Derrick: A Three-Layer Balancer for Self-Managed Continuous Scalability

ANDRZEJ JACKOWSKI, LESZEK GRYZ, MICHAŁ WEŁNICKI, and CEZARY DUBNICKI,
9LivesData LLC, Poland
KONRAD IWANICKI, University of Warsaw, Poland

Data arrangement determines the capacity, resilience, and performance of a distributed storage system. A scalable self-managed system must place its data efficiently not only during stable operation but also after an expansion, planned downscaling, or device failures. In this article, we present Derrick, a data balancing algorithm addressing these needs, which has been developed for HYDRAstor, a highly scalable commercial storage system. Derrick makes its decisions quickly in case of failures but takes additional time to find a nearly optimal data arrangement and a plan for reaching it when the device population changes. Compared to balancing algorithms in two other state-of-the-art systems, Derrick provides better capacity utilization, reduced data movement, and improved performance. Moreover, it can be easily adapted to meet custom placement requirements.

CCS Concepts: • **Computer systems organization → Secondary storage organization**; • **Information systems → Distributed storage**; **Storage recovery strategies**.

Additional Key Words and Phrases: data balancing, distributed storage, capacity utilization

ACM Reference Format:
Andrzej Jackowski, Leszek Gryz, Michał Wełnicki, Cezary Dubnicki, and Konrad Iwanicki. 2023. Derrick: A Three-Layer Balancer for Self-Managed Continuous Scalability. *ACM Trans. Storage* 19, 3, Article 27 (June 2023), 34 pages. https://doi.org/10.1145/3594543

1 INTRODUCTION

The management of physical data placement across devices is a fundamental problem that virtually any distributed system has to address. Especially distributed storage systems, which are normally responsible for maintaining data for other tiers, have to deal with the many intricacies of this problem. In particular, to decrease the risk of data loss and shorten failure handling, such systems have to replicate or erasure-code data chunks and disperse them across different physical devices. To optimize capacity utilization, they have to balance the amount of data between the devices while also accounting for the underlying network characteristic so that the cost of keeping the redundant chunks in sync is acceptable. When new devices are added, or existing ones fail, the systems have to move or reconstruct data chunks, ideally in a way that minimally affects the performance of the core functionality. These are just a few common examples of data placement requirements, and many applications also have their own specific ones.

Authors' addresses: A. Jackowski, L. Gryz, M. Wełnicki, and C. Dubnicki, 9LivesData, ul. Ekologiczna 1/19, 02-798 Warsaw, Poland; emails: jackowski@9livesdata.com, gryz@9livesdata.com, welnicki@9livesdata.com, dubnicki@9livesdata.com; K. Iwanicki, Wydział Matematyki Informatyki i Mechaniki, Ul. Banacha 2, 02-097 Warsaw, Poland; email: iwanicki@mimuw.edu.pl.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1553-3077/2023/6-ART27 $15.00
https://doi.org/10.1145/3594543

For these reasons, as we clarify in the next section, the problem of physical data placement in storage systems has received considerable research attention. A lot of that attention was focused on large-scale cloud-oriented storage systems, namely data-center-wide systems with thousands of machines or global geo-distributed systems that can be up to orders of magnitude larger. In contrast, relatively little work was dedicated to *on-premise scale-out storage systems*, such as HYDRAstor [12], Ceph [41], and Swift [16]. On the one hand, given the data deluge in today's digital societies, a market for such solutions is thriving (IDC expects a 4.7% CAGR in External OEM Storage [22]). On the other hand, however, managing data placement in such systems poses unique challenges, which cannot be effectively addressed solely by adopting solutions developed for public clouds.

More specifically, these challenges stem mainly from the life cycle of on-premise scale-out storage systems. Once deployed, such a system is controlled by the owning client. Consequently, it should hardly require human intervention, instead being largely self-managed. Furthermore, to accommodate the ever-accumulating data, the systems are typically expanded, often repeatedly, which has two key implications. First, an average system comprises multiple generations of hardware that inevitably offer different performance characteristics and capacities. Nevertheless, despite this heterogeneity, the system is expected to ensure high utilization of all available resources. Second, it is not uncommon for a single system to grow even by orders of magnitude. Supporting tiny configurations is thus as important as large ones to provide flexible scaling. For instance, Scality [18] reduced the minimal system size to three nodes to solve the challenges of small and medium-sized enterprises. Again, the performance overhead due to the growth must not be observable; on the contrary, the performance scaling should be as close to linear as possible to justify the expansion costs. In short, on-premise scale-out storage systems are expected to offer what we have dubbed *self-managed continuous scalability*: a single system must be capable of autonomously maintaining high resource utilization even when it is expanded by a few orders of magnitude with heterogeneous hardware.

While self-managed continuous scalability may seem like a natural requirement, it is hard to meet in practice. This observation is consistent with a common computer systems design principle, referred to as the incommensurate scaling rule, which states that changing a parameter of a system by a factor of ten usually requires a new design [35]. In the context of data placement, when formalized, many issues are NP-hard problems [15, 19]. Consequently, algorithms for large-scale deployments are inherently heuristics that rely on probabilistic or asymptotic properties (holding only for sufficiently large systems) while at the same time emphasizing different aspects, notably fault tolerance. In contrast, small-scale deployments sometimes make it possible to efficiently search the entire solution space to find optimal placements. In other words, algorithms employed for managing data placement do vary depending on the scale. Furthermore, as we elaborate shortly, they are heavily affected by practical issues, notably conflicting requirements with respect to placement, hardware heterogeneity, and traffic considerations, to name just a few. Controlling data placement in systems that ensure self-managed continuous scalability thus indeed requires special solutions.

The relevance of these issues is reinforced by a recent report by Gartner [32], which argues that scalability and flexibility (e.g., handling device additions without disrupting other operations) with simultaneous cost reductions are the current challenges in on-premise storage. New technologies, such as EAMR HDDs [34], bring new possibilities, but keeping up with the newest hardware requires significant software changes. According to Coughlin Associates [9], the vast majority of capacity still is—and will be in the foreseeable future—shipped in HDDs. The problem is that

the performance of HDDs does not improve as fast as their capacity,[1] resulting in less than 10 IOPS/TB in modern drives. High performance and flexibility are very hard to achieve with such a limited amount of IOPS, so attentive data placement across devices and thrifty data movements are necessary when aiming at continuous self-managed scalability.

This article presents Derrick, an algorithm for managing data placement. We implemented Derrick in our commercial backup and archival storage system, HYDRAstor, and verified it in production. A key observation behind Derrick is that data arrangement has different requirements and timing constraints depending on particular events that trigger its changes. Therefore, aiming at self-managed continuous scalability precludes a one-size-fits-all approach. To better explain the trade-offs and prioritizations due to conflicting needs, we analyze common requirements on data placement in on-premise scale-out storage systems. The study is based on: empirical data and our experience from thousands of HYDRAstor deployments worldwide, an examination of Ceph and Swift (state-of-the-art open-source systems used in multi-petabyte-scale installations), and a survey of other scale-out storage systems based on publicly available materials.

Instead of using a one-size-fits-all approach, the data arrangement in Derrick involves three sub-algorithms called Central Balancing (CentrBal), Transition Guide (TrGuide), and Distributed Balancing (DistrBal) to handle different cases. CentrBal computes a data arrangement for a perfectly stable system, disregarding hardware failures. To make data migration smooth, TrGuide computes a transition plan between two arrangements provided by CentrBal. Both CentrBal and TrGuide are allowed to run calculations for minutes or even a few hours, as they activate only if the device population changes, which is a time-consuming operation.[2] The situation is much different if a hardware failure occurs and immediate action is necessary to prevent service disruption. In such a case, DistrBal quickly finds a new placement for data from the failed devices. Despite the fact each of Derrick's three sub-algorithms has a different purpose, their structure is similar, as all of them optimize data arrangement through a hill-climbing technique. However, the assurance that each of the algorithms is able to find a solution in a given amount of time is not trivial. Therefore, we present novel techniques that are used to reduce computational complexity while giving guarantees that the outcome meets expectations.

As our experimental evaluation shows, Derrick achieves better results than the state of the art in meeting key data arrangement requirements that are indisputably important in most storage systems. Moreover, Derrick can be adapted to additionally meet very specific requirements of a particular storage system, as such being potentially broadly applicable.

The rest of the article is organized as follows. Section 2 surveys related work. Section 3 analyzes requirements on data arrangement in self-managed distributed storage and shows how they are met in Derrick. Section 4 describes Derrick's algorithms, and Section 5 presents further key details. Section 6 evaluates Derrick's implementation for HYDRAstor. Section 7 concludes.

2 DATA ARRANGEMENT PROBLEMS AND SOLUTIONS

Each distributed storage system, to a varying extent, adjusts its data arrangement methods to meet its needs. Some systems, such as HDFS [36] and Haystack [2], store data based on decisions of central metadata servers, which limits scalability, robustness, and performance. Therefore, in large-scale applications, these systems are often replaced by decentralized solutions. For instance, Tectonic is used in Facebook [33] to provide superior scalability and resource utilization. Since its workloads may require low latency or be IOPS-intensive, Tectonic arranges data dynamically to

[1] Multi-actuator HDDs can improve performance a lot, but the technology is fresh, so their pricing and availability in the next years are unclear.
[2] Device addition requires unpacking, connecting cables, moving data, etc.

meet the specific performance requirements. Similarly, other large-scale systems, like Windows Azure Storage [3] and Spanner [8], dynamically place and move data to improve resilience and performance.

A more static approach is taken by systems that make placement decisions based on hashes of the data. For many types of workloads, including storing content-addressable blocks in HYDRAstor, such methods are extremely efficient. Consequently, in our work, we focus on this kind of data arrangement.

Some of such systems, notably OpenStack Swift with its Rings [16] and Apache Cassandra [29], are based on consistent hashing [28], a technique that limits the number of data moves when the number of hash buckets (e.g., storage devices) changes. There are many publications improving consistent hashing. In particular, elastic consistent hashing [43] aims to reduce power consumption. Aye et al. [1], in turn, describe how to better data balancing specifically in GlusterFS. Consistent hashing can also be reduced to rendezvous hashing, which is a more generic algorithm suitable for storage systems. For instance, IBM Cloud Object Storage System utilizes so-called weighted rendezvous hashing [21].

Another state-of-the-art algorithm is CRUSH [42], employed in Ceph [41], which also distributes data based on a hash-like function. CRUSH supports a multi-level hierarchy of heterogeneous devices and introduces low computational overhead. Data movements in CRUSH have been reduced with the introduction of the straw2 [7] bucket type. Furthermore, since the arrangements calculated by CRUSH may underutilize capacity, Ceph features an additional balancer plugin [4] that alleviates this phenomenon. Another sample improvement of CRUSH is MapX [39], which calculates intermediate data placements to decrease the tail latency during data movements.

Both Swift and Ceph are utilized in thousands of deployments and are actively developed, so their algorithms are constantly improved to meet the needs of those systems. However, in the case of our system, aimed at self-managed continuous scalability, CRUSH and consistent hashing do not address some principal requirements, and their adequate modification seems very difficult, if not impossible. For this reason, we have devised Derrick, an alternative approach that is easier to extend to take into account additional constraints. Moreover, for common requirements of storage systems, Derrick also outperforms the state of the art. The requirements are described in Section 3 along with a brief comparison of Derrick, CRUSH, and consistent hashing in real-world systems.

The main technique underlying Derrick is hill climbing. It has already been applied to problems related to data balancing, for instance, allocating resources to tasks in a distributed system [10] or allocating data in a system built of devices with varying reliability [11]. The novelty of Derrick, however, comes not from the fact that it is based on hill climbing but rather from the way it utilizes this technique. Especially, without the separation of the aforementioned sub-algorithms and introduction of additional solutions, providing expected results in given time bounds would be hard, if possible at all.

In general, there has been a considerable amount of research on optimizations of the same metrics as in this article but using techniques other than finding a placement for erasure-coded or replicated fragments. To start with, there are various erasure coding schemes aiming to improve system performance [31], decrease repair degree [20], or minimize repair bandwidth [27]. Furthermore, there are techniques for dynamic replica management, such as CDRM [40], including adjusting the number of replicas to availability requirements. Another approach, adopted in HeART [26], is to leverage the fact that disk reliability changes over time to decrease storage utilization. Pacemaker [25] and Tiger [24] further improve this approach by reducing the transition overload, providing further space savings, and bettering robustness.

Because of its design, and notably, the score dimensions introduced shortly, Derrick is prepared to support many of those requirements at the same time. Therefore, by and large, it can be adjusted

to incorporate most of the aforementioned ideas. Moreover, integrating many of them into Derrick entails no changes at all. For instance, the LRCs [20] require storing additional erasure coding fragments, for which Derrick can find a proper placement out of the box. Likewise, Pacemaker [25] is meant to keep data in subclusters with homogeneous failure models, and Derrick can find placement within such subclusters as well. Methods requiring dynamic changes in data resilience can also be integrated, either by changing the resilience without modifying the fragment number (e.g., from 9+3 to 10+2 codes) or by modifying the score dimensions. In short, Derrick is truly versatile.

3 REQUIREMENTS ON DATA BALANCING

A recurring theme in on-premise storage systems, including Ceph, Swift, and HYDRAstor, can be summarized as follows. Every data piece (e.g., a file, an object, or a block) has its identifier (e.g., based on its pseudo-random hash). Since managing each such piece separately is infeasible, the identifier is used to assign the piece to a logical collection named a *group*. In other words, groups are a means of aggregating individual data pieces into manageable units.

Furthermore, for resilience, data are replicated or erasure-coded. Therefore, each group consists of *components* (e.g., a group with 3 replicas has 3 components). We denote components as $IdOfGroup$: $IdInGroup$, so with 2 groups and 3 replicas per group, the components are: 0:0, 0:1, 0:2, 1:0, 1:1, 1:2. The notation would be the same for erasure codes with 3 parts in total per group (e.g., for 2+1 codes, with 2 data parts and 1 parity part).

The data arrangement problem is finding an assignment of components to devices that maximizes metrics entailed by the requirements of the system. A basic sample data arrangement is presented in Fig. 1. In the rest of this section, in turn, we analyze specific requirements on data arrangement that are common in scalable real-world storage systems.

3.1 High Capacity Utilization

The most fundamental requirement on data arrangement is efficient utilization of the available storage capacity. We assume that data are assigned to groups evenly, as the hashing function is fair, so all components in the system have roughly similar sizes. In systems with deduplication, such as HYDRAstor, data is kept in small blocks, so the component sizes are naturally balanced. However, even systems that distribute entire files/objects, like Ceph, do not attempt to address the potential imbalance due to varying file/object sizes [5].

For this reason, we consider capacity utilization as maximized if the ratio of the number of components to the available device storage bytes is equal for all devices (cf. Fig. 1). If, in contrast, one device has a higher components-to-bytes ratio, the capacity of devices with lower ratios is wasted (cf. Fig. 2), because, per our assumption, all components are expected to have approximately the same size.

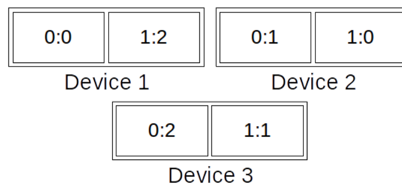


Fig. 1. Data arrangement of two groups (0,1) with thee components each (0:0, 0:1, 0:2 and 1:0, 1:1, 1:2) on thee same-capacity devices. Capacity utilization and resilience are optimal in such a configuration.

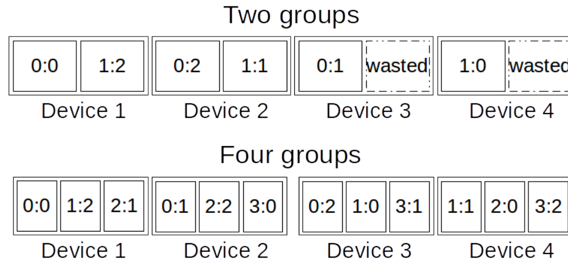


Fig. 2. Data arrangement of groups with 3 components each. With only 2 groups 25% of capacity is wasted. With the number of groups increased to 4, the capacity is fully utilized.

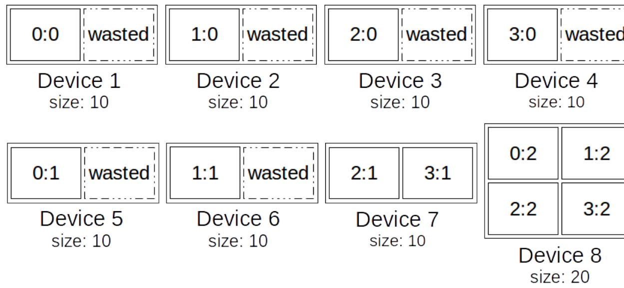


Fig. 3. Data arrangement of 4 groups with 3 components each in a heterogeneous system. Each device has at most 4 components but, system wise, more than 25% of the capacity is wasted because there are devices with half of their capacity wasted.

When the number of components is increased, the capacity waste may be decreased (as in Fig. 2), but this also burdens the system more. Therefore, for instance in Ceph, the recommended number of groups per device is 100 [6], while in HYDRAstor, we keep the number even smaller to increase data locality. If a system encompasses devices with different storage capacities (referred to as a heterogeneous system), the effect of wasted capacity is amplified because the size of each component constitutes a higher percentage of the capacity of a small device (see Fig. 3).

3.2 Resilience to Failures

Another crucial requirement on data arrangement is storing components resiliently, that is, in a way that, thanks to replication or erasure coding, they can survive device failures. To reduce the probability of data loss in case of a hardware failure, two components of the same group should be kept on different devices. Since devices form a multi-level hierarchy (e.g., a server has many drives, and a rack has many servers), this rule should be followed at the different levels of the hierarchy. The highest attainable resilience depends on the particular replication/erasure-coding scheme and the system size (see Fig. 4).

What is important, in heterogeneous systems, optimal capacity and optimal resilience may not be achievable simultaneously (cf. Fig. 5). Therefore, a need arises to describe how resiliently the data should be kept. In particular, in Ceph, crushmap rules specify how devices at each level of the hierarchy are chosen. The rules are strictly followed, so crushmap may need an update if the requirements on balancing change (e.g., when the system grows and the number of components

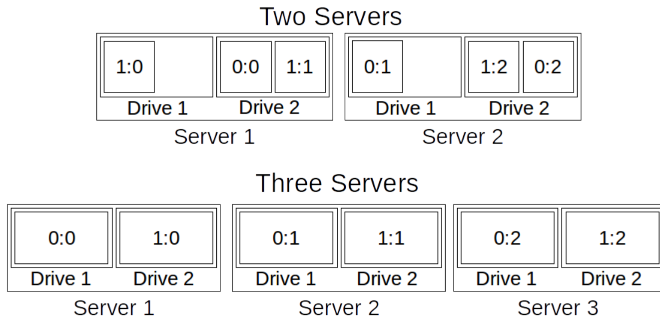


Fig. 4. The size of a system affects its resilience to failures. With 2+1 erasure codes, a system with 2 servers and 2 drives per server can survive a drive failure but not a server failure. In contrast, a system with 3 servers is also resilient to server failure.

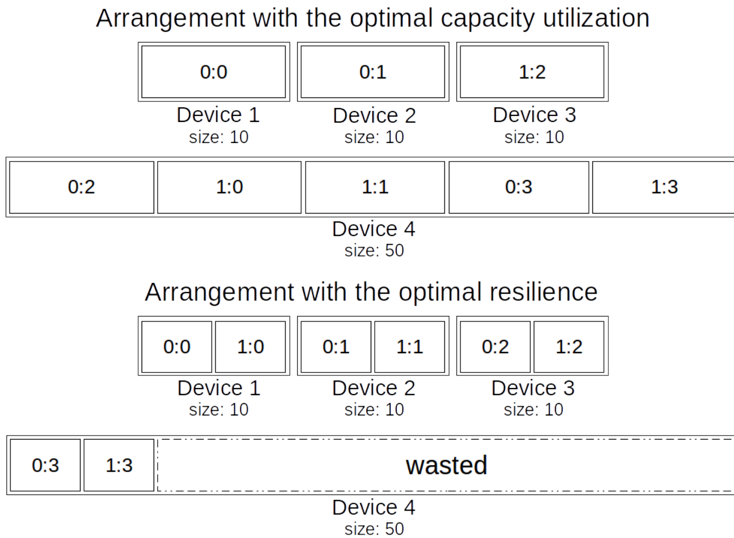


Fig. 5. A heterogeneous system with four devices: Device 4 is five times larger than Devices 1–3. With 3+1 erasure codes, the arrangement resilient to a device failure wastes 50% of the capacity. In contrast, in the arrangement with an optimal capacity utilization, Device 4 hosts 3 components from group 1, which precludes recovery of this group upon a failure of this device.

of the same group per server can be reduced). In Swift, Rings have a configurable overload factor, which specifies a fraction of additional components that can be accepted by each device to improve system resilience. Unless the factor is high enough, Swift may not find the most resilient solution, so a careful value selection is necessary, and capacity is underutilized anyway (details in Section 6.1.3). To provide self-managed continuous scalability, Derrick always finds the most resilient solution for the maximal capacity. However, it also allows an administrator to specify a minimal level of resilience that overrules the decisions stemming from maximizing capacity utilization.

3.3 Balancing Distinguished Components

Data arrangement affects not only resilience and capacity but also the overall system performance. In heterogeneous systems, the performance may be improved by placing more components on faster devices (e.g., servers with SSDs or better CPUs), but storing too many components on one device leads to underutilization of the capacity of other devices. Therefore, in some systems, there are distinguished components (*DistComps*, in short) that have additional tasks assigned. In particular, Ceph and Ursa [30] specify 1 distinguished *primary* per group, while in HYDRAstor there are 3. Overloading any device with too many DistComps should be avoided; otherwise, a bottleneck may arise. Network utilization also depends on data arrangement, as we will describe shortly afterwards.

3.4 Keeping Related Data in One Rack

With erasure codes, decoding multiple pieces is required to reconstruct failed data. Consequently, placing whole groups within the same rack decreases the expensive inter-rack communication during recovery. At the same time, keeping a group in one rack precludes resilience to rack failures, which is provided by many systems, including Ceph, Swift, and clouds [20]. However, some studies suggest that rack failures are less frequent than expected, and the overall system resilience is higher if data is located nearby to improve reconstruction speed [44]. Therefore, HYDRAstor gives this possibility as well. Ceph gives the option to specify a rule that keeps whole groups within one rack, but its balancer plugin tends to move components between racks more than necessary. Likewise, HDFS has a default policy to place two replicas in one rack and the third replica in another, which gives some locality and also resilience to a failure of one rack. There are also special erasure codes that trade capacity for decreased inter-rack traffic [17], but this conflicts with capacity maximization.

3.5 Limiting Data Movements

The data arrangement algorithm needs not only to calculate a good result for a stable system but also to react to changes that are often unexpected (e.g., hardware failures) or predictable shortly in advance (e.g., additions and removals of devices). Modification of a data arrangement requires moving data between devices, and hence it is desirable to minimize such movement as much as possible. In scale-out systems that can significantly change their size (e.g., from one to hundreds of devices), a change in the number of groups is necessary to maintain a similar number of components per device regardless of the system size. Ceph and HYDRAstor scale the number of groups automatically [6], while in Swift, functionality for altering the number of groups without cluster downtime is under development [14]. When the number of groups changes, some components should be moved (e.g., to improve capacity utilization as in Fig. 2). However, if the algorithm computes a placement for new groups independently of their previous locations (e.g., as in CRUSH), excessive amounts of data may be moved. A similar issue happens when other system parameters change (e.g., the configuration of the resilience hierarchy).

3.6 Limiting Non-stable Components

Another requirement related to data movement is how many components are moved at the same time (we refer to components during movement as *non-stable*). In Swift, only a single component from a group is moved at a time because the system cannot read data from non-stable components. HYDRAstor can read data from non-stable components, but it benefits from a limited number of such components in duplicate elimination, caching, and read-write deletion [38]. Ceph provides throttling, which also limits data moved at a time but does not limit movements per group. Keeping

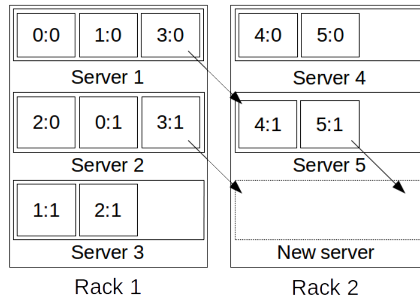


Fig. 6. Keeping entire groups in racks increases the number of required component movements upon a change to a system. In a system with 2 racks, 1 new server is added to the 5 existing ones. In effect, 2 components have to be moved to balance the capacity, and a movement of an entire group is needed to keep entire groups in racks. More specifically, Group 3 is moved. However, its two components cannot be stored on the new server because of the resilience requirements. Therefore, component 5:1 is moved as well to create space on Server 5. All in all, 3 components are moved instead of just 2.

Table 1. Summary of data arrangement algorithms in real-world use cases.

	Derrick	CRUSH in Ceph	Swift Rings
Capacity utilization	**The Highest**	High (with Ceph's balancer)	High
Resilience with multi-level hierarchy	**Optimal solution for given constraints and capacity**	Only given constraints are met	Space oversubscription needs to be configured
Balancing of distinguished components	**The Best**	Moderate	None
Groups can be kept within one rack	**Yes**	Yes, but Ceph's balancer spoils it	No
Data movement	**The Lowest**	Moderate	Low
Limiting non-stable components	**Precise** (preserve resilience and capacity)	None	Simplified ("move one at a time")
Computation time during failures	**Seconds**	**Seconds**	**Seconds**
Computation time when adding or removing devices	Minutes or hours	**Seconds**	**Seconds**

groups within a single rack makes maintaining components stable more difficult because it enforces the movements of entire groups (see Fig. 6).

3.7 Final Remarks

In Table 1, we summarize the discussed requirements and the way real-world implementations of the aforementioned state-of-the-art data arrangement algorithms meet them. As visible in the table, Derrick trades computation time when devices are added or removed for additional features and better results. As mentioned earlier, device additions and removals are predictable and take significant time anyway, so we find the trade-off profitable. In practice, the additional time spent on computations does not affect the system at all (i.e., the system is fully operational during such calculations), whereas better data placement provides considerable benefits.

4 DERRICK'S OVERVIEW

To meet all of the aforementioned requirements and provide self-managed continuous scalability, Derrick arranges data using three sub-algorithms. As we describe in this section, CentrBal, and TrGuide calculate placement for every component in the system, whereas DistrBal modifies their results in case of failures.

First, data arrangement for a healthy state is calculated using *CentrBal*, which is allowed to operate for minutes or even hours until a satisfactory solution is found. During this step, capacity utilization, resilience, balancing of distinguished components, placement of data between racks, and data movement is optimized. The calculation uses a small fraction of available resources (i.e., at most one core in a multi-server environment), and in practice has no negative impact on service quality, as our system has implemented mechanisms for effective resource sharing during various loads [37].

As the new data arrangement can differ a lot from the previous one, a mechanism is needed to prevent having too many non-stable components. Therefore, *TrGuide* operates efficient component movement between two CentrBal results that preserve the stability of components and other requirements (including capacity utilization, resilience, and managing network traffic). In the event of hardware failure, an immediate change is needed, and in such case, *DistrBal* can quickly override CentrBal/TrGuide decisions.

The results of each algorithm are distributed through the system to enable routing messages to proper nodes. As CentrBal and TrGuide compute placement for all components in the system, their result can have a significant size (e.g., multiple megabytes if there are millions of components), but the algorithms are executed occasionally. DistrBal modifies location only for components affected by a failure, so its results are much smaller.

4.1 Hill Climbing in Derrick

All three subalgorithms search the space of possible arrangements, which is exponential in system size. Therefore, a heuristic approach is taken to find a solution that meets many requirements at once. To be more specific, Derrick uses the hill-climbing method, which is an optimization technique that iteratively attempts to find a better solution by making incremental changes. To achieve that, each subalgorithm calculates a multi-dimensional score function that describes how much component arrangement fails to meet given requirements. In each iterative step, one or more components are moved at a time, using an *operation* as we explain shortly. If moving components decreases the score, the operation is applied and the procedure is repeated for a better arrangement.

As subalgorithms have different goals, their score functions differ. To be more specific, each score function describes components placement as a collection of *score dimensions* (called ScoreDims). ScoreDims are compared in lexicographic order, and each ScoreDim corresponds to a single requirement on data placement. For instance, to describe capacity utilization, a ScoreDim can contain sorted quotients of device size and a number of components assigned to it. In a situation from Fig. 1, assuming each device has 1 TB, the set is $\{0.5, 0.5, 0.5\}$ and if one component is moved, the set changes to $\{0.33, 0.5, 1.0\}$ (in which capacity utilization is worse, as the component size is reduced from 0.5 TB to 0.33 TB).

Another example is a ScoreDim which describes resilience by counting components from the same group on each device. Such ScoreDim contains a cartesian product of all groups and devices, so in a situation from Fig. 1 it is $\{1, 1, 1, 1, 1, 1\}$ (each device has one component from each of the two groups), and if a component is moved, it is $\{2, 1, 1, 1, 1, 0\}$ (one of the devices hosts 2 and one of the devices host 0 components from one of the groups). The score function orders ScoreDims by their priority, and if a more important ScoreDim has a higher value the score is worse, even if lower priority ScoreDims improve. In other words, a more important requirement is never violated to improve a less critical one.

To improve the score, components are moved between devices based on heuristics dubbed operations. The very basic operation is to try the movement of each component to another device and verify if any of such movements improve the score. Such an operation is not sufficient to reach the optimal score. For instance, to reach optimal resilience without decreasing the capacity

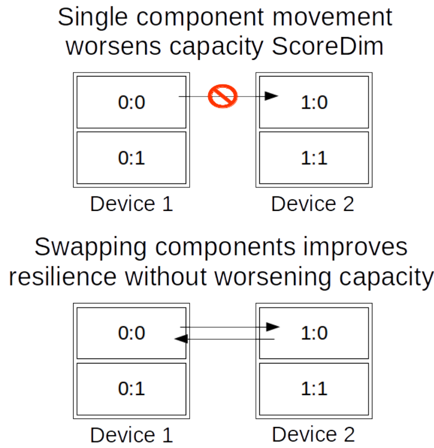


Fig. 7. An example in which a single movement cannot fix the resilience without worsening the capacity. Two simultaneous movements can lead to the optimal state without temporarily worsening the score.

utilization ScoreDim, movement of two components at a time may be necessary (Fig. 7). However, trying all possible component movements is $O(mn)$ with m components and n devices, whereas trying all possible movements of two components at a time is $O(m^2n^2)$, and three is $O(m^3n^3)$. As we explain in further sections, there are cases in which three or more components must be moved at the same time to leave a local minimum. On the one hand, staying in a local minimum means that one of the ScoreDims is not improved as much as possible (e.g., TrGuide can halt the transition). On the other hand, checking all possible movements of three components at a time for a system with $m = 1000$ and $n = 100$ (which are smaller than the maximal configuration we aim to support) requires checking more than 10^{15} states which is unacceptable. Therefore, we introduce techniques that reduce the set of tried movements (discussed in Section 5).

The general idea of subalgorithms is presented in Listing 1. The details of each subalgorithm are encapsulated in specific requirements, score calculation, and improvement heuristics.

```
# devices - List of devices with relevant details
# prev_arr - Previous arrangement of components
# reqs - List of ordered requirements
def derrick_subalg(devices, prev_arr, reqs):
  score, arr = calc_score(prev_arr, reqs), prev_arr
  while True:
    old_score = score
    for req in reqs:
      new_arr = improve(arr, req)
      new_score = calc_score(new_arr, reqs)
      if new_score < score:
        score, arr = new_score, new_arr
        break
    if old_score == score:
      return arr
```

Listing 1. Pseudocode of Derrick subalgorithm

The last remark regarding hill climbing is that, in general, such optimization technique can find a local minimum that is different from the global minimum [23]. In the case of a component arrangement problem, it means that the arrangement will not meet all requirements, for instance, resilience or capacity can be decreased, which is unacceptable. Therefore, we introduced techniques

that guarantee that the result always matches the global minimum for the most important ScoreDims. We explain these techniques in detail in Section 5.

4.2 Central Balancing

CentrBal computes component arrangement for a healthy system. Its input contains: a list of all devices with their sizes, a current component arrangement for a healthy system (calculated earlier or in a fresh system generated arbitrarily), and an ordered list of requirements. In this section, we explain a simplified CentrBal, which ensures optimal capacity and resilience while minimizing the number of transfers (advanced techniques are discussed in Section 5). Three ScoreDims are necessary to meet the aforementioned requirements:

(1) **Capacity**: whether a device has too many components for its size.
(2) **Resilience loss**: counts components from each group per device.
(3) **Movements count**: counts components with changed placement.

With those ScoreDims, CentrBal tries to find an arrangement that first maximizes capacity, then minimizes the number of components from the same group on one device. If any of the heuristics find a data arrangement that is as good or better in terms of capacity and resilience but requires fewer component movements, such an arrangement is chosen.

In heterogeneous systems, optimal resilience and capacity may not be possible at the same time (Fig. 5), so an additional ScoreDim above *Capacity* is added: *Accepted resilience loss*, which counts components from the same group per device, but only above a given threshold. For instance, if the threshold is 2 the system capacity is optimized as long as no device hosts more than 2 components from the same group. Moreover, the *Resilience loss* ScoreDim optimizes the resilience as much as possible without capacity decrease, so if a solution with the optimal capacity and at most 1 component from the same group on each device exists, it will be found.

Distributed systems typically have a device hierarchy (drives, servers, racks, etc.), so both resilience and capacity can have multiple ScoreDims which represent each device type. Drive utilization is more important ScoreDim than the utilization of servers, as drive utilization directly affects the system capacity. However, the arrangement of components on other levels of the hierarchy can also provide benefits (e.g., better utilization of network links). The ordering of such dimensions is important because, for instance, two utilization ScoreDims can be conflicting (an example in Table 2).

Table 2. Utilization requirements (e.g., for drives and servers) can be conflicting

	Server 1		Server 2	
	Drive 1	Drive 2	Drive 3	Drive 4
Size of drive [TB]	42.9	42.9	31.25	31.25
Total size of server [TB]	85.8		62.5	
Case 1: A component is added to the Server 2				
Drive components	19	18	14	**14**
Machine utilization ScoreDim	2.319		2.232	
Drive utilization ScoreDim	2.258	2.383	2.232	2.232
Case 2: A component is added to the Server 1				
Drive components	19	**19**	14	13
Machine utilization ScoreDim	2.258		2.315	
Drive utilization ScoreDim	2.258	2.258	2.232	2.404

In the Case 1, the drive utilization is lower than in the Case 2 as $\{2.383, 2.258, 2.232, 2.232\} < \{2.404, 2.258, 2.258, 2.232\}$, but for machine utilization $\{2.319, 2.232\} > \{2.315, 2.258\}$.

4.3 Transition Guide

TrGuide controls component movements from the current system state to the arrangement calculated by CentrBal. It ensures that system requirements are met, and network usage is balanced. Its major challenge is to keep many components stable. Just like CentrBal, TrGuide computes the score for the entire system. ScoreDims that describe the system resilience and capacity are the most important. After that, there are ScoreDims that count how many components are moved: within each group (it determines the number of non-stable components) and from each device (to balance network usage).

TrGuide needs the motivation to move components to the location calculated by CentrBal. Therefore, *ComponentsNotOnTarget* is a ScoreDim that counts components not located in their final location. *ComponentsNotOnTarget* is less important than the ScoreDim counting components moved within each group to prevent a situation in which all components are moved at once. It is expected that after a portion of component movements TrGuide will stop because every possible move will violate ScoreDims more important than *ComponentsNotOnTarget*. Later, when some movements are completed (and the number of non-stable components is decreased), TrGuide will move the next portion of components. Assuring TrGuide's liveness without moving too many components at a time is the difficult part (explained in Section 5.4).

4.4 Distributed Balancing

DistrBal overrides CentrBal and TrGuide decisions in case of hardware failures. Its goal is to find a good temporary placement for components that lost their devices. Therefore, DistrBal moves components from failed devices but does not move components hosted on their healthy CentrBal/TrGuide targets. That is because the movement of healthy data costs resources (e.g., disk IOPS, network traffic), which are needed to reconstruct missing data and handle the additional load. The components return to the location calculated by CentrBal/TrGuide when the issue is solved, so transient failures are handled efficiently.

Unlike the other two algorithms, DistrBal improves the placement only for a subset of components and devices, and the responsibility for calculating the arrangement is distributed across many instances of the algorithm. Each device has its own DistrBal instance that considers moving components to other devices. Limiting the responsibility of a single DistrBal instance decreases its complexity and facilitates the gathering of volatile information (e.g., a current utilization or a failure state of devices). If a device fails, its components are moved by other DistrBal instances that host components from the same groups. Since many DistrBal instances can make decisions about the same component, synchronization is needed. For example, DistrBal instances that host components from the same group can conduct voting to move only one component from their group at a time. To prevent a situation in which many components are simultaneously moved to one device, a locking mechanism is implemented (e.g., the device allows only one new component at a time).

5 DERRICK'S DETAILS

As described in the previous section, Derrick subalgorithms are based on a simple idea to improve score functions by moving components. However, the selection of proper ScoreDims and heuristics is non-trivial. In this section, we describe details of important techniques used in Derrick for HYDRAstor. Both theoretical lemmas, with their key ideas and practical observations, are presented to explain that even very detailed requirements can be met to provide self-managed continuous scalability. The methods are general and can be used in all subalgorithms, not only the subalgorithm chosen as an example to clarify each technique.

We use the following names of operations that move components to explain techniques and heuristics:

Relocation(c, n) moves the component c to the device n.
Swap(c_1, c_2) swaps c_1 and c_2 between their devices.
Push(c_1, c_2, n) moves c_1 to the device of c_2 and c_2 to the device n.
Cycle3(c_1, c_2, c_3) moves c_1 to the device of c_2, c_2 to the device of c_3 and c_3 to the device of c_1.

5.1 Capacity and Resilience in CentrBal

First, we show how the computational complexity of Derrick can be limited when heuristics are selected properly. Trying all possible relocations is not enough to find a data arrangement that has the best capacity for a given resilience target because the only possible relocation that improves the capacity may decrease resilience. Therefore, movement of more than one component at once may be necessary (Fig. 8). In our system, the groups have equal sizes (typically 12, so erasure codes like 9+3 and 10+2 are possible), and therefore we formalized a *Lemma 1* that limits the number of operations tried at once to improve resilience and capacity.

LEMMA 1. *If groups are equinumerous, then trying all relocations, swaps, and pushes is sufficient to find an arrangement with optimal capacity within resilience restriction.*

The key idea of the lemma is that if the system is not balanced, there must be a device n_1 that accepts a component c_1 from a group g_1 and a device n_2 that has too many components. However, due to resilience restriction, none of the components from n_2 might be accepted on n_1. As the groups are equinumerous, there must be a third device n_3 which accepts a component from n_2 and

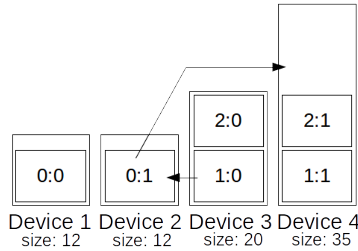


Fig. 8. Relocations are insufficient to balance capacity with a resilience restriction. Initially, the size of each component is 10, because there are 2 components on Device 3. None of the possible single relocations increases the score, as they either decrease the resilience or decrease usable capacity. After a single push, the size of the component can be increased to 11.67 (limited by Device 4), so the capacity increases by 16.7%.

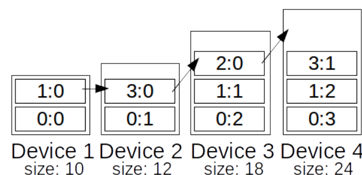


Fig. 9. If the groups have a different number of components, more than two simultaneous operations can be required to balance capacity with a resilience restriction. Device 4 has enough free space to get additional component, but cannot host any additional component from Device 1 or Device 2 without decreasing resilience. Relocating component 2:0 to Device 4 without performing other operations does not improve the score, so it is not done alone.

has a component from the group g_1. The formalized version of the lemma with formal proof is presented in Appendix A. The situation is much different if groups can have different sizes (Fig. 9).

5.2 Multiple ScoreDims in CentrBal

CentrBal optimizes capacity and resilience but also tries to meet all other requirements of a specific system. In our case, the device hierarchy consists of three levels – racks, physical servers, and logical nodes (which host a fixed number of drives). Therefore, to meet all of the requirements described in Section 3, our CentrBal score function consists of over 20 ScoreDims. The most important 12 of them are explained in Table 3, but there are some other system-specific ScoreDims which, for example, balance DistComps even further. However, the exact selection and order of ScoreDims depend on a specific system design. In our case, we find optimization of system capacity very important for storing backups, but we can think of a theoretical system in which optimization of DistComps (and therefore throughput) would be more important than maximal utilization of disk space.

Having that many dimensions enables optimizing resources effectively. For instance, DistComps are first optimized across logical nodes, as each logical node can handle a similar number of IOPS. After that, DistComps are also optimized across physical servers. After ScoreDim #8 is optimized, a possible situation is that one server hosts logical nodes with 7 DistComps each, and the other server has 8 DistComps each. In the described situation CentrBal tries to move DistComps to further optimize the utilization of resources shared between logical nodes, so each of the servers has 15 DistComps in total. However, having that many ScoreDims also has significant consequences.

First of all, the arrangement that optimizes all ScoreDims typically does not exist, and more important ScoreDims impact less important ScoreDims in a non-obvious way. For instance, ScoreDim for keeping entire groups in racks can spoil the balance of DistComps (Fig. 10), which came as somewhat of a surprise. Secondly, the calculation of the entire score function for all dimensions is expensive, as each score requires its data structure of significant size (e.g., it keeps information per device). Therefore, we implemented auxiliary data structures that are sufficient to verify how each ScoreDim changes after a single operation. Only when the best operation is selected, the full score is recalculated.

Finally, sometimes to improve a ScoreDim without spoiling another one, many operations need to be done at once (Fig. 11), which increases the complexity. Therefore, a possible solution to reduce

Table 3. Major ScoreDims of CentrBal

#	**Score Dimension**	**Description**
1	Accepted resilience loss	Ensures target resiliency in heterogeneous systems
2	System capacity	Optimizes system capacity
3	Resilience (logical node)	Decreases the number of components from the same group on each logical node.
4	Resilience (physical server)	Decreases the number of components from the same group on each server.
5	Num. components to the size of physical server	Balances components per physical server for even consumption of server resources.
6	Num. components to the size of logical node	Balances components per physical server for even consumption of logical node resources.
7	Keeping groups within a single rack	Reduces the number of racks on which each group is spread.
8	DistComps distribution across logical nodes	Balances distribution of DistComps across different levels of hierarchy to minimize unequal resource consumption.
9	DistComps distribution across physical servers	
10	DistComps distribution across racks	
11	DistComps from the same group (logical node)	Decreases the number of DistComps from the same group on one logical node.
12	Number of transfers	Decreases the number of transfers required to optimize all of the above.

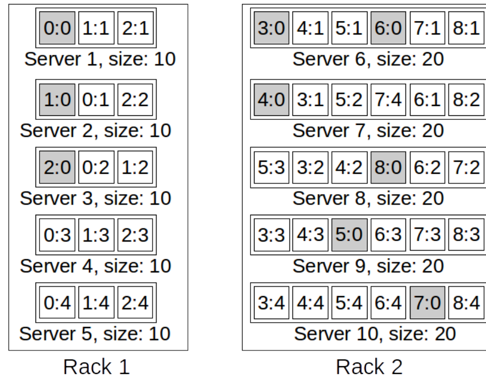


Fig. 10. An example in which placing whole groups in racks spoils the balance of DistComps. Components with IdInGroup=0 are distinguished. Rack 2 has servers with larger devices, so it receives more groups. As the result, 6 DistComps need to be placed on 5 servers, so Server 6 has 2 DistComps.

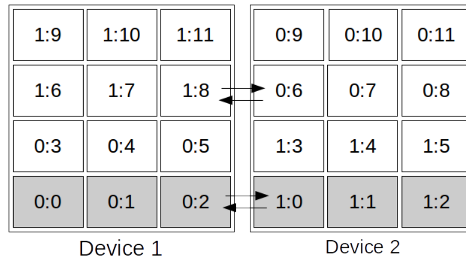


Fig. 11. An example in which two simultaneous swaps are required to limit the number of DistComps of the same group on one device. In that example, components with IdInGroup=0,1,2 are distinguished. Operations that move fewer components would decrease resilience, capacity, or the number of DistComps per device.

the complexity, is to greedily select an operation when it improves the score, without checking if there are any better options, but then an operation that also improves less important ScoreDims can be missed. Therefore, in our implementation, we often try a few improvements and choose the best. Another technique is to limit the components and devices that are considered for each heuristic. For example, when the resilience is not optimal, the component movement can be initially limited to components from devices that host most components from one group. Only when all heuristics related to system resilience were tried, heuristics optimizing other ScoreDims are started.

5.3 DistrBal ScoreDims

DistrBal tries to find a placement for components from failed devices, which is as good as the placement that would be computed by CentrBal. In fact, DistrBal score consists of very similar ScoreDims as these of CentrBal (3). However, the time constraints for the computing time of both algorithms are different. Therefore, the first difference is that the set of heuristics used by DistrBal is limited to the least complicated ones. Moreover, the set of components that are even tried to be moved is limited only to components from the failed devices.

As DistrBal communicates with other devices hosting components from a group, it has access to additional information, like the very recent capacity utilization of each device. Therefore, if a device

currently has some additional capacity (because the system has some free space), a component can be placed there. If the device becomes full before the failed device is restored, the recovered component will be moved to a different location, which has free capacity but is worse in terms of other ScoreDims.

5.4 Components Stability in TrGuide

The goal of TrGuide is to move all components to their target locations calculated by CentrBal while keeping requirements on data arrangement. In our system, one of the requirements is to not exceed 3 non-stable components within each group. Therefore, a transition plan is generated which does not move more than 3 components from each group at a time. After the maximal possible number of components is moved, TrGuide waits until any of the components is fully transferred (and therefore stable again). The operation is repeated until all of the components reach their final location provided by CentrBal.

The transition plan is prepared based on the following TrGuide_Score:

TrGuide_Score

(1) Nodes with too many components (exceeding the capacity)
(2) Groups with decreased resilience
(3) Groups with more than 3 non-stable components
(4) Components not in their target locations

TrGuide is capable of moving all components to their targets, without violating resilience and capacity restrictions, while moving at most three components of one group at a time. Therefore, it is guaranteed that the algorithm can progress without making more than 3 non-stable components within each group. The fact that TrGuide can move forward by moving 3 components also limits its computational complexity, as fewer component movements need to be tried at a time.

The key idea of why moving only three components at a time is sufficient is based on a construction of a component arrangement in which *relocations*, *swaps*, and $cycle_3$ violate resilience restrictions, as *swaps* and $cycle_3$ do not change the capacity. Such an arrangement must contain a cycle longer than 3 but we found two ways of effectively breaking such a long cycle into smaller one (Fig. 12). One of the methods requires keeping a reserve for one additional component on each node, which decreases the system capacity. Therefore, we introduced another method that breaks the long cycle into parts. Such breaking of the cycle is always possible (an example is presented in Fig. 13), as we explained in formal proof of lemma 2 and lemma 3 in Appendix A.

5.5 Stability of DistComps in TrGuide

An extreme example of meeting requirements by Derrick is maximizing the system performance by ensuring that at most one (out of three) DistComps is non-stable in each group. TrGuide is able to ensure such requirements, however additional ScoreDims were needed to make TrGuide progressing. That is because there is a difficult case in which two DistComps from the same group need to be swapped. Such a swap cannot be done straightforwardly without making both components unstable. Therefore, TrGuide needs to make three swap operations with non-distinguished components (as presented in Fig. 14). To enforce such an operation, an additional *Unwanted Distinguished Components* dimension was needed, that counts DistComps that are placed on the device that according to CentrBal will host another DistComp from the same group. In this way, TrGuide has the motivation to swap the DistComp with a non-distinguished component to improve that ScoreDim. Additionally, *Components not on the final target of components from its group* ScoreDim, that counts components that occupy place for a different component from the same group, enforces that the swap is done with another component of the same group.

To sum up, the final score function looks as follows:

TrGuide_Score_Extended

(1) Nodes with too many components (i.e., too much data)
(2) Groups with decreased resilience
(3) Groups with more than one non-stable distinguished component
(4) Groups with more than 3 non-stable components
(5) Components not on the final target of components from its group
(6) Unwanted Distinguished Components
(7) Distinguished Components not in their final target
(8) Components not in their target location

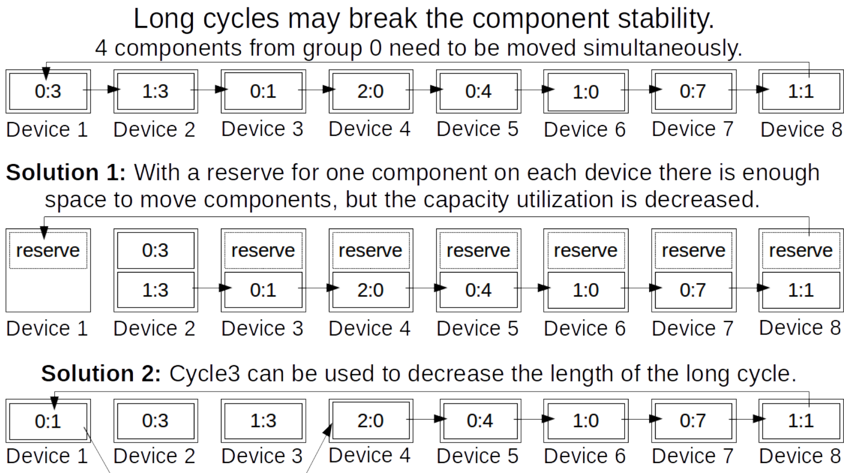


Fig. 12. Moving longer cycles at once can violate the stability of components. A possible solution is to keep an additional reserve for one component on each device, but it decreases system capacity. Therefore, we break longer cycles into smaller ones with a different technique.

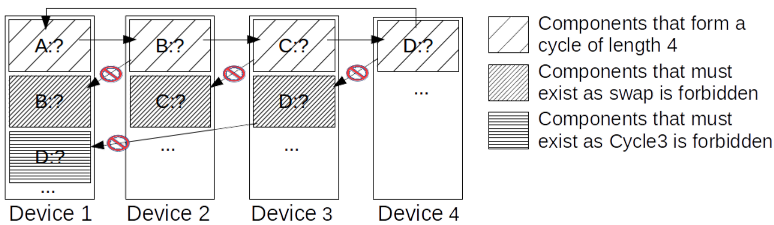


Fig. 13. *If a cycle of length 4 can be done without worsening resilience or capacity, then a swap or cycle3 can be done. Otherwise, the Device 1 does not accept a component D:? from the Device 4 which is a contradiction.*

5.6 Final Remarks

Derrick is very flexible and, as explained hitherto, it can meet diverse system requirements. Using the techniques described in Sections IV and V, we were able to express every needed requirement in adequate ScoreDims. Typically, an addition of a single straightforward ScoreDim is enough, but in some cases, a more complex design needs to be used (as in our TrGuide). Similarly, the

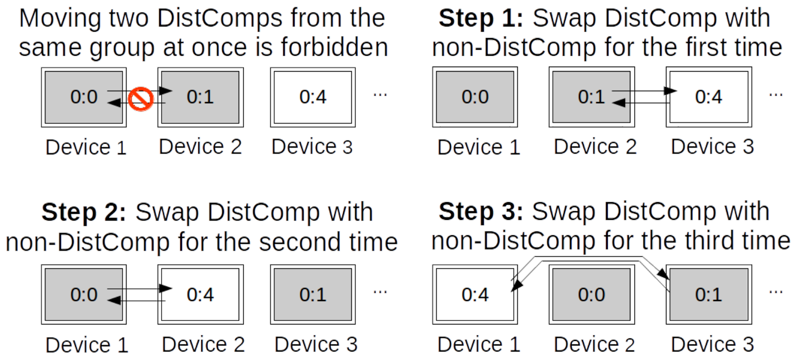


Fig. 14. Additional operations are needed to swap two DistComps from the same group without making both of them non-stable at the same time.

basic operations or their compositions are typically enough to implement heuristics that find data arrangement meeting each requirement. We are well pleased with the overall performance of Derrick, which we discuss in the next section.

6 EVALUATION

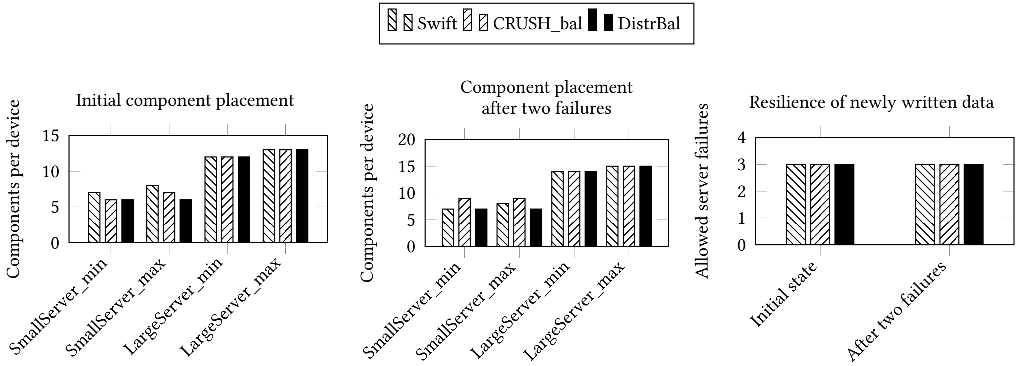
The evaluation is divided into three parts. First, Derrick is compared with the state-of-the-art algorithms used in Ceph and Swift in terms of meeting the requirements from Section 3. Secondly, we analyze the differences between DistrBal and CentrBal. Lastly, we measure the computation time of our implementation.

6.1 Comparison with Ceph and Swift

To ensure self-managed continuous scalability, a system needs to meet many requirements on data placement, so we compare data arrangements generated by Derrick, CRUSH in Ceph, and Swift Rings. A series of experiments was conducted to show how the algorithms differ in optimizing capacity and resilience, limiting the number of transfers required after changes in the system, keeping groups within racks, and balancing DistComps.

In most experiments, we used average-sized, heterogeneous configurations, which are typical for on-premise storage. They are additionally easy to understand. However, in the relevant cases, we present the results from larger configurations to show how the algorithms behave during scaling. Many experiments used a variable number of groups, as it often affects the results. The number of components per group in the presented experiments is 12, which is the default value for both the erasure-code scheme in Ceph documentation and for HYDRAstor, but we have not observed any meaningful differences between experiments with 2-15 components per group. In our system, each logical device internally manages its drives in a manner similar to RAID0, while the other systems assign components per drive. Our approach improves capacity utilization when the number of components does not equally divide the number of drives. However, we decided to ignore this difference in the presented results to avoid favoring our system.

The first two experiments were conducted using actual multi-server installations and real data being written. In these experiments, we carefully evaluated the three algorithms in terms of integration with the entire system and its practical behaviour. In further experiments, we only used tools that calculate the placement of the components offline, as it saved us from setting up a new testbed for each experiment. The tools that can compute component placement are already



(a) Minimal and maximal number of components at the beginning.

(b) Minimal and maximal number of components after two failures.

(c) After two server failures, new data is placed with resilience to three server failures.

Fig. 15. Deployed system evaluation in a heterogeneous system with 16 servers.

provided with Ceph (*osdmaptool*) and Swift (*swift-ring-builder*), and we implemented a similar tool for Derrick. All three tools share code with the production systems.

In all of the experiments involving Ceph, CRUSH was configured to use the *straw2* bucket type, which minimizes component movement. In relevant cases, the CRUSH result was further improved by Ceph's balancer (noted as *CRUSH_bal*). Swift required modification of the *overload* parameter in some experiments (noted as *SWIFT_overl*).

6.1.1 Deployed Systems Evaluation. In the first experiment, we used fully deployed systems to compare how each system is balanced initially and how failures are handled by each algorithm. We built one of the smallest possible heterogeneous configurations capable of storing data resiliently with 9+3 codes and up to three device failures. Therefore, the system consisted of 16 servers with the following configuration. Each of the servers had two drives formatted to have equal sizes. In 14 of the servers, the devices had 10 GB (denoted *largeServer*), and in 2 the devices had 5 GB *smallServer*, as Ceph does not allow devices below 5 GB. The system hosted 384 groups. The servers used Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz and Intel(R) Xeon(R) CPU E5-2430 @ 2.20GHz.

In the beginning, the systems had no data, and all of the servers were up and running. Therefore, Derrick placed all components just as computed by CentrBal, as there was no need to make any adjustments with DistrBal. Similarly, Ceph's balancer and Swift Rings placed components for the stable system. Already at this stage, the difference is considerable, as each algorithm placed a different number of components on small servers (Fig. 15a). On devices of small servers (5 GB), Derrick placed at most 6 components, Swift placed 7 and Ceph's balancer placed 8. On devices of large servers (10 GB), each algorithm allowed at most 13 components. Therefore, according to the placement, the maximal component size for Swift is $5/7$ *GB*, for Ceph's balancer $5/8$ *GB*, and for Derrick $10/13$ *GB* (as $10/13 < 5/6$). We elaborate on the utilization of capacity by Swift and Ceph in Section 6.1.3, as differences occur in other cases as well.

6.1.2 Failure Handling in Deployed Systems. In the next step, we simulated a hardware failure of two large servers, one after another. First, we killed processes responsible for handling storage

on one server. After a minute, we killed processes on the other server. The systems responded as follows:

- **Derrick** started DistrBal computations to adjust the component placement. In both cases, components ended up in locations optimal in terms of capacity (Fig. 15b) and resilience (Fig. 15c). In wall-clock time, the computations took in total just below 3.5s, but during that time the CPU consumption of the service which conducts the DistrBal computation (and implements some other functionalities) was below 10% of a single core on each node.
- **Swift** does not remove failed devices from Rings automatically but uses pre-computed handoff locations [13]. We experimented with the handoff locations but they do not balance load perfectly and, as a result, writing data to the system using handoff locations resulted in having some nodes without any free capacity, while other nodes had over 30% of free space. Therefore, we assumed that Ring rebuilding can be handled automatically after a failure, and included such an operation in our experiment. Rebuilding of the Ring after each failure took 2.2s but the component placement was not optimal in terms of capacity. Moreover, changing the Ring twice causes problems. As Swift cannot read objects while they are moved, and it does not implement any mechanism similar to our TrGuide, Swift requires waiting for transfers and reconstructions to be finished before the next balancing can be started. In fact, it is implemented by a timer which prevents changing the system balance until a given number of hours. Therefore, safe handling of two consecutive failures which happen shortly after each other is difficult.
- **Ceph** detects failed services promptly, but by default it waits for 10 minutes[3] before the service is considered down and removed from the CRUSH map. When a service is removed from the map, the computations takes milliseconds but typically the balancer needs to be executed a few times before it reaches its final results. As the balancer is started from a timer (which is by default 60 seconds), some time is required before it reaches its final state. Nevertheless, in terms of capacity, the placement was the worst of the three, as 9 components were placed on the small servers.

Finally, we started to write data. Data was written in small (512 KB) files / objects, to minimize the impact of uneven distribution of objects to components. All systems were able to accept the amount of data proportional to their component sizes, without any negative impact on data resilience (Fig. 15c) — it was possible to restore it even after three additional server failures.

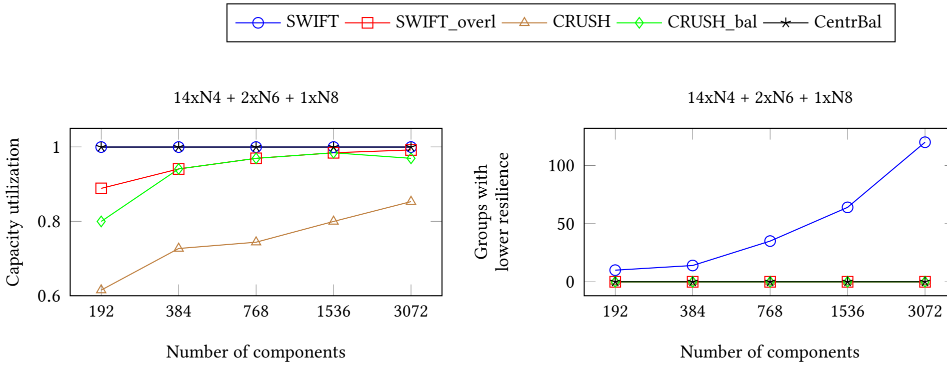
This experiment leads to the following conclusions. First of all, despite the simple scenario, each algorithm delivered different usable capacity and Derrick provided the best result of the three. In a stable system, DistrBal bases on CentrBal results, and during a failure DistrBal moves components from failed devices to optimal locations. However, this does not mean that CentrBal can be entirely replaced with DistrBal, as we evaluate differences between CentrBal and DistrBal further in Section 6.2.

Secondly, the model of handling failures in each system is different. Despite the fact that algorithms used in Ceph and Swift are able to compute results within seconds, their default use-cases rely on a manual intervention or are delayed by minutes.

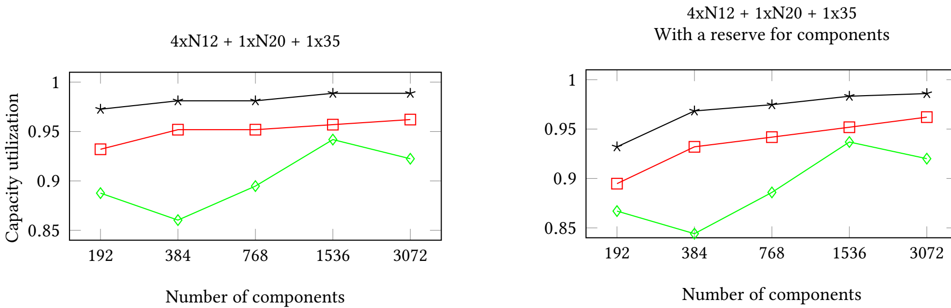
Finally, all of the algorithms provided resilient component placement. However, as we will describe shortly afterwards, it is not always the case.

6.1.3 Capacity Utilization and Resilience. Using the aforementioned tools, we verified capacity utilization and system resilience in a configuration that contained 17 servers: 14 with 1 TB drives, 2 with 6 TB drives, and 1 with 8 TB drives. In such a configuration, with 12 components per group,

[3]The value is configurable with mon_osd_down_out_interval setting.



(a) Capacity utilization and resilience violation in a heterogenous system with 17 nodes total.



(b) Capacity utilization in a heterogeneous system with 6 nodes.

(c) Lowered capacity utilization with an additional reserve for one component on each device.

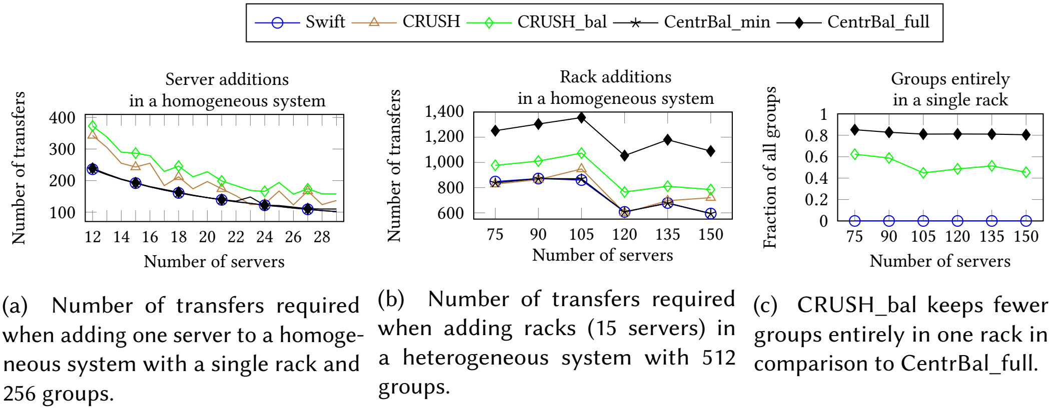
Fig. 16

two components of the same group are never on the same server and almost all available disk space is consumed (<0.0005 of space was not available because the 4 TB drive was not exactly 2x smaller than 8 TB drive).

Nevertheless, data arrangements calculated by each algorithm differed significantly (Fig. 16a). First of all, only CentrBal and Swift Rings optimally utilized the capacity. CRUSH was 1%-40% off, depending on the total number of groups and whether Ceph's balancer was used. For instance, with 256 groups, one of the 4 TB nodes received 150 components, which is 22 over the optimal result. Therefore, the component size (and so the capacity of the whole system) was decreased by 15%.

Swift Rings did failed to provide optimal resiliency, placing two components from the same group on one server. In heterogeneous systems, Swift allows changing the *overload* factor, which is a float value x that determines whether up x to additional device capacity can be used to improve system resilience. Although it was theoretically possible to find a perfect arrangement without overloading any node, Swift required a change of the overload factor to 0.0002 to find an arrangement that provided optimal resilience, but it decreased the capacity by 1%-11%.

Underutilization of the capacity contradicts with the market demand for cost reduction in scalable storage. Moreover, Swift Rings requires additional attention to make sure that the data is kept resiliently. To clarify, we present the results of another experiment (Fig. 16b), which uses a system with 4x1.2 TB servers, 1x2.0 TB server, and 1x3.5 TB server (an analogue of Example 1 from Fig. 8 for 12 components per group). In that experiment, our goal was to keep at most 4 components per group on each server, which is more than in the most resilient solution (2 components per group), but allows much better capacity. For 192 groups, the optimal solution in terms of capacity

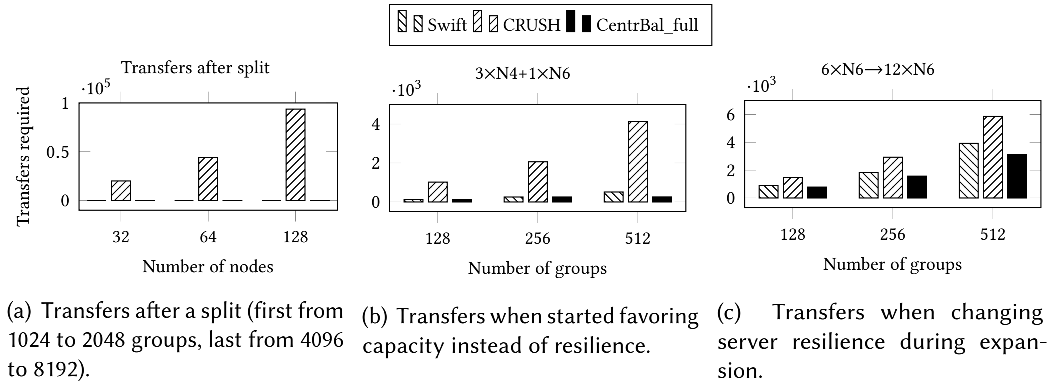


(a) Number of transfers required when adding one server to a homogeneous system with a single rack and 256 groups.

(b) Number of transfers required when adding racks (15 servers) in a heterogeneous system with 512 groups.

(c) CRUSH_bal keeps fewer groups entirely in one rack in comparison to CentrBal_full.

Fig. 17



(a) Transfers after a split (first from 1024 to 2048 groups, last from 4096 to 8192).

(b) Transfers when started favoring capacity instead of resilience.

(c) Transfers when changing server resilience during expansion.

Fig. 18. Number of transfers in different systems.

allows at most 4 components per group, but Swift may not find such an arrangement unless the overload is increased to a proper value (e.g., 0.01 overload was too small). When the overload was set to the smallest value that guaranteed the expected resilience, 2.6%-4.1% of the capacity was wasted compared to CentrBal with the same resilience. The aforementioned experiments show that CRUSH exactly follows the resilience requirements, but it achieves the lowest capacity utilization. Swift Rings does not find a resilient arrangement unless the overload value is set higher than it is really necessary. CentralBal results had an optimal capacity for the given resilience requirements in every experiment. The differences in capacity utilization are 1%-40%.

The reason for the capacity utilization differences is as follows. CRUSH depends on probability distribution, so its results are the worst due to variance. Therefore, Ceph's balancer improves the CRUSH results, but both the balancer and algorithms of Swift Rings are implemented in a way that allows one or two components off the perfect result, mostly to speed up the calculations. Therefore, capacity loss depends on how many components are on each device: from a fraction of a percent if there are hundreds of components per device, to even tens of percents if there are few. Especially, if the difference between the largest and smallest devices is high, the capacity loss on each misplaced component increases (Fig 3). When a 20 TB disk hosts 100 components, 0.5 TB hosts only 3 and misplacing two components makes a big difference.

In the end, to confirm a need for TrGuide that does not require an additional reserve of one component per device (as described in Section 5.4) we verified how such reserve affects the capacity. As presented in Fig. 16c, the reserve decreases the system capacity by up to 4%.

6.1.4 Transfers Required During Transition. In the next group of experiments, we verified how many components need to be transferred when a data arrangement changes. Limiting the transfers is important because component movement consumes resources, especially network and disks as data needs to be read, sent, and written. In the presented results we checked how many components need to be transferred, which can be converted to the amount of data that needs to be moved by multiplying the number of transferred components by the total system capacity, then dividing by the total number of components in the system. For instance, if a 100 TB system has 3072 components (256 groups), performing 400 transfers requires reading, sending, and writing of 13 TB.

We compared CRUSH, CRUSH_bal, and Swift Rings with two versions of CentrBal to show how additional requirements affect the number of transfers: CentrBal_full optimizes all requirements, including DistComps and in-rack placement, and CentrBal_min only optimizes resilience, capacity, and the number of transfers (as in Section 4.2).

The simplest experiment adds servers one by one to a homogeneous system that starts with 12 servers (where there is exactly one component of each group on each of the servers). The system had a single rack and 256 groups. Swift Rings and CentrBal_min required the same number of transfers, and CentrBal_full required on average 2.3% additional movements to optimize the placement of DistComps (Fig. 17a). CRUSH required on average 25% more transfers, and as in the previous experiments, the capacity was underutilized, so CRUSH_bal required even more. Therefore, in the basic experiment, CRUSH required many more transfers than necessary.

In a multi-rack heterogeneous system, the results were more diverse. We verified the number of transfers needed during the addition of entire racks (15 servers) of randomly selected servers with 4/6/8 TB drives and 512 groups (Fig. 17b). Swift Rings and CentrBal_min required almost identical numbers of transfers. As Swift Rings finds only an approximate solution, on average its result had 1.9% lower capacity and a manual[4] improvement to maximize the capacity required on average 3.5% more transfers. Furthermore, in Swift Rings and CentrBal_min there is no option to keep components from one group in the same rack. Such a requirement can be described in CRUSH, but the balancer plugin spoils it. With CentrBal_full the distribution of groups across racks can be imperfect as well because there are more important requirements such as capacity, but CRUSH_bal gave 27%-45% worse results (Fig. 17c). In terms of data movement, CRUSH_bal needed on average 18% more transfers than CentrBal_min and CentrBal_full required 30% more to improve a lot of placement of groups within racks.

Finally, we compared data movement in three scenarios in which not only the number of servers changes. If the number of groups is doubled in a homogeneous system with 32 nodes Swift Rings and CentrBal_min require no changes, CentrBal_full moves a handful of components to optimize DistComps, but CRUSH moves half of the data (Fig. 18a). Similarly, when changing whether capacity or resilience is more important in a system with 3x 4 TB and 1x 6 TB servers, Swift Rings requires 36% more transfers than CentrBal_full, and CRUSH moves far more components (Fig. 18b). When a requirement for server-level resilience is changed, because the size of a homogeneous system is doubled, Swift Rings requires 13%-30% more transfers than CentrBal_full, and CRUSH requires 86%-89% more (Fig. 18c).

To sum up, CentrBal_min not only ensures superior capacity utilization and resilience but also requires the lowest number of transfers. The main reason for the differences in the number of transfers is that CentrBal can spend additional time seeking solutions that provide the same results with the reduced number of data movements. CentrBal always uses previous component

[4]In that experiment, Swift Rings could not find optimal placement even when the *force* option was set. In other experiments, we observed that sometimes using the *force* flag helps to find optimal result, but it also dramatically increase the number of transfers, as it moves many components unnecessarily.

placement as a starting point, and with each moved component it tries to improve as many metrics as possible. For comparison, in other algorithms, a change of requirements such as desired resilience, means practically balancing data from scratch. This is why even CentrBal_full, which additionally optimizes other metrics, moves less data than others.

6.1.5 Distinguished Components Placement. Balancing of DistComps is important to evenly utilize system resources, which leads to improved performance. In our system, the first three components of each group are distinguished, and in Ceph the first component is distinguished, so we evaluated balancing of 1 or 3 DistComps (Fig. 19). In both CRUSH and CRUSH_bal, some nodes have up to 15%-40% more DistComps than necessary because neither of the algorithms tries to balance the distinguished components more than based on the probabilistic distribution. Swift does not use DistComps, so Swift Rings can even put all components with indexes 0 or 0,1,2 on the same node (such a node gets several times more DistComps than others). CentrBal_full finds a perfect or almost perfect distribution of DistComps in every case, which allows the highest resource utilization.

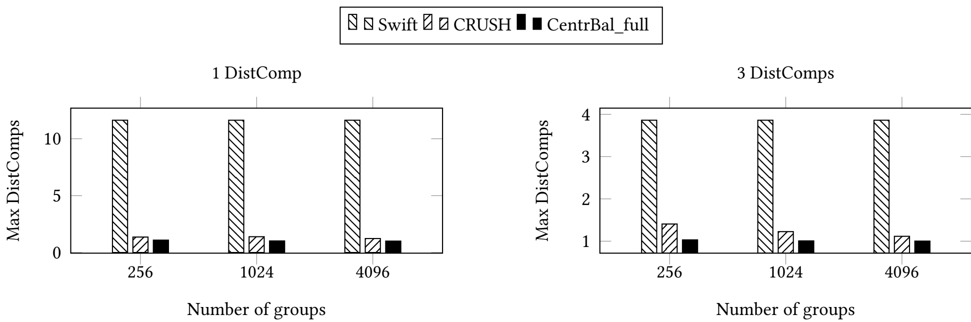


Fig. 19. Maximal number of DistComps per server, normalized to the value in optimal components arrangement.

6.2 Distributed Balancing Evaluation

In a stable system (without recent failures or changes in the number of devices), DistrBal simply uses the results already computed by CentrBal. In case of failures, DistrBal quickly adjusts the results provided by CentrBal. Therefore, we evaluate DistrBal by comparing its results with results computed by CentralBal in longer calculations. We randomly chose 25 heterogeneous configurations. In each configuration, we removed 1, 2, or 3 logical nodes in two ways. First, we killed the chosen nodes, so DistrBal moves the components from the removed nodes. Then, we completely removed these nodes from the system, so CentrBal could calculate its data arrangement. In 84% of cases, CentrBal found a better arrangement in terms of requirements on data arrangement, and in the remaining 16% the arrangement was equivalent to DistrBal result. As DistrBal does not try to move as many components as CentrBal it cannot find every improvement. Therefore, there were frequent differences on less important ScoreDims e.g. in 56% of cases the maximal number of DistComps per server was higher (not plotted).

The data arrangement calculated by CentrBal is superior to the one provided by DistrBal. However, the fact that DistrBal does not perform more complex operations is also its advantage, as it only computes a temporary state during failures. More complex operations require additional data movement, therefore higher resource consumption, but there are already fewer resources due to missing devices and data reconstructions.

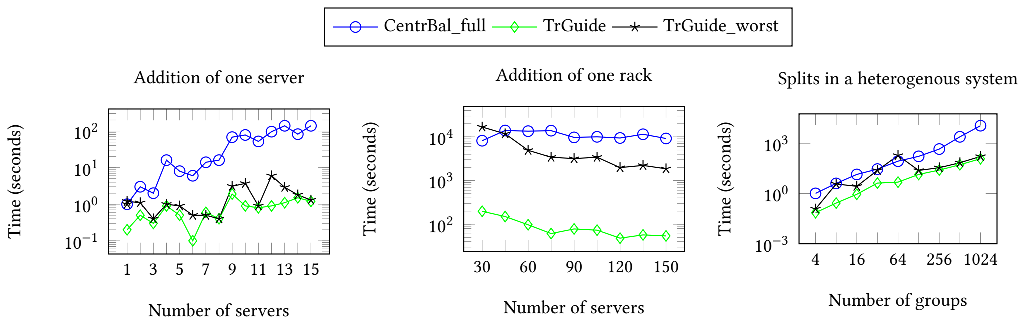
6.3 Performance Evaluation

6.3.1 CentrBal and TrGuide Performance. The performance was evaluated on a server with Intel Xeon E5640 Westmere 2.66GHz and 20GB of RAM (experiments needed even less memory). Fig. 20a presents the computation time of CentrBal when the size of a one server system (starting with 8 groups) is increased one server at a time. Initially, the computations take around 1 second, but as the number of groups is doubled each time the system size doubles, the final computation (with 256 groups total) took over 2 minutes. Fig. 20b shows the execution time of CentrBal_full during rack additions (15 servers) in a heterogeneous system (randomly selected servers with 4/6/8 TB drives) and 512 groups. The computation time does not increase with the number of servers because with more servers, fewer components require movement. Another experiment shows how the execution time depends on the number of groups (Fig. 20c). In all experiments, CentrBal_full finishes within 5 hours.

TrGuide typically moves most components using relocations, so the computations take less than a few minutes. The very worst case (TrGuide_worst in Fig. 20b and 20c) is a theoretical situation in which there are a lot of components to move because the system size was increased a lot, and immediately the system was filled to the brim. In such a case, TrGuide must exceed the limit of not-stable components to avoid hitting out of space, and more computations are needed to find an optimal plan in terms of resilience and non-stable components.

Our implementations of CentrBal and TrGuide were tuned to find a solution within hours using a single core, and the goal was achieved. Moreover, both algorithms work much quicker in smaller systems, where there are typically fewer CPU resources. Therefore, in every conducted experiment, the execution of all three algorithms consumed far below 1% of daily CPU resources available in the system.

6.3.2 Speed-Up Considerations. We find the computation time of Derrick good enough, but the execution time can be further reduced by orders of magnitude. First of all, we expect that our implementation of Derrick still has room for optimizations. Secondly, the number of heuristics used to improve less significant ScoreDims can be reduced. As we will describe shortly afterwards,



(a) Computation time after addition one node in small heterogeneous configuration. Number of groups scales with the system from 8 to 256.

(b) Computation time after addition of a rack (15 servers) with 512 groups.

(c) Computation time of splits in a heterogeneous system with 150 servers.

Fig. 20. Computation time of different configurations.

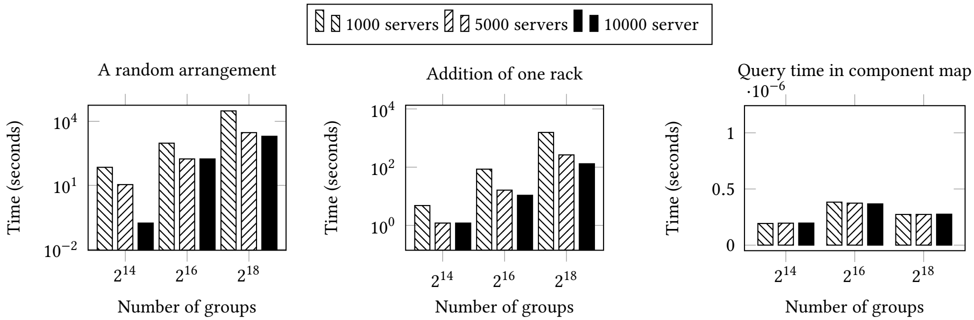


Fig. 21. Computation time of CentrBal_min.

Fig. 22. Query time in Derrick's result.

our evaluation of CentrBal_min confirms that finding a solution that only uses basic heuristics can be very quick. Finally, in large systems, the computation can be parallelised. In a larger system, two racks typically do not share any groups. Therefore, a system can be divided into smaller parts which are balanced separately and then merged into one system.

To evaluate the parallelisation idea, we randomly selected 10 heterogeneous configurations with 4 racks in total and executed CentrBal_full as follows. First, we divided the system into two parts (2 racks each). Secondly, we calculated component arrangement using CentrBal_full for each part. Finally, we executed CentrBal_full for the system with 4 racks, starting with the arrangement computed for the system parts. Such a method reduced computation time on average by 31%. For instance, in one of the experiments computation of arrangement for one part took 61s, and merging the results took 73s. Computing the arrangement for the entire system at once took 199s. Therefore, the total amount of work was similar, but the wall-clock time is reduced.

6.3.3 CentrBal_min Performance. We evaluated CentrBal_min, to verify the performance with the limited number of requirements. CentrBal_min can promptly compute its result even in a massive system with thousands of servers and 3M components (Fig. 21). Computation after the addition of 15 machines took only 26 minutes, and in a theoretical scenario when the initial arrangement is random the computing took under 10 hours.

6.3.4 Query Time in Derrick's Results. In the end, we also evaluated what is the overhead of finding a component location using a result generated by Derrick (Fig. 22). As components are represented by two numbers, a hash map representation of Derrick's result is very efficient. Our evaluation showed that even with millions of components and thousands of servers, performing a million queries using a single core does not even take half a second.

7 CONCLUSIONS

We examined challenges in providing self-managed continuous scalability in heterogeneous distributed storage systems. As a solution, we presented Derrick, which is a novel algorithm for finding data arrangements that meet multiple requirements. We implemented Derrick in HYDRAstor and evaluated it against existing state-of-the-art solutions in Ceph and Swift. Application of the described techniques in production also confirmed their effectiveness. Our approach guarantees maximal resilience, higher capacity utilization, and less data movement. For specific requirements, such as balancing distinguished components or keeping groups within racks, Derrick achieves respectively 15%–40% and 27%–45% better results. Moreover, to ensure flexible scalability, the

changes in data arrangement are introduced gradually, without superfluous disruption of system operations.

Derrick finds satisfactory solutions in systems from a few to thousands of devices within the given time. The general idea can be adapted to systems with a different set of requirements.

APPENDIX

A FORMALIZATION

The appendix contains the formalization of proofs using mathematical notation. First, we formalize the problem of finding a specific component arrangement. Then, we formulate *lemma 1*, *lemma 2*, and *lemma 3* using the notation and prove them.

A.1 Problem Statement

The problem of finding a specific arrangement is selecting a particular $a_i \in A$ for a given system

$$s =< N, G, C, A, group, capacity >: S \tag{1}$$

where

$$N = \{n_1, n_2, ..., n_n\} \text{ is a finite set of nodes.} \tag{2}$$

$$G = \{g_1, g_2, ..., g_n\} \text{ is a finite set of groups.} \tag{3}$$

$$C = \{c_1, c_2, ..., c_n\} \text{ is a finite set of components.} \tag{4}$$

$$group : C \longrightarrow G \text{ is a function that maps each component to a group.} \tag{5}$$

$$capacity : N \longrightarrow \mathbb{N} \text{ maps each node to its capacity.} \tag{6}$$

$$a_x : C \longrightarrow N \text{ maps each component to a node. We denote } a_x \text{ an arrangement.} \tag{7}$$

$$A = \{a_1, a_2, ..., a_n\} \text{is a finite set of all possible arrangements.} \tag{8}$$

A.2 Auxiliary Functions, Definitions and Corollaries

FUNCTION 1. *Function **components** provides a subset of components for a given arrangement and node:*

$$components : A \times N \longrightarrow 2^C; \tag{9}$$

$$components(a_x, n) = \{c | a_x(c) = n\} \tag{10}$$

FUNCTION 2. *Function **componentSize** computes a quotient of node capacity and its components number:*

$$componentSize : A \times N \longrightarrow \mathbb{N}; \tag{11}$$

$$componentSize(a_x, n) = capacity(n)/|components(a_x, n)| \tag{12}$$

FUNCTION 3. *Function **componentSize'** computes a quotient of node capacity and its components number increased by 1, which is a componentSize after a node accepts an additional component:*

$$componentSize' : A \times N \longrightarrow \mathbb{N}; \tag{13}$$

$$componentSize'(a_x, n) = capacity(n)/(|components(a_x, n)| + 1) \tag{14}$$

FUNCTION 4. *Function **systemCapacity** computes the capacity available in the system:*

$$systemCapacity : A \longrightarrow \mathbb{N}; \tag{15}$$

$$systemCapacity(a_x) = |C| * min(\{componentSize(a_x, n) | n \in N\}) \tag{16}$$

Definition 1. *a_x is an arrangement with optimal capacity iff*

$$systemCapacity(a_x) = max(\{systemCapacity(a) | a \in A\}) \tag{17}$$

Function 5. *Function **conflicts** returns the number of components for a given group on a node:*

$$conflicts : A \times N \times G \longrightarrow \mathbb{N}; \tag{18}$$
$$conflicts(a_x, n, g) = |\{c | group(c) = g \land c \in components(a_x, n)\}| \tag{19}$$

Function 6. *Function **maxConflicts** returns the maximal number of conflicts in the system:*

$$maxConflicts : A \longrightarrow \mathbb{N}; \tag{20}$$
$$maxConflicts(a_x) = max(\{conflicts(a_x, n, g) | n \in N, g \in G\}) \tag{21}$$

Definition 2. *a_x is an arrangement with an optimal resilience iff*

$$maxConflicts(a_x) = min(\{maxConflicts(a) | a \in A\}) \tag{22}$$

Definition 3. *a_x meets a resilience restriction of $r : \mathbb{N}$ iff*

$$maxConflicts(a_x) <= r \tag{23}$$

Definition 4. *a_x has the optimal capacity within a resilience restriction of $r : \mathbb{N}$ iff*

$$maxConflicts(a_x) <= r \text{ and} \tag{24}$$
$$systemCapacity(a_x) = max(\{systemCapacity(a) | a \in A \land maxConflicts(a) <= r\}) \tag{25}$$

Corollary 1. *If a_x meets a resilience restriction of $r : \mathbb{N}$ then*

$$\forall n \in N \ |components(a_x, n)| <= r * |G| \tag{26}$$

Function 7. *Function **groupSize** returns the number of components which are in the group:*

$$groupSize : G \longrightarrow \mathbb{N}; \tag{27}$$
$$groupSize(g) = |\{c | c \in C \land group(c) = g\}| \tag{28}$$

Definition 5. *Groups are equinumerous iff*

$$\exists_{n \in \mathbb{N}} \forall_{g \in G} |groupSize(g)| = n \tag{29}$$

Function 8. *Function **sumConflicts** returns the sum of conflicts for one group in total:*

$$sumConflicts : A \times G \longrightarrow \mathbb{N}; \tag{30}$$
$$sumConflicts(a_x, g) = \sum_{i=0}^{n} conflicts(a_x, n_i, g) \tag{31}$$

Corollary 2.

$$\forall a_1, a_2 \in A \ \forall g \in G \ sumConflicts(a_1, g) = sumConflicts(a_2, g) = groupSize(g) \tag{32}$$

A.3 Operations

Derrick solves the problem of finding arrangement by moving components between nodes. Therefore, Derrick has a set of possible operations which move components between nodes. Each operation is a partial function which is defined only if the components are indeed located on their initial nodes. We specify the following operations:

Operation 1. ***Relocation*** *is an operation which moves a component from its initial location to a different node:*

$$relocation : C \times N \times N \times A \nrightarrow A; \tag{33}$$

$$relocation(c_a, n_{initial}, n_{new}, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} a_x(c) & if\, c \neq c_a \\ n_{new} & if\, c = c_a \end{cases} \tag{34}$$

Operation 2. ***Push*** *is an operation which moves a component from its initial location to a different node, then takes a different component from that node and moves it somewhere else:*

$$push : C \times C \times N \times N \times A \nrightarrow A; \tag{35}$$

$$push(c_a, c_b, n_a, n_b, n_c, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} n_b & if\, c = c_a \\ n_c & if\, c = c_b \\ a_x(c) & if\, c \neq c_a \wedge c \neq c_b \end{cases} \tag{36}$$

Definition 6. *An arrangement a_x* ***can be reached*** *from a_y using operations $o_1, o_2, ..., o_n$, if there exists a composition of operations ϕ for which $\phi(a_x) = a_y$.*

A.4 Lemma 1

Lemma 1. *Assume a system and $r \in \mathbb{N}$ for which an arrangement meeting a resilience restriction of r exists. If groups are equinumerous, for every arrangement a_x there exists an arrangement a_y with optimal capacity within capacity restriction r, which can be reached from a_x using relocations and pushes.*

Proof: The proof consists of two steps. First, we show that there exists an arrangement a_r which meets a resilience restriction r and can be reached from a_x. Then, we show that a_y can be reached from a_r using relocations and pushes.

If a_x meets a resilience restriction r, then $a_x = a_r$. Otherwise, at least one node n_a and one group g_a for which $conflicts(a_x, n_a, g_a) > r$ exist. Moreover, for at least one node n_b, $conflicts(a_x, n_b, g_a) < r$, otherwise $sumConflicts(a_x, g_a) > r * n$ which contradicts the existence of an arrangement meeting a resilience restriction r. Therefore:

$$relocation(c_a, n_a, n_b, a_x) = a_i \tag{37}$$

$$conflicts(a_i, n_b, g_a) <= r \tag{38}$$

$$conflicts(a_x, n_a, g_a) - 1 = conflicts(a_i, n_a, g_a) \tag{39}$$

Repeating (possibly more than once) such steps, for every node and group which does not meet the resilience restriction r, leads to a_r.

If a_r has the optimal capacity within resilience restriction r, then $a_r = a_y$. Otherwise, at least one node n_a which lowers the system size exists ($componentSize(a_r, n_a) < \frac{systemCapacity(a_y)}{|C|}$). $Capacity(n_a)$ is given but the usable component size can be increased by moving a component out of n_a.

As capacity of a_r is not optimal within resilience restriction r, there must also be a node n_b satisfying the formulas:

$$componentSize'(a_r, n_b) >= \frac{systemCapacity(a_y)}{|C|} \tag{40}$$

$$\exists g_b \; conflicts(a_r, n_b, g_b) < r \tag{41}$$

The existence of n_b can be explained as follows. As n_a underutilizes the capacity, the other nodes must be able to take at least one additional component. Therefore, at least one node can accept additional component in terms of the capacity. Assume that for every node n_f accepting the additional component $\forall g_b\ conflicts(a_r, n_f, g_b) = r$. It means that each of the nodes hosts exactly $r * |G|$ components, and none of these nodes can accept any component. It is a contradiction, as it means no arrangement meeting the resilience restriction r exists in which n_a hosts fewer components (as none of the other nodes can accept it).

Therefore, n_b can take a component in terms of capacity and we want to move one of components from n_a. If any component c_a on n_a with $g(c_a) = g_a \wedge g_a = g_b$ exists, then c_a can be moved to n_b with the relocation. Otherwise, placing c_a on n_b breaks the resilience restriction r. Therefore, $conflicts(a_r, n_a, g_a) > conflicts(a_r, n_a, g_b) = 0$ (as there are no components from g_b), and $conflicts(a_r, n_b, g_a) > conflicts(a_r, n_b, g_b)$ (as n_b accepts g_b but not g_a). But the groups are equinumerous, so there must be at least one node n_c for which $conflicts(a_r, n_c, g_b) > conflicts(a_r, n_c, g_a)$ (there are two nodes with more components from g_a, we need to place components from g_b somewhere). Therefore, n_c hosts component c_b with $group(c_b) = g_b$, so $push(c_a, c_b, n_a, n_c, n_b, a_x)$ can be done.

The reasoning can be repeated until the optimal capacity with resilience restriction r is reached QED.

A.5 TrGuide Definitions

To formalize lemma 2 and lemma 3, additional definitions and operations related to TrGuide are introduced:

Operation 3. ***Swap*** *is an operation which changes the location of two components between nodes:*

$$swap : C \times C \times N \times N \times A \nrightarrow A; \tag{42}$$

$$swap(c_a, c_b, n_a, n_b, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} n_a & if\, c = c_b \\ n_b & if\, c = c_a \\ a_x(c) & if\, c \neq c_a \wedge c \neq c_b \end{cases} \tag{43}$$

Operation 4. *$Cycle_n$ is a generalization of a swap to more than two components:*

$$cycle_n : C \times ... \times C \times N \times ... \times N \times A \nrightarrow A; \tag{44}$$

$$cycle_n(c_1, ..., c_n, n_1, ..., n_n, a_x) = a_y \Leftrightarrow a_y(c) = \begin{cases} n_2 & if\, c = c_1 \\ ... & \\ n_1 & if\, c = c_n \\ a_x(c) & if\, c \neq c_1 \wedge ..n \wedge c \neq c_n \end{cases} \tag{45}$$

Definition 7. *Each operation* ***moves at a time*** *a number of components which is equal to the number of components which change their position during the operation. Relocation moves 1 component at a time, swap / push 2 at a time, and $cycle_n$ moves n components at a time.*

Definition 8. *For any two arrangements a_x and a_y, $nextBalancingStep(a_x, a_y)$ is an arrangement a_z meeting all of the following criteria:*

$$crit.\ \#1: |\{c | a_x(c) = a_y(c)\}| < |\{c | a_z c = a_y(c)\}| \tag{46}$$

$$crit.\ \#2: max(maxConflicts(a_x), maxConflicts(a_y)) >= maxConflicts(a_z) \tag{47}$$

$$crit.\ \#3: min(systemCapacity(a_x), systemCapacity(a_y)) <= systemCapacity(a_z) \tag{48}$$

A.6 Lemma 2

Lemma 2. *For any given arrangements a_x and a_y, TrGuide can find $nextBalancingStep(a_x, a_y) = a_z$ by moving at most three components at a time.*

Proof: Let's select a component c_a for which $a_x(c_a) = n_a \land a_y(c_a) = n_b \land n_a \neq n_b$. If n_b can accept c_a without violating $crit.\#2$ and $crit.\#3$, then the $nextBalancingStep(a_x, a_y)$ is $relocation(c_a, n_a, n_b, a_x)$.

If moving c_a to n_b violates $crit.\#2$, then n_b already hosts $max(maxConflicts(a_x), maxConflicts(a_y))$ components of $group(c_a)$, so a component c_b with $group(c_a) = group(c_b)$ for which $a_y(c_b) \neq n_b$ exists. Therefore $swap(c_a, c_b, n_a, n_b, a_x)$ is the $nextBalancingStep(a_x, a_y)$. Swaps can always be done without violating $crit.\#3$, as swap does not change the number of components on any machine.

If moving c_a to n_b violates $crit.\#3$ but does not violate $crit.\#2$, a component c_b for which $a_y(c_b) \neq n_b$ exists. However, c_b can be swapped with c_a only if it does not violates the maxConflicts. If the maxConflicts is violated, the entire algorithm can be repeated to move c_b to $a_y(c_b)$. If c_b cannot be moved to $a_y(c_b)$, just as c_a cannot be moved to $a_y(c_a)$, the procedure can be repeated again, up to $x < |C|$ times, until a component c_x can be moved to $a_y(c_x)$ or a cycle is formed. In either case, such an algorithm can move more than 3 components at a time, which is forbidden. However, if there is no cycle, the $relocation(c_x, n_a, a_x)$ is the $nextBalancingStep(a_x, a_y)$. If a cycle longer than 3 exists, we use the *lemma 3* to find the $nextBalancingStep(a_x, a_y)$ within the cycle.

In this way, all cases have been examined QED.

A.7 Lemma 3

Lemma 3. *For any given arrangements a_x and a_y, if a cycle p of length $n > 3$, formed by components $c_1, c_2, ..., c_n$ for which $a_y(c_1) = a_x(c_2) \land ... \land a_y(c_n) = a_y(c_1)$ exists, TrGuide can find the $nextBalancingStep(a_x, a_y) = a_z$ by moving at most three components at a time.*

Proof: First, consider the situation that $\exists i, j \in \{1..n\}((i < j) \land (a_x(c_i) = a_x(c_j)))$. In such case, reduce the considered cycle to $c_i, ...c_j$. If the length of the cycle is 2 or 3, use *swap* or $cycle_3$ as the nextBalancingStep. Otherwise, repeat the reasoning with the shorter cycle.

At this point, we can assume that the length of the cycle is $n > 3$ and $a_x(c_1), ..., a_x(c_n)$ are pairwise different. Using $cycle_n(c_1, ..., c_n, a_x(c_1), ..., a_x(c_n), a_x)$ does not change the capacity but moves more than three components at a time. If possible for any i, one of the following operations should be selected as the $nextBalancingStep(a_x, a_y)$:

$$relocation(c_i, a_x(c_i), a_y(c_i), a_x) \tag{49}$$

$$swap(c_i, c_{(i+1)\%n}, a_x(c_i), a_x(c_{(i+1)\%n}), a_x) \tag{50}$$

$$cycle_3(c_i, c_{(i+1)\%n}, c, a_x(c_i), a_x(c_{(i+1)\%n}), a_x(c_{(i+2)\%n}), a_x) \tag{51}$$

If none of the three operations is allowed, there are the following consequences. First, the $cycle_n$ does not violate the resilience requirement of nextBalancingStep, otherwise a swap would be possible (the same argument as in the *lemma 2*). Therefore, $cycle_n$ would be the $nextBalancingStep(a_x, a_y)$ but it moves too many components.

We can prove that if *swap* and $cycle_3$ violate the $crit.\#2$, then $cycle_i$ violates it as well for $i > 3$, which leads to a contradiction. For every $j <= n$, device $a_x(c_j)$ hosts maximal allowed number of components from the $group(c_{(j+1)\%n})$, otherwise swap is possible. Device $a_x(c_j)$ also hosts maximal allowed number of components from the $group(c_{(j+2)\%n})$, otherwise the $cycle_3$ would be possible. As $cycle_3$ moves two components to their a_y location, moving any of the component from $c_{(j+2)\%n}$ (even the one already on its a_y device) does not violate $crit.\#1$. Therefore, any of $c_{(j+2)\%n}$ can be moved to $a_x(c_{(j)})$ to not violate $crit.\#1$, but as $cycle_3$ violates the $crit.\#2$, every $a_x(c_{(j)})$

must host the maximal allowed number of components from every group hosted on $a_x(c_{(j+2)\%n})$. It implies that any cycle of even length violate the $crit.$#2, as every second node needs to host the maximal allowed number of all components from every group from other hosts with the same parity. Moreover, cycles of odd length violate $crit.$#2 as well, as swaps violating $crit.$#2 implies $a_x(c_j)$ hosts the maximal allowed number of components from the $group(c_{(j+1)\%n})$. Therefore, the limit of allowed components from $group(c_{(j+1)\%n})$ is achieved on every node, which contradicts the fact that $cycle_i$ is possible. Therefore, $relocation$, $swap$ or $cycle_3$ is the nextBalancingStep QED.

REFERENCES

[1] Kyar Nyo Aye and Thandar Thein. 2014. *A Data Rebalancing Mechanism for Gluster File System.* Ph. D. Dissertation. MERAL Portal.

[2] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10).*

[3] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* 143–157.

[4] Ceph. 2022. *Balancer Plugin.* Retrieved April 16, 2023 from https://docs.ceph.com/en/mimic/mgr/balancer/

[5] Ceph. 2022. *CHAPTER 3. PLACEMENT GROUPS (PGS).* Retrieved April 16, 2023 from https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/5/html/storage_strategies_guide/placement_groups_pgs

[6] Ceph. 2022. *Placement Groups.* Retrieved April 16, 2023 from https://docs.ceph.com/en/latest/rados/operations/placement-groups/

[7] Ceph. 2022. *V0.94.10 HAMMER.* Retrieved April 16, 2023 from https://docs.ceph.com/docs/master/releases/hammer/

[8] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[9] Tom Coughlin. 2021. *C1Q 2021 HDD Update.* Retrieved April 16, 2023 from https://forbes.com/sites/tomcoughlin/2021/05/04/c1q-2021-hdd-update/

[10] Fausto Distante and Vincenzo Piuri. 1989. Hill-climbing heuristics for optimal hardware dimensioning and software allocation in fault-tolerant distributed systems. *IEEE transactions on reliability* 38, 1 (1989), 28–39.

[11] John R Douceur and Roger P Wattenhofer. 2001. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*

[12] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. 2009. HYDRAstor: A scalable secondary storage.. In *FAST*, Vol. 9. 197–210.

[13] OpenStack Foundation. 2022. *Administrator's Guide.* Retrieved April 16, 2023 from https://docs.openstack.org/swift/latest/admin_guide.html

[14] OpenStack Foundation. 2022. *Increasing partition power.* Retrieved April 16, 2023 from https://specs.openstack.org/openstack/swift-specs/specs/in_progress/increasing_partition_power.html

[15] Lukasz Golab, Marios Hadjieleftheriou, Howard Karloff, and Barna Saha. 2013. Distributed data placement via graph partitioning. *arXiv preprint arXiv:1312.0285* (2013).

[16] Greg Holt. 2011. *Building a Consistent Hashing Ring.* Retrieved April 16, 2023 from https://docs.openstack.org/swift/latest/ring_background.html

[17] Hanxu Hou, Patrick PC Lee, Kenneth W Shum, and Yuchong Hu. 2019. Rack-aware regenerating codes for data centers. *IEEE Transactions on Information Theory* 65, 8 (2019), 4730–4745.

[18] HPE. 2018. *StorageExperts What's new with HPE Scalable Object Storage with Scality RING?* Retrieved April 16, 2023 from https://community.hpe.com/t5/Around-the-Storage-Block/What-s-new-with-HPE-Scalable-Object-Storage-with-Scality-RING/ba-p/7008183

[19] Hung-Chang Hsiao, Hsueh-Yi Chung, Haiying Shen, and Yu-Chang Chao. 2012. Load rebalancing for distributed file systems in clouds. *IEEE transactions on parallel and distributed systems* 24, 5 (2012), 951–962.

[20] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12).* 15–26.

[21] IBM. 2018. *IBM Cloud Object Storage System™, Storage Pool Expansion Guide.* Retrieved April 16, 2023 from https://www.ibm.com/docs/en/STXNRM_3.14.1/coss.doc/pdfs/storagePoolExpansion_bookmap.pdf

[22] IDC. 2022. *Enterprise Storage Systems Market Share.* https://idc.com/promo/enterprise-storage-systems.

[23] Alan W. Johnson and Sheldon H. Jacobson. 2002. On the convergence of generalized hill climbing algorithms. *Discrete applied mathematics* 119, 1-2 (2002), 37–57.

[24] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. 2022. Tiger:{Disk-Adaptive} Redundancy Without Placement Restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 413–429.

[25] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory R Ganger. 2020. {PACEMAKER}: Avoiding {HeART} attacks in storage clusters with disk-adaptive redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 369–385.

[26] Saurabh Kadekodi, KV Rashmi, and Gregory R Ganger. 2019. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 345–358.

[27] Govinda M Kamath, N Prakash, V Lalitha, and P Vijay Kumar. 2014. Codes with local regeneration and erasure correction. *IEEE Transactions on information theory* 60, 8 (2014), 4637–4660.

[28] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[30] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[31] Michael Luby, Mark Watson, Tiago Gasiba, Thomas Stockhammer, and Wen Xu. 2006. Raptor codes for reliable download delivery in wireless broadcast systems.. In *CCNC*, Vol. 6. 192–197.

[32] Julia Palmer, Jerry Rozeman, Chandra Mukhyala, and Jeff Vogel. 2021. Magic Quadrant for Distributed File Systems and Object Storage. ID: G00738148.

[33] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[34] John Paulsen. 2021. *Energy Assisted Magnetic Recording Will Solve the Need for Capacity.* Retrieved April 16, 2023 from https://blog.seagate.com/enterprises/energy-assisted-magnetic-recording-will-solve-the-need-for-capacity/

[35] Jerome Saltzer and M Frans Kaashoek. 2009. *Principles of computer system design: An introduction.* Morgan Kaufmann.

[36] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[37] Piotr Skowron, Marek Tomasz Biskup, Łukasz Heldt, and Cezary Dubnicki. 2013. Fuzzy adaptive control for heterogeneous tasks in high-performance storage systems. In *Proceedings of the 6th International Systems and Storage Conference*. 1–11.

[38] Przemyslaw Strzelczak, Elzbieta Adamczyk, Urszula Herman-Izycka, Jakub Sakowicz, Lukasz Slusarczyk, Jaroslaw Wrona, and Cezary Dubnicki. 2013. Concurrent Deletion in a Distributed Content-Addressable Storage System with Global Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 161–174.

[39] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. 2020. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 1–11.

[40] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. 2010. CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster. In *2010 IEEE international conference on cluster computing*. IEEE, 188–196.

[41] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 307–320.

[42] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. 2006. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 31–31.

[43] Wei Xie and Yong Chen. 2017. Elastic consistent hashing for distributed storage systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 876–885.

[44] Mi Zhang, Shujie Han, and Patrick PC Lee. 2017. A simulation analysis of reliability in erasure-coded data centers. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 144–153.

Received 5 May 2022; revised 30 September 2022; accepted 17 March 2023