made production-ready.

 Thus real-life OO productivity is often lower than promised by advocates.

 In general, programming language paradigms are *orthogonal* to levels of genericity. For example, compositorical genericity can be implemented with OO languages.

Example 7.14: **Genericity in the Linux kernel**

The Linux kernel has more than 20 millions of lines of code written in C. Many people are regarding C as an imperative language, some even condemning it as "high-level assembler". However, the kernel has many parts like stackable filesystems where OO techniques are used. Several parts, like the dm = device mapper infrastructure, are more or less following many principles from compositorical genericity. Universal genericity is also present, for example in firewall rules execution engines. Few people seem to know that even FP = Functional Programming style is possible in C, if you know how to do it.

 Good C programming requires some skills. People who *really* have those skills are reaching a similar productivity than with other programming languages. Notice that C has some unique application areas where other languages are practically out of the game[a], such as kernel and deep system programming.

 A good programmer is treating programming languages as **tools**, which have *no global* pros and cons, but each of them is more or less well-suited for each specific

application area.

[a]Several years ago, some Java advocates were claiming that operating systems would be better written in Java, thus C will vanish in the long term. This has not become true. Reason: it is not *reasonably* possible to write a JVM = Java Virtual Machine in Java, while all major JVMs are written in C.

7.3. From OpenSource Consumers to Contributors to Leaders

The basic idea of OpenSource is very simple:

Manager Hint 7.15: Fundamental idea of OpenSource

Several competitors and enthusiasts are meeting together in a common neutral playground, also called **commons** or **common land**. Each is contributing something useful to the commons.

As a *result* of collaboration, *each* of them is **getting back more value** than *each* of them have contributed.

OpenSource is much more than a particular component. In fact, it is a **whole ecosystem**.

This means: by definition, only globally useful software (see section Usefulness Scope of Software) can qualify as OpenSource commons. In some cases, domain-specific generic software may qualify also, but this needs to be checked.

This *usability gap* leaves you an opportunity for **company-individual or product-specific customization** even of your own OpenSource components, provided you manage to get an appropriate degree of genericity (see section Architectural Levels of Genericity).

Manager Hint 7.16: Best areas for OpenSource

Several people are fearing that OpenSource might help their competitors too much.

When OpenSource is used for **basic infrastructure** instead of finished competitive products, then a **win-win** situation is always **improving competitiveness**.

There are a few *examples* where even giving away a *full product* can improve competitiveness. An example is Google Android.

In addition, each competitor may make better business by **strengthening a whole ecosystem**, e.g. attracting more customers in total, etc.
In particular, this can make sense when competition is more between *whole ecosystems*[a], than between individual companies.

[a]Example: today there is an increasing competition between the webhosting ecosystem (including public blog software like WordPress), and account-based social media (e.g. Facebook & co).

Example 7.15:

The vast majority of companies will profit from the Linux kernel, because selling OS software is *not* their core business.

Details 7.2:

Several ecosystems are already **dominated by OpenSource**. Commercial competitors would not have a chance anyway, because the **world-wide total productivity** (e.g.**scaling effects**) of OpenSource is unbeatable in such areas.

Example 7.16:

MARS would not have a chance for long-term survival if it weren't OpenSource.

How to take advantage of OpenSource? The OSAMM = Open Source Adoption Maturity Model is explained in simplified form at https://baloise.github.io/open-source/docs/md/goals/uplift.html. More context can be found in Lofi Dewanto's presentation https://drive.google.com/file/d/1GHLogE3ibdyjPaYfK_O4ELVtvUcE051R/view, in particular slide 24. There are 3 levels of OpenSource adoption which are interesting for most companies:

1. **Use**. Typically, OpenSource software is just downloaded, possibly compiled (depending on development model), and installed.
2. **Contribute**. Some code / documentation / feedback is flowing from in-house users back to the public project.
3. **Champion**. Somebody in the company has a leading role in the public project, and is thus leading a **movement**.

Manager Hint 7.17: **Important OpenSource specialities**

 There are certain misconceptions about OpenSource, which can lead to **fatal failures**. Here are some extremely important explanations:

- OpenSource is more about a *movement* than about the "software as such".
- Several important OpenSource projects, like the Linux kernel, have been **founded by individuals** and *not* by companies. Such projects are following **different rules than company projects**.

Manager Hint 7.18: **Rules in personal OpenSource projects**

 Managers who don't know the written and un-written rules of **personally led OpenSource projects** can easily create **substantial damage**, up to the destruction of a (sub-)project.

 The principles behind OpenSource movement rules can be found at Eric Raymond's articles from the 1990s and early 2000s. You need to understand that OpenSource communities are a **gift culture**, aka **meritocracy**.

Example 7.17: **Personal leadership in Linux**

Practically everybody knows that Linus Torvalds has founded the Linux kernel. His name is even encoded into the project name.

 As you can read at http://www.kernel.org, there is no chance to submit a patch originating from a company. Linus and the kernel hackers will simply ignore it. Only patches submitted by *individuals* are acceptable at all. It would be bad style to argue "you must accept this patch because I am from company XYZ, and I am paid by my company to create this patch". Even if the company name had three capital letters, it wouldn't help.

Example 7.18: **Kernel modules like MARS**

 As mentioned above, MARS would have no long-term chance for survival unless OpenSource. Since it is a Linux kernel module, it ***cannot exist* independently from Linux**.

 Consequence: anyone who wants to work at the core of the MARS project ***must accept* the same rules** as for the Linux kernel[a].

 Eric Raymond's famous articles need to be obeyed, too. For example, as a company you *cannot decide* to replace the founder of the project (which started only upon personal initiative and *not* as a company project), with another person. Otherwise, the public OpenSource project would be either dead, or it would necessarily lead to a project fork. Only one of the forks could survive in long term, and be included into mainstream Linux. Which one will become clear to you, once you have read Eric Raymond's articles (if you cannot guess it anyway from terms like **meritocracy**).

[a] Example: `grsecurity` was *technically* a sub-project of Linux, but did not comply to the rules of the Linux community. Therefore it failed in 2017, after more of a decade of OpenSource activity. Some of its remains are now migrated into mainstream, but not by the original founder of the technical sub-project.

7.4. Recommendations for Design and Operation of Storage Systems

7.4.1. Recommendations for Managers

When you are responsible for **masses of enterprise-critical data**, the most important point is to get people with **the right skills**, in *addition(!) to* the *right mindset*, and to assign the right roles to them.

Practical observation from many groups in many companies: which storage systems / architectures are in use, and how much they are *really* **failure resistent** and **reliable**, and how much they are *really* **scalable** for their workload, and what is their **TCO = Total Cost of Ownership**, does often *not* depend on real knowledge and on facts. It often depends **randomly** on **personal habits** and **pre-judgement** of staff[3].

 In essence, this results in a **gambling game** how safe / cost-effective etc your critical data *really* is.

 In particular after company mergers, suchalike varieties need not remain a permanent disadvantage. You may turn it into an advantage. Once you have enough reliable and validated

[3] This can be seen in a bigger company (e.g. after mergers etc) when very different architectures have been built by different teams for very similar usecases, although they are sometimes even roughly comparable in size and workload.

KPIs about each of the systems, and after you have checked that they are *really* comparable, you can derive a detailed comparison of competing architectures and/or of their actual implementations. Then you may start **merging** some of the technical platforms, provided there is a business case for it. Or, you may **bleed out** some old / obsolete technology.

When the game is about building up **new functionality** from scratch, it is much different. There are two main possibilities:

1. check whether your *best* platform can be extended with the new functionality. Good architectures are also **easily extensible**.
2. build a new platform.

The rest of this section focusses on architecture of *new* platforms. Always check whether existing *experience* can be re-used.

 As explained throughout section Scalability Arguments from Architecture, there are many pitfalls, and there are only few people who know them, because more people are working in small-scale systems than in large-scale enterprise ones. There are so many lots of people at the market who *claim* to have some experience, but in reality they don't know what they don't know (**second-order ignorance**).

Second-order ignorance is very dangerous, even for affected people themselves, because they are in good faith about their own skills, and that they would be able to control everything (sometimes they really want to control literally *everything*, even other people who have more real experience and knowledge). See for example wrong assumptions and "false proofs" about scalability, derived from different use cases (or even from workstation workloads). See the failed scalability scenario in section Example Failures of Scalability where some freelancers were consulted as "external experts".

Manager Hint 7.19: **Pitfall "false experts"**

 Check your information sources! There is a *systematic reason* for ill-informed "experts": the internet.

 On the internet, you can find a lot of so-called "best practices". Many of them propagating badly scaling storage architectures for enterprise workloads, sometimes even *generally* claiming they would "scale very well", which is however often based on *assumptions* instead of knowledge (and rarely based on *measurements* at the right measurement points for deriving substantial knowledge about your *real* application behaviour). Literally *anyone* can post incorrectly generalized "best practices" to the internet. Together with second-order ignorance about the non-transferability of "success stories" from usecase A to usecase B (resulting in *false "proofs"*), the internet is creating **information bubbles**.

Example 7.19: **Superfluous load balancers**

Good examples are HTTP or other IP-based load balancers placed in front of VMs. Almost always, this is an **expensive ill-design**.

 Notice: as long as *multiple* VM instances are hosted on *one* hypervisor iron, load balancers are most likely completely useless[a]. Instead, just assign more physical resources to a single VM. Only when the application load is *really* so high that 1 VM would fill up a hypervisor *completely*, only then a load balancer *might* be potentially useful. However, *first* check that there are enough RAM and SMP hardware threads. Only when state-of-the-art multi-socket CPUs with $\approx$ 128 or more CPU threads would be insufficient for a very high connection rate, and after tuning measures like PHP OpCache

were not sufficient, a load balancer or another means for load distribution *could* become necessary.

Even then, there are often more intelligent alternative solutions, like wide-area *distributed* **input traffic partitioning** to geo-distributed servers, in place of a central load balancer acting as a SPOF in a single datacenter. For example, source-IP based routing can partition global traffic into per-continent datacenters, drastically reducing application traffic latencies. In essence, this is coarse granularity sharding at global level.

[a] Reason: on SMP servers, there *already exists* a "load balancer". The kernel and its **process scheduler** can do even better than any external load balancer, by better distribution of physical CPUs to processes, and by exploitation of **shared memory**, for example shared filesystem kernel caches, such as the Dentry Cache, and the fscache / Page Cache. Exceptions would only occur when there were per-VM global bottlenecks, such as interdependent processes. For instance, it is easy to *misconfigure* Apache logfiles to become such a bottleneck. Just fix such misconfigurations, before claiming that SMP scalability would be limited.

In a nutshell: compared to the scalability of sharding, load balancers would be **only suitable for small-scale scalability**. However, small-scale scalability is much easier to achieve via hardware-based SMP = Symmetric MultiProcessing, at least in *most*[4] cases.

Never start a design with a load balancer *by default*. Only use load balancers when there is *well-founded strong evidence* that other scalability measures won't suffice. In particular, it needs to be very clear that sharding is really impossible, which in turn implies that there exists only 1 big customer, and that its data cannot be partitioned at all.

Manager Hint 7.20: Cost explosion by superfluous load balancers

Unnecessary load balancers are causing **follow-up cost by increased complexity**. In addition to the load balancer hardware and its setup / administration, *multiple* servers and/or VMs need to be set up and administered.

If you just need a redirection mechanism, read sections What is Location Transparency and Where to implement Location Transparency.

For example, the traffic from BGP = Border Gateway Protocol is executed by your **ordinary network routers**, without additional hardware, and they can distribute sharded traffic to wide-area geo-locations. In comparison, load balancers are just restricted **overkill**.

Never accept a system design with a *mandatory* load balancer. It will likely imply a BigCluster-like *architecture*, though typically only *implemented* as a SmallCluster.

Details 7.3:

Mandatory load balancers are often[a] creating some $O(n^2)$ behaviour,

[4] Personally, I have never seen a situation where a load balancer was really necessary. In all example cases, they were superfluous. In a few cases, they were even counter-productive.

showing up somewhere, often unexpectedly. Even when reduced to $O(n)$, load balancers are close to the **opposite of sharding** at *concept level*, because they try to *distribute* an *unpartitioned load* to servers needing **shared data** similar to DSM (see section 4.3.5), instead of first *partitioning the data* and thus also partitioning the corresponding traffic. Read section Error Propagation to Client Mountpoints about typical *real* scalability and reliability. When this doesn't help, read section Example Failures of Scalability where the load balancer was a major *source(!)* of massive scalability problems.

 Do not mis-use load balancer hardware for achieving location transparency. Suchalike would need to be called "load *redirector*" in place of "load *balancer*". You pay a lot of money for functionality you don't need, see also section Layering Rules and their Importance. Traffic redirection is both cheaper and more performant when executed by your ordinary network routers.

 **Sharding** architectures typically don't need any load balancers, although they are **massively scalable *horizontally***. Typically, they rely on the scalability of DNS, and of IP routing. Notice: when DNS would reach its scalability limit, then the internet as such would not scale anymore.

 In comparison, a load balancer is a SPOB = Single Point Of **Bottleneck**, where the traffic must physically **flow through** (thereby increasing hops and latencies), instead of dynamic wide-area routing.

[a] There are some rare potential exceptions, like **game servers** rendering scenes in **real-time**, consuming *massive* CPU and/or GPU power in relation to network bandwidth. Even there, sharding is often a better alternative. In contrast, ordinary video streaming typically consumes very low CPU power, because file streaming is executed by kernel `sendpage()` and partly offloaded to DMA hardware acceleration.

Manager Hint 7.21: **Load balancers vs sharding**

 As a manger, if you "buy" a *mandatory* load balancer, there is a high risk for **architecturally hindering long-term scalability** by sharding.

 Check whether people are *really* experts, when they want to solve suspected(!) scalability problems via mandatory load balancers. It is just poor system design, often inducing DSM problems, and producing unnecessary follow-up cost. Unfortunately, load balancers are systematically promoted by **internet information bubbles**.

 Real knowledge originates from evaluated sources, such as **scientific publications** which have undergone at least some minimum *quality check*, and which are trying to describe their preconditions and operating environments as precisely[5] as possible.

 Real experts will tell you when they don't know something. In addition, they will tell you *multiple* ways for obtaining such information, such as measurements, simulation, etc. In addition, real experts are able to do well-founded measurements

[5] Therefore, chances are better to get a real expert when he has some (higher) academic degrees, and was working in the area for a longer time.

and deriving forecasts from them. Later, when it works, their forecasts were roughly correct. Check the quality of forecasts afterwards!

If you don't have anyone in your teams who knows how **caching** *really* works, or if it is a single guy who cannot withstand the pressure from a whole group of "alpha animals", you are running an **increased risk** of unnecessary expenses[6], worse services (indirect cost), failed projects, and sometimes even resulting in loss of market share and/or of stock exchange value.

The problem is that it *looks so easy*, as if everyone could build a *large(!)* storage and/or application system, with ease. It looks easy once a small prototype is running at a workstation. Some people believe that "just spend some more money" would all which is needed. Unfortunately, both "marketing drones" from commercial storage vendors, and even a few OpenSource advocates, are propagating this **dangerous mindset**.

As a responsible manager, **how can you detect** dangerous partly knowledge?

Good indicators are wrong usage of the term "architecture" (see definition in section What is Architecture), and/or **confusion of architecture with implementation**. When somebody confuses[7] this, he does not really have an overview of different architectural solution classes. Instead, such people are tending to propagate their random "favourite solution" or their random "favourite product". For you as a responsible, this increases the **risk** of getting a non-optimum, or possibly even a bad / dangerous solution.

Another good indicator is advocacy of load balancers. See above boxes about the size of their real application area and their real value. Do not confuse people's belief with deep knowledge about Operating Systems and Distributed Systems. The latter also requires substantial theoretical background, in addition to practical experience.

Not everything which works in a garage, or in a student pool, or in the testlab (whether it's yours or from a commercial storage vendor), or in a PoC with so-called "friendly customers", is well-suited for large enterprises and their critical data (measured in petabytes / billions of files / etc), or is the optimum solution for TCO. Some rules of thumb, out of experience and observation:

- For each 1 or 2 orders of magnitude of the **size** of your data, you will need **better methods** for safe construction and operation, as would be sufficient for lower demands.
- For each 3 to 4 orders of magnitude (sometimes even for less), you will need **better architectures**, and people who can deal with them.
- For each 1 or 2 orders of magntitude of **criticality** of your data (measured by *losses* in case of certain incidents), you will also need better architecture, not just better components.

Manager Hint 7.22: **Important advice**

If you start a new platform from scratch, always **start with a *good* architecture**.

Once a platform is in production, even with a small number of customers, it becomes increasingly difficult to change its fundamental architecture. While bugs can be relatively easily fixed, and while single components can be exchanged with some effort, changing an architecture may turn out *close to impossible*, or at least very expensive.

7.4.2. Recommendations for Architects

In order of precedence, do the following:

1. **Fix and/or limit and/or tune the *application*.**
   Some extreme examples:

[6] I know of cases which have produced unnecessary *direct* cost of at least € 20 millions, not counting further indirect cost such as power and rackspace consumption.

[7] Notice that there exist people who use the term "architecture" inadvertly. They even don't even know that they are confusing architecture with implementation. Pure usage of a certain term is no clear indicator that somebody is really an expert.

- When you encounter a classical Unix **fork bomb**, you have no chance against it. Even the "best and the most expensive hardware[8]" is unable to successfully run a fork bomb. The only countermeasure is *limitation of resources*. Reason: unlimited resources do not exist on earth.
- If you think that this were only of academic interest: several types of internet **DDOS attacks** are acting like a fork bomb, and **Apache** is also acting similar to a fork bomb when not configured properly. This is not about academics, it is about *your survival* (in the sense of Darwin).
- If you think it cannot hurt you because you are running `fast-cgi` or another application scheme where forks are not part of the game (e.g. databases and many others): please notice that **network queues** are often acting as a replacement for processes. Overflow of queues can have a similar effect than fork bombs from the viewpoint of customers: they simply don't get the service they are expecting.
- If you think this cannot hurt you, because you are working in a completely different area from Apache: *any* type of IP-based network traffic can show queueing behaviour. Complex queuing systems can show "unexpected" behaviour, and sometimes even a dangerous one.
- Real-life example for application-level problems: some percentage of `WordPress` customers are typically and *systematically* **misconfiguring** their `wp-cron` cron jobs. They create backups of their website, which *include* their old backups. Result: in each generation of the backups, the needed disk space will roughly *double*. Even if you had "unlimited storage" on top of the "best and the most expensive storage system", and even if you would like to give "unlimited storage" to your customers, it simply cannot work at all. Exponential growth is exponential growth. After a few months of this kind of daily backup, you would need more storage than atoms exist in the whole universe. You *must* introduce some quota limits somewhere. And you *must* ensure that the `wp-cron` misconfiguration is fixed, whoever is responsible for fixing it.
- Another `WordPress` example: the `wp-cron` configuration syntax is not easily understandable by laymen. It is easy to **misconfigure** such that a backup is created *once per minute*. As long as the website is very small, this will not even be noticed by sysadmins. However, for bigger websites (and they are typically growing over time), the IO load may increase to a point until even asynchronous replication over 10Gig interfaces cannot catch up. Even worse: the next run of `wp-cron` may start before the old one has finished within a minute. Again, there is no chance except fixing the *root cause* at application level.

2. **Choose the right *overall* architecture** (not limited to storage).
   An impressive example for architectural (cf section What is Architecture) ill-design can be found in section Example Failures of Scalability. Important explanations are in section 4.4.2, in particular subsection Influence Factors at Scalability, and section 4.4.4 on page 71. A strategic example is in subsection Case Study: Example Scalability Scenario. It is absolutely necessary to know the standard cache hierarchy of Unix (similarly also found in Windows) from section Performance Arguments from Architecture. More explanations are in this manual at many places.

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

    Similarly for reliability: if you have problems with too many and/or too large incidents affecting too many customers, read sections Reliability Arguments from Architecture and Reliability Differences CentralStorage vs Sharding.

[8] There is an old joke from the 1980s: a Cray is a computer capable of running an endless loop in 10 seconds.

3. **Choice and tuning of components**.
   No further explanations necessary, because most people already know this. In case you think this is the *only* way: no, it is typically the *worst* and typically only the *last resort* when compared to the previous enumeration items. See example in section Example Failures of Scalability.
   Exception: choice of wrong components with insufficient properties for your particular application / use case, or even hard restrictions as mentioned in section What is Architecture. But this is an *architectural* problem in reality, and belongs to the previous item, not to this one.

A. Mathematical Model of Architectural Reliability

The assumptions used in the model are explained in detail in section 4.3.1.2 on page 51. Here is a quick recap of the main parameters:

- n is the number of basic storage units. It is also used for the number of application units, assumed to be the same.
- k is the replication degree, or number of replicas. In general, you will have to deploy $N = k * n$ storage servers for getting n basic storage units. This applies to any of the competing architectures.
- s is the architecture-dependent spread exponent: it tells whether a storage incident will spread to the application units. Examples: $s = 0$ means that there is no spread between storage unit failures and application unit failures, other than a local 1:1 one. $s = 1$ means that an uncompensated storage node incident will cause n application incidents.
- p is the probability of a storage server incident. In the examples at section 4.3 on page 49, a fixed $p = 0.0001$ was used for easy understanding, but the following formulae should also hold for any other $p \in (0, 1)$.
- T is the observational period, introduced for convenience of understanding. The following can also be computed independently from any T, as long as the probability p does not change over time, which is assumed. Because T is only here for convenience of understanding, we set it to $T = 1/p$. In the examples from section 4.3.1.2 on page 51, a fixed $T = 10,000$ hours was used.

A.1. Formula for DRBD / MARS

We need not discrimiate between a storage failure probability S and an application failure probability A because applications are run locally at the storage servers 1:1. The probability for failure of a single shard consisting of k nodes is

$$A_p(k) = p^k$$

because all k shard members have to be down all at the same time. In section 4.3.1.2 on page 51 we assumed that there is no cross-communication between shards. Therefore they are completely independent from each other, and the total downtime of n shards during the observational period T is

$$A_{p,T}(k, n) = T * n * p^k$$

When introducing the spread exponent s, the formula turns into

$$A_{s,p,T}(k, n) = T * n^{s+1} * p^k$$

A.2. Formula for Unweighted BigCluster

This is based on the Bernoulli formula. The probability that exactly $\bar{k}$ storage nodes out of $N = k * n$ total storage nodes are down is

$$\bar{S}_p(\bar{k}, N) = \binom{N}{\bar{k}} * p^{\bar{k}} * (1 - p)^{N-\bar{k}}$$

Similarly, the probability for getting k or more storage node failures (up to N) at the same time is

$$S_p(k,N) = \sum_{\bar{k}=k}^{N} \bar{S}_p(\bar{k},N) = \sum_{\bar{k}=k}^{N} \binom{N}{\bar{k}} * p^{\bar{k}} * (1-p)^{N-\bar{k}}$$

By replacing N with $k*n$ (for conversion of the x axis into basic storage units) and by introducing T we get

$$S_{p,T}(k,n) = T * \sum_{\bar{k}=k}^{k*n} \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

For comparability with DRBDorMARS, we have to compute the application downtime A instead of the storage downtime S, which depends on the spread exponent s as follows:

$$A_{s,p,T}(k,n) = n^{s+1} * S_{p,T}(k,n) = n^{s+1} * T * \sum_{\bar{k}=k}^{k*n} \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

Notice that at $s = 0$ we have introduced a factor of n, which corresponds to the hashing effect (teardown of n application instances by a single uncompensated storage incident) as described in section 4.3.1.2 on page 51.

A.3. Formula for SizeWeighted BigCluster

In difference to above, we need to introduce a correction factor by the fraction of affected objects, relative to basic storage units. Otherwise the y axis would not stay comparable due to different units.

For the special case of $k = 1$, there is no difference to above.

For the special case of $k = 2$ replica, the correction factor is $1/(N-1)$, because we assume that all the replica of the affected first node are uniformly spread to all other nodes, which is $N-1$. The probability for hitting the intersection of the first node with the second node is thus $1/(N-1)$.

For higher values of k, and with a similar argument (never put another replica of the same object onto the same storage node) we get the correction factor as

$$C(k,N) = \prod_{l=1}^{k-1} \frac{1}{N-l}$$

Hint: there are maximum k physical replicas on the disks. For higher values of $\bar{k} \geq k$, there are $\binom{\bar{k}}{k}$ combinations of object intersections (when assuming that the number of objects on a node is very large such and no further object repetition can occur execpt for the k-fold replica placement). Thus the generalization to $\bar{k} \geq k$ is

$$C(k,\bar{k},N) = \binom{\bar{k}}{k} \prod_{l=1}^{k-1} \frac{1}{N-l}$$

By inserting this into the above fomula, we get

$$A_{s,p,T}(k,n) = n^{s+1} * T * \sum_{\bar{k}=k}^{k*n} C(k,\bar{k},k*n) * \binom{k*n}{\bar{k}} * p^{\bar{k}} * (1-p)^{k*n-\bar{k}}$$

B. GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not

allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If

there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements",

and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to

60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

    Copyright (c) YEAR YOUR NAME.

```
    Permission is granted to copy, distribute and/or modify this document
    under the terms of the GNU Free Documentation License, Version 1.3
    or any later version published by the Free Software Foundation;
    with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
    A copy of the license is included in the section entitled "GNU
    Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts,
replace the "with...Texts." line with this:

    with the Invariant Sections being LIST THEIR TITLES, with the
    Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other
combination of the three, merge those two alternatives to suit the
situation.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```