University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Leszek Gryz

Impact of Data Organization on Distributed Storage System

PhD dissertation

Supervisors

prof. dr hab. Krzysztof Diks

Institute of Informatics
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw

dr Cezary Dubnicki

9LivesData, LLC

January 2012

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

January 09, 2012	....
date	*Leszek Gryz*

Supervisors' declarations:
the dissertation is ready to be reviewed

January 09, 2012	....
date	*prof. dr hab. Krzysztof Diks*
January 09, 2012	....
date	*dr Cezary Dubnicki*

Abstract

With the explosive growth of data stored in digital format, there is a need for a new approach to data storage. Large amount of stored data requires modern storage systems to be scalable and easily extendable on-line. Moreover, the data must be resilient and highly available, which in turn requires failure-tolerant and highly available storage. To address these needs a new storage segment called scalable distributed storage systems (DSS) has recently emerged.

One of the key architectural decisions for a DSS system is the design of its data organization defining data distribution among multiple nodes as well as local data placement on each node. The fulfillment of almost all the requirements of DSS systems depends directly on proper data organization.

In this thesis we describe DSS requirements dependent on data placement. Moreover, we identify trade-offs among these requirements and discuss how trade-off resolution is related to design choices for data organization. Next, we propose a novel data organization that resolves the trade-offs in a reasonable way. This is verified by using the proposed organization in a commercial DSS system called HYDRAstor. We have a substantial first-hand experience with this system, as for the last few years we have been part of a core team designing and implementing HYDRAstor.

In this system, data is organized around a distributed hash table with virtual supernodes spanned over physical nodes. Data resiliency is provided with erasure codes, with fragments of erasure-coded blocks distributed among supernode components. Fragments are stored in containers that are organized into chains to allow fast storage of data streams and enable efficient data consistency management, data health verification and data reconstruction.

We conclude that the design of data organization in HYDRAstor allows for the balancing of conflicting DSS requirements which has resulted in the creation of a break-through, highly innovative storage system.

Keywords: data organization, distributed storage system, storage system requirements, failure resilient system, scalable system, self-manageable system, backup and archival system, failure tolerant distributed hash table

ACM Classification: E.2, H.3.1, H.3.2, H.3.4, C.2.4, C.4

Streszczenie

Z powodu gwałtownego wzrostu ilości danych cyfrowych, potrzebne jest nowe podejście do sposobu ich przechowywania. Bardzo duże ilości danych wymagają, aby nowoczesne systemy przechowujące dane były skalowalne, łatwo rozszerzalne on-line, cechowały się wysoką dostępnością i zapewniały, że dane nie zostaną stracone w przypadku awarii. Aby zaspokoić te potrzeby, ostatnio powstał nowy typ systemu zwany rozproszonym systemem pamięci masowej (RSPM) (ang. distributed storage system).

Jednym z kluczowym elementów architektury systemu RSPM jest projekt organizacji danych, który określa rozproszenie danych pomiędzy wiele serwerów, jak również ich lokalne rozmieszczenie na każdym serwerze. Właściwa organizacja danych jest kluczowa, ponieważ wpływa ona bezpośrednio na spełnienie prawie wszystkich wymagań systemu RSPM.

W rozprawie zostały opisane wymagania RSPM, których realizacja zależy od organizacji danych. Ponadto zostały zidentyfikowane konfliktujące wymagania oraz została przeprowadzona dyskusja w jaki sposób rozwiązanie tych konfliktów jest związane z organizacją danych. Następnie zaproponowana została nowatorska organizacja danych, która uwzględnia problemy wynikające z tych konfliktów. Skuteczność tej organizacji danych została przez nas zweryfikowana poprzez użycie jej w systemie HYDRAstor, który jest komercyjnym systemem RSPM. W ciągu kilku ostatnich lat byliśmy główną częścią zespołu, który zaprojektował i zaimplementował system HYDRAstor. Dzięki temu zdobyliśmy doświadczenie w projektowaniu organizacji danych tego typu systemów.

W systemie HYDRAstor dane są zorganizowane wokół rozproszonej tablicy mieszającej opartej na wirtualnych super-węzłach rozpostartych na fizycznych serwerach. Odporność danych na awarie jest zapewniona poprzez użycie kodów korekcyjnych typu “erasure codes” i rozdystrybuowanie zakodowanych fragmentów danych na wszystkie serwery danego super-węzła. Fragmenty danych są przechowywane w łańcuchach następujących po sobie kontenerów fragmentów. Umożliwia to szybkie zapisywanie danych strumieniowych, wydajną weryfikację spójności i jakości danych oraz rekonstrukcję utraconych danych.

Podsumowując, projekt organizacji danych w systemie HYDRAstor umożliwił równoważenie sprzecznych wymagań RSPM, co w rezultacie doprowadziło do zbudowania przełomowego i innowacyjnego systemu pamięci masowej.

Słowa kluczowe: organizacja danych, rozproszony system pamięci masowej, wymagania systemu pamięci masowej, system odporny na awarie, skalowalny system, system samozarządzający, systemy kopii zapasowych i archiwizacji, rozproszona tablica mieszająca odporna na awarie

Klasyfikacja tematyczna ACM: E.2, H.3.1, H.3.2, H.3.4, C.2.4, C.4

Contents

Chapter 1

Introduction

1.1 Motivation

With the development of information technology, the amount of information generated is increasing exponentially [40]. For example, nearly 800 exabytes were created in 2009, which is 62% more than the amount generated a year earlier. For 2010, this amount was nearly 1200 exabytes, an increase of 50% [32]. This study also projects that by 2020 the amount of digital information created annually will be 44 times as big as it was in 2009.

Since practically all of the new data created is stored digitally, the exponential growth in the amount of data created leads directly to similar increase in the demand for storage. The increase in the transactional data stored in the database is 30-50% annually on average. The growth of WORM data (write once, read many) e.g. medical data (such as X-rays), financial, insurance, multimedia data, is 100% per annum [19]. Additionally, in many areas, legislation [60, 5] requires keeping of data for a long time, which further increases storage needs.

Currently, there are two common main types of storage systems used for long term data retention - storage based on hard disk drives and storage based on tape libraries (optical storage is less popular today, so we omit it here). Hard disk drives are typically organized in one of the following forms: Direct-Attached Storage (DAS), Network Attached Storage (NAS), and Storage Area Network (SAN). The tapes are organized in a tape library.

DAS is the simplest hard disk form of organization. Several hard drives are directly attached to computers. This causes every computer to become small *storage island*, since storage is not shared among computers. Thus, DAS-based solution are not scalable and utilize capacity in a very inefficient manner. They are practically unmanageable and unusable for a large amount of data.

NAS integrates multiple disks into a single device with file level interface accessible over a computer network. NAS-based solutions are more scalable than DAS-based ones but still form storage islands around NAS servers. The scalability of each NAS is limited by the number of disks each server can handle. To store huge amount of data, administrators have to use many NAS servers and manually assign data to them. Static allocation results in inefficient capacity utilization and performance. Each NAS server is a single point of failure. Finally, manual management increases significantly operational cost of these solutions.

SAN makes use of a computer network to connect many hard disk drives that can be divided into logical units. Computers maintain their own file systems on those logical units in a non-shared manner and thus still generate storage islands. Moreover, storage usage is inefficient, as space needs to be pre-allocated to file systems and databases. To make hard disks truly shared, dedicated shared or distributed files systems are built on top of SAN, something which is beyond topic of this thesis.

A tape library contains many magnetic tapes, tape drives and a robot for loading tapes into the drives for reading or writing. Even though tape-based solutions can reach capacity of petabytes, they cannot be extended gradually to grow with increasing needs, i.e. large up-front investment is required. Moreover, tapes have very long access time, because the robot has to load tapes into drivers and rewind them to the right location. Last but not least, tapes are very fragile - they must be stored at the appropriate temperature and humidity which increases cost of their maintenance.

All these solutions have additional common disadvantages. They do not provide flexible failure tolerance. Typically used disk redundancy schemas like RAID-5 [61] and RAID-6 [83] fail to protect data when more than one or two disks fail respectively. Tape libraries do not provide any failure protection - tape damage results in loss of data it stores. To reach the required level of failure resiliency administrators create excess data copies which results in inefficient capacity utilization. Last but not least, current solutions do not eliminate duplicated data. Such feature would greatly improve efficiency of storage usage for backup and archival data, because it contains a lot of duplicates – consider full backup done weekly which usually contains very little new data. Elimination of duplicated data is impossible to accomplish in the case of tapes because of lack of random access, but it is feasible in the case of disk-based solutions.

The above review of existing storage paradigms shows that currently available solutions are not designed to manage large amount of fast growing data in an effective, failure resistant, highly available and scalable manner. Fortunately, the recent increase in magnetic disk capacity and multi-core CPU performance, together with their falling costs, allow for a new approach. Scalable, distributed storage systems (DSS) are specifically targeted for the efficient storage of huge amount of data. Key features of DSS include automated management, efficient storage usage by avoiding space pre-allocation and supporting deduplication of stored data; and high availability and resiliency to failures.

Recognizing the challenges related to efficiently storing the rapidly expanding amount of digital data, NEC Corporation (the largest Japanese IT company) started a project called HYDRAstor in late 2002, aimed at building a DSS for backup and archival data. I have been privileged to be a member of the core team working on architecture and implementation of the system since the beginning of the project. Initially, it was a research project run by NEC Laboratories America in Princeton, NJ, USA. After a few years, HYDRAstor was introduced as a commercial product available for sale in Japan and the US. Since then, this product has been recognized as a break-through technology. This recognition has been documented with multiple awards and distinctions:

- *2007 US Product of the Year* in the Backup and Disaster Recovery Hardware category awarded by TechTarget's Storage magazine and SearchStorage.com,

- *2008 Product Innovation Award* given by the Network Products Guide,

- *2008 American Business Award* for the first grid-based storage solution optimized for backup and archiving with unlimited scalability, effortless management, self-evolving capabilities and secure, reliable data deduplication,

- *2009 publication [26] in the 7th USENIX Conference on File and Storage Technology (FAST)* conference, which is the best storage-related conference in the world,

- *2010 the fastest deduplication storage system in the world* as shown by analysis of the most famous independent storage analyst Curtis W. Preston [68].

While working on HYDRAstor, my area of responsibility was the design and refinement of the load balancing algorithm and implementation of the related fault tolerance distributed hash table which is the backbone of the DSS system. In the course of this work I gained a lot of experience and insights into the design of DSS in general. One of the major challenges in creation of disk-based DSS is the design and implementation of proper data organization. All major features of DSS like scalability, fault tolerance, high availability, efficient storage usage and performance directly depend on the careful design of data organization in the system.

1.2 Problem Statement

Satisfaction of many functional requirements of distributed storage systems depends directly on the organization of data stored in the system. However, organizing data in a way that allows the system to meet each requirement separately in an optimal way is an impossible task, as many requirements are indirectly conflicting with respect to data organization. That is, for many pairs of requirements, organizing data in a way that improves the fulfillment of one requirement reduces the level of fulfillment of the other requirement. As a result, the selection of proper data organization is a balancing act. In this thesis we describe trade-offs among the DSS requirements influenced by data organization. Next, we show how to address these trade-offs with proper data organization using as an example a commercial DSS HYDRA-stor, which to our knowledge is the only system that reasonably satisfies all the DSS requirements.

1.3 Thesis Contributions

The main contributions of this thesis are:

- Identification of DSS requirements dependent on data organization and description of these dependencies.
- Identification of trade-offs among indirectly conflicting DSS requirements (as defined above).
- Proposal of a novel data organization that addresses all the distributed storage requirements.

- Description of functional implementation based on proposed data organization in a highly successful, commercial system.
- Analysis of the requirements satisfaction and trade-offs resolution of proposed data organization.

1.4 Outline of Dissertation

This thesis is structured as follows. In chapter 2 we define distributed storage systems (DSS), discuss the motivation for them and the importance of DSS data organization. Chapter 3 contains a description of DSS functional requirements whose fulfillment is dependent on the organization of data stored in such systems. In chapter 4 we show that many of these requirements are conflicting and discuss details of related trade-offs. Chapter 5 describes our solution in a commercial DSS called HYDRAstor, and discusses how the requirements and trade-offs are addressed in this solution. Chapter 6 contains a review of related work, in which we outline data organization in other DSS systems and show differences and similarities between these approaches and our solution. Conclusions and future work are given in chapter 7.

Chapter 2

Distributed Storage Systems

2.1 What is DSS

Data storage plays a crucial role in information technology. All storage devices like magnetic and optical disks and magnetic tapes have physical limits to their capacity and performance. One approach to address these limitations is to group storage devices into multi-device entities like RAID arrays for disks and tape robots for tapes. The resulting systems may have quite substantial capacities. However, these architectures have serious drawbacks - they are very difficult to grow both in terms of storage and performance. Upgrading frequently is destructive, i.e. the old, smaller device needs to be got rid of and replaced completely with a new, larger device instead of adding more capacity to the old device. Moreover, both failure tolerance and availability are limited and are not satisfactory for the most demanding customers like enterprise companies. To address these limitations distributed storage systems are created. These systems consist of multiple storage devices connected via a fast computer network. The capacity and performance of such systems is the sum of capacity and performance of the individual storage devices. Additionally, in contrast to RAID arrays and tape robots, such systems can grow from one or few nodes incrementally by adding new nodes without interrupting system operation. Distributed storage systems provide interface to store and retrieve data, but in general these systems hide the number and type of equipment with which the system is built from the user.

According to DSS taxonomy[63], the following categories of DSS exist based on application functional requirements:

- *Backup and Archival.* Systems in this category provide the ability to backup and retrieve data (backup) and store data for long-term read-only retention (archival). High reliability, availability and scalability are the main objectives. For backup, efficient storage usage and high streaming performance are also very important. Such systems have to provide strong consistency i.e. in case of any failures the system should resolve any data inconsistency without user interaction. The usage pattern is write-once and read-many with no data updates. Examples of research systems are Pergamum [86], CFS [21], PAST [25] and OceanStore [75]. Examples of commercial systems are ExaGrid [1], EMC Centera [3], EMC GDA [4], Symantec NetBackup 5020 [88] and EMC Avamar [2].

- *General Purpose Filesystem.* Systems in this category provide persistent storage with filesystem like interface that complies with most, if not all, POSIX API standards [47]. Examples of such systems are NFS [77], AFS [53], Coda [78], xFS [8], Farsite [7] and Ivy [55].

- *Publish/share.* The main objective of systems in this category is providing (to some extent) volatile storage for publishing or sharing files among users. The most famous system in this category is Napster [57] and its variants like Gnutella [57], MojoNation [96] and BitTorrent [42]. Often systems in this category have completely decentralized peer-to-peer architecture (Free Haven [23], Freenet [17], Publius [92]).

- *Performance.* Systems in this category satisfy the requirements of I/O-intensive parallel applications. They stripe data across multiple nodes to reach high bandwidth. Many of the systems in this category are parallel file systems operating within a computer cluster. Examples of systems in this category are PPFS [44], Zebra [41], PVFS [38], Lustre [14], GPFS [79].

- *Federation Middleware.* Systems in this category integrate heterogeneous storage systems belonging to many institutions into single, consistent interface. Examples of such systems and related research are data grids [16] and SRB [10, 71].

- *Custom*. Systems in this category are characterized by a unique set of functional requirements, typically being a mixture of the above system categories. Examples of such system is Google File System [34].

In our work we focus on the *Backup and Archival* category. There are many reasons for this decision. First of all, the market for backup and archival storage is enormous [33], as practically every company needs to backup all its data. The customer needs on this market for scalable capacity and performance are to a large degree unsatisfied. Backups are done mostly to tape robots and single-node RAID arrays, both of which are not scalable. Backup data often contains many duplicated blocks, as full backups are done usually at least once every week. As a result data deduplication is very important in this market. Additionally, building DSS is much more feasible for the backup and archival market than for the primary storage market. This is because distributed storage usually comes with high latency, which is well tolerated by backup applications, but can be highly problematic for primary storage. Last but not least, recent research advances, hardware developments and pricing trends make building a commercial DSS for the backup and archival market possible. Techniques like distributed hash tables [85, 73, 51], erasure codes [12], Rabin fingerprinting [65] are all directly applicable to such systems. The advent of very fast and cheap multi-core CPUs, multi-terabyte SATA drives, 10 GigE networks are the main developments which offer an opportunity to create a scalable, disk-based data distributed storage system which supports deduplication. Such a system would perfectly meet unsatisfied needs for the retention of highly redundant backup data.

2.2 Motivation for DSS for Backup and Archival

As discussed in Section 1.1 the main motivation for creating the DSS is an exponential growth of data that must be retained. Furthermore demand for greater application availability shortens both backup window and time for recovery after failure or disaster. A common trend is that more data has to be backed up in a shorter space of time. Existing systems do not meet these capacity and performance requirements. Additionally, demand for higher capacity and performance increases backup environment complexity and management overheads. These problems are reflected in customers' perceptions of quality of backup and recovery processes. According to Enterprise Strategy Group report [89] 66 percent of users claim that backups take too long, 49 percent claim that recoveries take too long, 40 percent complain that backups and recoveries consume too many human resources, 37 percent complain that it is difficult to validate backup/recovery success and 33 percent claim that backup media management is a problem for them.

Tape libraries are still the most popular for backup storage today. Tapes must be stored in special conditions ensuring proper temperature and humidity, otherwise their lifespan is degraded [18]. They require strong commitment from the backup administrator in the processes of data backup, data management and data recovery. Because of sequential access, tapes also have a very long data seek and access time, which slows down the recovery process and makes single-file restoration very long. Sequential access makes tapes unusable for an on-line archive solution.

Another problem is that most tape drives can efficiently operate only at their maximum designed throughput. To record a high-quality signal, the recording head must be moved across the tape at high, constant and predefined speed. If a tape drive does not receive enough data, the drive spends some time waiting for its buffer to be filled up with data, rewinds the tape to the exact place where the last write took place, writes at the maximum speed and repeats the whole cycle again. This effect is called *shoe-shining* [66]. It results in a throughput lower than the throughput of data arriving to the drive and negatively impacts the tape lifespan as each tape has a limited number of passes (i.e. the number of times the tape passes over the drive head). To avoid shoe-shining, backup applications use a technique called *multiplexing* - they send multiple data streams simultaneously to a tape drive to reach its designed speed. However, such technique results in data being interwoven on each tape. It has negative impact on restore operation, because the whole tape has to be read, but most of the data is disregarded. Multiplexing also decreases reliability, because a single tape stores streams of many backups, so if one tape is broken many, instead of one, backups become unrecoverable.

As stated earlier, the scalability of tape libraries is not incremental. Even though tape libraries have a huge capacity, a single instance may not provide the required throughput. If the throughput of existing equipment is exhausted, moving to the next level involves considerable capital investment not only in new resources, but also in additional space in a data center. In addition, multiple libraries complicate administration and create non-unified storage islands.

With respect to the price of raw capacity, magnetic tapes may look attractive when compared to other media like magnetic disks. However, typical tape usage schema grandfather-father-son (GFS) may require up to 25TB of raw tape capacity to backup 1TB of data of production environment, depending on retention periods [20]. This 25TB contains highly redundant data. If it was deduplicated then only around 1TB of raw data would have to be required to store the backup data. Deduplication is feasible with a disk-based solution, it is not feasible with tape-based ones.

Tape library solutions are also a logistical challenge. For example to keep backup of 40TB of production data organization needs to have 1250TB. This assumes GFS policy with factor 25 and an 80 percent efficiency usage for each tape. 1250TB is around 3125 of LTO-3 tapes [43] to manage - keeping track, retirement, offsite transport and storage of the tapes.

Last but not least, tapes are hardly ever checked for errors and corrupted data. There is no way to make automatic checks for smaller appliances that do not have robots that load and unload tapes automatically. Even in the case of sophisticated tape libraries tapes may be stored offsite and cannot be automatically loaded by library robots. What is more, only recently tape library manufacturers have added tape verification to their products [70]. Lack of backed up data verification leads to serious problems with data recovery. Gartner estimates that 10 to 50 percent of all subsequent restores from tape fail, depending on the time elapsed since the backup occurred. What is worse these problems are often unnoticed. Both Gartner and Storage Magazine report that some 34 percent of companies never test a restore from tape. Of those that do test, 77 percent experienced failures in their tape backups [72].

Because of problems with tapes and with the advent of lower-priced SATA-based disk arrays, software backup vendors added options to backup data to magnetic hard drives. Backing up to disk drives solves some of the tape-based solution problems. Disk drives provide random access and provide flexible throughput. Using disk drives backup software does not require the sending of multiple data streams simultaneously to a tape to avoid shoe-shining. Disk accept writes at any throughput. Disks can support much faster recovery. They provide fast single backup or even single file restoration. They are less sensitive to environment factors like temperature and humidity. Finally, since disks are always on-line they can be periodically and automatically checked for data corruption to avoid failures during data recovery.

However, disk-based backup using SAN and NAS as backup targets also has serious problems. First of all, provisioning and static assignment of space on target to backed-up volumes is a management nightmare. Performance is also limited, as backup applications write to filesystems, which impose significant overheads. Resulting backup files are usually very large and get fragmented all over the disk [67]. To avoid this problem, reduce the hassle with provisioning, and additionally make backup up to disks more friendly for existing backup applications *Virtual Tape Libraries* (VTL) were introduced.

VTL emulates a standard tape library, but writes data to disks. A VTL solution facilitates disks management because users only define the number of virtual tapes to emulate. Actual space provisioning and allocation of the appropriate amount of disks is done by the VTL. VTL also provides good performance because they write to a raw disk device. VTL makes use of a backup application usage pattern to store continuous, big chunks of data on disks which avoids fragmentation.

Existing disk-base solutions solve some of the tape libraries problem, but they are far from complete solutions that would address all backup and archive requirements. First of all, these solutions are not reliable. They are all based on RAID-5 or RAID-6 that have many disadvantages:

- They only provide a fixed failure resiliency meaning that users cannot define reliability levels in an uniform way.

- In a typical configuration there is no standby hard disk that would take over a failed one - until the administrator manually replaces a failed disk (which may take a long time) RAID runs in degraded mode with decreased resiliency.

- RAID typically has only one controller which is a single point of failure.

- The RAID array is also inefficient in data recovery - after disk fails and is replaced by a new one the entire disk is rebuilt even though it may have very little data.

Disk-based solutions are not easily scalable, because all of them form storage islands. These islands cannot provide global data deduplication which would greatly improve storage usage efficiency, decrease storage costs and simplify management by reducing required amount of hardware. Even though recently manufactured VTL and NAS are equipped with deduplication [99], this deduplication is done only within a single VTL or NAS, not among all appliances which reduces the deduplication ratio.

Due to all these factors there is a clear need for a new distributed solution that would integrate all the disks into a single coherent, reliable and scalable system which utilizes capacity efficiently.

2.3 DSS Model

We consider a system consisting of multiple physical nodes connected with an underlying transport network. Physical nodes can differ in the number of disks and their storage space, processing power and other resources. Communication between nodes is asynchronous. Nodes send messages to each other. The messages can be lost, duplicated, delayed for arbitrarily long time before being delivered; they can be also delivered out-of-order. However, if delivered, a message content cannot be corrupted or modified by a malicious user. In most cases, messages are delivered within a reasonable time and in the order they were sent. Physical nodes and network links are subject to failures:

- physical node fail-stop: a node hosting components stops operating forever; all data on local disks of this node is lost.
- transient failure: an event or series of events occur which render a node or a set of nodes temporarily unreachable. This case includes network partition, as well as temporary node failure that is later repaired by restarting this node in its last consistent state.

Physical nodes are assumed to be trusted. The system does not handle Byzantine failures, in particular malicious nodes conspiring to break the system. In the case of the failure of a destination node or the failure of the network link leading to that node, the communication layer does not provide information to other nodes whose messages cannot be delivered. Each physical node maintains its own clock. Clocks can differ by no more than some constant. We expect this constant to be in the order of seconds at most. Servers are stored in a controlled environment and are always available unless there are failures or maintenance work.

There are two types of operations:

- Interactive operations that handle direct user action like storing and retrieving data. Such operations are initiated directly by the user.
- Background operations that are not directly related to user actions. Such operations are started by the system itself. Example of such tasks are data transfer between nodes and data rebuilding.

User writes to system data blocks. These data blocks create user data stream when concatenated in order they were received. Users read data streams i.e. they read blocks in the same order as they were stored.

We define three types of systems:

- homogeneous system - all nodes are exactly the same,
- heterogeneous with proportional nodes - system consists of different nodes, but components of all nodes are proportional i.e. ratio between CPU power and node capacity is the same on each node,
- heterogeneous with non-proportional nodes - system consists of different nodes and relative resources of components that the nodes are built of are different.

2.4 Importance of Data Organization in DSS

Data organization is the physical and spacial arrangement of data hosted by a system. In the context of distributed systems we distinguish 1) high level organization that manages the placement of data among physical nodes - we will call such organization *data placement* and 2) low-level organization that manages data blocks on physical disks - we will call such organization *disk-level organization*.

Feasibility of DSS requirements is a derivative of proper data organization:

- For efficient storage usage, data should saturate the capacity of all nodes and disks when system is filled up.
- For high resiliency and high availability, the system should have redundant data and place it in a way that minimizes probability of data loss or data unavailability.
- For fault tolerance, data organization has to be distributed and there should not be a single point of failure.
- For scalability, the system should have its data organized in a way that is independent of the system size.
- For ease of system management, data organization should support self healing and self balancing.

- For performance, data organization should support both fast locating of data on nodes and disk and high throughput of write and read operations.

DSS requirements that depend on proper data organization are described in detail in section 3 on page 15.

Chapter 3

DSS Requirements Related to Data Organization

This chapter describes DSS requirements dependent on data organization.

3.1 Efficient Storage Usage

In distributed storage systems user data is placed on a subset of system nodes. Data placement policy should distribute the data in such a way that all available storage space on all nodes and disks is used up when system becomes full. All data should be organized as an automatically managed, global common storage pool. Such data organization would avoid over-utilization on some nodes and under-utilization on others. To reduce the likelihood of excess capacity data organization should require no, or very limited need for provisioning and the system should be easily scalable to allow capacity to be extended by adding new nodes just when it is needed, not up front preallocated. Additionally, data organization should not require setting sizes of any volumes or pools - such settings quickly become outdated during the system's lifetime and would result in constant human intervention.

We measure *efficiency of storage usage* as a ratio of size of user data stored in the full system to the total raw disk space of all disks of the system. Higher the ratio better the efficiency of storage usage. The ratio has no upper limit. This is because system can store more data than its total capacity, for example due to data compression.

In this thesis we also use term *storage utilization* which means ratio of raw disk space used by the data (user data, redundant data, metadata) stored in the system to the total raw disk space of all disks that are in the system. This ratio maximums value is 1. The high storage utilization is one of the factors determining the efficient use of disk storage.

3.1.1 Support for Deduplication

Proper data organization should allow to detect duplicated data and stores only one copy of it. Such feature used to be unfeasible mostly due to the high demand placed on the CPU. But due to hardware and pricing trends (increasing capacity of SATA drives and increasing performance of new multi-core CPUs coupled with a decrease in their prices), new technology like distributed hash tables [85, 73, 51] and exponential increase in the amount of data to be backed up, data deduplication becomes a must.

The deduplication ratio depends on user's backup policy. A common scenario is the creation of over twenty or so full backups of user data. Such backups typically contain data that differs very little. In such cases data deduplication is at least 20 to 1. By finding such duplicates, a system can store at least 20 times more data using the same capacity. Thus data organization must provide support for deduplication. For the best results deduplication:

- should be done globally against all data already stored in the system to make it as efficient as possible,

- should detect deduplication in storage objects i.e. after they are slightly changed after sequence of edits or modification, for example after insertion of one byte at the beginning of a file.

Further to lowering storage costs deduplication also decreases consumption of electricity (to power and cool the disk drives) and bandwidth (e.g. for system replication).

However, note that deduplication achieves the best results for full backups policy. If other backup policies, like differential or incremental, are used deduplication efficiency might be very low. For example, under incremental backup differences between a successive and the preceding backup are stored. Such differences contain almost no duplicates.

3.1.2 Deletion on Demand

The user has to be able to delete data stored in the system. On the one hand this requirement seems to be an obvious one. On the other hand an operation that simply removes an individual block conflicts with data deduplication, because deduplication leads to multiple ownership of data blocks. A block that is to be removed by a user may have been already used as a base deduplication for another write by this or another user. One could consider keeping counters for each block with a number of references. A delete operation would decrease the counter and a deduplicate write would increase it. However, since messages can be duplicated, counters would be decreased or increased in an inconsistent way unless some techniques were used to guarantee idempotency of these operations. Additionally, such counters would require writing data to disks in case of duplicate writes even though logically such writes do not require the storage of any data and thus do not require disk access. Unnecessary disk access would slow down the duplicate write. Since it is common that ratio of duplicate to non-duplicate writes is in the range of 8:1 to 9:1 such a solution would negatively impact the performance of write operations experienced by users. Thus proper data organization must be able to handle the above mentioned problem.

3.1.3 Handling Almost Full System

A system should operate normally with high, close to 100%, system capacity utilization i.e. in the case when the system is almost full. In a dynamically changing DSS there are cases when some nodes are full while other are underutilized. For example, after the disk failure of an almost full node, or when a full system was extended with new nodes. In such cases some nodes become full while others are not. Data organization has to be able to handle it gracefully without any major performance drop or other user noticeable symptoms. In particular, a system has to avoid the situation when one full node, by trying to get rid of too much data, transfers away the data to other nodes and ”infects” them with ”fullness”. In turn, the other nodes try to get rid of their data. Such chain of data transfers would lead to system instability.

3.2 Fault Tolerance

A fault tolerant system is able to continue operation, possibly at gracefully reduced level, instead of failing completely when some part of the system unexpectedly fails. Such a requirement is obvious and crucial for any storage system. In the case of DSS it is even more important than in centralized systems because DSS is constructed of a large number of components. Each component's failure probability stays at the same level, thus the increased number of components increases the probability that some part of the system fails. In particular, data must be organized in such a way that no single point of failure results in its loss.

To achieve fault tolerance data placement policy has to provide high data resiliency.

3.2.1 Data Resiliency

DSS has to store user data reliably. Therefore data should be organized in such a way that the probability of data loss caused by nodes or disks failures is very low. This is achieved by adding redundant data and spreading the original and the redundant data across nodes in a way that minimizes data loss when some parts of the system, especially disks, fail. Apart from appropriate arrangement of data, fast data reconstruction after failure is also very important in order to limit the time during which system operates with a lowered data redundancy level.

With the increase of the system size its data resiliency should at least stay at the same level. Note that this might be hard to achieve because a bigger system has more components, in particular disks, thus probability of failures also increases.

We measure data resiliency by two statistics:

- Number of concurrent nodes or disks failures tolerated by the system without data loss.
- Ratio of redundant data to raw user data.

There should be a flexible data resiliency policy to allow users to increase resiliency in exchange for decrease in efficiency of capacity usage due to higher ratio of redundant data to raw user data.

Data Reconstruction Data organization has to support the data reconstruction background process. After a disk or node fails data originally stored on it has to be efficiently reconstructed by this process in another location in order to reach the initial, desired resiliency level.

Data Scrubbing Data organization has to support data scrubbing [81]. Data scrubbing is a background task that periodically inspects disks data for errors, and corrects them using redundant data. This process is required to detect and fix potential latent errors i.e. errors that are undetected until the disks sectors are accessed.

3.2.2 Operation under Faulty System

Data organization policy has to handle a system while some of its components have failed. Such policy needs to:

1. Enable non-stop execution of data storing, retrieving and deletion. The user should not notice any symptoms except:
    - performance drop if before the failure system was utilized at close to maximum performance,
    - rejection of new block writes if they are not duplicates and disks were failed causing total system capacity dropping below amount of raw data written by users.
2. Assign a new place for data that is stored on failed or unreachable nodes and disks and then cause system to rebuild the data to reach the original level of resiliency. Resiliency has to be rebuilt to increase chances of surviving any additional failures. Finding new places for data hosted by broken nodes/disk should not cause the reassignment of data that is hosted by healthy nodes and should not cause any transfers except the ones that are required to rebuild unreachable or lost data.
3. Keep system in balanced state so as not to cause too much of a drop in performance. Perfect policy should ensure that the proportional decrease in performance is not greater than proportional decline in resources caused by failure.

3.2.3 Operation after Failure Repair

After failure repair system should reach resiliency level required by the user. Performance of the repaired system should not be lower than before the failure.

Data should be organized in such a way that after a failure is repaired the system should minimize the amount of data transferred between nodes. In particular, if the failure was caused by network partition so that some of the nodes where not reachable but did not lose any data, and during the failure no data was written or updated, there should be no data transferred after the failure is repaired. Data placement should be the same as it was before the failure.

3.2.4 Failure Reporting

Even though a system has to support non-stop read, write and deletion operations under failures, it should also report any failures to users and inform them of possible threats. In particular, data organization should support the easy detection and reporting of data state health. Users should be able to easily find out if their data is lost and if not, to know what is the chance of such a loss. There are at least two important statistics that have to be reported:

- resiliency level - reports the minimum number of nodes and disks whose failure will cause data loss,
- data lost - reports which data has been lost.

3.2.5 Data and Metadata Consistency During and After Failure

While a system operates under failure data organization must guarantee data consistency. The most problematic case is when a node or disk is unreachable while new user data is being written or existing data is being updated. After a failure is repaired data organization has to support proper integration of temporarily unreachable data with user updates done during the failure.

Another important issue is metadata consistency. Besides recreating missing user data a data placement policy algorithm has to rebuild proper information about metadata. This means that a system has to heal itself in a distributed environment that does not have any central, managing entity. Such healing must be properly designed in order not to lead to the creation of two or more equivalent instances of the system (i.e. split-brain) that leads to desynchronization of metadata and user data.

3.3 Scalability

Scalability is a very important requirement for DSS. Ideal data organization should guarantee that performance and capacity of the system grow linearly with the addition of new hardware. Availability, resiliency, fault tolerance and efficiency of storage usage should stay at the same level. It allows users to start with small setup and grow with increasing data volume.

While taking care of scalability data placement policy must also limit the amount of data transferred after node addition/removal. This is to limit the number of background tasks that use up system resources and to limit the time window in which the system is changing its data to nodes assignment. Such transient states in most cases are not optimal with respect to data placement requirements. Ideally, when a new node is added to or removed from a homogeneous system consisting of n nodes only $1/n$ of data stored by it should be transferred.

Scalability can be measured by comparing the increase in throughput after system extension to increase in system resources (i.e. number of nodes in case of homogeneous system). The ratio of increase in throughput to increase in resources is 1 for an ideally scalable system. Values of less than one means that system is not ideally scalable.

3.4 Self Management

As the complexity of distributed system increases, systems management tasks become significantly more complicated [84]. Management operations relied mainly on highly skilled human intervention and administration, but the availability of human resources is limited and their costs are increasing. At the current rate of expansion, there will not be enough skilled IT people to administrate the world's computing systems [45]. Besides, about 30-40% of all computer problems are attributable to system administrators errors [35, 36, 56]. A large distributed system requires high level of automation and self management.

In the context of data organization this boils down to at least the following requirements listed below.

3.4.1 Self Healing

Data organization has to support automatic failure discovery and rebuilding missing metadata and user data.

3.4.2 Self Balancing

Data placement policy has to automatically reassign mapping from nodes to data that they host when a system is rebuilding after failure or when an administrator adds or removes nodes. Balancing should be both accurate and responsive to changing conditions that are result of failures.

3.5 High Availability

In DSS data should be placed in such a way that temporary failures of any system component should not stop serving user requests and should not cause loss of user data. This requirement is motivated by the requirements for systems to be constantly on-line. A common industry requirement for large systems availability is ”four nines” (99.99%) or even ”five nines” (99.999%). The first means that a system can be unavailable for maximum 4.32 minutes per month, the second means that a system can be unavailable for maximum only 25.9 seconds per month.

3.5.1 On line Maintenance

In addition to unplanned events data organization has to support planned events related to system administration like hardware and software upgrades. It should be possible to mark nodes or disks for temporal removal from the system to let data temporarily migrate in non-disturbing manner to other locations.

3.6 Performance

Effective data organization is crucial to performance for any system which has multiple nodes and disk. Data should be placed in such a way that the load on each node and disk is proportional to its resources in order to avoid non-uniform distribution of the load and the formation of bottlenecks. There should be no hot spots i.e. nodes that are over-utilized due to storing data that is frequently accessed by users.

In a distributed storage system there are two important performance metrics:

- Aggregated throughput of write operations executed by all system users. It is measured in MB or GB per second. It is an aggregated value, because for large systems one user does not have enough resources like network bandwidth to fully drive a big system and because commonly all users store their backup at the same time. This is the main statistic that is used to compare different systems performance.
- Throughput of read operations. However aggregated throughput, specially in big systems, is less critical than in case of write operation, because it is unlikely that all users read data at the same time. Thus system should provide high read throughput for fraction of users reading their data concurrently.

Low response time of user write and read operations is less important than in case of primary systems like file systems, because backups are stored and retrieved as consecutive streams of data, not a separate data objects. High latency is well tolerated by backup applications.

There are several areas of performance requirements in distributed storage system that have to be supported by proper data organization.

3.6.1 Fast Direct User Operations

Data organization has to support good performance for operations directly visible by users like read, write and deletion. These operations require high throughput in case of read and write and a short time to complete in case of deletion.

3.6.2 Fast Background Operations

Background task operations are data transfer between nodes, data reconstruction after failures, data scrubbing, space reclamation, etc.

Tasks that increase resiliency (data reconstruction) or stabilize system by reaching desirable state (data transfer between nodes) are more important then other ones, thus they should have higher priority and be optimized for short time of execution.

Space reclamation should reclaim space fast enough to make the system able to accept new writes.

Other tasks like data scrubbing have lower priority, however they should be fast enough to execute within reasonable time like several days.

3.6.3 Fast Data Location

Data must be organized in such a way that locating and retrieving user data is fast and ideally does not require any central, prone to failure, directory. It is necessary to satisfy fast direct user operations. In addition to that, to support deduplication, data must be quickly found by its content to check if it is already stored by the system. Even in centralized systems this requirement is hard to achieve [99]; in distributed systems it becomes a big challenge, especially in a scalable, dynamically changing system where data is transferred from one node to another, where nodes with previously stored data may be unreachable, may contain outdated data (i.e. data that was already deleted while node was detached from the rest of the system) or contain subset of data it is supposed to store (i.e. in case some disks on given node were broken).

3.6.4 Disk Data Locality

Data organization should also take into account the locality of data on disks i.e. it should allow for the creation of disk level storage structures that would put blocks of a file ”near” each other so that many blocks of data can be read at the same time with a single disk head movement thus avoiding the disk seek penalties.

3.6.5 Optimal Initial Data Placement

Initial data placement for given hardware configuration should be the final data placement. The reorganization of data placement is an expensive process; thus a change in type of user load and amount of data in the system should not impact data placement

3.6.6 Performance under Failure

Under failure performance should be degraded proportionally to the percentage of failed nodes and disks. Tasks executed by failed nodes and disks should be distributed among healthy ones.

Chapter 4

Trade-offs and Mutual Benefits among Requirements

4.1 Trade-offs

Determining the optimal data placement with respect to all system requirements is problematic. For many pairs of requirements organizing data in a way that improves fulfillment of one requirement reduces level of fulfillment of the other requirement. Supporting all the requirements within one system is not possible. Some conflicts can be mitigated by proper data organization, whereas others are resolved by design decisions that prioritize some requirements and sacrifice others.

	Efficient Storage Usage	Fault Tolerance	Scalability	Self Management	Availability	Performance
Efficient Storage Usage	N/A	4.1.1	+	+	4.1.2	4.1.3
Fault Tolerance	4.1.1	N/A	4.1.4	+	+	4.1.5
Scalability	+	4.1.4	N/A	+	4.1.4	4.1.6
Self Management	+	+	+	N/A	+	+
Availability	4.1.2	+	4.1.4	+	N/A	
Performance	4.1.3	4.1.5	4.1.6	+		N/A

Table 4.1: *Requirements matrix.*

In addition, requirements may positively influence each other. Supporting one requirement might help to satisfy the other one. We call such relations mutual benefits among requirements.

Table 4.1 provides all combinations of any two requirements. If there is a trade-off between two requirements corresponding intersection contains number of section of this document that describes that trade-off. If there are mutual benefits they are marked with '+' sign and are described in section 4.2 on page 35.

4.1.1 Efficient Storage Usage vs. Fault Tolerance

There are the following trade-offs regarding data organization supporting both fault tolerance and efficient storage usage:

- Fault tolerance requires the storage of redundant data. Such data lowers the efficiency of storage usage. The tension can be mitigated by using efficient methods for redundant data like erasure coding [12] instead of data replication, but cannot be eliminated entirely. Erasure codes add resiliency to the stored data with fine-grain control between required resiliency level and resulting storage overhead. Details of erasure codes are described in section 5.2.2.3 on page 43.

- Data organization supporting fault tolerance imposes an arrangement of data that can cause inefficient use of disk space. For example, if there is data whose resiliency is to be provided by additional n replicas to handle n failures of system built upon $n+1$ nodes each node would have to store original data or its replica, thus each node would store the same amount of data. If the nodes did not have the same capacity then the bigger nodes would never by fully utilized. A similar problem exists when instead of replicas correction codes are used. For example all disks in RAID systems have to be the same size to fully utilize their capacity.

4.1.2 Efficient Storage Usage vs. Availability

If each storage node had unlimited storage space then each piece of data could be stored on each node to maximize the availability of that data. Obviously, such a solution is not acceptable due to economic factors i.e. the cost of storage. In large systems it would also lead to problems with the efficient management of a huge number of data copies and problems with the network's bandwidth that would have to transfer copies of each piece of data. On the other hand, there is a solution where each node stores only original pieces of data without any redundant data. Such a solution maximizes efficiency of storage usage but results in very low availability. Inaccessibility of any node causes some data to be unavailable. Thus the final data organization solution has to be somewhere in between.

Efficient Storage Usage vs. Reliable Detection of Duplicated Blocks on Write

While some nodes are not reachable it may not be possible to deduplicate new data being stored because a system cannot verify if the same data has already been stored. In such a case one possible option would be to favour availability over efficient storage usage and store the data even if the duplicate was not discovered. Another option would be to block the write until nodes are reachable, which would sacrifice availability in order to gain efficient storage usage. To limit this trade-off data should be organized in such a way that deduplication checks can be reliably done even if multiple nodes are unreachable.

4.1.3 Efficient Storage Usage vs. Performance

We have identified the following areas for trade-offs regarding performance and efficient storage usage.

Providing Resiliency

There are two main data organization techniques for providing resiliency - data replication and some kind of data coding i.e. erasure codes. Replicas are better suited for performance than coding - the creation of replicas is a relatively fast process in terms of CPU usage; only one replica is needed to retrieve data, thus the system only needs to read data from one disk. In contrast, data coding and decoding is CPU intensive process. Further to this, retrieving data requires reading data from many disks. On the other hand coding uses disk space much more efficiently than replicas for the same level of resiliency [94], so it is much better suited to efficient storage usage.

Heterogeneous, Non-proportional Nodes

In the case of a system built upon heterogeneous nodes that do not have the same ratio of storage capacity and node speed (in broad sense the speed is the number of CPU cores and their speed, RAM speed, network speed and disk speed) there is a conflict between reaching the highest possible performance and using all available storage space. If data was organized in such a way that each node storage space was to be consumed on an ongoing basis proportional to node capacity, the system would have to slow down to the slowest node in the system i.e. node with the smallest ratio of speed to capacity. On the other hand, if the system was to perform as fast as possible then the slowest node would have to receive a smaller fraction of writes. This fraction of writes would be less than the ratio of this node disk space to total space of all the other nodes. This is because this fraction would be proportional to node speed, not to node capacity. This would result in unequal utilization of storage space on nodes and the system being unable to accept new data without the execution of some background process that would transfer data from over-utilized nodes to under-utilized ones. Such processes would only make sense if there were enough idle time windows, i.e. when there are no user activities to consume system resources. If this was not the case such background process would be in conflict with direct user operations i.e. writes and would decrease write performance anyway. Furthermore, such a background process does not help the performance of data retrieval that would be slowed down by the slowest nodes.

Deduplication vs. Performance

The impact of deduplication on write performance is ambiguous and it depends on the ratio between duplicated and non-duplicated data in the user data stream. A write with deduplication is more complicated than non-deduplication write because before data block is stored on disk the system needs to check if that block already exists. In the case that it does not exist, the system does additional work before the actual data storing which has negative impact on performance. On the other hand, if it is a duplicate write then the check for duplicate is the only operation that has to be done. If the data organization is properly designed checking for a duplicate could be faster than actually storing user data. This is because in order to provide resiliency a system needs to access many disks to store replicas or coded data on many nodes. However, verification that a given piece of data is a duplicate may not require any disk access at all. For example some kind of small data summary can be kept in RAM to answer deduplication queries directly without disk I/O. Thus the performance of writing streams containing many duplicates might be faster on a system supporting deduplication than on a system without this feature.

Deduplication has negative impact on read performance. Deduplication devastates data locality because during a write deduplicated parts of user data stream are scattered over nodes and disks where the data was previously stored. Reading such stream might result in close to random access to disks that hurts performance.

Efficient Storage Usage vs. Fast Data Location

For the best storage usage efficiency there should be no metadata information about the location of stored blocks, because such information occupies storage, thus lowers efficiency of storage usage. Blocks should be simply stored on free sectors of disks. Obviously, such a solution would be extremely inefficient when it comes to locating a given block. The only possible method would be to scan all blocks stored in the system. An alternative to this would be a data organization with a global directory mapping each block's id into its physical location on the node and disk. The directory would serve requests for block's data location very quickly. However, the directory would have to be replicated on many nodes and disks to avoid single node/disk bottleneck. Such a solution provides fast data location but storage usage wise it is inefficient. The final solution should be found somewhere in between these two methods.

4.1.4 Fault Tolerance and Availability vs Scalability

Every increase in system size increases the number of hardware components it is built from. Since the probability of each component failure does not change, the larger the system and higher the number of components, the greater the risk of failure. For big systems failure is the norm rather than the exception. Even only taking into account magnetic disk empirical results, analysis in [80, 62] show that annualized failure rates (AFR) range from 1% to 13% which is much higher than the information provided by disk vendors. Common AFR is around 6%. This means that a system built of 1000 disks experiences disk failure every 6 days on average. Similar to the trade-off between scalability and failure tolerance, availability may be decreased in a big system because of the increase in probability of failure in a situation when a system has a single point of failures. On the other hand, a bigger system has more components, thus data can be organized in a way that provides redundancy increasing both fault tolerance and availability.

4.1.5 Fault Tolerance vs. Performance

There are several data organization issues regarding performance vs. fault tolerance:

- Fault Tolerance is improved by spreading each redundant data over many or all nodes. However, such a wide spread has a negative impact on the performance of writes because the additional data has to be stored and the write operations are slowed down by the slowest node storing the data. The latter problem can be mitigated by ignoring the slowest nodes as long as certain, minimal level of redundancy is preserved and hoping that later background process will reconstruct and store the missing data. Albeit, such an approach is a pure example of trade-off between fault tolerance vs. performance.

  However, data redundancy may improve performance of read operation, because there are many nodes and disks that can serve the request, and the fastest ones can be chosen.

- Similar to capacity vs. performance trade-off there is a tension between types of redundancy i.e. replications vs. coding. On the one hand, erasure coding gives better fault tolerance than replicas [94]. On the other hand replicated data is much faster to read - only one disk on one node needs to be accessed in contrast to coding that requires reads from multiple disks. Additionally, in both read and write operations coding requires more CPU power which also has negative impact on performance.

- A fault tolerant system must execute additional processes which verify data health and rebuild data after failure to reach original data resiliency. Such additional processes consume system resources that cannot be used by regular read and write operations. Thus, the additional process may lower system performance.

4.1.6 Scalability vs. Performance

In a distributed system, data placement policy should distribute load and user data evenly over all nodes to reach both high performance and even space utilization. This solution may become troublesome in a big system because of the assumption that all nodes participate equally in processing requests - in case of a node or a disk failure the system's performance drops until the failure is repaired manually or automatically. Since probability of any failures increases in bigger systems, it may negatively impact performance. The problem may be even more severe in cases when a failed component does not stop but executes with lowered performance - such a failure negatively impacts performance, but may not be easily detected for a long time.

Scalability vs. Fast Data Location

DSS stores pieces of user data over all nodes. The efficient management of the locations of each data block is harder to achieve in bigger systems. A naive data organization solution like keeping some kind of central directory of data blocks and their locations provides fast data location, but is not scalable. This is because in a big, dynamically changing system it would be difficult to keep the directory up to date. On the other hand, there is a solution where each node keeps directory of a subset of blocks, for example blocks stored locally. Such a solution is scalable in terms of keeping the directories information up to date, but to find a single block all nodes would have to be queried which would overload the nodes. As a result, ideally we need a solution which is both scalable and allows for fast data location.

Scalability vs. Fast Background Operations

The performance of background operations may suffer in a big system due to the following reasons.

Similarly to regular operations, background operations might have to operate over all system's data thus they would have to contact all or most of the nodes of the system. A transient performance problem with any of the nodes may slow down background operation processes. This is more serious for bigger systems.

In bigger systems, especially the ones without any central point of control, it is harder to control interoperability of many background processes. They may interfere with each other and slow themselves down. In other words, it might be hard to distribute background processes work evenly over all nodes.

Scalability vs. Disk Data Locality

Data organization preserving disk data locality is crucial for efficient disk read and write operations. It may be hard to achieve this in systems built upon many disks. Data that is written to the system should be stored on as many disks as possible to utilize total throughput of all disks. As long as only one user writes data that is distributed over all nodes, locality on disks may be accomplished. Problems may occur when many users write data streams concurrently which is common in big systems. To reach high throughput of disk writes those streams should be stored on consecutive sectors of every disk to avoid head seeks. However, this leads to streams of data coming from different users becoming interwoven on the disk. Thus, there would be no disk locality with respect to one stream. Later on, read operations for that stream would be inefficient because data would have to be read from non-consecutive parts of the disk.

4.1.7 Intra Requirements Trade-offs

While working on design of DSS, in addition to trade-offs between separate requirements, we have identified noticeable trade-offs within some requirements. These intra trade-offs refer to fault tolerance, self balancing and deduplication requirements.

Fast Data Recovery vs. Resiliency For Concurrent Failures

Within fault tolerance requirement alone there is a tension between data placement policy, that minimizes probability of data loss when many concurrent failures occur, and data placement policy that supports fast data reconstruction. Reconstruction time determines the length of time that the system is vulnerable to data loss caused by additional failures. Fast reconstruction gets system quick back to faulty-free, resilient state.

In order to examine this trade-off in detail let us consider system built upon n nodes. Assuming that the system guarantees that k nodes can fail without data loss, there must be some kind of data redundancy. For simplicity and without loss of generality let us assume that redundancy is provided by storing k replicas for each user block. We will consider two data placement policies. The first policy randomly spreads each user data block and its replicas over all nodes. If there is enough data written to the system, then for each $k+1$ element subset S of nodes there is a data block such that this block and its all replicas are stored by nodes from S. Thus, a concurrent failure any of $k+1$ nodes causes data loss. The probability of data loss is 1. However, data recovery under such data placement is very fast, because when a node fails all its data can be recreated by reading replicas from all other nodes in parallel. The second policy divides n nodes into g disjointed groups of size $k+1$. The user data block and its replicas are stored within one randomly chosen group. Such data placement has lower probability of data loss when $k+1$ nodes fails, because data loss occurs only when nodes building a group fail. For $(k+1)$ being a divisor of n probability of data loss is $\frac{n/(k+1)}{\binom{n}{k+1}}$, where $n/(k+1)$ is the number of groups and $\binom{n}{k+1}$ is the number of all subsets of size $k+1$. Thus, the probability of data loss is lower than 1 as in case of the first policy. However, data recovery under second data placement policy is slower, because after a node fails the recovery process can only read data from k nodes, thus it is slower by $(n-1)/k$ factor compared to the first policy. When a second node within that group fails, recovery is lowered by factor $(n-2)/(k-1)$ and so on.

To sum up, a system that supports many simultaneous node failures (the second policy) tends to need more time to reach its original resiliency level than a system that is more prone to concurrent node failures (the first policy). The first policy is better for failures uniformly distributed in time, the second one is better for multiple failures occurring at the same moment followed by long non-failure periods of time.

Local Balancing vs. Global Balancing

Within balancing itself there is another intra trade-off. Any data placement policy has to dynamically balance a system after events that change a set of available nodes and their characteristics like capacity or CPU power. Such events may be planned events like adding, removing and replacing hardware and unplanned events that are results of failures i.e. some node being unreachable, disk failures, node failures, network error, etc. Such dynamic balancing can be done at two levels: local and global. By balancing at a local level we mean actions taken by subset of nodes that balance a system without any global management. A node takes action based on limited information received from nodes that are its neighbours, thus that information is incomplete and does not describe the state of all nodes. By global balancing we mean taking actions that are coordinated by some global entity that decides what actions to execute based on information gathered from all alive nodes.

Both types of balancing have advantages and disadvantages. Local balancing is quickly adaptable and starts balancing actions shortly after an event. However, its decisions may not be globally optimal and are not stable. This is because they are based on incomplete information that may vary and actions are not coordinated, possibly leading to conflicting decisions that abandon each other. Our experience is that local balancing works properly as long as the requirements for it are simple and decisions require very limited information. On the other hand, global balancing finds data placement that is globally optimal and stable. One problem with global balancing is that it requires long time to execute actions because it needs to gather information from all nodes, leaving a system in unbalanced state for a long period. It also requires a global entity thus it is prone to failures and may be a bottleneck limiting scalability.

Inline vs Offline Deduplication

There are two main methods of implementing deduplication: inline and offline. Inline deduplication finds duplicated blocks on the fly while data is written to the system and only non-duplicated blocks are stored. Offline deduplication first stores all the written data on some short term storage, finds duplicates, removes them and moves non-duplicated data to long term storage later on in a background process.

Inline deduplication minimizes disk capacity requirement but its performance might be negatively impacted by deduplication being performed during regular write operation. Inline deduplication data organization is simple to manage because data is stored only once it is in its final version. This limits the number of disk I/O operations. The inline deduplication system has a simple administration policy, because the next backup operation can be started just after the previous one is finished. Inline deduplication also provides a safe and efficient storage system replication on a remote site for disaster recovery. It is common policy to set up a secondary system that mirrors data of the primary one. In the case of disaster of the primary system, the secondary one can be used for data restore. The primary system can send newly stored blocks (ones that were not duplicates) to the secondary system just after a write operation is finished - the time lag is very small.

The advantages and disadvantages of offline deduplication opposite to those of inline. Offline deduplication requires more storage for storing written data in original form to be used later by background operation process. However, since deduplication is not carried out while data is being written, the performance of non-duplicated write can be better. With regards to administration offline is more complicated than inline because the administrator needs two polices for each backup - one for writing the backup and the other for letting the system do the deduplication as a background process.

Offline deduplication also complicates disaster recovery and makes it more dangerous because it has to wait for the background deduplication process to be finished before sending non-deduplicated data to the secondary system. During that window time data is not mirrored, which increases the chances of user data loss in case of a primary system disaster. Last but not least, two distinguished capacity areas make the data organization more complicated and harder both to implement and support.

To sum up, apart from performance, inline deduplication is better in all aspects of data storing. Performance depends on actual implementation of the system.

4.2 Mutual Benefits

For some pairs of requirements, organizing data in a way that improves the level of fulfillment of one requirement increases the level of fulfillment of the other one.

Data organization which supports availability also supports resiliency. This is because to support both of these requirements, redundant data has to be created. Another reason is that one might consider availability as a stronger version of resiliency. Resiliency guarantees that user data exists, however it may not be reachable. Availability guarantees that data is reachable which implies that the data exists.

Scalability may positively influence efficiency of storage usage. This occurs when data organization supports global deduplication. Since a bigger system has more data, there are more options for data blocks to be deduplicated against each other. More options result in better data deduplication which improves storage usage efficiency.

Self balancing and self healing results in the automatic adaptation of the system to changes in the environment. Automatic system reaction is commonly faster and more accurate than human intervention. Thus, self management shortens the time during which efficiency of storage usage, fault tolerance, availability and performance are degraded.

Chapter 5

Proposed Solution

5.1 HYDRAstor DSS

5.1.1 History and Current Status

In 2002 I began working on a new DSS project run by NEC Laboratories America, Inc. We designed and developed a novel distributed storage system called HYDRAstor. To my knowledge, this system is the first commercial implementation of a highly scalable, high-performance content-addressable storage system supporting global duplicate elimination, per-block user-selectable failure resiliency, self-maintenance including automatic recovery from failures with data and network overlay rebuilding. From the beginning, HYDRAstor was designed to address the DSS requirements described in the previous chapter. This was achieved with original ground-breaking research utilizing creatively the latest technologies. One-way secure hashing like SHA-1 [29] and content-addressable storage paradigm (CAS) [3, 69, 99] were our inspirations for a fast and safe duplicate elimination. Distributed hash tables (DHT) [27, 51, 73, 76, 85, 98] allowed us to build a scalable, failure resistant system and extend duplicate elimination to a global level. Erasure codes [12] added space-efficient redundancy with fine-grain control between redundancy levels and storage overhead.

I have been involved in the project from the first day of its existence. During this time I have been working on design and implementation of data placement policy and data organization that were the key for reaching the above requirements. The result of this work is described in this chapter.

5.1.2 Additional Requirements

Beside the generic distributed storage system requirements described in the previous chapter we had to take additional specific requirement into account while working on design of the system:

- The system should survive a configurable number of disk or physical nodes (machines) failures. That number is in the range of 2-11. The default number is 3.
- The system should scale from one to hundreds of physical nodes.
- In big systems, the computer network is not homogeneous. Physical nodes are grouped into racks which have a default number of 10 machines. An intra-rack network has a higher throughput than an inter-rack network.

5.2 Architecture of HYDRAstor

5.2.1 Overview

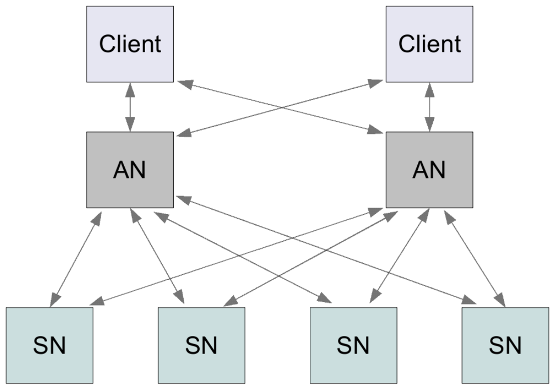


Figure 5.1: *HYDRAstor architecture. Clients communicate with ANs. Each AN communicates with all SNs. Physical node may host singe AN, single SN or their combinations.*

HYDRAstor system consists of two types of logical nodes: (1) frontend nodes called *Access Nodes* (ANs), and (2) backend nodes called *Storage Nodes* (SNs). ANs and SNs can be hosted by separate physical nodes or can be combined and hosted by one physical node. Figure 5.1 shows example of ANs and SNs configuration.

Clients communicate with ANs which role is to distribute read and write requests to SNs. SNs take care of storing data on disks and finding duplicated data. The entire system can be accessed with a regular file system interface which is exported by *Hydra File System* (HFS) [91] built on top of ANs. HFS uses HYDRAstor driver interface exported by ANs. That low-level interface may be used to implement other standard access protocols, for example VTL or SMB/CIFS protocol [82]. ANs can also be used to decrease network traffic, because ANs can cooperate with SNs to find out duplicated data before actually sending it to the backend. Deduplication done on ANs is called *deduplication on source* in opposition to deduplication on SNs called *deduplication on target*.

In this work we concentrate on the functionality of the backend, as it alone constitutes a full blown example of DSS. It provides support for read, write and deletion operations, automatically reconfigures data placement after node additions, removals and failures; and automatically rebuilds data resiliency to required levels after failures.

High-level data placement in backend is organized around *Fixed Prefix Network* (FPN) (section 5.2.2.2) that is our version of distributed hash table (DHT) (section 5.2.2.1). In HYDRAstor, each FPN nodes is replaced with a *supernode* (section 5.2.2.3) which spans over many physical nodes for high resiliency and availability. Local, disk-level data organization supports streaming access to disks. Data is stored in containers that are organized into chains (section 5.2.3) to allow fast storage of data streams and enable efficient data consistency management, data health verification and data reconstruction.

In the next section (5.3) we describe how these global and local data organization allowed us to implement functionality of the system.

5.2.2 Global Data Organization

5.2.2.1 Distributed Hash Tables

Since scalability and failure resilience are major requirements of data organization, the use of distributed hash table was a natural choice.

Distributed hash tables provide a scalable mechanism of mapping keys onto values, delivering standard functionality of a hash table. It is possible to store a key and value pair, and to pick up a value based on the key.

Mapping from keys to values is distributed among the nodes, that is the entire hash space is divided into disjoint regions which in turn are assigned to participating node. DHTs are fully decentralized and they scale very well. Node addition or removal causes usually minimal disruption of the mapping. In most cases they provide *consistent hashing* - adding or removing a node results in remapping only k/n keys where k is number of keys and n is number of nodes. DHTs are also failure resilient - they rebuild their structures even with nodes continuously joining, leaving, and failing.

To satisfy these requirements, each node of a DHT maintains links to only a few other nodes in the system commonly called *neighbours*. These links form the overlay network. Typically each node has $O(\log n)$ neighbours in a system with n nodes. The method of selecting neighbours determines the network topology.

The hash table's key space is a set of bit strings, typically ranging from 128 to 256 in length. Each node is assigned an identifier and a node owns keys that are closest to its identifier according to some pre-defined metric. To find a node owning given key k, a node checks if it is the owner. If not it passes the key to the neighbour that is closest to k in terms of this metric. Typically, the maximum number of hops in any route (route length) is $O(\log n)$ when node degree (number of neighbours) is $O(\log n)$ (Chord [85], Kademlia [51]). Other common options are route length $O(d\sqrt[d]{(n)})$ for degree $O(d)$ (CAN [73]) (where d is system constant) and route length $O(\log n)$ for degree $O(1)$ (Koorde [48]).

There are many proposal of new DHTs in recent research. The most well known of these are Chord, CAN, Kademlia, Tapestry [98], Pastry [76]. All these proposals were designed for peer-to-peer networks [93] that have thousands of nodes and are characterized by a high ratio of nodes arrival and failure. The system designers were more focused on small node degree and fast reconstruction of nodes state after failure or node departure than short routing paths. In a commercial system that consists of no more than a thousand of nodes and operates under controlled environment where nodes are added and removed in planned way the priorities are the opposite - short routing paths are more important than the size of a node state. Thus, slightly different DHT than the ones available so far was needed . That is why HYDRAstor is built upon a specialized DHT named Fixed Prefix Network (FPN) [27].

5.2.2.2 Fixed Prefix Network

This section presents some preliminary material. It is presented for completeness to make the thesis self-contained. The reader familiar with Fixed Prefix Network can move straight to section 5.2.2.3 ‘FPN with Supernodes‘ on page 43.

Fixed Prefix Network (FPN) is a distributed hash table. Hash keys have the same length S. FPN divides hash key space into *FPN nodes*. FPN nodes are hosted by physical nodes. Each physical node may host many FPN nodes simultaneously.

FPN node is identified by string of bits. It is called an FPN node *fixed prefix*. An FPN node with fixed prefix

$$f_1 f_2 \cdots f_k$$

is responsible for hashes starting with this prefix i.e. hashes of the form

$$f_1 f_2 \cdots f_k x_{k+1} x_{k+2} \cdots x_S$$

where $x_{k+1} x_{k+2} \cdots x_S$ are any bits.

Figure 5.2 illustrates prefix tree of FPN nodes. Leaves represent FPN nodes. Digits along the path from the root to the leaf are FPN node prefix. Two values with keys $K_1 = 10000101$ and $K_2 = 10101011$ would be stored on nodes H and J, respectively.

FPN ids do not overlap with each other i.e. for each hash key there is only one FPN node with id being the hash key prefix. Additionally, a healthy FPN network i.e. FPN with all FPN nodes alive has hash key space fully covered i.e for every hash key there is an FPN node with an id that is a prefix of the hash key.

Two FPN nodes are neighbours if their fixed prefixes differ only on one bit position and the rest of the bits are the same. If fixed prefixes differ in length only the first $shorter\_length$ bits are considered, where $shorter\_length$ is the length of the shorter fixed prefix. Thus, in FPN with all fixed prefixes length equal to l each node has l neighbours. In FPN with different lengths of fixed prefixes, FPN nodes with shorter prefixes may have more than one neighbour along a given bit. For example, consider node E with fixed prefix 010 from figure 5.2. Its neighbors are nodes: K (prefix 1100) and L (prefix 1101) along the first bit; A (prefix 0000) and B (prefix 0001) along the second bit; and F (prefix 0110) and G (prefix 0111) along the third bit. If node E was replaced by nodes with fixed prefixes longer by one bit $E0$ (prefix 0100) and $E1$ (prefix 0101) they would inherit E's neighbours in such a way that they would have only one neighbour along any given bit. $E0$'s neighbours would be K, A, F, and $E1$'s neighbors would be L, B and G.

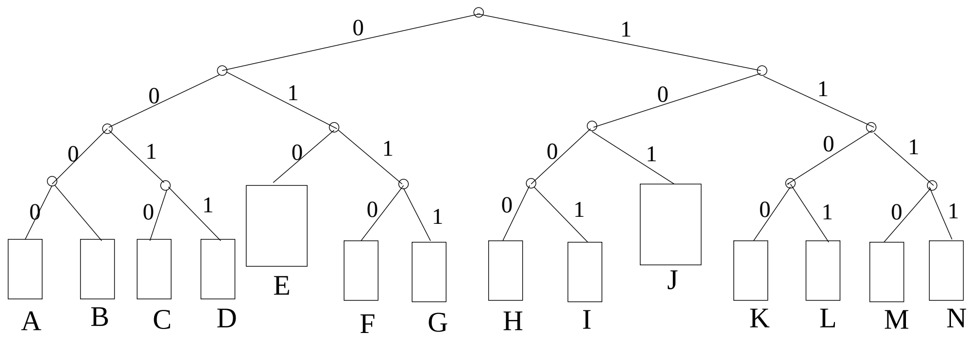


Figure 5.2: *FPN example.*

source: [28]

Routing Basic routing is done by resolving bits from left to right. It guarantees that already resolved bits are not lost and that there will be no more than max_prefix_length hops where max_prefix_length is the longest fixed prefix in the system. max_prefix_length is in the order of the logarithm of number of FPN nodes.

To decrease the number of hops, each FPN node keeps *jump tables*. Each fixed prefix is divided into groups of bits of size D. Groups start at bit position 0, D, $2D$ and so on. Such groups are called digits. With each digit there is associated information about the location of FPN nodes having fixed prefix of the form:

$$f_1 \cdots f_i d_1 \cdots d_D f_j \cdots f_k$$

where f denotes bit value of local FPN node and d denotes any bit of the digit. Such information is called jump tables. As with bits they are used for routing. Instead of resolving bit by bit they are used to resolve digit by digit from left to right. Resolving by digits decreases number of hops. Digits can be dynamically increased while system increases. It allows the maintenance of a limited number of hops while the system increases in size. By keeping additional $O(dn^{1/d})$ node locations, where n is number of nodes and d is is the number of digits in fixed prefix, the number of hops is limited to $O(\log d)$

Pings Neighbouring FPN nodes periodically (every T seconds) send each other short messages called *pings*. Pings are used to detect neighbours' failures and propagate changes of the network. Pings carry information about sender location and its version (the version is increased when an FPN node changes the physical node hosting it). Pings also carry information about changes in a jump table for a given digit if it is sent to a neighbour along a bit belonging to that digit. After receiving a ping the node reconciles its state. A node version is used to find out the latest information. After $D * T$ seconds (where D is digit size) information about the jump table changes is updated on all involved nodes.

FPN Operations

FPN Node Split FPN Node split operation creates two children nodes in place of the parent node. New nodes have their parent's fixed prefixes extended with zero and one. Children inherit neighbours and jump tables information that is relevant to them. This operation is local i.e. children's physical nodes are hosted by the parent's physical node. An FPN node is split if the number of hash keys it is responsible for exceeds the system wide constant. Since hash keys are equally distributed over hash key space FPN nodes should split more or less at the same time assuming that new hash-value pairs are continuously being added to the system. The difference in the length of their fixed prefixes should not be higher than one.

There is no operation that would revoke a split - two children nodes are never merged.

FPN Node Transfer FPN nodes can change physical nodes that host them. Such an operation is called an FPN node transfer. After a node is transferred information about its new location is propagated through pings. Additionally, the old physical node keeps information where the FPN node was transferred to forward there message sent to that FPN node by sender that did not yet update information about its new location.

Transfers are used for balancing purposes when physical nodes are added or removed and when FPN nodes are recovered after they were lost due to failures.

Handling Failures

FPN Node Recovery FPN has defined an algorithm that re-assigns part of the hash space handled by dead FPN node (i.e. hosted by dead or unreachable physical node) to other physical node. The algorithm only works properly under assumption that a physical network is not partitioned into to separated subnetworks of physical nodes. New instances of the FPN node rebuilds the FPN network. However, it does not rebuild information about lost hashes and their values, because this information has no redundancy - one failure causes permanent data loss.

Routing Under Failures In the case of failures, routing which consists of resolving bits or digits from left to right, may not be possible. In such a case information about neighbours and jump tables is used to pass the message to any FPN node that reduces value of XOR between the hash key and the fixed prefix of a destination node. If there is no such node, a random node is chosen as the next destination and it is kept by the message in a list of already visited FPN nodes to avoid loops.

5.2.2.3 FPN with Supernodes

The original Fixed Prefix Network design introduced a complex recovery scheme of a failed physical node which was difficult to implement. Additionally, it did not guarantee that a series of physical node recoveries would not create two or more equivalent versions of the system. Furthermore, since there is no redundancy in FPN, even if the overlay network can be repaired, some of user data is lost. RAID-like resiliency could have been built on top of FPN. However, such a solution would make it difficult to satisfy DSS requirements like high resiliency for large configurations, tolerance for entire physical nodes failures and fast rebuilding of data resiliency after failures. To address these shortcomings, FPN was extended with the concept of *supernodes*. We call this system FPN with supernodes (FPN/SN).

Supernodes A Supernode represents one FPN node, but it is spanned over several physical nodes in order to increase its resilience to physical node failures. Each supernode consists of a fixed number of *supernode components*. A component can exist on no more than one physical node in a given point in time. Components of the same supernode are called *peers*.

The number of components of each supernode is given by a constant called *supernode cardinality*. Usually, this constant ranges from 4-24. For the commercial system HYDRAstor it is set to 12. For a given system incarnation, supernode cardinality is the same for all supernodes and is constant throughout entire system lifetime. As a supernode is equivalent to FPN node, it is identified with a fixed prefix. One physical node can host multiple components of different or the same supernodes (in case of small systems), although peers are usually placed on separate physical nodes to maintain high resiliency to failures. A given component distribution is defined by supernode incarnation. Whenever a component changes its location, a new incarnation of this supernode is created.

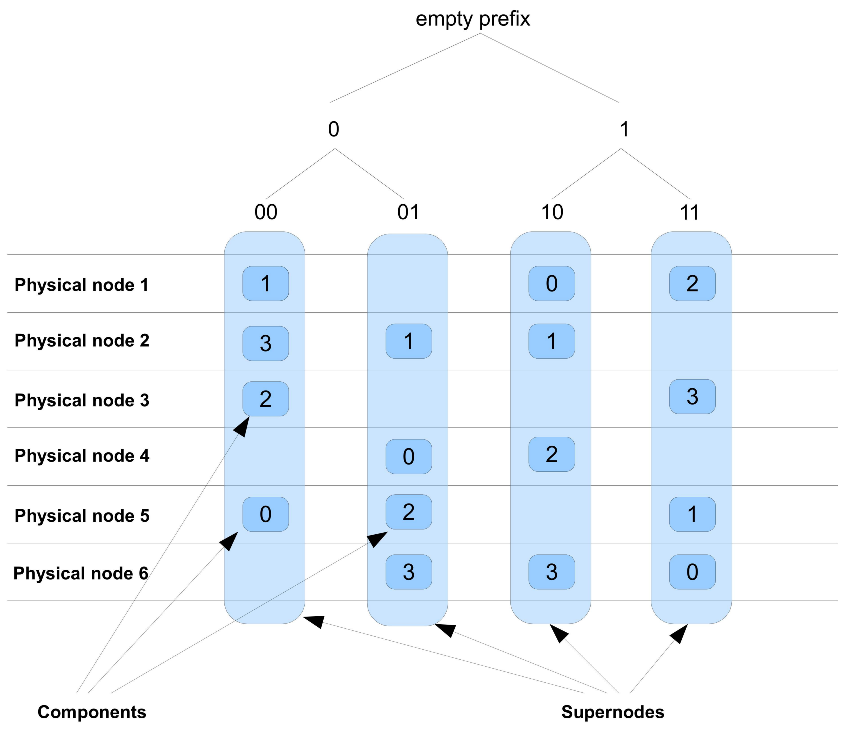


Figure 5.3: *Supernodes and Components. 4 supernodes of fixed prefixes 00, 01, 10 and 11 spanned over 6 physical nodes. Each supernode has 4 components, i.e. supernode cardinality is 4. Each component within a given supernode is identified with component index.*

Figure 5.3 shows an example of FPN/SN system. The upper part of the figure contains prefix tree showing four FPN nodes as leaves dividing the prefix space into four disjoint subspaces. The lower part of the figure shows four supernodes of cardinality 4 spanned over 6 physical nodes. These supernodes are FPN nodes representation.

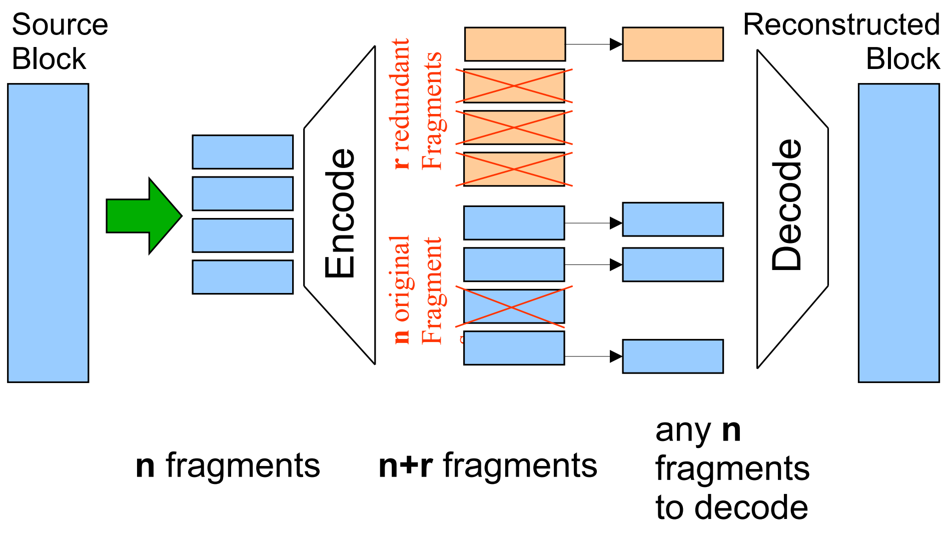


Figure 5.4: *Erasure codes schema.*

Supernodes solve two FPN problems:

- **Simple overlay network recovery** - FPN has a very complex recovery scheme and does not guarantee that a series of recoveries will not lead to the creation of more than one instance of the system. Due to the introduction of supernodes, this inter FPN nodes recovery scheme is no longer needed. Instead, a much simpler intra supernode recovery mechanism was introduced. A supernode quickly recovers its components in case of failure of any physical machine that the supernode is spanned over. Under standard configuration that mechanism is able to rebuild a supernode as long as there is *supernode cardinality/2 + 1* components alive i.e. *supernode cardinality/2 + 1* components is hosted by non-failed and reachable physical nodes. We assume that the supernode cardinality is large enough so that the probability of simultaneous permanent loss of more than *supernode cardinality/2-1* components is very small.

- **User data resiliency** - a supernode is basic logical entity that stores user data in a failure resistant manner. This is achieved by erasure coding [12].

  Figure 5.4 shows erasure codes schema and possibility to use it for resilient storage of data blocks. An erasure code is an error correction code originally used in telecommunication. Erasure codes divide a source block into n fragments and adds additional r redundant fragments. $k = \frac{n}{n+r} < 1$ is the *rate* of encoding. Storage cost is increased by a factor of $\frac{1}{k}$. The key property is that the source block can be reconstructed from any n encoded fragments.

Erasure codes are a superset of replication and RAID schema. A data block with c replicas can be described by an ($n = 1$, $r = c$) erasure code. RAID level 1 can be described by an ($n = 1$, $r = 1$) erasure code, RAID level 4 and 5 can be described by an ($n = 4$, $r = 1$) erasure code.

To provide failure resiliency each user data block is encoded into supernode cardinality fragments. These fragments are spread over the physical nodes that the supernode is spanned over. In case of very important data, instead of coded fragments, supernode cardinality copies of the block are spread over the physical nodes.

Supernode Components A component is described by a component description that contains:

- Fixed prefix of the supernode a given component belongs to.
- Component's index - a number between 0 and supernode cardinality - 1 which is an id of this component in any incarnation of its supernode.
- Component's version (or incarnation) that is increased when this component changes host.
- IP of physical node hosting component.

A component's prefix and index are fixed for a given component lifetime.

Supernode Composition A supernode is a logical entity. The physical representation of the supernode incarnation is a supernode composition. It consists of a fixed prefix identifying a given supernode, a supernode version, which is a sequence number and an list of component descriptions. Each such description has information about component location and its version. A supernode composition always contains supernode cardinality of components. Dead components (hosted by failed or unreachable physical machines) remain in the composition they occurred. A component recovered on another physical node is described by a newer composition. A composition version i.e. sequence number is increased on every supernode change.

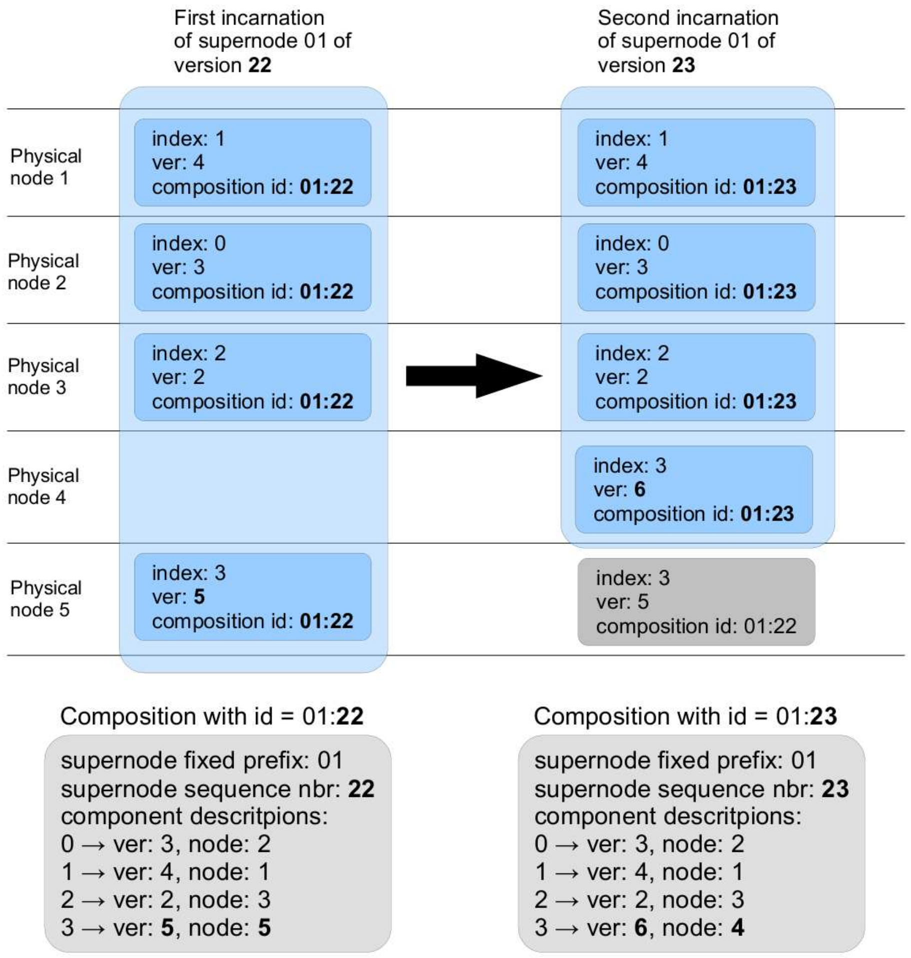

First incarnation of supernode 01 of version **22**
Second incarnation of supernode 01 of version **23**
Physical node 1
Physical node 2
Physical node 3
Physical node 4
Physical node 5
index: 1 ver: 4 composition id: **01:22**
index: 0 ver: 3 composition id: **01:22**
index: 2 ver: 2 composition id: **01:22**
index: 3 ver: **5** composition id: **01:22**
index: 1 ver: 4 composition id: **01:23**
index: 0 ver: 3 composition id: **01:23**
index: 2 ver: 2 composition id: **01:23**
index: 3 ver: **6** composition id: **01:23**
index: 3 ver: 5 composition id: 01:22
Composition with id = 01:**22**
supernode fixed prefix: 01
supernode sequence nbr: **22**
component descritpions:
0 → ver: 3, node: 2
1 → ver: 4, node: 1
2 → ver: 2, node: 3
3 → ver: **5**, node: **5**
Composition with id = 01:**23**
supernode fixed prefix: 01
supernode sequence nbr: **23**
component descritpions:
0 → ver: 3, node: 2
1 → ver: 4, node: 1
2 → ver: 2, node: 3
3 → ver: **6**, node: **4**

Figure 5.5: *Example of two incarnations of supernode with fixed prefix 01 and compositions describing them. Left part shows the first incarnation. Right part shows the second incarnation after component with index 3 has changed location from physical node 5 to 4. The component version is increased from 5 to 6. All active components has updated id of the latest composition from 01:22 to 01:23. The component marked gray is an old, not active, version of component of index 3.*

Figure 5.5 shows two incarnations of supernode identified by fixed prefix 01. Left part of the figure shows first incarnation which consists of components hosted by physical nodes 1, 2, 3 and 5. This incarnation is described by supernode composition identified with id 01:22 (01 stands for fixed prefix of the supernode, 22 is a supernode incarnation sequence number). Each component has a local copy of this supernode composition. Right part of the figure shows incarnation of the supernode after component identified with index 3 has changed location from physical node 5 to physical node 4. Information that has been update due to this change is bold. New supernode incarnation with version equal to 23 is described by new supernode composition identified with 01:23. Components that still are part of the supernode keep local copy of the new supernode composition. New incarnation of component with index 3 has changed version number from 5 to 6. The previous incarnation of the component, marked gray, is still located on physical node 5. However, the component is not active and is not part of the newest incarnation of the supernode.

A supernode composition is changed by a distributed consensus protocol [11] run by the components currently belonging to this supernode. This protocol is described below in section 5.3.1 on page 55. Consensus guarantees that two compositions with the same fixed prefix and sequence number have exactly the same content i.e. description of peers location.

Each component maintains one most recent and active supernode composition which reflects the most up-to-date knowledge this component has about the state of its supernode. Each composition is stored persistently on physical nodes that host components described by that composition.

The compositions are partially ordered by their fixed prefix and sequence number. We say that a composition A is younger than B if they have the same fixed prefixes and A has higher sequence number or A's fixed prefix is proper prefix of B's fixed prefix. We define composition C as *the youngest* if there is no younger composition than C.

Compositions are identified by their ids that consist of fixed prefix and sequence number.

The sequence of consecutive, ordered compositions is called a *chain of compositions*.

Except the list of components compositions keep some additional auxiliary information about the supernode. In particular compositions keep *global state* described in section 5.2.2.3 on page 50.

Overlay Network   Similar to FPN nodes, supernodes create overlay network. Each component keeps compositions of its *neighbouring partners*. A component's neighbouring partner is a component which belongs to a neighbouring supernode (neighboring in terms of FPN node) and has the same index as the component. A component also keeps information about its peers whose locations are described by the current composition. Thus, components create an overlay network by keeping location information about peers and neighbouring components.

Supernode Changes A supernode can be changed by three operations: supernode split, component transfer and component recovery.

A supernode split is a global supernode operation. It is the counterpart of the FPN node split. Two compositions with fixed prefixes extended with zero and one are created on physical nodes hosting components of the supernode. After the physical nodes receive these two compositions, the components of the old supernode are deactivated and two new ones are created in their place. Children components are hosted by the physical node hosting parent component. As with FPN, components inherit information about appropriate neighbours. The aim of the split is to keep the size of components at limited level to efficiently balance them between physical nodes.

A component transfer changes the location of one component. A new composition is created both on physical nodes belonging to the previous composition and to the physical node hosting component after the transfer. A new composition deactivates the old incarnation of a component on the old location and creates a new incarnation on the new physical node. After the transfer, the component immediately rebuilds its information about its neighbours. Component transfer operation is a base unit for balancing algorithm described in section 5.3.6 on page 66.

Component recovery is an operation that creates a new instance of a component after the physical node hosting it becomes unavailable. This process is executed by component peers. After half of supernode cardinality peers decide that the component is not reachable they find new location for the recovered component and create new composition describing component in the new location. This process is described in more details in section 5.3.4 on page 60.

User Data Failure Resiliency Supernodes enable the application of erasure codes to provide failure resiliency. Each block that is to be stored is divided into K original fragments. Based on these fragments, L redundant fragments are created, where $L =$ *supernode cardinality* - K. Fragment size is equal to the size of the original block divided by K. All the fragments are distributed across destination supernode components. Using any of the K fragments, the original data block can be reconstructed, thus data of that supernode is available even if L components of that supernode are lost.

When a physical node is lost, all components that it hosted are recovered on other physical nodes as described in section 5.3.4 on page 60.

Global Information and Global State

Global information HYDRAstor provides a mechanism that allows the aggregated values of various statistics reported by components to be gathered. For example, components may report if they have finished a given task for reporting: 0 for not finished yet and 1 for finished. The minimum of all such statistics is enough for the system find out if the task is finished or not. First, statistics are reported by components to their *local leader*. The local leader is a component of index 0. The local leader aggregates statistics according to a predefined aggregation method depending on the statistic type - it may be minimum, maximum, average, total, etc. Then the local leader reports the statistics up the supernode tree. This is done by sending statistics to a virtual node which has a prefix shorter by one bit. This virtual node is handled by the local leader of the component responsible for hash equal to the virtual node prefix extended with zeros. After such a component receives statistics from two children components, it applies the appropriate aggregation function and sends them to a virtual node which has a prefix shorter by one bit. Finally, the statistics are delivered to a virtual node of prefix ϵ (i.e. prefix of length zero) handled by a *global leader*. The global leader is a component of index 0 which has a prefix consisting only of zeros i.e. a prefix in form $0*$. The global leader passes the aggregated statistics down the supernodes tree to each local leader.

Together with aggregated statistics, global information includes also information about components' locations that is used by the balancing algorithm described below in section 5.3.6 on page 66.

Global state All components have access to globally available information called *global state*. Global state is a set of various small pieces of information like the list of all physical nodes the system is built upon, status of certain background operations, etc. Global state is kept by each composition. The global leader is responsible for changing and propagating the global state to all components. The global state is changed when the global leader receives a request to change a given element in the set. Depending on the type of information it may verify that a given change is acceptable - for example, a request that carries the change also carries information about the value of the element considered as up to date by the requester. If it differs from the value global leader has, the request is dropped. This is to handle potential races when the global state is changed by many sources at the same time, or messages with the request are reordered. If the change is acceptable, the global leader creates a new global state with the change applied and propagates the new global state to its peers. Every new global state change has the sequence number increased. The global leader compares the global state sequence number reported in global information (this is the minimum of global state sequence numbers reported by all supernodes). If the global leader has a higher sequence number than the one in global info, then the leader periodically propagates the global state down the supernodes tree. Finally, such a message is received by all local leaders. Local leaders propagate the new global state to all theirs peers.

Global information and global state are used by many HYDRAstor algorithms, in particular the deletion algorithm described in section 5.3.7 on page 74.

5.2.3 Local Data Organization

Disk Level Data Organization There are several goals that local data organization has to reach. First, it has to support efficient storage and retrieval of data streams. Data from one stream should be placed on the consecutive sectors of disks to limit disk drive heads movement during write and allow efficient stream prefetch to RAM cache during reads. Second, data organization has to support easy identification of availability and reliability of the stored data. In a case of a failure, data organization has to allow for both fast identification of missing fragments and location of fragments to be used for missing fragments reconstruction. Thirdly, data organization should support fast location and retrieval of data from old component locations after components transfer between physical nodes. Fourth, the data organization should support distributed deletion. Last, the data organization have to support data deduplication.

A stream of user data blocks is converted into supernode cardinality streams of erasure coded fragments distributed over physical nodes. Fragments belonging to a consecutive limited number of blocks form a logical unit of data management called a *synchrun*. Single stream of fragments belonging to a synchrun is called *synchrun component*. Figure 5.6 shows an example of such organization.

Adjacent synchrun components are stored on one disk for fast disk write and read operations. Similarly to a stripe in a RAID group, a synchrun

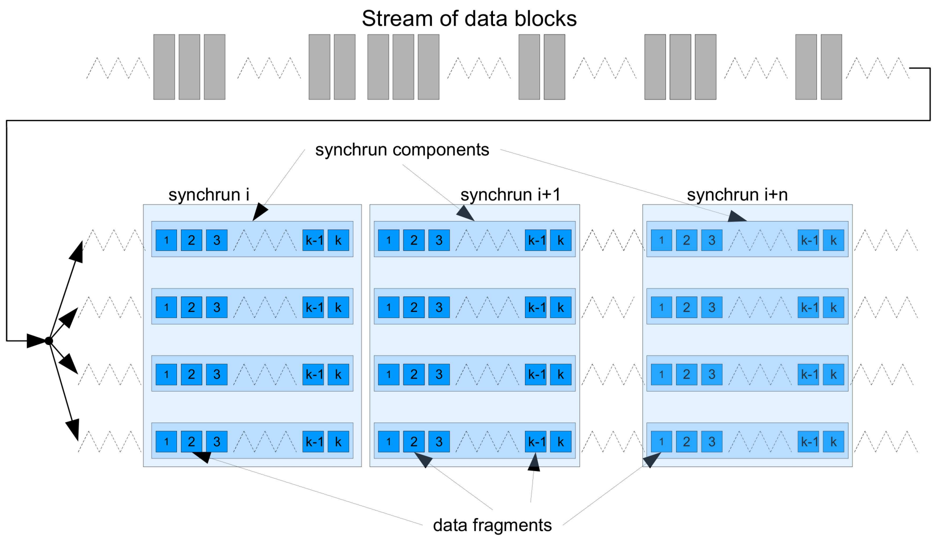


Figure 5.6: *Synchrun and synchrun components schema for supernode cardinality = 4. The upper part of the figure shows stream of data blocks routed to a given supernode. Each block from the stream in converted into 4 erasure coded fragments. Each fragment is stored in a synchrun component. Synchrun components form synchruns, which are logical data management units. Synchrun keeps erasure coded data blocks.*

harness multiple disks to write and read data faster than any single disk can do. Implementation of write operation is described in section 5.3.5 on page 63.

For a given supernode, user data blocks are written within the current composition. Synchruns are stamped with the composition sequence number under which the write is executed and a sequence number within the composition. Thus, synchruns can be ordered in chains. Since fragments are also stamped with a sequence number within a synchrun, they also can be ordered in a chain. In a fragments chain, every given fragment is identified by pair of synchrun stamp and sequence number within the synchrun. That identification is used by read operation to quickly find appropriate fragment on disk as described in section 5.3.5 on page 65. Figure 5.7 shows examples of chain of synchruns and chains of fragments.

Due to failures, chain of fragments may contain *holes* i.e. sequences of missing fragments. Fragments chains can be scanned for easy identification of such holes. Data is available as long as for each block there are enough

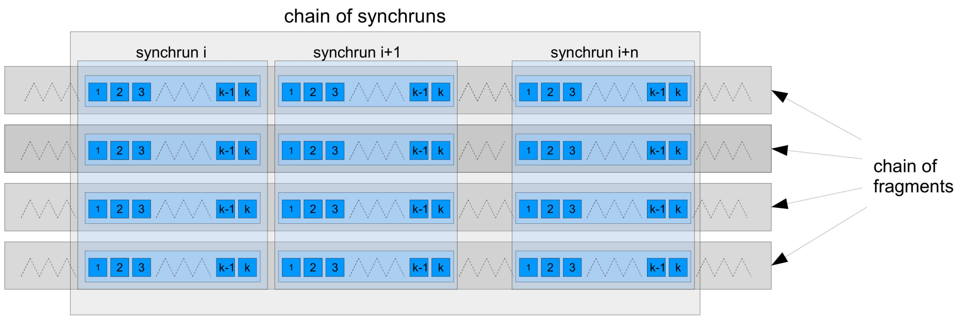


Figure 5.7: *Chain of synchruns and synchrun components.*

fragments chains without holes in places where the block's fragments are stored. The available fragments are used to created the missing fragments and remove the holes.

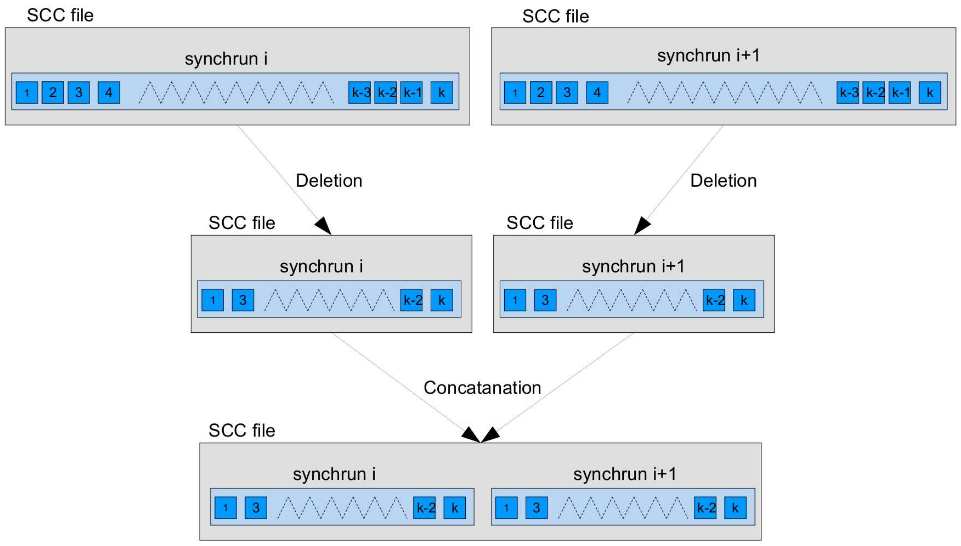


Figure 5.8: *Synchrun component containers (SCCs). The upper part shows two SCCs. Each of them originally stores one synchrun component. After some fragments are deleted SCCs become smaller. To keep their size similar to the original the SCCs are concatenated into one SCC containing both the synchruns.*

Each synchrun component is stored in a file called *synchrun component container* (SCC). SCCs are limited in size and in number of fragments they keep. By default an SCC is well below 100 MB in size and has several thousand fragments. SCC is the base unit for both data management on disks and the base unit for data movement between the physical nodes. Initially, one SCC stores one synchrun component. However, size of SCCs may be decreased, for example after some fragments are deleted. System tries to keep SCCs size similar to the original value to limit number of local SCCs and be able to keep their metadata cached in RAM. Thus small, adjacent SCCs may be concatenated. After such operation an SCC may keep multiple consecutive synchrun components. Figure 5.8 shows an example of two SCCs originally storing one synchrun component each and then concatenated into one SCC.

Each SCC has another file associated with it. The file is called an *SCC index*. SCC index keeps metadata information about fragments like a list of fragments, the offset of each fragment in SCC, deletion counters keeping number of blocks/fragments pointing to each fragment (there are details of deletion in section 5.3.7 on page 74), etc. SCC indexes of a given supernode keep exactly the same data, thus each supernode has supernode cardinality copies of each SCC index providing that its peers have their data in sync.

Separation of metadata from data allows efficient data management after component transfers or component recovery. After component is transferred or recovered, it downloads its SCC indexes. Their size is relatively very small comparing to the size of SCC, thus such operation is fast. After downloading the SCC indexes the component has complete view of all data it manages and allows it to identify SCCs that have to be downloaded or reconstructed from peers' fragments. There are more details about this process in section 5.3.4 on page 60.

To provide support for deduplication queries there is an additional data structure that maps part of the hash keys stored locally to the list of potential SCC indexes that may contain metadata about the associated block. To check that a given hash belongs to a previously stored fragment, potential SCC indexes have to be read from the disk. This map keeps part of the hash keys instead of the whole hash keys to limit memory they consume. More details about deduplication is described in section 5.3.5 on page 65.

Each server keeps a RAM cache of SCC and SCC indexes. SCCs and SCC indexes are prefetched to serve succeeding requests belonging to the same stream without disk access.

5.3 Functionality Implementation of HYDRAstor

5.3.1 Supernode Consistent State

Ensuring the consistent state of a supernode is crucial for providing consistent overlay network. Consistency is provided by a consensus algorithm. HYDRAstor uses a modified version of Ben-Or algorithm [11]. Participants are peers of the supernode. The input for the algorithm is the current composition called *base composition* and changes proposed by components, for example new location of given component or a supernode split. A typical result of the algorithm is a composition with the non-conflicting proposed changes merged and applied to the base composition. The new composition has the sequence number increased by one. In the case of a split operation consensus results in two compositions describing a supernode after the split. In such a case, the compositions have the fixed prefixes extended with zero and one and also their sequence numbers are higher by one than the sequence number of the base composition. A split vote cannot be merged with any other voting proposals, thus children components are created on the same physical node as the parent component i.e. supernode split is a local operation.

Under standard configuration a new composition can be created if at least the majority of peers which know up to date composition participate in the voting procedure and agree on the change. Peers that did not participate in voting, for example due to transition network problems, are notified about the composition by peers participating in the voting.

HYDRAstor can also be configured to successfully finish the voting procedure when only half of supernode cardinality peers are active. This is done with the help of an *arbiter* process that can be executed on any machine other then the physical nodes that the system is built upon. This is to handle a very small HYDRAstor setup when the system is built on only two physical nodes. In such a case, single node failure blocks the voting procedure. An arbiter allows peers from one physical node to temporarily decrease the quorum to half of supernode cardinality (instead of half plus 1) and it additionally guarantees that two physical nodes will not receive such permission for the same base composition. For a system built upon more than two physical nodes an arbiter is not used because any single node failure does not block the consensus algorithm.

The base composition carried by consensus messages is a guard against participation of out of date components in the voting. Only components active according to the base composition can take part in the voting process. Each peer verifies if it is eligible to vote by verifying that its composition is the same as the base composition carried by consensus messages. If its composition is older then it updates the composition. If component is still active after the update it participates in the voting, otherwise it does not. If a component's composition is newer than the base composition it means that other peers have an old composition. In such a case the component ignores the voting messages and propagates the newer composition to the out of date peers. That newer composition stops the voting process initiated by out of date peers and updates their information about the supernode.

If the new composition results in a change of location of a component, peers send this composition to that new location. The physical node that receives it stores it persistently and creates the new instance of the component. In the case of a transfer the old location creates a so called *inactive component*. This is used to coordinate the transference of the component's data from the old to new location.

The HYDRAstor system model assumes that messages can be lost, reordered and retransmitted. Thus composition may be duplicated, lost and reordered. Composition sequence number is used by components to find out the most up to date composition.

Every physical node persistently stores all received composition that are the youngest among all the stored compositions so far. It does not matter whether there are local active components associated with those compositions. It is done this way because such compositions may describe location of neighbouring supernodes. Information about supernodes locations is required for routing. To reclaim the storage space the physical node removes compositions that are not the youngest.

5.3.2 Overlay Network Monitoring and Supernode State Propagation

Peers monitor each other in order to detect failures by periodically exchanging messages called *internal pings*. Internal pings are also used to propagate various auxiliary information within the supernode. The most important information carried by the ping is the current component's composition. Pings propagate the newest composition among peers. Pings are also sent to old incarnations of components that do not exist in the current composition in response to pings received from them. This is to handle a case when an component is disconnected from the system, its new incarnation is created in another location and then the component is reconnected back to the system. Such a component does not know about the new composition so it acts as an active component and sends pings to its peers. Any peer that receives such ping from old component incarnation replies with a ping carrying the latest composition that deactivates the old component incarnation.

External ping messages are sent between neighbouring partners. External pings correspond to base FPN pings - they propagate routing information between neighboring supernodes and monitor neighbours availability. Additionally, external pings propagate information about known physical nodes. This information is used by the balancing algorithm described below.

Under default settings pings are sent every 5 seconds. If a component does not receive several pings (by default 6) from its remote peer or partner, the component treats the remote peer or partner as unreachable.

5.3.3 Routing

Messages in FPN/SN can be routed to a specific supernode or to a specific supernode component. Routing to a specific supernode is very similar to routing in the original FPN, except that instead of routing messages through FPN nodes, they are logically routed through supernodes. In practice this means that messages are passed through components that belong to these supernodes. Messages are passed over network links which are monitored by neighbouring partners or peers.

If there are no failures, messages are passed between supernodes along components which have the same index as the component initiating the routing. When routing to a specific component, the routing differs only in the last hop, in which the message is passed from the first component receiving this message in the destination supernode to the destination component in this supernode.

Figure 5.9 shows how messages are routed. Because each physical node hosts multiple components it may occur that apart from next hop component it also hosts a component that is closer to the target supernode or target component. Thus, as an optimization such components take over routing the message and make a routing shortcut.

Routing in changing network Components may change their location while a message is being sent to them. Since information about new locations is not propagated instantly a message may be sent to a physical node

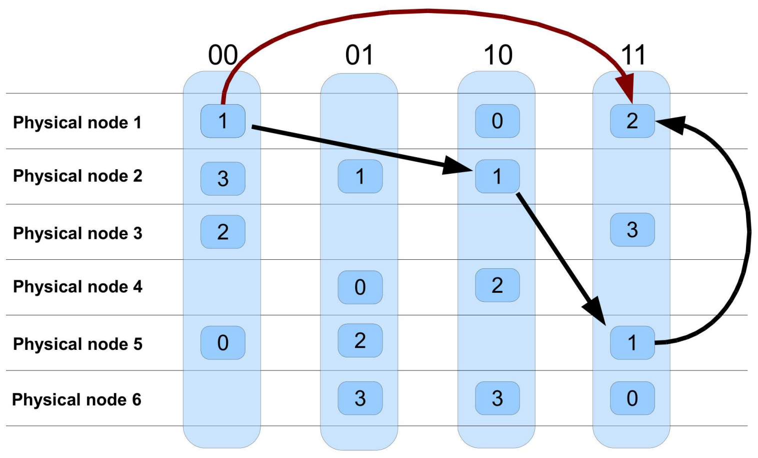


Figure 5.9: *Example of routing a message from component of index 00:1 (component of index 1 belonging to supernode of fixed prefix equal to 00) to component 11:2. Supernode cardinality is 4. Black arrows show routing under assumption that physical nodes do not optimize the routing by making shortcuts through locally stored components. In such a case message goes from component 00:1 to component 10:1, then to 11:1 and finally is passed to component 11:2. However, since physical nodes hosts multiple components there can be routing shortcuts. Since 00:1 and 11:2 are hosted by the same physical node message can be routed faster (red arrow).*

that does not already host the destination component. To handle this location change gracefully compositions stored on the old location are scanned to find a composition that describes new location of the component (such composition must exist if component was deactivated in the physical node). This message is then passed to new location.

Routing under failures

Links along which messages are routed are monitored by pings. As described above, if a component does not receive several pings from a peer or a partner it treats them as unreachable. Messages are not passed to unreachable peers or partners. Figure 5.10 shows how routing is handled around a broken partner. If a message cannot be passed to a partner because it is unreachable, the message is passed to any reachable peer. That peer will try to pass the message to its appropriate partner to reach the next hop supernode. A component that passes a message to a peer adds itself to message's list of visited components within the current supernode. This list is used to avoid sending message to already visited components in the case when

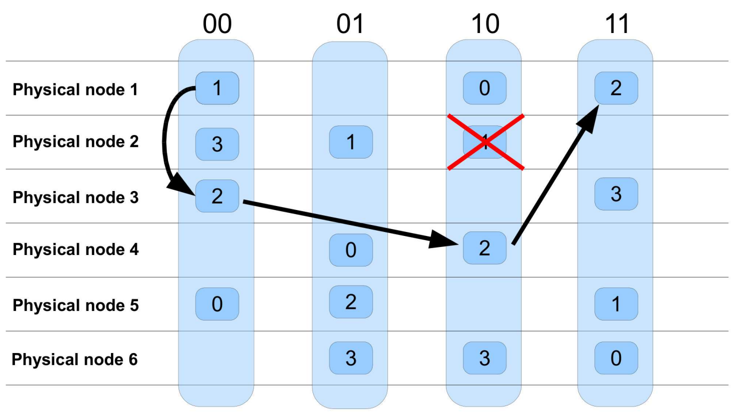


Figure 5.10: *Example of routing a message from component of index 00:1 (component of index 1 belonging to supernode of fixed prefix equal to 00) to component 11:2 when 10:1 is unreachable. Supernode cardinality is 4. In this example we assume physical nodes do not make routing shortcuts through locally stored components. 00:1 cannot send message to its neighboring partner 10:1, because it is unreachable. To bypass the unreachable component 00:1 passes the message to its peer 00:2. 00:2 passes it to 10:2 which finishes the routing by passing the message to 11:2.*

several peers have broken links to partners of the next hop supernode and the message is being routed within a supernode trying to find a way out to the next hop supernode. The list is cleared when the message is passed to the next supernode, because routing guarantees that the message will not get back to the same supernode again. This is because routing resolves bits from left to right and each hop between supernodes resolves at least one additional bit.

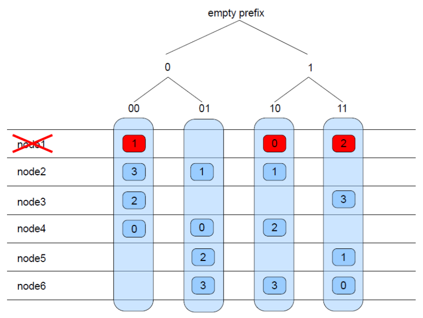
Note that messages might be sent to a failed physical node before a component passing the message decides that next hop destination is unreachable. That is because on default a component is treated as unreachable after no ping is received from it during a 30 second time window. During that 30 second window messages might be lost. Such cases are handled by message retransmissions done by the source component that started message passing. Each component that initiates message passing and wants to make sure that the message is delivered has to retransmit the message until it receives an acknowledgment that it is delivered to the target. This is consistent with our system model that assumes that a message can be lost, thus such retransmission and acknowledgment mechanism has to be done anyway. However, we are currently working on making use of TCP/IP sessions to shorten that time from 30 to 10 or less seconds.

Initially we planned to implement FPN jump tables that would limit the number of routing hops between supernodes. During the project implementation it turned out that jump tables are not important for the current version. That is because HYDRAstor is delivered to customers together with the *Hydra File System* [91] (HFS). HFS stays on top of HYDRAstor and provides regular file system interface. Since the biggest currently available systems are built upon no more than 100 physical nodes, HFS is able to keep a cache of all components' locations. In a stable system (i.e. when no components are transferred or recovered) it passes write or read requests directly to the destination physical nodes. Requests are routed only when components change locations. Furthermore, such location changes are immediately reflected in the cache when HFS receives a read or write reply from HYDRAstor. Subsequent requests are again passed directly to destination physical nodes.

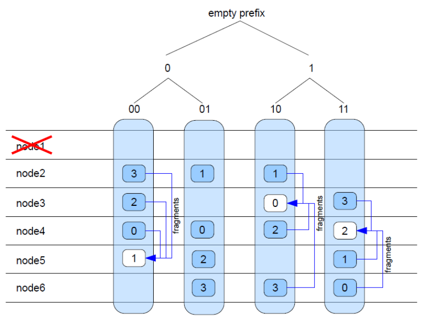
5.3.4 Failure Discovery and Recovery

A supernode is a basic logical entity that provides failure resiliency. We assume that the probability of concurrent loss of half components i.e. the concurrent loss of physical nodes hosting these components is very low. Such an assumption greatly simplifies system design, because there is no need for complex recovery of lost supernodes by other supernodes. Instead, each supernode monitors its state and has the ability to recover itself in case some of its components are lost (we say that a component is lost if its hosting physical node fails).

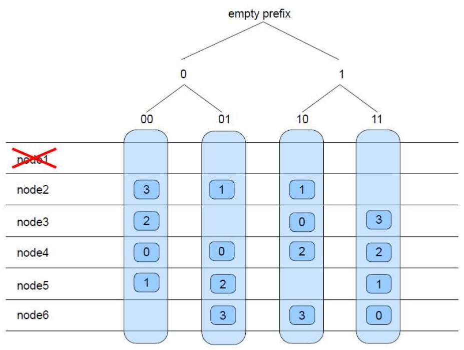
As peers ping each other, they detect unreachable peers. A peer P1 is declared unreachable by another peer P2 after a threshold number of periodically sent pings (by default the period is 5 seconds and the threshold number is 6) from P1 is not delivered to P2. P2 may reconsider and declare P1 reachable again after even only one ping from P1 is delivered. Each internal ping includes a vector of bits indicating peers declared unreachable by this ping source peer. Peer P2 decides to recover P1 only if the majority of peers declared P1 unreachable in the most recent pings received by P2. To recover P1 component P2 finds a physical node that will be P1's new host. It does this through the entropy function described below. Following this P2 initiates voting over changing the location of P1. At least *supernode*



(a) *Node 1 failed.*



(b) *Components are recovered, but missing data is not rebuilt. Components get fragments from their peers.*



(c) *Components recovered and their data rebuilt.*

Figure 5.11: *Overlay network recovery and data reconstruction.*

cardinality/2 + 1 peers have to participate in the voting. If P2 wins the voting the new composition describes new location of P1. Otherwise, if the new composition does not reflect P2's vote, P2 may start the recovery procedure again if P1 is still declared unreachable.

After the lost component is recovered on a new location it has no data stored locally. It receives from its peers list of its SCC indexes (copy of each SCC index is kept by each peer). Based on the SCC indexes the component finds out which fragments are missing. Then it downloads appropriate SCCs from the peers and rebuilds the missing fragments as described in section 5.3.8 on page 77 below. After all the missing fragments are rebuilt supernode resiliency reaches the desired, original level.

Figure 5.11 shows the recovery and data rebuild process in a four supernodes system built upon six physical nodes with a supernode cardinality set to four. Picture (a) shows the system with a failed physical node 1. Lost components are marked red. Each of the supernodes 00, 10 and 11 finds out that it has lost one of its components. Within each supernode, peers of each lost component start the recovery procedure that leads to creation of new incarnations of the lost components (picture (b)). The new component incarnations are marked white, to indicate they do not have their data stored locally. To rebuild missing fragments each ”empty” component downloads appropriate SCCs from peers and rebuilds missing data. On picture (c) all components have their data stored locally - the system has reached its normal state with original redundancy.

5.3.5 Write and Read Operation

Block Interface HYDRAstor exposes block interface. Data blocks are of variable size for better duplicates elimination [65, 52, 50]. Usually their average size is 64 KB. Blocks are immutable. They consist of data and optional pointers to ids of previously stored blocks. A block hash is computed as SHA-1 of its contents. A client provides information about required block *resiliency class* on a block write. Higher resiliency classes have more redundancy with higher storage overhead. HYDRAstor replies with the id of the block. This id is provided to facilitate the read operation for retrieving the block. Pointers stored in blocks are exposed to HYDRAstor to facilitate deletion operation. A client provides its unique id that is used to identify its streams of data in the write operations.

Blocks form a directed acyclic graph (DAG). Clients store data as a tree of blocks similar to file system structures. Because of deduplication trees may share some blocks. This is why blocks form DAGs instead of trees. The root of the tree is pointed by special block called a *retention root*. The retention root guarantees that a tree pointed by it will not be deleted as long as it is not pointed by another special block called a *deletion root*. After a tree of blocks is created the client uses a retention root to mark the tree not to be deleted. When client decides that the tree should be deleted it additionally marks it with a deletion root.

Figure 5.12 shows blocks organized into a DAG with 3 source vertices. One of them is a retention root (SP2), one is a deletion root (SP1) and the last one is a regular block (A), which indicates that this part of DAG is still under construction. Assuming that user will not add any new retention roots all blocks not reachable from live retention roots (i.e. ones not pointed

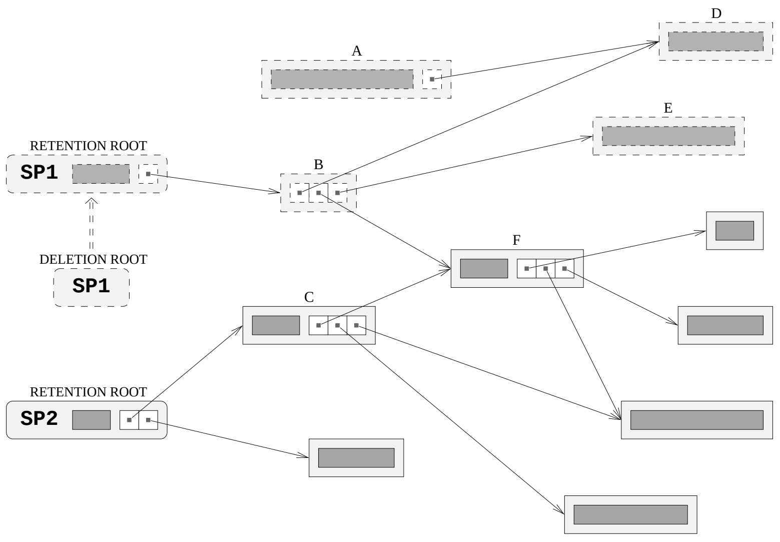


Figure 5.12: *Blocks organized in a directed acyclic graph. Data part of each block is shaded, pointers are not.*

source: [26]

by deletion roots) will be deleted (blocks with dotted lines). Such block are B, E, A and D. However, if user creates retention root pointing to block A before the deletion operation processes this block, blocks A and D will not be deleted.

Write Operation An access node (AN) computes the hash key of the block it wants to store. Like the the original FPN, the hash key determines the destination supernode. In addition, it also determines index of the component of the supernode that manages the write operation. Such a component is called a *write initiator*. A write request message containing a block to be written is routed to the write initiator. Based on resiliency class it creates an appropriate number of original and redundant erasure coding fragments (unless the block keeps pointers to other blocks - in such a case fragments are copies of the block) and propagates them to itself and all other peers. Peers store the fragments in the latest SCC in the SCC chain. The write initiator waits for *write threshold* acknowledgments and confirms the write with the client. The reply contains the block id that is required to read the data. Among other information the block id contains the synchrun id and the serial sequence number of the write within the synchrun. This information is required to localize data on the disk by the read operation (see 5.3.5 on page 65 for details).

Figure 5.13 shows the write operation of block with key starting with bits 010111. Component of index 1 is a write initiator.

The write threshold can vary from a number of original fragments to supernode cardinality. Its value determines the trade-off between latency and resiliency: the greater the write threshold, the higher the resiliency and latency. Under the current version the write threshold is system wide constant.

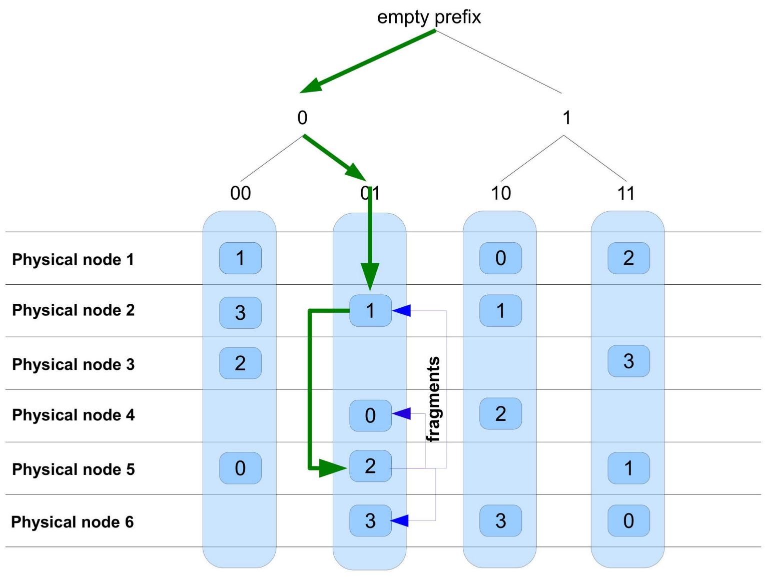


Figure 5.13: *Writing of block with key starting with bits 010111.*

A write initiator manages creation and closing of all its peers SCCs. It also stamps synchruns with the current composition sequence number and with consecutive sequence numbers within the composition and stamps fragments with sequence numbers within the synchruns. Thus, the write initiator takes care of forming SCCs and fragments into chains.

Any component can be a write initiator. However, practically speaking it turned out that only limited number of components within a supernode should create and propagate data fragments. There are two reasons. Firstly, write initiators are more CPU intensive than non-write initiators. This allows us to balance a heterogeneous system with physical nodes without having an equal ratio of CPU to capacity. More details are included in section 5.3.6 on page 66. Secondly, each write initiator causes fragments to be written to supernode cardinality SCC files. This is due to the fact that each peer stores fragments received from a given write initiator in a separate SCC file. Later, when many users read the data concurrently (i.e. many data streams are read at the same time) each disks serves read requests from multiple files which results in many disk seeks and degraded performance. Lowering number of write initiators reduces the performance degradation. Performance problems related to reading data concurrently from many files are described in detail in section 5.4.6 on page 103. By default, there are three write initiators per supernode.

Data Deduplication HYDRAstor uses content-addressable storage paradigm to implement data deduplication. Data block key is computed based on the block's content. Thus it is very easy to achieve data deduplication. Deduplication query is routed to a write initiator handling the hash. First it is checked if a matching SCC index is already kept by RAM cache. If not, based on the hash and the data structure that maps part of the hash keys stored locally to the list of potential SCC indexes, matching SCC indexes are found and read from disk to RAM cache. Also SCC indexes followed in the chain are prefetched to limited disk access while serving succeeding deduplication queries. If any of the matching SCC indexes have information about the hash it means that the duplicate is found. Otherwise, there is no duplicate in the system. If the duplicate is found, the system has to verify that the previously stored block is readable. The write initiator receives from its peers summary of remote SCCs it has created and manages. Such summaries are sent whenever the state of the remote SCCs is changed. Based on the summaries write initiator builds information about ranges of SCCs chains that have complete and missing data fragments. This information together with id of the previously stored block taken from local SCC index suffices to verify readability of the block.

Read Operation To read a block from FPN/SN system, a client provides a block id that was returned by the write operation executed earlier. This address contains the hash key of the data block. Using this hash key the read request message is routed to the *read initiator* component which manages the read operation within the supernode. Beside the hash, the block id also contains synchrun id under which the write was executed and the sequence number of the write within the synchrun. These information allows to find, in the fragments chain, SCC index with information about

SCC that contains the fragment, offset in the SCC where the fragment is stored and number of minimal fragments required to reconstruct the block. As in the case of data deduplication query, SCC index and SCCs followed in the fragments chain are prefetched to RAM cache to speed up succeeding read operations. Then the read initiator sends fragment read requests to peers storing original fragments. This is because reconstructing a block from original fragments is more efficient than reconstructing one from redundant fragments. In addition to the block id, peer receives from the read initiator id of SCC that should store the fragment and offset within it. Since all fragments chains in a synchrun are loosely synchronized on all peers, such additional information allows a peer to bypass its local SCC index and read the fragment directly from SCC to limit number of disk accesses. If it turns out that the data was not synchronized between the peers, then the peer has to read its local SCC index. However, it is uncommon scenario. After the read initiator receives enough fragments, it reconstructs the block and sends it to the client.

In the case of failures peers may not have the fragments. In such a case the read initiator sends fragment read requests to all the remaining peers and reconstructs the block from the redundant fragments.

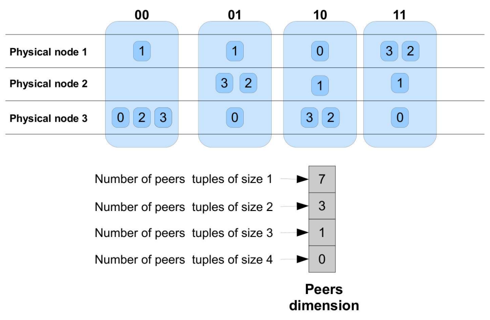
Even in a healthy system peers may also not have the required fragments stored locally. This occurs after the peer was transferred from other physical node, but its data is still hosted by the old one. In such a case the request is passed to the old physical node. Information about its location comes from two alternative sources. One is a compositions chain that can be used to find out the previous peer location. The other is reports that the old physical node sends to the peer about its orphaned fragments.

5.3.6 Balancing

Proper distribution of data among physical nodes is critical for data resiliency and availability, efficient storage usage, and system performance. There are many criteria that a balancing algorithm has to satisfy.

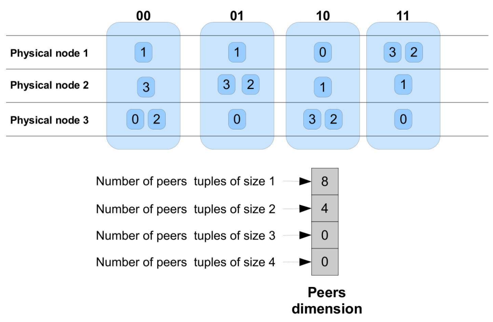
Desired data distribution is achieved by proper components distribution over the physical nodes. We have defined a multi-dimensional function called *system entropy* that prioritizes balancing criteria. The function is fully ordered. The lower the value of the function, the better the system is balanced.

The entropy function has the following dimensions ordered by their priority (highest priority first):



00
01
10
11
Physical node 1
Physical node 2
Physical node 3
Number of peers tuples of size 1
Number of peers tuples of size 2
Number of peers tuples of size 3
Number of peers tuples of size 4
7
3
1
0
Peers dimension

(a)



00
01
10
11
Physical node 1
Physical node 2
Physical node 3
Number of peers tuples of size 1
Number of peers tuples of size 2
Number of peers tuples of size 3
Number of peers tuples of size 4
8
4
0
0
Peers dimension

(b)

Figure 5.14: *Example of peer dimension of two components distributions. System is built from four supernodes with SNC=4. (a) shows an example of components distribution such that one physical node (3) hosts three peers (00:0, 00:2 and 00:3). (b) show a better components distribution - there is no physical node hosting three peers. Peer dimension of the left configuration is higher that the peer dimension of the right configuration (peer dimensions are compared as vectors with reversed order of their elements).*

- *Disk full* - measures system capacity overload. For each physical node there is defined a relative overload. It is equal to

$$\max(0, \frac{sum\ of\ components\ size\ hosted\ by\ physical\ node}{physical\ node\ usable\ capacity} - 1)$$

  where usable capacity is around 99% of total capacity of a physical node. The value of disk full is equal to the sum of all physical nodes relative overload. Component size is the size of all its non-deleted fragments.

- *Components on retired physical nodes.* A system administrator may temporarily set physical node to be retired. In such case a physical node transfers to other physical nodes all its components with their data fragments and can be turned off for maintenance tasks. This dimension is a number of components hosted by such retired physical nodes.

- *Peers* - measures system resiliency to physical nodes failures. This dimension is an integer vector of supernode cardinality size. Peers[i] = j means that in the system there are j peer tuples of size i such that each tuple is hosted by one physical node. Failure of such node results in the loss of i peers. Figure 5.14 shows example of peers dimension for two distribution of components.

Peer dimension comparison operator treats higher indexes with higher priority i.e. for two peer dimensions $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$ $A < B$ if:

$$a_n < b_n$$

or

$$a_n = b_n \ and \ (a_1, a_2, ..., a_{n-1}) < (b_1, b_2, ..., b_{n-1})$$

Such comparison prefers system with many tuples of peers of size n even over a system with only one tuple of peers of size $n+1$ i.e. in case of physical node failure it is better to lose n peers in many supernodes than $n+1$ peers in one supernode.

- *Components over capacity density* - measures the degree of balanced use of physical node disk space by components. It is a standard deviation of components over node capacity density computed for all physical nodes. The component over node capacity density is defined as
$$\frac{number\ of\ components\ hosted\ by\ physical\ node}{usable\ capacity\ of\ physical\ node}$$

- *Write initiators over physical node CPU density* - measures the degree of balanced use of physical node CPU power by write initiator components. It is standard deviation of write initiators over node CPU power density computed for all physical nodes. Write initiator over CPU power density is defined as
$$\frac{number\ of\ write\ initiators\ hosted\ by\ physical\ node}{physical\ node\ power}$$
where physical node power is an integer value defined by an administrator. It should be proportional to physical node CPU power.

- *Data to download* - measures amount of data that has to be transferred or recovered to reach a state in which all components have their data stored locally. The aim of this dimension is the preference for putting components on physical nodes that already host their data. For example, when a physical node is temporarily down its components are recovered on other physical nodes. After the node is started it is desirable that these components are transferred back to it, even if the other set of components would be equally efficient according to higher dimensions.

HYDRAstor uses a hybrid balancing algorithm that consists of so called *local balancing* and *global balancing*.

Local Balancing

Internal and external pings carry sender physical node summaries called *pnodeStats*. PnodeStats contain limited information that is almost sufficient to compute entropy. PnodesStats contain:

- total disk space,
- used disk space,
- physical node power,
- prefix space covered by components,
- prefix space covered by write initiators,
- information about retirement

Additionally each component keeps information about:

- the size of its data which is stored locally,
- its target size i.e. size after all its data is recovered on or transferred to the local physical node (the size of component data stored locally is equal to its target size if all its data is locally complete) and
- the size of data located on remote physical nodes (physical nodes periodically send reports about data which does not belong to any local component to components that are responsible for this orphaned data).

This information together with pnodeStats and information about peer locations taken from locally stored compositions is sufficient to compute the *physical node's view of entropy*. Such entropy is computed for a system consisted of physical nodes that send pings to the local physical node i.e. this is entropy for a local physical node neighbourhood defined by peers and supernode neighbours of all components hosted by the physical node. Besides limited number of physical nodes such view of entropy is also limited in the sense of peers dimension. This dimension only has information about peers of supernodes that have at least one component hosted by a local physical node.

Each physical node periodically (every 10 seconds by default) considers all possible transfers of locally hosted components to neighboring physical nodes. For each such potential transfer local node computes physical node view of entropy as if the component was transferred. Then it chooses the view of entropy with lowest value. If it is lower than current view of entropy appropriate transfer is scheduled.

Comparing physical nodes views of entropy instead of the regular entropies is good enough. That is because such transfers impact only nodes neighbourhood based on which the views of entropies are computed, but does not impact wider neighbourhood based on which regular entropies would have to be computed. Two regular entropies and two views of the entropies computed for systems before and after the transfer would have exactly the same differences on all dimensions, except *Components over capacity density* and *Write initiators over physical node CPU density*. Since *Components over capacity density* and *Write initiators over physical node CPU density* are standard deviations (and depend, inter alia, on number of physical nodes the deviations are computed based on), the difference are not the same for regular entropies and for views of entropies. However, they have turned out good enough approximations in practice.

Before a component is transferred, an initialization message is passed to the target physical node. The message carries destination pnodeStats to verify that target node pnodeStats are similar to the ones about which the transfer decision was made. If they are similar the destination node creates a *component skeleton*. It exists longer than the balancing period i.e. for 15 seconds. During that time it gathers any other transfer proposals and chooses the one with the best view of entropy computed by physical nodes that are trying to transfer components to the target physical node. Then the transfer leading to the system with the best view of entropy is executed. The aim of the component skeleton is to select the transfer that best improves the entropy among many concurrent transfer proposals. Such concurrent transfers may for example occur after the physical node not hosting any active components is started up. It would cause that many other physical nodes at the same time would decide that it is a good transfer target. If they were not serialized all such transfers would be executed leading to the physical node being overloaded with components that in turn would transfer away. In turn this would lead to component thrashing.

Local balancing is also used to find out location for an unreachable component to be recovered. Component that executes the recovery, called recoverer, simulates that the unreachable component is hosted by the local physical node. Then the recoverer executes local balancing with this unreachable component only to find out the best location for it. Then it executes the same steps as in case of regular transfer. The only difference is that under component recovery component skeleton immediately replies for creation request without gathering any other transfer proposals. This is to speed up component recovery. If recovered component is not hosted by the optimal physical node, then the component can be later transferred to a better matching physical node.

Initially HYDRAstor used local balancing only. However this approach had several disadvantages:

- Entropy is optimized by one transfer only and within limited number of physical nodes. Thus, the entropy function is optimized locally. Globally optimal components distribution may not be achieved.
- Components distribution depends on races between physical nodes trying to execute transfer operations. Even if a component skeleton accepts the best transfer there may still be situations where a non optimal transfer is executed (non optimal even in the sense of the improvement of local view of entropy). For example, this happens when the physical node that should initiate the best transfer operation is temporarily unavailable. During this period of unavailability other components may be transferred to the destination physical node, occupy it and block the best operation.
- Due to the above mentioned problems and the very low priority of *data to download* entropy dimension there were problems with components not being transferred to physical nodes hosting the most of their data. Quite often after a physical node was restarted it received components that were not hosted by it before its restart. This led to many unnecessary transfers of fragments between physical nodes. If the original components were transferred then most of their data would already be in the correct location - a restarted physical node would only have to download fragments written while it was down.
- There was no guarantee that required resiliency would be preserved in heterogeneous system with physical nodes with small and large disk space. When small physical nodes becoming full they transfer their components to the bigger ones thus overloading them with many peers belonging to the same supernodes. This decreases data resiliency - the failure of a big node would result in data loss. To block too many peers

being hosted by a big physical node there could be added a limit on maximum number of peers a physical node can host. But this in turn would block recovery of components and their data on machines with free capacity under scenario when many physical nodes fail in such a way that only the big physical nodes can host the reconstructed data. Even though such physical nodes cannot resiliently host reconstructed data it is still better to reconstruct the data on them and, after failed physical nodes are repaired and restarted, quickly transfer the data to proper location than not doing the reconstruction at all.

- It is hard to determine what is the raw capacity of the heterogeneous system, because it depends on the final components distribution that is not deterministic.
- The addition of new physical nodes results in many components being transferred to them at the same time. This results in many peers of each supernode not having their fragments stored locally. In such cases execution of deletion algorithm (described below) is blocked. This is because deletion algorithm in order to execute needs within each supernode certain minimal number of peers with all their data fragments stored locally i.e. peers with their SCC chains locally complete.

The above problems were solved by introduction of global balancing.

Global Balancing

Global balancing extends local balancing to globally computed affiliation of components to physical nodes.

HYDRAstor keeps list of physical nodes the system is built upon in the global state. For each physical node there is information about its total capacity, its CPU power and optionally rack number it belongs to. Additionally, the global state keeps information about *minimum resiliency class*. Minimum resiliency class is the weakest data resiliency class that the user is planning to use. Both types of information are provided by the administrator on system initialization and whenever physical nodes are added, removed or replaced or when user wants to change the minimum resiliency class. Based on this information the global leader computes *components global distribution.*

Components global distribution is globally optimal in terms of entropy function within the restriction of the minimum resiliency class that defines maximum number of peers hosted by one physical node. Entropy function is slightly modified to cluster supernodes along racks. For example if there are two racks, each having 12 physical nodes then for SNC=12 half of the supernodes will be hosted by physical nodes of the first rack and other half will be hosted by the physical nodes of the second rack. Since network traffic within supernodes is higher than between supernodes such clustering limits inter-rack network traffic which is lower then intra-rack network.

Additionally, the new global distribution is close to the current components distribution taken from global information. It is close in terms of number of components that have to be transferred to reach the new components global distribution. Global distribution is stored in the global state.

There is an additional flag associated with a list of physical nodes the system is built upon. The flag is reset on every change of the physical nodes list or change of the minimum resiliency class. The global leader sets the flag after the leader computes the global components distribution for the given list of physical nodes and minimum resiliency class. This is to deal with the possibility of the global leader being "killed" while computing the distribution - its new incarnation will start the computation from the beginning if the flag is not set.

Each supernode adopts to the new distribution. This process is managed by local leaders and local balancing. Compositions keep additional information called *target pinning*. Target pinning defines the target location of each peer. Entropy used by local balancing is modified. After *Components on retired physical nodes* dimension, and before *Peers*, new dimension *Pinning* is added. This represents the number of components that are not hosted by their target physical nodes as defined by target pinning. Additionally, local balancing is modified in such a way that:

- components hosted by the target physical node are never transferred away from it even if it is full, as long as all its disks are healthy,

- components are always transferred to the target physical node even if it is full as long as all its disks are healthy.

These two additional conditions are motivated by the fact that components global distribution is optimal for healthy physical nodes; thus components should be located according to the distribution as long as their target hosts are healthy. Healthy full physical node hosting only components defined by target pinning means that the system is totally full and no more data can be stored anyway. It makes no sense to transfer components to some other physical nodes. If the target hosts have some broken disks then their capacity is lower than assumed by global distribution. In such a case some of its pinned components are allowed to be transferred to other physical nodes. Otherwise, broken physical node would block user writes.

Local balancing adopts the location of a component to target pinning. In turn, target pinning of each supernode is adopted to components global distribution by the local leader. The local leader does not update target pinning to conform to global distribution in one step. This is because it would result in many concurrent component transfers. In turn this would result in many peers not having their fragments stored locally and would block the deletion algorithm. Therefore the local leader monitors the number of components which do not have locally stored fragments and gradually adopts target pinning to the global distribution. It changes a limited number of target pinnings in order to sustain at least ten (default, configurable value) components with their fragments stored locally. After each component transfer it waits until the component fragments are transferred to the new location before changing the next target pinning.

Global balancing solves problems local balancing was experiencing:

- Components distribution is optimal according to the entropy function. Local balancing is still used to handle failures, but it only changes components distribution locally so that distribution is still close the optimal and more importantly, after the failure is fixed components are transferred to their original locations.
- After healthy physical node is restarted, it hosts exactly the same components.
- In heterogeneous system components do not transfer to bigger physical nodes losing resiliency while system becomes full. By defining minimum resiliency class the user defines acceptable resiliency loss for such a case.
- Components global distribution deterministically defines components location thus raw system capacity can be easily computed.
- Addition or removal of physical nodes does not cause too many concurrent transfers that would block execution of the deletion algorithm.

5.3.7 Deletion Algorithm

The deletion algorithm used in HYDRAstor is similar to the distributed garbage collection algorithm [64]. With each block there is an associated counter that keeps a number of other blocks with a pointer pointing to this block. Blocks with a counter equal to zero can be reclaimed. Counters are not updated on block writes because it would negatively impact write performance. Instead, there is an update counters process that periodically scans block trees and updates the counters. It starts with roots pointed by retention roots written after the last deletion process was executed. Deletion increases the counters of blocks that directly or indirectly are pointed by such roots. After the counters incrementation phase is done a very similar process is repeated for decrementing counters of blocks that are directly or indirectly pointed by roots that in turn are pointed by deletion roots written since the last deletion execution. There is other independent background process called space reclamation that removes from disks fragments with counter equal to zero later on.

Actual counters incrementation and decrementation for blocks stored by a given supernode are executed by peers. Since a block with pointers is kept in a resiliency class *copy* (fragments are copies of the block) and counters are kept per fragment each peer can compute the counters on its own without the cooperation of other peers. To increase resiliency of counters' computation process and their values to intermittent failures, peers compute them independently. Since the counters should have the same values after they are computed, peers can verify them to make sure that the process was correct and that there were no software or hardware errors.

In order for a peer to be able to update counters it needs to have its SCCs chains complete on a local physical node. Such a peer is called a *good peer*. To maximize the resiliency of counter values, all components should compute the counters. However, in a large, dynamically changing system the chance that all components are good peers is low. Thus the counters upgrade process is started when at least nine peers of each supernode are good peers (nine is a default and configurable value). During the process computed counters are kept in temporary files. If during the process, the number of good peers of any supernode drops below seven (another configurable value), the process is aborted for safety reasons (i.e. there is a higher chance of counters being lost). Counters computed so far are disposed and computation is repeated after system has more peers with complete local data. Otherwise, when all counters are updated and each supernode has at least seven good peers, temporary counters are stored in SCC indexes and become regular ones. Write initiators distributed the new counters to peers that were not good peers and did not compute the counters on their own.

Aborted deletion means that during the counters update the system experienced failures that broke chains of SCCs on too many physical nodes. After such broken chains are repaired by background operations the update counters process can be started again.

A system providing both deduplication and deletion must guarantee that when a block is deduplicated against an older block, it is not scheduled for deletion. HYDRAstor solves this problem by imposing additional restrictions on the block interface and by temporary marking blocks non-deletable if they are base for deduplication. HYDRAstor introduces user visible virtual time. Time is divided into *deletion epochs*. A deletion epoch is a counter with increase operation. At any given point in time the system is in only one epoch. Within given epoch only one update counters process can be executed i.e. before the process is started the epoch is increased. The counters update process processes the pointers that were written in epochs earlier than the current one. The id of each block returned to the user by write operation contains current epoch E. The user can only write blocks with pointers to blocks which have ids with the epoch E or $E-1$. Ids with older epochs are rejected during the write operation. When a client is notified that the epoch is to be increased, he has to write a block with pointers to the blocks that were written in previous epoch in order to guard them against deletion. Additionally, while the counters upgrade process is active, a write initiator which deduplicates block against an older block, requests that all its good peers mark the older block with special marker. The marker guarantees that the block will not be deleted until next epoch, even if its counter drops to zero. In next epoch the user will guard the block by writing a pointer to it or system will be allowed to delete it. Markers are kept in good peers' RAM, not on disk. Thus their overhead on write operation is very small. However, keeping this information in RAM is safe with respect to loosing this information after server restart, because it has to last until the counters upgrade is finished. If in the meantime peer is restarted it is not good peer any more and no longer participates in the counters upgrade process, so it will not mistakenly allow for the deletion of the block that was used as a duplicate.

The deletion algorithm makes use of global information and global state to synchronize and drive the various phases of the counter upgrades. Initially, each supernode reports the number of good peers it has by global info. This information is propagated up the supernodes tree to the global leader as the minimum of all such numbers. If it is nine or higher, it initiates the upgrade counters process by adding information to the global state that the deletion process can start. Good peers execute actions according to the global state phase. After they finish, they report it by global information. The global leader receives these reports and moves the process to the next phase by changing the global state. Such iterations last until good peers report that all the work is done. It means that the latest computed counters stamped with current deletion epoch are persistently stored in SCC indexes. The global leader finishes the process and stores the current deletion epoch in the global state. This information is used by space reclamation background tasks (described below) to verify that they reclaim disk space (i.e. remove fragments) based on up to date counter values.

The above description of the deletion algorithm is very simplified as a detailed description is beyond the scope of this document.

5.3.8 Background Tasks

Chains of SCCs allow the easy implementation of various background operations. The separation of data from metadata i.e. SCCs from SCC indexes allows components to easily find out what data should be stored locally. First of all, component can verify that it has all SCC indexes by checking if the SCC index chain is complete from the beginning of the system's lifetime up to SCC index of current composition. If there are holes in the chain the component gets missing SCC indexes from its peers. Then it verifies that the SCC indexes are up to date. Since the only operation that might make SCC index outdated is the deletion counters update process, the component verifies that the deletion epoch associated with SCC indexes is the same as the epoch kept in the global state. If there are SCC indexes with an older epoch the component gets up to date indexes from its peers. With all SCC indexes up to date the component can perform various background operations over the data.

Rebuilding of missing fragments The component scans its SCC indexes and associated SCCs. If an SCC is missing or it does not have all required fragments the component first checks if there are physical nodes that reported to it that they have the missing SCCs. Such case is possible after the component is transferred or it was recovered, but the original physical node is reachable (for example, the component was recovered while the original physical node was restarted, but for some reason the component is not transferred back after the node is up and running again). The missing SCCs are downloaded if they are available on remote physical nodes.

Otherwise, SCCs have to be rebuilt. The components download appropriate SCCs from its peers and rebuilds the locally missing SCCs.

Space reclamation The component scans SCC indexes and removes SCCs that have fragments with counters set to zero and which are not marked as non-deletable (they are marked as non-deletable if during current deletion epoch fragment was used as a duplicate for new fragment write). It starts with the SCC that has the most reclaimable fragments to maximize the amount of capacity freed within a singe SCC recreation.

SCC concatenation Physical nodes try to store limited number of SCCs. Due to space reclamation and splits the size of an SCC may drop below the desired threshold. In such a case the adjacent SCCs are concatenated. This operation is synchronized within a supernode in order to have SCCs covering the same chains of fragments on all peers.

SCC sorting Just after a new SCC is closed for writing its fragments are sorted by unique client id. This operation speeds up reads because data is read from consecutive sectors of disks.

SCC split After component is split its SCC are split too. The actual SCC split is executed as a background process.

Data scrubbing Background process sequentially reads SCCs along their chain. If an SCC cannot be read due to disk errors it is removed and rebuilt from SCCs stored on peers.

All the above data operations result in the creation of new SCCs. SCCs are not changed in place in order to avoid fragmentation of data on disk.

SCCs chains also allow for the creation of reports about data health. Each write initiator receives information from peers about the state of the SCC chains it has created. Based on the information the write initiator finds out what is the minimal resiliency of the data in various resiliency classes. It reports it by global info to the global leader. Based on information from all supernodes the global leader can create reports about the whole system. This information can be used by administrators to find out what is the health of the users' data.

5.4 Requirements Satisfaction Discussion

The proposed solution addresses all data organization requirements. Most of the requirements are fully satisfied, although a few of them have imperfect solutions.

5.4.1 Efficient Storage Usage

Metadata Overhead Efficiency of storage usage is decreased by metadata. HYDRAstor has constant, system size independent, 2% overhead of metadata associated with SCCs. This overhead comes from 1 MB SCC index and 20 bytes metadata associated with each fragment of an SCC.

Additionally, around 3% of storage is reserved for temporary data generated by deletion operation. Without this reserve deletion could not be executed when the system is full.

Storage Utilization Storage utilization is determined by the lack of differences in the sizes of components and the uniformity of components distribution over physical nodes.

If the components had different sizes then physical nodes hosting smaller components would be underutilized. However, the size of data managed by all component is almost the same. Standard deviation from the mean size of component is $O(\sqrt{\frac{p}{n}} * \mu)$ where p is the number of supernodes, n is the number of data blocks stored in the system and μ is an average component size (the formula is derived in the appendix on page 142). Assuming that SHA-1 uniformly distributes hash keys, average block size with erasure coding overhead is 100 KB, maximum component size is 0.5 TB, supernode cardinality is 12 and each physical node has 12 TB of capacity this standard deviation from the mean is not more than 0,017 percent of the average component size. It practically means that all components have the same size. Negative impact of difference in components size on storage utilization is negligible.

Uniformity of components distribution over physical nodes is determined by number of components and number of physical nodes. The number of components is equal to supernode cardinality multiplied by the number of supernodes. The number of supernodes is power of 2 and depends on the number of physical nodes, their size and configurable variable defining maximum component size. The system splits components in such a way that in a full system the size of data managed by each component is no

Number of Physical nodes	1	2	3-4	5-8	9-16	17-32	33-64	65-100
Number of Supernodes	2	4	8	16	32	64	128	256
Number of Components	24	48	96	192	384	768	1536	3072

Table 5.1: *Number of supernodes and components for different number of physical nodes.*

greater than the maximum component size. For the current homogeneous commercial version of the system physical node capacity is 12 TB, maximum component size is 0.5 TB and supernode cardinality is 12. Table 5.1 shows the total number of supernodes and components for such physical node size and configuration parameters for systems which have physical nodes ranging from 1 to 100.

The components are evenly distributed over physical nodes if the total number of components is a multiple of the number of physical nodes. Otherwise some physical nodes host k components and others $k+1$. Consider the case of a system built upon five physical nodes. Such a system has 192 components. They can not be equally distributed among physical nodes - three physical nodes host 38 components and two physical nodes host 39 components. When a system becomes full physical nodes with 38 components will be utilized at $38/39 * 100\% = 97,44\%$. There will be $3 * 2,56\% * 12TB = 0,9230TB$ space unused. Thus maximum system utilization will be

$$\frac{5 * 12TB - 0,9230TB}{5 * 12TB} * 100\% = 98,46\%$$

Figure 5.15 shows the percentage of non-utilized total capacity for different system sizes with a range of 1-100 physical nodes. When the number of physical nodes is power of 2 and in some other cases when number of component is a multiple of the number of physical nodes, the system is utilized at 100% (13 system configurations). In most cases (69 configurations) utilization is between 98% and 100%. In other cases system utilization is between 97% and 98% (15 configurations) and in the worst three cases it is below 97%. This figure shows that some configurations should not be

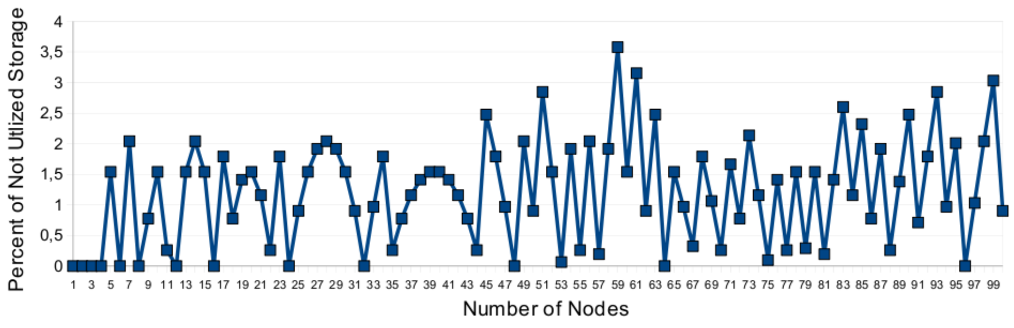


Figure 5.15: *Storage Utilization*

instantiated by system administrators if high utilization is their primary concern.

In the case of heterogeneous systems with physical nodes of different capacity, apart from degree of match of components to physical nodes, data resiliency may also limit storage utilization. Consider a system that is built on four physical nodes which is designed to handle the failure of one physical node without data loss with minimal redundancy overhead. Its supernode cardinality is set to default value of 12. To satisfy the requirements each block has to be split into 9 original and 3 redundant fragments. Each physical node has to host 3 of these 12 fragments. If any physical node fails each block could be rebuilt from the remaining fragments. This means that each physical node hosts 3 components of each supernode. If the physical nodes are not the same capacity then the physical nodes that are not the smallest ones will never be fully utilized. This problem becomes less severe in bigger systems where the number of physical nodes is greater then supernode cardinality. Under such systems there is more freedom of supernode to physical nodes assignment within the same resiliency level. This allows the system to assign more components to bigger physical nodes.

Support for Deduplication By using a distributed hash table with key computed based on block content using a hashing function we have created an elegant solution for truly global duplicate elimination in distributed system. Data is deduplicated against all blocks stored in the system.

Studies have shown that variable-size blocks provides better deduplication than fixed-size blocks ([65, 52, 50]). HYDRAstor supports such blocks. It allows external to HYDRAstor systems i.e. HFS to implement deduplication efficiently. HFS uses content-defined division of files into blocks, similar to one in [54], that produces blocks between a given minimum and maximum size. Content-defined division handles deduplication of files after they have been shifted by additions, removals, or modifications.

System deduplication ratio depends very much on the pattern of data stored in the system and backup data retention policy, thus it is hard to measure its effectiveness by running tests with artificial data. There are currently no academic or industry standard benchmarks that could be used to compare deduplication efficiency. However, the deduplication ratio of subsequent backups reaches more than 95% at some NEC clients. Total deduplication ratio i.e. ratio of the size of duplicates discovered by the system to the size of logically stored data, depends on number of full backups kept in the system. For example, if last 20 full backups were kept by such clients, then the total deduplication ratio would be 90%. This is because deduplication for first backup would be 0% and deduplication ratio of subsequent backups would be 95%. Total deduplication would be

$$\frac{0\%\ backup\ size\ +\ 19 * 95\%\ backup\ size}{20\ *\ backup\ size} \approx 90\%$$

Deletion on Demand Deletion on demand and data deduplication are hard to achieve within one system because of multiple ownership of data blocks. Distribution over many physical nodes, failure tolerance and content-addressability make it even harder to implement. By proper data organization, notion of explicit pointers to blocks in programming model and the separation of marking data for deletion from data reclamation, deletion on demand is successfully accomplished. To our knowledge HYDRAstor is the first highly scalable system supporting deduplication and deletion in a distributed environment.

The explicit pointers and DAG structure of blocks stored by HYDRAstor allowed the implementation of a deletion algorithm similar to a distributed garbage collection that computes reference counters for each data block. Splitting deletion into deletion epochs, and making guarantees that blocks will be not deleted only when they are directly or indirectly pointed by retention root stored in the same or next epoch than the block was stored, allowed us to solve the problem with deduplication against blocks scheduled for reclamation. In other words, whenever a block with reference counters set to 0 is used as a duplicate it is marked to survive until the next deletion epoch; within next epoch the counters computing process will include the newly stored retention root, thus the block will not be removed.

Keeping supernode cardinality copies of per-block reference counters provides robustness in the face of failures. Stamping counters with deletion epoch allows the detection of obsolete counters on physical nodes that were temporarily detached from the system that executed subsequent deletion rounds. Additionally, since counters have higher redundancy than data and pointers are kept in class copy deletion and space reclamation may be executed even if data stored with a lower redundancy is lost. This high metadata resiliency allows the system to reclaim space occupied by fragments even if blocks related to these fragments cannot be rebuilt.

Multiple ownership of data blocks makes any mistake in computing counters much more devastating compared to non-deduplication storage systems because all the data that the block was used for as a base for deduplication is lost too. Additionally, the scale of the system and its dynamism increases chances of software or hardware error. That is why several components of each supernode computes counters in parallel to verify that all the values are the same at the end of the process. If they are not the same counters computing is aborted and the administrator is notified about the problem. However, as yet HYDRAstor has never reported such a problem in production environment.

The deletion operation has a low impact on write and read operations. First of all counters are not updated on regular user writes. This is very important in the case of duplicate writes which are much faster than non-duplicate writes, because they do not need to write any data to disk. In fact, in most cases they do not even need any disk read operations due to the RAM cache with hash keys of data stored on the disks and SCC chains summary kept by write initiators. Counters computation is executed by several peers of each supernode. Since supernodes are balanced over the physical nodes and data is evenly distributed over the supernodes, counters computation executed as a batch processing is well balanced over all physical nodes. The amount of work that has to be done in any given deletion epoch is proportional to the number of pointers written since the previous counters computation, the number of stored deletion roots (i.e. number of trees marked for deletion) and to the trees' depth and size. It does not depend on the total number of blocks in the system. The counters updating process finds out pointers and deletion roots that have to be processed in the current round by scanning SCCs belonging to the tail of SCCs chains created after the last counters update. Last but not least, SCCs metadata (i.e. SCC indexes) is separated from SCCs itself. By doing this the deletion operation only updates metadata. Compared to SCC, this is a very small file. This results in a smaller number of disks accesses required to update counters information. Actual space reclamation is done by other background processes that create new SCCs updated according to the metadata. The process starts with the SCCs with highest ratio of space to be gained to maximizes reclaimed space within one SCCs update.

Handling Almost Full System While working on the HYDRAstor project, there were many problems with handling an almost full system in a stable and predictable way. We suspected that the problems are generic and are related to the fact that there are many conditions imposed on balancing data in a distributed storage system. Most of the conditions are static for a given list of physical nodes and can be computed and set only once. For example, in the case of HYDRAstor such conditions are related to peers and write initiators distribution. However, there are conditions that are more dynamic and depend on other, unstable factors. An example of such condition is the physical node being full. This depends both on the physical nodes list, the amount of data in the system and its distribution over the physical nodes. Since the amount of data may constantly change, the condition for the physical node being full may also constantly change. If the balancing algorithm is sensitive to these changes (i.e. the system tries to be instantly and perfectly balanced all the time) it results in data constantly being transferring back and forth. The first version of HYDRAstor clearly showed these problems. It only had local balancing implemented - without implemented global balancing, components pinning and conditions that limit cases when a full physical node can transfer out some of its components. Each physical node decided independently what balancing action should be executed by analyzing possible transfers to its neighboring physical nodes and values of related entropy improvements, including improvements of physical nodes fullness. We encountered three problematic cases:

- System with failed physical nodes. Components are recovered on physical nodes that are the best according to a local entropy change. The recovered components may cause “alive” physical nodes to become full due to the reconstruction of data previously hosted by failed ones and new writes coming in. Additionally, after component is transferred, it may turn out that the recovered component disrupts other entropy dimensions. For example, after a physical node that hosts the new component becomes full, it should transfer away other, originally hosted component, to another physical node in order to optimize write initiator dimension. Thus, another transfer is executed. This transfer in turn may disrupt entropy of the additional physical node and induce next transfer. This leads to a chain of transfers and causes the system to become unstable at least in terms of components not having their data stored locally. This leads to problems with write and read performance and deletion execution.

- A restart of multiple physical nodes may cause two or more components swapping their hosting physical nodes. If the physical nodes are almost full, the components cannot change back their locations - they would require two single transfers, but the first single transfer would increase the value of entropy function, so it is not allowed. Two "atomic" transfers would be required, but system does not support such an operation. Such case results in unnecessary transfers of data belonging to the swapped components.

- Heterogeneous system with the smaller physical nodes reaching full state. Such physical nodes transfer components out to the bigger ones. This may lead to a situation when a significant number of components of each supernode is hosted by a limited number of physical nodes. As a result, the system loses failure resiliency because a failure of a big physical nodes may cause data loss if a failed physical node hosts more components than the number of resilient fragments. Such failure may even cause metadata loss if a failed physical node hosts at least half of the supernode cardinality components.

To address above problems, global balancing, pinning and limiting the freedom of component transfers out of full physical nodes were introduced. The general rule for handling full physical nodes is that pinned components are not transferred unless they are hosted by a physical node with a failed disk or their pinning target physical node is down. This greatly limits the possible number of transfers. Only components that are originally hosted by a failed physical node can be transferred between physical nodes. This practically solves the problem with transfers that induce chain of transfers and makes system unstable.

In the current version, there are three scenarios for a physical node being full:

- Full healthy physical node (i.e. a physical node with all healthy disks) hosting pinned components only.

- Physical node hosting non-pinned components becomes full. Such components come from broken or unreachable physical nodes. At the time when components are recovered, they are placed on physical nodes that are best according to various entropy criteria i.e. they provide the best resiliency level. They do not have to be placed on physical nodes that have the most free space. Later, when a user writes data it may turn out that the physical nodes that host recovered components become full.

- Physical node that becomes full after one or more of its disks fail. Data that was hosted by the failed disk is reconstructed on the remaining ones making them full.

The first scenario means that system has become full globally. This is because global balancing places components in the optimal way with regard to storage utilization and other balancing conditions (i.e. resiliency level). Thus, no component transfer can improve utilization without worsening other conditions. A system does nothing when such type of full physical node occurs. The second and third scenarios occur when the physical node is full, but system is not full globally. In such a case if a component or components were not transferred away from the full physical node it would stop accepting writes - this would result in writes not being accepted totally. This is due to the uniform distribution of keys over components; all components receive the same number of blocks thus, if one component blocks writes, writes are blocked globally. In the second scenario only non-pinned components can be transferred. Pinned components are not transferred for the same reason as in the first scenario.

The second and third scenarios are dealt with by a balancing algorithm and resource management system. If in the second or third scenario a physical node becomes close to full (depending on configuration it is 98% - 99% of used disk space) a balancing algorithm finds physical node that has enough free space to accept a new component and is optimal with regards to other balancing requirements. Then component is moved to the new physical node and transfer of its data from the old to the new physical node is started. However, if during this process the user is writing data at pace that exceeds pace of data being transferred out then it is possible that the near full physical node may reach 100% utilization and block all writes globally. Thus to handle such a situation data transferred out of the near full physical node gets higher priority in order to ensure that data is transferred out at least as fast as the new user writes are coming. This guarantees that the physical node accepts writes without interruption, but it may temporarily slow down total system throughput by 50% because 50% of full physical node resources is used by handing the transferred out component.

5.4.2 Fault Tolerance

HYDRAstor architecture has no single point of failure. Data is spread over physical nodes in such a way that enables the system to operate non-stop as long as there is no massive failure of many physical nodes at the same time (i.e. more than half of supernode cardinality minus 1) or there are consecutive failures that occur in time window short enough for metadata and data lost in a previous failure to not have been recovered yet. Users can configure failure tolerance by changing supernode cardinality. The higher value the higher failure tolerance (assuming that number of physical nodes is higher than supernode cardinality). Users can also choose a redundancy level for each written data block which allows them to create flexible data storing policies for various types of backed up data.

Data Resiliency By using erasure coding HYDRAstor is very flexible and efficient in providing resiliency. For each data block the user decides the resiliency level and related storage overhead. A solution based on erasure codes is much more efficient than RAID-1, RAID-5 and RAID-6 that are commonly used in industry. Tables 5.2 shows flexibility in terms of number of possible physical node/disk failures HYDRAstor can accept before data is lost and related storage overhead for supernode cardinality is equal to 12 and 16.

Number of acceptable failures before data loss (i.e. number of redundant fragments)	1	2	3	4	5	6	7	8
Storage overhead (%) for SNC=12	9,09	20	33,33	50	71,43	100	N/A	N/A
Storage overhead (%) for SNC=16	6,67	14,29	23,08	33,3	45,45	60	77,78	100

Table 5.2: *Number of Accepted Failures and Storage Overhead.*

RAID-1 accepts one disk failure with 100% storage overhead. For such overhead HYDRAstor handles 6 and 8 disk failures for supernode cardinality equal to 12 and 16 respectively. RAID-5 accepts one disk failure with 25% storage overhead. With similar storage overhead of 20% and 23,08% HYDRAstor handles 2 and 3 disk failures for supernode cardinality equal to 12 and 16 respectively. HYDRAstor handles the failure of one disk with 9,09% and 6,67% overhead for supernode cardinality equal to 12 and 16 respectively. Similarly, compared to RAID-6 HYDRAstor has a greater resilience at lower storage overhead. RAID-6 accepts two disk failures with 50% storage overhead. For that level of resiliency HYDRAstor requires only 20% and 14,28% of storage overhead for supernode cardinality equal to 12 and 16 respectively. With 50% and 45.45% of storage overhead HYDRAstor can handle 4 and 5 disk failures for supernode cardinality equal to 12 and 16 respectively.

HYDRAstor does not satisfy one of the resiliency requirements described in chapter 3 on page 15. As the system increases the number of physical nodes and supernodes spanned over them also increases. The level of resiliency each supernode provides stays at the same level, but since number of supernodes increases so does the probability that at least one of them will lose data. If one supernode losses data all data is lost (after a supernode is lost there are data holes in the user data stream which makes it practically unusable). To solve this problem HYDRAstor should automatically increase supernode cardinality and proportionally increase the number of redundant fragments of each data block together with increasing system size. This would overcome the increased probability of data loss coming from increased number of supernodes. HYDRAstor does not contain this feature because it is quite hard to implement and so far no client has requested it. However, such a feature is feasible and can be added if requested by the storage market.

Data Reconstruction The data placement organization of HYDRAstor allows fast detection of a failed or unreachable physical node and efficient data reconstruction. After a physical node fails all components that were hosted by it are recovered by their peers on other physical nodes. After a component is recovered it initiates checking completeness of its SCCs chain to find out which fragments are missing locally. It then communicates with its peers and reconstructs this data locally by downloading fragments from peers and reconstructing missing local fragments. To make this operation faster fragments are reconstructed at the level of SCCs i.e. the whole new SCCs are created - there is no in place operations. This limits random disk access and limits fragmentation of data stored on disks.

Data reconstruction in HYDRAstor is superior to RAIDs because:

- HYDRAstor starts reconstruction just after it finds out that a failure has occurred. RAID starts reconstruction after failed disk is replaced by the administrator, thus reconstruction depends on human intervention that may be very delayed. During that time the system operates with decreased resiliency.

- HYDRAstor reconstructs only the missing data. RAID reconstructs the whole disk regardless of amount of data originally stored by that disk.

- HYDRAstor tries to spread recovered components in such a way that number of recovered components hosted by one physical node is minimized. By doing this re-computation of missing fragments from excising fragments is distributed over many physical nodes. In the case of RAID only one physical node does all the work.

In a fully utilized system data reconstruction tasks compete for system resources with other tasks in particular write and read operations. The user can choose a system wide policy that assigns specified percentage of resources to data reconstruction tasks and the rest to store and retrieve operations. For example, a *fast* policy assigns all resources to store and retrieve operations. Such a policy allows the user to store backup data as quickly as possible but with increased risk of data loss. It should be used by users that have very small time windows for making the backup, but they seldom do the backup i.e. once a day, and the resources can be consumed by reconstruction tasks the rest of the time. In opposition to fast policy is *reliable* policy that assigns most of the resources, for example 80%, to critical tasks i.e. tasks that reconstruct data that has very low or no redundancy left. Tasks that are less critical get fewer resources. Such a policy increases the data resiliency at the expense of read and write performance.

Data Scrubbing SCCs chains allow each physical node to scan disk data that belong to components hosted by the physical node in an orderly manner. If some fragments are not readable they are reconstructed from fragments read from peers and stored in different location on the disk. Additionally, data scrubbing can be done by one piece of SCCs chain at once, thus the process can be divided into smaller operations that are executed when a physical node is lightly loaded.

Operation under Faulty System HYDRAstor enables the non-stop execution of data storing, retrieving and deletion upon physical node or disk failure. Upon physical node failure components hosted by it are quickly recovered by their peers on other physical nodes. By doing this the system can accept new writes even if the user requires that all components of each supernode store fragments (i.e. in the case when the write threshold is set to supernode cardinality). Data deduplication works properly even if some peers of each supernode do not have their SCCs complete. As long data blocks can be read and their resiliency level is not too low they can be used for deduplication. With the respect to read operation it works properly as long as there is enough data fragments to reconstruct blocks. Deletion operation can proceed as long as each supernode has several (by default 3/4 of supernode cardinality) full compositions chains in order to be able to reliably analyze all written pointers to blocks and reliably store computed reference counters. System strives to keep several (by default 3/4 of supernode cardinality) full chains within each supernode. If number of full chains in a given supernode is low, component transfers of this supernode are blocked until some of the component's peer reconstruct its full local chain. Thus, only massive failures causing one or more supernodes having less than seven SCCs chains stored locally would block deletion.

Upon disk failure physical node rebuilds missing fragments locally on other available disks. If this disk loss, resulting in physical node capacity decrease, causes the physical node to become full, some of its components are transferred to other physical nodes as described above in section 5.4.1 on page 84.

In summary HYDRAstor survives failures as long as no more than half of supernode cardinality minus 1 physical nodes hosting components belonging to the same supernode fail at the same moment. In such a case missing components and important metadata are recovered and reconstructed to reach the initial level of failure resiliency. User data resiliency depends on the level of redundancy associated with each data block. Even if data with low redundancy is lost due to a failure, data with higher redundancy is available. Further to this, since metadata related to deletion operation like chains completeness is kept by each component of each supernode, the user may delete fragments related to missing blocks to reclaim space occupied by lost data.

Upon failure the system gradually decreases performance. The performance loss after components are recovered depends on the degree of uniform dispersion of recovered components over available physical nodes. For example, if a homogeneous system's component are evenly recovered on other physical nodes then the performance loss is proportional to the loss in resources. If they are not uniformly distributed then the performance drop is slightly higher because system performance is limited by the physical nodes being mostly loaded i.e. in case of homogeneous system is limited by the physical nodes hosting the most components . However, since the difference in the number of components hosted by each physical node is not greater than one, then that additional performance drop is around *1/number of components hosted by a physical node* . For a standard commercial configuration the number of components hosted by a physical node is between 24 and 48 depending on number of physical nodes the system is built upon. Thus the additional performance drop which results from an unequal number of components on physical nodes is between 2 and 4 percent.

Figure 5.16 shows the system behavior just after physical node failure. In the experiment there were four physical nodes each having twelve 1 TB SATA disks, 20 GB or RAM, two quad-core 3GHz CPUs.

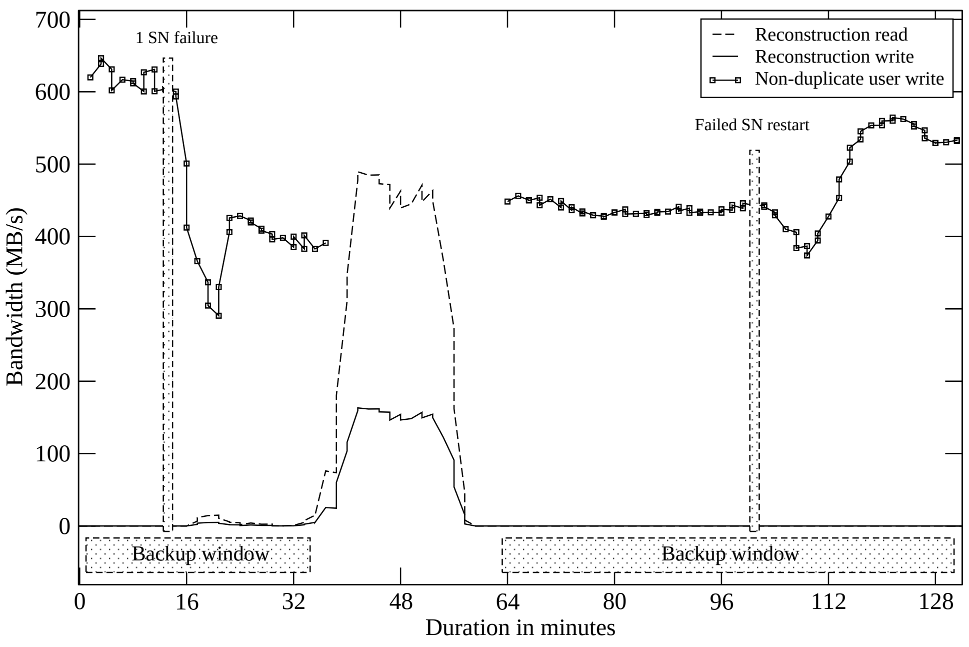


Figure 5.16: *Writes Under Failed Physical Node.*

source: [26]

The test started with writing to the healthy system, achieving throughput over 600 MB/s. After 14 minutes one physical node failed. Write performance initially dropped to 300 MB/s and then stabilized at around 400 MB/s. The initial drop is caused by timeouts on messages to the failed physical nodes. After missing components are recovered throughput stabilizes at 400 MB/s. Each recovered component initiates data rebuilding. This experiment was executed with the fast policy set i.e. one that prefers high write performance over quick data reconstruction. Thus reconstruction tasks are suppressed because of the ongoing user backup. After the backup session is finished reconstruction started to work with full bandwidth (around 33th minute). The reconstruction of each block required reading 9 fragments to rebuild 3 missing ones. The reconstruction read bandwidth reached 480 MB/s and reconstruction write bandwidth reached 160 MB/s. In the 58th minute rebuilding was finished leaving a healthy system running over three physical nodes. In the 64th minute the next backup session started achieving write bandwidth of 430 MB/s. In the 100th minute the failed physical node was recovered and connected back to the system. Just after the reconnection write bandwidth dropped to 380 MB/s, because of destabilization caused by components being transferred back to their original location. After component re-balancing was finished write performance increased to about 550 MB/s. It was lower than the initial 600 MB/s because some of the data i.e. SCCs was not in the correct place. After they were transferred back to the correct location i.e. to the reconnected physical node, write throughput was back around 600 MB/s (not shown on the figure).

Operation after Failure Repair After a physical node is repaired and reconnected to the system components that were hosted by it before the failure, i.e. components pinned to it, are transferred back. By doing this the system minimizes the amount of data that has to be transferred. Most of the components data is hosted by the failed physical node because failure typically results in no data loss (i.e. the physical node was considered as failed because it was unreachable due to network problems) or small part of the data is lost (i.e. one out of twelve disks failed). Transferring components back to their original configuration guarantees that the data distribution is optimal, because it was optimal before the failure.

The reconnected physical node has to integrate its local data with the corresponding data that was recovered on other physical nodes. This data might have been deleted while the physical node was down and new data might have been stored. By scanning compositions chains and communicating with peers of all components the physical node hosts it finds out about the newest SCC indexes related to its local, possible outdated, SCCs. Since these remote SCC indexes contain up to date metadata information it is used to update local SCC indexes. After local SCC indexes are updated the physical node downloads missing fragments and removes data that was deleted by the user while the physical node was down.

Failure Reporting Data organization of HYDRAstor allows the system to easily verify data health. Each component scans chains of its SCCs and sends reports to write initiators with the status of each SCC. Among other information such reports contain information about fragments data classes and information about missing fragments. Based on this information write initiators build information about the state of all local and remote SCCs they have created. That information is sufficient for write initiator to figure out the state of all data blocks related to writes executed from beginning of the system lifetime. In particular the write initiator finds out what is the least number of potential failures that will cause data loss if they occur. If that number is negative it means that data has already been lost. The write initiator sends this report to the global leader component that aggregates it to build a report for the whole system. Then the user can send a request to the global leader to get this information.

Data and Metadata Consistency During and After Failure HYDRAstor provides consistency on several levels. On the highest level it guarantees that overlay network with the self-repairing feature does not lead to network disintegration in the case of failures. The overlay network is based on a distributed hash table. At the beginning of the project we analyzed available distributed hash tables (together with our in-house fixed prefix network) and were not convinced that any of these DHTs guarantees that in case of a very unusual sequence of failures it provides network integration. In particular it does not create more than one instance of the network which obviously would lead to major metadata disintegration. To avoid this potential problem we have decided to use DHT without any recovery operations, but instead assume that DHT nodes are entities that provide very high failure tolerance. Thus the idea of the supernode was born. Since supernodes are spanned over a dozen physical nodes, the physical nodes can cooperate to support the supernode's consistent state. It turned out to be a relatively easy task, since there were well known algorithms described in the literature that provide consensus [11] and thus a consistent state of data over physical nodes. Thus overlay network consistency is provided by limited functionality DHT that is spanned over failure resilient nodes i.e. supernodes. Consistency of user data is provided on a supernode level. Since data blocks can only be stored or deleted, but cannot be updated, providing consistency boils down to the recreation of missing fragments and the removal of deleted ones. Simply put, after a physical node is reconnected to the system its components download SCC indexes about SCCs that where created while the physical node was disconnected from the system and update SCC indexes information about deleted fragments in the existing SCCs. Based on the updated SCCs indexes the physical node rebuilds and removes appropriate data fragments. Data blocks consistency is also provided by recomputing hash keys of blocks and comparing them with hash keys stored together with data fragments. If it turns out that they do not match most probably due to corruption of a fragment, blocks can be reconstructed by choosing different subsets of fragments to find out one that can rebuild the block with the proper hash. If such set is found other fragments are removed and recomputed from the proper ones.

5.4.3 Scalability

Building system upon a distributed hash table gave as solid base for reaching scalability.

Scalability of Capacity Scalability of capacity is a direct consequence of efficient storage usage described in section 5.4.1 on page 79. Since storage utilization in general does not depend on the number of physical nodes and since data deduplication is done globally, its efficiency does not depend on system size, thus the capacity of HYDRAstor scales very well.

Scalability of Write and Read Performance Figure 5.17 shows scalability of performance of non-duplicated write, write of 100% duplicated blocks and read operation.

Each storage node (SN) was running under Linux version Red Hat EL 5, and had twelve 1TB SATA disks, 24GB RAM, two quad-core 3 GHz CPUs and four GigE cards. Each access node (AN) was running under the same Linux version, and had two 146GB disks, 8GB RAM and the same two quad-core 3 GHz CPUs, four GigE cards for connections with SNs and six GigE cards for external connections.

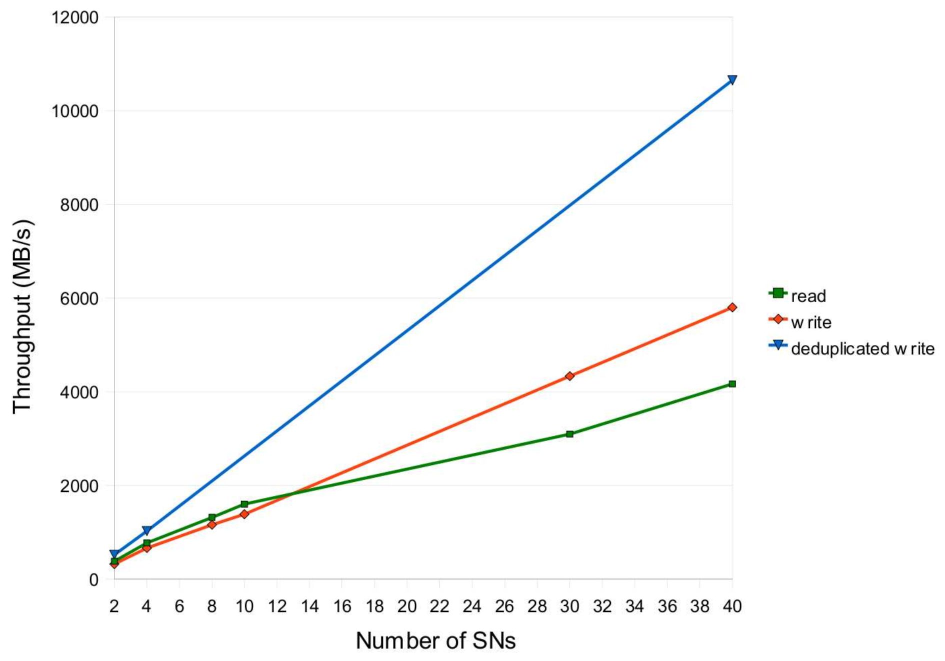


Figure 5.17: *Scalability of write and read operations.*

Performance was tested for five hardware configurations: 1 AN + 2 SNs, 2 ANs + 4 SNs, 4 ANs + 8 SNs, 5 ANs + 10 SNs, 15 ANs + 30 ANs, and 20 ANs + 40 SNs. Unfortunately we do not have results for this exact hardware setup for middle configurations for deduplicated write. However, there are other performance results (no presented here) executed on slightly modified versions of software and hardware that are consistent with the figure 5.17 for deduplication writes.

For each initially empty system data was written. There were no duplicated blocks. In the figure it is shown as "write". Then the same data was stored again, making all blocks duplicates. This is shown as "dup-write". Then the stored data was read. It is shown as "read".

These results show that write operations, both in case of non-duplicates and duplicates, scale linearly. The throughput of a duplicated write is around 260 MB/s per storage node. The throughput of a non-duplicated write is around 145 MB/s. Read operation for small configurations, i.e systems having ten or less physical nodes scales linearly. For a bigger system the scalability is not that good - it is lower by 30% than the expected throughput. This is related to problem with efficiency of handling many user streams.

This problem is described below in section 5.4.6 on page 103.

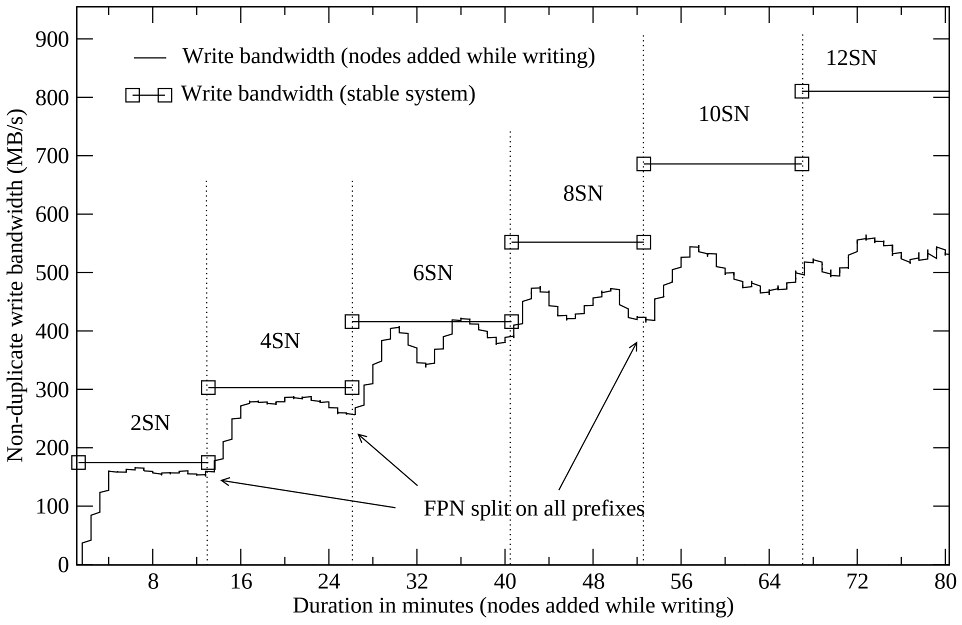


Figure 5.18: *Write Scalability.*

source: [26]

Scalability of Write Operation under Dynamic System Extension Figure 5.18 shows how performance of a write operation is scaled when the number of physical nodes increases dynamically. Each physical node was running one backend HYDRAstor server under Linux version Red Hat EL 5.1, and had six 500 GB SATA disks, 6GB RAM, two dual-core 3 GHz CPUs and two GigE cards. There were two experiments done:

- dynamic one - physical nodes are added while user writes are coming
- static one - number of physical nodes is stable - each measurement was performed independently; test initialized system from scratch and loaded the same amount of random, non-duplicated data. Time on X axis refers to the dynamic case only.

The results indicate that in range of 2 to 12 homogeneous physical nodes tested, the system performance scales linearly with the system growth in the static example. The hash space is equally divided across the storage of physical nodes. It guarantees that every machine is equally loaded and does not become a bottleneck. The dynamic example shows the cost of the system growth. It is manifested in a lower bandwidth than in the static example. This is due to most of the data being stored on the oldest physical nodes. In order to check for duplicate elimination the system has to query these physical nodes on every write. The older physical nodes becomes bottlenecks and slow down the whole system. However, after all data transfers are completed the system reaches the same throughput as in the static case (this is not shown on the figure).

Scalability of Heterogeneous Systems The results presented above demonstrate performance of a homogeneous systems. Since our priority was so far to deliver system which run over homogeneous physical nodes we did not execute scalability tests of heterogeneous systems. To some extend we already verified that scalability of capacity could be reached because the module that computes components distribution over physical nodes had already been executed against various heterogeneous systems, and it was generating components distribution that results in capacity utilization similar to that of homogeneous systems.

We expect that scalability of performance of heterogeneous systems with proportional physical nodes will scale in the same manner as homogeneous systems. By assigning supernode components proportionally to physical nodes resources the system should utilize each physical node at the same level, thus performance should scale.

HYDRAstor architecture also handles the most difficult heterogeneous systems with non-proportional physical nodes. From a DSS standpoint it is hard to solve imbalances in physical nodes computing power and capacity. Generally if a physical node throughput is assigned proportionally to its capacity then physical nodes with weaker computing power slow down the system, because they are the bottlenecks. If throughput is assigned proportionally to computing power than physical nodes with higher than average ratio of capacity to computing power will not use up all its capacity. The proposed solution to this challenge is to divide components into two groups - write initiators and non-write initiators. Write initiators are more computing power intensive than non-write initiator components. Balancing algorithm assigns components proportionally to physical nodes capacity and within this assignment it gives more write initiators to more powerful physical nodes.

5.4.4 Self Management

HYDRAstor is highly self managing system. The role of administrator is very limited. First of all, there are no volumes or storage configuration parameters that have to be estimated in advance and changed during system extension. In HYDRAstor the only action an administrator has to take is to update the list of physical nodes that a system should use. Data migration and balancing is done automatically by the system and does not require any human intervention.

The system automatically detect failures, recovers missing components and rebuilds lost data to reach the resiliency level required by the user. After being notified about a failure the administrator only has to replace the failed physical nodes or disks. After this operation the system automatically incorporates reconnected or fixed hardware and reaches the same state as before the failure.

5.4.5 High Availability

The high availability of HYDRAstor is a result of failure tolerance and self management, especially self healing. Upon temporary failures components are quickly recovered on other physical nodes and are ready to serve write requests. Data redundancy provides non-stop data retrieval even if some part of the system is not reachable.

Physical node retiring operations allow for the temporary removal of physical nodes for planned administration actions like hardware or software upgrade. Physical node retiring in big systems which have more than supernodes cardinality nodes allows for such actions without sacrificing of failure resiliency, because all components and their fragments are transferred to other physical nodes in the system in such a way that no physical node hosts more then one peer of any supernode.

5.4.6 Performance

Duplicate and Non-Duplicate Write and Read Throughput

Figure 5.19 shows the result of an experiment that shows write throughput as a function of the fraction of blocks detected as duplicates. The tested system was built upon four physical nodes. Each storage node had twelve 1 TB SATA disks, 20 GB of RAM, two quad-core 3GHz CPUs and four GigE cards and was running Red Hal EL 5.1 version of Linux operating system.

Four client machines generated stream of blocks with a specified percentage of duplicates. The sequence of duplicated blocks were written in the same order as the base data, thus they recreated the same stream as it

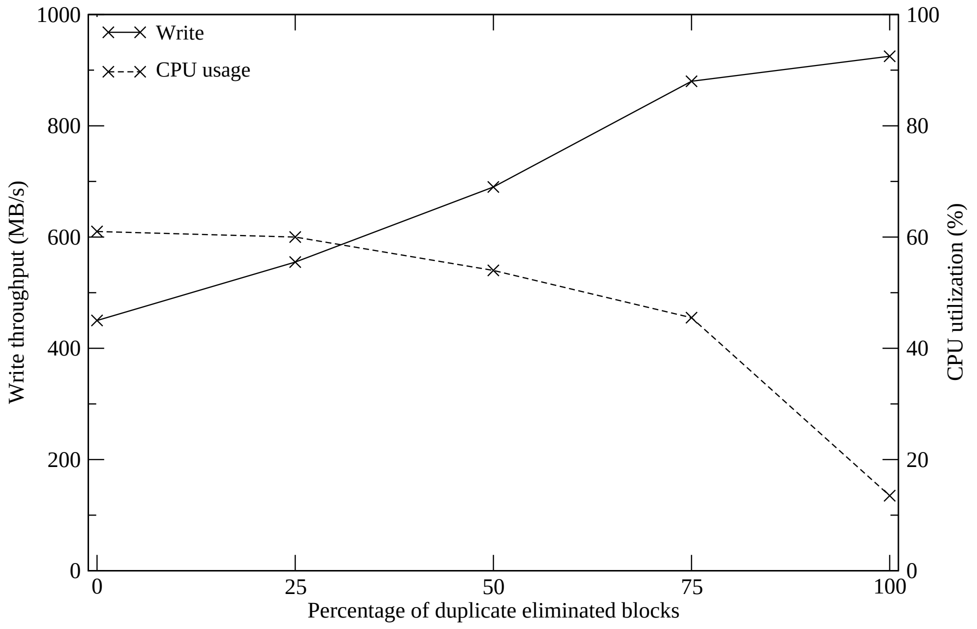


Figure 5.19: *Write throughput as a function of duplicate ratio.*

source: [26]

The results show that HYDRAstor provides very high bandwidth, especially of duplicated blocks. The graph shows that with increase of duplicated blocks write throughput increased almost linearly. Only at 100% of duplicates is the increase slightly lower, but this is a result of the limits of network bandwidth between HYDRAstor and clients writing data. Tests of results executed by other teams working on the project suggest that without this network bottleneck throughput for 100% duplicates would be slightly higher and would reach 950 MB/s.

This high throughput is the result of proper design of both high level data organization i.e. hash table with supernodes and low level organization on disks. High level organization results in the even distribution of data over the system, balanced load over all physical nodes and fast mechanism for data location. On low, disk level, non-duplicate writes result of bulk transfer to disks - for a loaded system fragments are written to disk in 2MB chunks. Duplicate writes are fast due to SCC-based organization. Such organization allows the prefetching of SCC indexes to the RAM cache - it allows for quick checks if a given block is already stored. If it is stored checking SCC summary reports submitted in the background by the remaining peers is enough to verify that the block is reconstructable. Through such data organization writing duplicate blocks does not require access to disk in most cases. Furthermore, a high duplicates ratio decreases network utilization between physical nodes because no fragments are sent and decreases CPU utilization because no erasure coding processing is required. This decrease in CPU is shown on Figure 5.19. Since in the typical backup stream deduplication ratio is over 80% there is enough free CPU for processing background operations like data reconstruction, space reclamation and data scrubbing, thus the background operations do not impact user-visible performance.

The performance of read operation is highly dependent on factors like the sequentiality of the data read, number of clients reading simultaneously, distribution of the duplicates and degree of stream sequentiality and locality on disks. These factors determine how much disk data prefetching into RAM cache limits access to the disks. In this experiment read throughput was between 400MB/s and 550MB/s, so it was similar to throughput of non-duplicate writes.

HYDRAstor Performance Compared to Other Systems

It is hard to compare efficiency of data organization of a commercial systems like HYDRAstor to other commercial systems for two reasons. First, there are no standard benchmarks that vendors can use to test their systems. Thus performance results may not be exactly comparable. Second, vendors use different hardware with different characteristics of main components like CPU speed, disks throughput and size of RAM, so the solutions are not compared based on the same hardware. However, we would try to outline the position of HYDRAstor among other main competitors. Not much can be done with the first problem. Thus we assume that the performance results published by vendors to some extend are comparable and we will compare the maximum throughput the systems provide. For the second problem one could compare ratio of throughput to purchase price for the system. Price of the system can be considered as a price for "abstract units of hardware". Thus such ratio is a performance of single "abstract hardware unit" and can be compared. Unfortunately, in most cases vendors do not publish price information officially. However, system prices tend to be correlated with system capacity. Thus it it reasonable to compare ratio of performance to capacity.

	Total Write Throughput (MB/s)	Usable Raw Capacity (TB)	Write Throughput per Usable Raw Capacity (MB/s/TB)
HYDRAstor 20AN + 40SN	10646	480	21,8
EMC GDA	3555	384	9,26
EMC DD880 w/Boost	2444	192	12,73
Greenbytes GB4000	950	230	4,13
HP D2D4312	666	48	13,86
IBM ProtecTier	1500	1000	1,5
Symantec NetBackup 5020	3250	288	11,28

Table 5.3: *HYDRAstor and Competing Systems Performance.*

Table 5.3 shows the total maximum write throughput of duplicated blocks, maximum raw capacity and write duplication throughput per TB of capacity for systems competing with HYDRAstor. These are the systems that support global inline duplicate elimination. Selection of the system to compare is taken from their performance comparison [68] done by W. Curtis Preston who is an expert in backup and recovery systems.

Maximum write throughput and maximum capacity are taken from W. Curtis Preston's comparison. However, we have updated some of the information. The original comparison has mixture of raw capacity (i.e. capacity that is consumed by both user data and redundant data like RAID parity data or erasure coded redundant fragments) and usable capacity (i.e. capacity without space required for redundant data). We unified the data and compared raw capacity only. However, in case of IBM ProtectTier we are not sure that reported capacity is in fact raw capacity because it is not clearly stated in IBM's specification. NetBackup 5000 is replaced by its newer version NetBackup 5020 having higher capacity. Its maximum throughput is corrected from 7166 MB/s down to 3250 MB/s (source [88]).

IBM ProtectTier's throughput is updated from 1000 MB/s to 1500 MB/s (source [46]). In case of HYDRAstor data is taken from our performance experiments for system built upon 40 storage nodes and 20 access nodes. Data reported by Curtis is an expected throughput for 110 storage nodes and 55 access nodes.

This comparison shows that HYDRAstor is the best performing system both in total throughput and in throughput per TB of provided capacity. Moreover, data organization of HYDRAstor allows the write throughput to be scaled up by adding additional physical nodes to the system. In fact, NEC in its commercial offer has HYDRAstor built upon 110 storage nodes and 55 access nodes. Such system, if scaled linearly to that size, would give a total write throughput around 29000 MB/s. Other systems cannot be scaled by adding additional nodes.

	Total Write Throughput (MB/s)	Usable Raw Capacity (TB)	Write Throughput per Usable Raw Capacity (MB/s/TB)
HYDRAstor + HFS 20AN + 40SN no duplication	5800	480	12,09
Thecus N5500 (*)	54,5	5	10,9
QNAP TS-559 Pro (*)	105,5	10	10,55
Synology DS1010+ (*)	103,2	15	6,88
Blue Arc Titan 1100 Series (**)	312,5	128	2,44
Hitachi NAS Platform 3080 (**)	700	4000	0,18
DDN S2A6620 (**)	2000	240	8,3
DDN SFA10000 (**)	10000	2400	4,17
Panasas PAS 12 (**)	1500	40	37,5

Table 5.4: *Write performance of NAS systems.*

HYDRAstor and HFS together implement NFS and CIFS interface thus together they are a NAS system. Since they are NAS system they can be compared to standard NAS systems (i.e. systems without deduplication) to find out how HYDRAstor's novel data organization with support for deduplication impacts the performance of regular non duplicated writes. Table 5.4 shows the write throughput of various small, middle and enterprise size NAS systems that have performance results published by vendors (marked with '**') or the performance results were published by service www.smallnetbuilder.com (marked with '*') that tested them. The table also shows the throughput of non duplicated write of HYDRAstor running with HFS.

The comparison shows that HYDRAstor can compete with NAS systems as long as the NAS systems were used for storing backup files (HYDRAstor would not reach such a high throughput if small files were written - in such a case regular NAS systems would be superior). Apart from Panasas PAS 12 (this is probably the fastest NAS currently available) HYDRAstor has similar level of performance compared to NAS systems. It means that HYDRAstor's novel data organization provides performance at least as good as the old well known and tuned data organization based on regular file systems.

Read Performance Problem

Figure 5.17 in section 5.4.3 shows that there is a problem with the scalability of the read operation. This reflects symptom of inefficiency when many users read data that was stored by many users writing simultaneously. Further to this, theoretically there is another related problem that might impact throughput of duplicate write.

The first problem is related to a number of concurrent users writing data blocks at the same time. Since the number of users in the tests shown on figure 5.17 was proportional to system size that problem is visible on this figure. The stream of blocks generated by each user is distributed over the whole system. Each supernode receives the same number of blocks from each stream. Each stream of blocks is divided into supernode cardinality streams of fragments. Each fragment is stored at the end of the last open SCC file. If there are many users writing data simultaneously then fragments coming from different streams are interwoven in each SCC. Even though the stream sorting process merges the interwoven fragments within each SCCs into consecutive disk sectors, with high number of streams of fragments there is not enough consecutive space on the disk of sufficient length to efficiently prefetch stream data fragments into the RAM cache.

The prefetch size has a constant size. For a high enough number of streams the consecutive space on disk has smaller size then the prefetch size. This results in reading into the cache not needed fragments which kicks out other prefetched ones that would be used by following read operations. It lowers the efficiency of the cache and increases random accesses to disks. Another problem is the number of total streams of fragments that are read from one storage node at the same time. It is proportional to number of streams that are read simultaneously by all users. Each such stream is read from a different SCC, thus it is read from different file. The increase in the number of streams read increases the number of random reads on disk, which are inefficient. These problems might even be multiplied by the data deduplication. Deduplication devastates data locality because during a write parts of user data stream may be deduplicated over all previously stored user streams. It results in user data stream being randomly scattered over the whole storage space. When users read stream of blocks that were stored as a duplicates against many previously stored streams the data may have to be gathered from multiple SCCs uniformly distributed over the SCC chains. This further increases random disk accesses and degrades read operation performance.

There might be another scalability performance problem that concerns the performance of write of duplicated blocks. For small systems with a small number of streams written concurrently almost all deduplicated blocks are identified in the RAM cache. Bigger systems with more concurrent streams written have higher frequency of disk I/O for deduplication checks. The mechanism is very similar to the problem with scalability of reads. The system checks if a new block that is to be stored is a duplicate by sending a query to the physical node that is responsible for the key space the block belongs to. This check is done by reading parts of matching SCC indexes. The size of SCCs index is around 1,2 MB. The smallest reasonable size of a block to read from disk is around 100 KB, because reading smaller blocks takes the same time. Problem shows up when there are so many streams written concurrently in the SCCs by users that were writing at the same time. If there are more than 12 streams in an SCC each stream has less than 100 KB of metadata in SCC index. This degrades prefetching efficiency. It might be noticeable when there are tens of concurrent streams. Additionally, to simplify implementation, the SCCs index cache keeps all entries read from disk within single 100 KB I/O, not only the entries that are related to prefetched stream. This causes quicker removal of cache entries that still contain data that would be needed by following deduplication requests which results in additional random access to disks and throughput degradation. However, in practice we have not seen the problem with drop of write performance of duplicated blocks of regular backups.

The decrease of read performance is only a potential problem in most cases. It is not noticeable for small and middle range systems. The problem affects big systems but only for uncommon usage pattern i.e. when all users read all their data at the same time. Such pattern would occur when all users lost their data at the same time. It is quite unlikely, because the data comes from separate sources. In case of big system only limited number of users read the data at the same time in a typical case. Read performance of such usage pattern is not degraded comparing to smaller systems, because problems described above are mitigated by higher number of disks serving read requests in parallel.

Some of the problems with read operation will be improved by increasing efficiency of RAM caches. It will be done by keeping only the needed data, in contrast to the current version which keeps in RAM all data read from disk.

Total Performance Limited by Slowest Physical Node

On very high level HYDRAstor data organization is based on two foundations - distributed hash table and keys of blocks computed based on their content. Since keys are randomly and evenly distributed over hash key space it is very easy to evenly distribute load over the whole system. This is because each component statistically receives the same number of fragments to store. However, there is the other side of the coin. Since all components receive the same number of fragments the system can only be as fast as the slowest component. Since component's speed depends on the storage node it is hosted by, the system is as fast as the slowest storage node. The balancing algorithm takes into account each storage node's performance and assigns components in such a way that each storage node load is more or less the same. Problems may appear when a physical node fails in such a way that it does not stop, but its performance is decreased. The current version of HYDRAstor does not handle such problems automatically. However, the system detects disks with degraded performance which allows administrator to take manual actions.

Fast Background Operations

Data organized as chains of SCCs provides support for fast background operations. It is seen on figure 5.16. Data recovery operation reaches 160

MB/s/physical node which is slightly higher then 150 MB/s/physical node reached by write operation.

Fast Data Location

Proposed data organization supports fast data location based on data content or id returned by previous write operation. DHT with hash of the data content allows for fast locating of component handling the hash, and therefore the physical node which hosts the data. Separation of metadata (SCC indexes) from data (SCCs) allows for fast location of data on disk, because stream access and prefetched SCC indexes result in most of the location queries being resolved in RAM cache without disk access.

In a dynamically changing system where data is transferred from one node to another, the component may not have its SCC index chain locally complete. In such a case one of its peers with complete SCC index chain is used to locate the SCC storing the data. In an extreme case after massive failures when non of the peers has complete SCC index the location may be found by checking partial SCC index chains stored by physical nodes previously hosting the component (such physical nodes send to the component reports about its orphaned chains, thus the component has information about the part of the chains hosted by other nodes). After the proper SCC index is found the component knows the SCC which stores the data. If the SCC is not hosted locally then it is located on one of the physical nodes previously hosting the component or other peers can be used to reconstruct the data.

Disk Data Locality

Proposed data organization takes into account locality of data on disks. Streams of users data fragments are stored in the tail of SCC chain (i.e. the latest open SCC file). This allows the write operation to store data on consecutive sectors of disks and thus provides high throughput. Then, after an SCC is closed for writing, the stream sorting background task consolidates data of different users streams in such a way that the fragments of the same stream are located next to each other on the disk. Later on, it allows the read operation to prefetch users data into RAM to avoid disk seek penalties.

Optimal Initial Data Placement

Initial data placement computed by global leader is the optimal and final one in terms of entropy. Due to pinning components to physical nodes any temporal disturbances caused by failures or maintenance works results in limited amount of data changing their locations (i.e. limited number of components being transferred). After the disturbances disappear data is transferred to its original locations.

Performance under Failure

If under failure components can be evenly reconstructed then the performance drop is almost proportional to the percentage of failed nodes. It can be seen on figure 5.16. Write throughput under 4 healthy physical nodes is 600 MB/s. After one node is failed and all recovery background tasks are finished, the throughput stabilizes at 430 MB/s.

If components cannot be evenly reconstructed on the remaining physical nodes then there would be an additional performance drop proportional to relative inequalities in number of components hosted by physical nodes.

5.5 Trade-offs resolution in HYDRAstor

5.5.1 Efficient Storage Usage vs. Fault Tolerance

As previously described there are two issues related to efficient storage usage vs. fault tolerance trade-off:

- fault tolerance requiring redundant data that worsens the efficiency of storage usage, and

- fault tolerance may impose placement of data that is not optimal with regard to storage utilization.

Both trade-offs are not resolvable. To support fault tolerance there needs to be some kind of redundant data to handle disk failures. Redundancy cannot be removed totally, but the extent of its negative impact on storage utilization depends on data placement policy. By using erasure codes HYDRAstor provides much better storage utilization than other commercial storage systems that are based on RAID technology (see section 5.4.2 on page 87 for more details). Additionally HYDRAstor gives users freedom in deciding how much additional storage they want to sacrifice for increased data resiliency on each block level.

Non optimal storage utilization imposed by data placement providing a given level of resiliency cannot be resolved either. The most simple case of such an unresolvable trade-off is a system that is built on n non-equal capacity physical nodes and is to guarantee no data loss if $n-1$ physical nodes fails. To satisfy such a requirement each data block needs $n-1$ its replicas. Original block and replicas have to be spread over all physical nodes. Such data placement makes the system full when the first physical node is full. This means that physical nodes with a higher than minimal capacity cannot be fully utilized. Storage systems can only strive to optimize capacity utilization within the boundaries of data resiliency.

Due to data organization based on supernodes and components HYDRAstor has many points of freedom in assigning data to physical nodes, much more than commonly used RAID solutions. However, HYDRAstor does not find optimal solutions for all cases. One such case is the above example with SNC=n=12. Another case is the example above with n set to 7 but with the same capacity physical nodes. Then HYDRAstor with supernode cardinality set to 12 could only keep original data in one component and its replicas in the 11 remaining components. Components would have to be spread over n physical nodes in such a way that each physical node hosted at least one component. Such data placement results in inefficient storage usage, because 5 replicas do not add any additional resiliency (i.e. there are 5 physical nodes hosting two replicas). To solve such problems within its current architecture HYDRAstor would have to either support supernodes with different supernode cardinality depending on requested resiliency or some of the components for some data classes would be allowed to be empty i.e. do not store fragments/replicas for chosen user data blocks. This would greatly complicate the system and we do not plan to introduce such changes. In any case, the problem with inefficient storage utilization fades away as the system size increases. Systems built upon more than supernode cardinality physical nodes have more freedom in distributing components within given resiliency levels in such a way that capacity is better utilized - bigger physical nodes can host component with many different supernodes. This improves physical nodes utilization but does not decrease resiliency.

5.5.2 Efficient Storage Usage vs. Availability

To provide highest availability each piece of data should be stored on each disk of each physical node. To reach the maximum storage usage efficiency each piece of data should only be stored once. HYDRAstor addresses this trade-off in several ways. First of all, it uses erasure codes that are at once storage efficient and provide both high resiliency and availability. Because only fraction of data fragments are required to reconstruct the data block, data can be read even if some physical node are not reachable. Secondly, quick components recovery after failures allows the system to accept writes almost without any noticeable interruption. Thirdly, after physical nodes failure, automatic data recovery of missing fragments on other physical nodes rebuilds data resiliency to the required level. This minimizes the chances that any following failures would result in data unavailability. Fourthly, the user decides how much resiliency given block should have. This allows him to flexibly define the trade-off resolution.

Efficient Storage Usage vs. Reliable Detection of Duplicated Blocks on Write

The architecture of supernodes provides very high redundancy of information about previously stored blocks even if the block is not readable. This is because each peer keeps the block's fragment together with its hash key. Thus only one component with complete metadata information about its fragments (i.e. complete SCC index chain) is enough to verify that a block to be written is not a duplicate. This means that the availability of information that a new block is not a duplicate is very high. The availability of information that a block is a duplicate is lower and depends on redundancy of the previously stored block. To verify that a block is a duplicate at least one component has to report that it has its metadata with block's hash key and the block has to be readable. A block is readable if enough peers have readable fragments to be able to reconstruct the block. Block readability depends on the level of its redundancy. If it is low and many physical nodes are not reachable the block may not be readable. In such a case the write operation stores the block as a non-duplicate by default. Users may change system wide setting so as not to store block if they are discovered as duplicates, but are not readable.

In summary, HYDRAstor is highly available when it comes to replying to queries for blocks that are not duplicates. In the case of duplicates the availability of such a reply depends on redundancy of the *base block* i.e.

an older block that is the base for deduplication. If the base block is not readable HYDRAstor stores the new block again by default. This decision comes from the fact that in typical backup polices users remove backups when they are old enough. This means that inefficiency related to storing a new block is temporary, because sooner or later the old unreadable block will be deleted and the new one will be the base for further duplicates. If the base block became readable (i.e. physical nodes that were storing its fragments were connected back to the system) it would be the base for further duplicates and the second block would be removed after its backup was removed. In any case one of the duplicates would be removed from the system if users applied typical backup polices.

5.5.3 Efficient Storage Usage vs. Performance

Providing Resiliency

HYDRAstor uses erasure coding instead of replicas. This is due to the fact that replicas are not acceptable due to their inefficient storage usage - the resulting solutions are very costly. Erasure coding is worse than replicas in terms of CPU usage and disk access pattern for read operations. Due to the constant increase in commodity CPUs computation of erasure codes becomes a minor problem. The problem with performance of read operations is more severe. It is associated with the fact that, to read a block several fragments have to be read from several disks. This is in contrast to replicas where only one fragment (copy) would have to be read. This is one of the reasons for read performance issues described in section 5.4.6 on page 103. Compared to replicas erasure codes increase the number of concurrently read streams on disks which increases disk random reads. This is the cost we pay for higher storage usage efficiency.

Heterogeneous, Non-proportional Nodes

HYDRAstor address the problem of the disparities between the CPU and the capacity among physical nodes by introducing two types of components and distributing them according to physical nodes resources. There are two types of components: regular components and write initiator components. Regular components are responsible for storing block fragments on disks. They do not generate any CPU intensive work. Write initiator components are responsible for data block compression and computing erasure code fragments. They generate much more CPU work than regular components.

Due to these two types of components all components are entities that utilize capacity and write initiators are entities that generate CPU work. The balancing algorithm spreads components in such a way that physical nodes have a similar fraction of their capacity utilized. Within this distribution the balancing algorithm reassigns components in such a way that CPU richer physical nodes receives more write initiators.

Deduplication vs. Performance

The impact of deduplication on performance depends on the type of operation. It turned out that HYDRAstor data organization is very efficient in providing highly performing write operation both non-duplicate and duplicated. The comparison of HYDRAstor to NAS systems in table 5.4 on page 102 shows that data organization that supports deduplication has non-duplicated block write throughput comparable to regular NAS systems. What is more, due to efficient support for deduplication, verification if a block is a duplicate is done without disk access in most cases. This results in the deduplication write being twice as fast as regular writes. This means that for a common write stream where 80-90% of blocks are duplicates write throughput is 80-90% faster than systems without deduplication. From this we can see that the trade-off between storage utilization and write performance was successfully resolved.

As described in section 5.4.6 on page 103 deduplication is one of the reasons for performance problems of read operation. A stream of written data is deduplicated against previously stored streams. Each new stream representing the same of set of users' data is slightly modified compared to the previous one. In the worst case scenario it may happen that the new stream is deduplicated against parts of all previously stored streams. This degrades the locality of such a stream on the disk. Later, when the stream is to be read, its performance suffers, because small parts of all previously stored streams need to be read instead of reading one continuous stream. This is a general problem which affects all systems with deduplication. We are not aware of any deduplication system that resolves this problem without sacrificing storage efficiency.

Efficient Storage Usage vs. Fast Data Location

The HYDRAstor system architecture based on DHT provides very fast data location. Data blocks address space i.e. hash key space is divided into non-overlapping subspaces, each handled by one supernode and its components.

Each physical node hosts tens of the components. Components create an overlay network that can provide very short routing paths. After jump tables are implemented these paths can be as short as one hop. So far the jump tables have not been required because HYDRAstor clients like HFS cache locations of all components in order to access them directly without any intermediate nodes. For big system built upon 100 physical nodes there is around 3000 components. Keeping in RAM their locations is easily manageable by HFS. However, such a coarse division of key space comes with a price of non-optimal storage utilization. This occurs when nodes' capacity divided by the number of components hosted by them is not equal on all nodes. Because components are the same size it results in system that cannot accept new writes because some physical nodes are full even though there are other physical nodes which still having free space. For a system with a default size of maximum component size (0,5 TB) it leads to up to 3.5% of storage not being utilized as seen on figure 5.15 on page 81. Utilization could be improved by increasing the number of components in the system by decreasing their maximum size. However, this would increase the number of streams of fragments and result in worse read performance.

5.5.4 Fault Tolerance and Availability vs Scalability

The basic entity that provides both fault tolerance and availability is a supernode. Supernode resiliency is defined by supernode cardinality that is a system constant set during system initialization. As the system increases so the number of supernodes also increases. However, since resiliency and availability provided by each supernode stays at the same level total system availability and resiliency decreases. To properly handle the trade-off between fault tolerance and availability vs scalability the system would have to dynamically increase supernode cardinality together with system growth. We decided not to implement such feature so far because on the one hand it would make the system implementation more complex and on the other hand supernode cardinality can be set during system initialization to a high enough level if the system is planned to be very big. However, we might add dynamically growing supernode cardinality in the future.

5.5.5 Fault Tolerance vs. Performance

The tension between performance and the potential negative impact of redundant data that has to be stored on each write operation turned out to be nonexistent in practice. Even though each block is divided into smaller fragments that together with redundant fragments are passed over the network and stored on supernode cardinality disks, the write operation has high throughput. This is because fragments coming from different blocks are merged into one stream of data stored on disk within one SCC (i.e. one file) which allows it to reach a high throughput. Additionally, since the number of minimal fragments required to restore a block is lower than supernode cardinality, the write operation does not have to store all fragments. Later the missing fragments can be reconstructed. Such schema can be used to provide high write throughput in case of a slow or temporarily unavailable physical nodes. However, current version of HYDRAstor required all supernode cardinality of fragments to be stored by write operation.

Although HYDRAstor experiences problems with big systems aggregated read operation performance, they are related more to deduplication that scatters data user streams and with the fact that all users streams are interwoven on disks than to the data redundancy. Data redundancy increases the number of concurrent streams by a constant factor in contrast to deduplication and the number of concurrent users. Deduplication and high number of concurrent users may increase number of streams in a unlimited manner.

In summary, the most critical factor for storage systems is the short time needed for storing backup because the time window when such backup can be done is very short. HYDRAstor fulfills this requirement very well. The aggregated read operation is less critical, because the chances that all users of a big system will read backups at the same time are low. In a typical case one or several users lose primary system data and only those users read the backup. Performance of read operation for several concurrent users does not suffer much.

With respect to the tension between types of redundancy i.e. replication vs. erasure coding, replication would provide better results for read operations. This is because to read a block only one fragment (copy) would have to be read. If there were many users reading data concurrently they would be divided into supernode cardinality groups. Each group would be handled by a separate peer of each supernode. This would result in a lower number of streams read by disks and thus it would partially solve problems with read performance. However, storing replicas instead of coded blocks is so much more inefficient storage-wise.

With respect to performance being negatively impacted by operations that rebuild data after physical node failures HYDRAstor data organization address it in two ways. First of all, after failure components that are recovered are distributed over "living" physical nodes in such a way that evenly distributes the load generated by rebuilding data. Thus data rebuilding after physical node failure does not generate data reconstruction hot spots. However, theoretically there might be performance issues on physical nodes serving fragments (i.e. source nodes). Data for reconstruction is read from peers of recovered components. If supernodes are clustered into separated groups of physical nodes (i.e. in a case when supernodes are assigned to racks) only one such group serves the fragments required to rebuild data and this may cause a hot spot. Albeit the reconstruction process does not read the fragments separately. Instead whole SCCs are read which is very efficient in terms of disk access. In practice source physical nodes are not performance bottlenecks. Second, HYDRAstor data organization allows quick verification of the resiliency of data after a failure. This allows the system to prioritize the background rebuilding processes according to loss in resiliency. If this loss is minor or moderate background processes are slowed down while users regular operations like read and write are executed. Since storage system typically has short intensive periods of regular operations and long time windows when no user actions are executed, non-critical background operations are executed during that idle time and do not interfere with users regular operations.

However, there are two cases when the current version of HYDRAstor creates hot spots. One is when one or several disks of a physical node fail. The other case is when a failed physical node is replaced by a new one before its data is reconstructed on other physical nodes. Both these scenarios require the reconstruction of data fragments which, due to resiliency loss, may be a high priority task. The fragments are only written to one physical node. This physical node becomes a hot spot and a bottleneck. However, the proposed data organization is flexible enough to support a solution to this problem. Data fragments could first be reconstructed on physical nodes where write initiators are hosted to reach required resiliency level. Later on, when the system is idle, fragments would be transferred to the final location as a low priority tasks not overloading the target physical node. Since write initiators are distributed over multiple physical nodes and they should not become hot spots. We plan to implement this improvement.

5.5.6 Scalability vs. Performance

The probability of any failures increases with growth in system size. Any such failures may negatively impact performance because it may stop or slow down users' operations. On the one hand HYDRAstor is sensitive to such errors because the load generated by users is evenly distributed over all supernodes and their components. Total throughput slowdown is proportional to the slowdown of any supernode. On the other hand each supernode is to some extent an autonomous entity that recovers from failures and has redundancy that allows it to handle users' requests even if there are failures. In case of deletion (i.e. marking blocks to be reclaimed) for default configuration only seven out of twelve components of each supernode have to be healthy (i.e. have full chain stored locally) to successfully execute the operation. In the case of write and read operations, as long as a supernode has "living" write initiators, a read operation can be executed by reading fragments from remaining peers and writing *write threshold* fragments instead of supernode fragments (writing write threshold feature is not implemented yet, but it is designed and relatively easy to add). A more problematic case is when the write initiator is hosted by a failed node. In such a case read and writes are stopped until the component has recovered which takes up to thirty seconds. The problems with dropping performance can be mitigated by an additional layer between the user and the system. For example HFS buffers users' requests to hide the intermittent performance drops.

There is another problem with hardware failures that manifest in decreased performance. In the case of DSS, the most important are problems with disks, that due to failures have degraded performance, but do not fail completely. In the case of HYDRAstor these types of failures slow down total throughput at the same level as the physical node with the failed disk (HYDRAstor is as fast as the slowest physical node). These types of failures are more likely to happen in bigger systems which have many disks. To address this problem, HYDRAstor discovers such disks and the administrator is notified about the problem. Then the administrator decides if such disks should be replaced.

Scalability vs. Fast Data Location

HYDRAstor architecture based on DHT is highly scalable. Due to the distribution of mapping from keys to blocks among all the physical nodes and very low disruption after nodes' arrival or departure DHTs scale to very large number of physical nodes. DHTs provide fast data location. FPN with the to-be-implemented jump tables would have a very low constant path length with little increase in node state.

Scalability vs. Fast Background Operations

Almost all background operations are processed within supernodes that coordinate and manage them. This makes background operations resistant to potential scalability issues because background tasks have high physical nodes locality i.e. they depend on a limited number of physical nodes. The performance problem of one physical node only impacts supernodes that have at least one component hosted by that physical node. For default configuration clustering supernodes along racks this is not more than 48 supernodes and this number does not depend on system size. Although there is one background operation that depends on all supernodes. This is the deletion upgrade counters process that is finished after all supernodes finish their work. The performance of this process may suffer if even one supernode has degraded performance, which is more likely in a bigger system.

Scalability vs. Disk Data Locality

HYDRAstor spreads each data stream equally among all supernodes, and in consequence, among all physical nodes. Main advantage of such an organization is load balancing among all physical nodes. However, it may be a drawback for very big systems in case when all stream are read at the same time. As an increasing number of data streams is associated with an increase in the number of physical nodes, each physical node keeps growing a number of substreams of smaller length. This finally results in a situation when each physical node stores so many short substreams that reading them all at the same time, regardless of organization of data on disks, results in decrease of disk read operations performance caused by decrease in disk data locality.

However, note that current high level data organization provides high performance of read operation when not all, but limited number of streams are read at the same time. In such a case data is read from a smaller number of disk sectors, so there is fewer random reads. Additionally, since there is fewer streams, RAM cache keeps more prefetched data of each stream which results in further decrease in disk I/O. And because data is spread over all physical nodes, system harnesses all disks to handle single or several stream reads which results in high performance.

Since reading limited number of streams is more common than reading all of them at once, current high level data organization is good enough in practice.

5.5.7 Intra Requirements Trade-offs Resolution

Fast Data Recovery vs. Resiliency For Concurrent Failures

HYDRAstor deals with this trade-off in several ways. First of all users can choose if they prefer clustered data placement by providing information to the system about physical nodes division into machine racks. If that information is provided supernodes are clustered along racks. If the information is not provided supernodes are more randomly distributed over all physical nodes. To guarantee the highest level of declustering the entropy function could be easily changed to prefer that distribution. Having said this, typical network topology has better network throughput between physical nodes in the same rack than between physical nodes in different racks. Thus it makes clustering supernodes along racks more a preferable distribution. This is because components belonging to the same supernode generate more network traffic than components belonging to different supernodes. The entropy changes that would prefer random supernodes distribution where not required by users and are not implemented. Secondly, the data recovery algorithm is very similar to the one described in [9]. The algorithm prioritizes the data to be recovered in such a way that data with a critical loss in resiliency is reconstructed first and its tasks, (in the case where a reliable policy is set for data reconstruction), have higher priority than write and read operations. This allows the system to quickly reach a safer level of resiliency. It mitigates the problem with prolonged data recovery in clustered systems. Thirdly, due to asymmetry in resource consumption between physical nodes that provide fragments for reconstruction and physical nodes that do actual reconstruction of missing fragments the trade-off to some extent melts away in favor of clustered distribution. Physical nodes are heavily loaded by processes that reconstruct data, because they recompute erasure coded missing fragments, which is a CPU intensive process. Physical nodes that provide fragments that are used to reconstruct missing ones are very lightly loaded because they only read the whole SCCs from disks (it is fast stream access and does not result in random I/O on disks because SCC is tens of megabytes file) and pass them through the network. This means that in the erasure coded data reconstruction process the source physical node can handle many target physical nodes for the same load on source and target. Recovered components are distributed evenly over all physical nodes (recovered components are not affiliated with racks). Such distribution of recovered components together with asymmetry in resource consumption between target and source physical nodes results in fast erasure coded data recovery performance, even when supernodes are clustered along racks to improve resiliency for concurrent failures.

Local Balancing vs. Global Balancing

HYDRAstor addresses this trade-off by the hybrid balancing algorithm that joins both local and global balancing. This results in system that has the advantages of both types of balancing, but does not have their disadvantages. The method of pinning components according to globally computed components distribution guarantees that component distribution is optimal and stable as long as the system has no failures. Since global balancing is computed by one central entity which gathers information from all physical nodes it can take many balancing criteria into account. Pinning makes system reach optimal, initial component distribution after the distribution is temporarily disturbed by planned or unplanned events like administration tasks (i.e. node retirement) or failures. After such events are gone (i.e. node is unretired or started after a failure) components are transferred back to their original locations. It both stabilizes the system and minimizes the amount of data being transferred between nodes after failures are repaired because components are transferred back to where most of their data is stored. Local balancing kicks in only in the case of physical node or disk failures. It makes decisions about the placement of components which have pinning target physical node broken, thus they operate only on very limited number of components and introduce little imbalance into the system. Additionally, local balancing executed by a physical node makes decisions based only on information received from neighboring physical nodes, which makes it fast and quickly adaptable to changes in the system caused by failures. Furthermore, even though the global distribution is computed by one chosen component (global leader) balancing algorithm is failure resistant. If that component dies while computing the distribution, its newer recovered incarnation executes components global distribution again. Unless there are continuous failures the optimal distribution is computed and delivered to all components.

Inline vs Offline Deduplication

As described in section 4.1.7 on page 34, offline deduplication may have only one advantage over inline deduplication - performance. Despite this advantage it has many disadvantages: - higher disk capacity requirements, more complicated administration, worse support for disaster recovery, and more complicated system design and implementation. Taking this into account HYDRAstor was designed as an inline deduplication system. This decision turned out to be the right one. HYDRAstor is a highly complex system. An additional subsystem temporarily storing users data and providing at least the same failure resiliency as the long term storage could make the implementation unfeasible. However, more importantly HYDRAstor performance results show that offline deduplication would not be faster and could in fact be slower. To support an offline deduplication the system would have to temporarily store data in a fashion very similar to NAS storage. However, the performance results presented in table 5.4 on page 102 shows that HYDRAstor stores streams of non duplicated data at least as fast as common NAS storage systems. This means that storing temporarily data for later offline deduplication processing would not give better performance. What is more, in the case of deduplication writes, HYDRAstor is around twice as fast as NAS storage which means that in case of deduplicated data offline deduplication would be even slower than the current solution.

Chapter 6

Related Work

A review of related work can be organized along many dimensions, but we have decided to concentrate on the dimension of scalability. This is because scalability is the most important feature of DSS systems; if scalability is not important, a DSS can be replaced, usually by a single node, which today can have a significant storage capacity and deliver many other DSS features.

With respect to scalability, existing disk-based solutions for backup and archiving can be divided into the following groups:

- Non-scalable, single node systems.
- Limited scalability clustered systems.
- Highly-scalable distributed systems.

Below we consider both commercial and research systems and concentrate on how data organization of these systems impacts their functionality.

Non-scalable systems In this group there are single-node systems targeting backup and archiving. They focus on efficient storage usage by supporting data deduplication. Some of them also focus on high performance. Flexibility of data resiliency is commonly limited to the RAID method of organization.

Venti [69] is a single server research solution targeting archiving. It cuts streams of data into variable size data blocks of maximum size 52 KB. Blocks are kept in an append-only log on a RAID array of disk drives (Venti does not support deletion). This log is similar to HYDRAstor's chain of SCCs.

The log is divided into sections called *arenas*. Arenas are similar to HYDRAstor' SCCs. For each block the SHA-1 key is computed to find duplicates. A separate index structure is used to locate a block based on a key. This index is stored on the disk drive. Venti keeps an RAM LRU cache for both the log and the index. The log cache with prefetching improves performance by limiting access to the disk log. However, the cache does not work for the index, because keys are random which results in no spatial locality in the index access. What is more, there is very little temporal locality in backup data stream. This causes LRU cache policy to be very inefficient. It results in random disk access and poor performance. The Foundation [74] is a research solution that improves Venti's design by storing a separate *summary* file for each arena. This file has list all of (hash, offset) pairs of the arena. The Foundation prefetches summary files and take advantage of the spatial locality inherent in sequential reads. It improves hash cache efficiency and increases the throughput of read and deduplicated write operations. HYDRAstor's SCC indexes are similar to summary files and are also prefetched to the RAM cache to improve performance. However, since Foundation is designed for personal use, it does solve the problem of multiple streams written concurrently but later read separately.

IBM ProtectTIER [59] a single node commercial solution with a very high capacity of 1 PB, but relatively slow throughput of 1000 MB/s. This system has unique data organization. Unlike HYDRAstor and other commercial solutions it does not use hash of data blocks to find duplicates. Instead it uses HyperFactor technology [58]. It retains the RAM index of summary of stored blocks. When new data is received, the system checks for data similarities in the index. When the matches are found, the similar blocks are read from the disk. Binary differential is performed and only unique data is stored. This is a different approach the one we used in HYDRAstor. HYDRAstor deduplicates data against exactly the same blocks and does not store any binary differences between data blocks. IBM claims that it uses differential solution to avoid possible hash collisions in CAS systems.

DataDomain [99] is another commercial solution. Since its architecture is described in detail we can compare its data organization with that of HYDRAstor. Originally DataDomain was a single node solution. It implements a file system interface. The file system divides streams of data into chunks of variable size, on average 8 KB vs. 64 KB in HYDRAstor. Smaller chunks provide a better ratio of data deduplication. Compressed chunks are combined into 1000-element groups. Groups are stored in 4 MB containers. Containers are very similar to HYDRAstor SCCs. However, since DataDomain is a single node solution containers store chunks of user data blocks vs. SCCs that store erasure code fragments. DataDomain uses a raw disk for storing containers vs. SCCs that are ext3 file system files. As with HYDRAstor DataDomain keeps containers separated form their metadata to allow metadata prefetch for better throughput. DataDomain creates a separate sequence of containers for each data stream to increase effectiveness of data prefetching. HYDRAstor achieves a similar result by sorting fragments stored in SCCs by their stream ids. HYDRAstor cannot use separate SCC chain for each data stream because in big systems it would result in very large number of containers (i.e. files) written concurrently which would have a negative impact on write throughput.

Limited scalability systems In this group there are multi-node storage systems targeting backup and archival. The data organization of these systems can potentially provide higher capacity and performance than systems of the previous group. However, scalability is still limited.

Pergamum [86] is an archival system built with disk-based storage nodes that store data reliably and in an energy-efficiently manner. The aim is of this research project was to replace tapes with long-term disk-based archival storage. Each node is equipped with flash-memory that keeps metadata like index of blocks stored in disks, data signatures and information about pending writes. It allows as many as 95 percent of the disks to be spun down while still providing reasonable performance and high reliability. Data organization based on flash memory that would keep content of existing SCC index cache and other metadata is an interesting option that we intend to investigate to increase performance of HYDRAstor. To provide reliability segments on disks are grouped into *regions*. Regions from multiple disk together with redundant erasure coded data form a *redundancy group*. Redundancy group are reliability entities similar to HYDRAstor's supernodes. However, Pergamum does not provide deduplication.

Designers of DataDomain are trying to add a scalability feature to their system [24]. They created commercial Data Domain Global Deduplication Array (DDG) [4] with this goal in mind. The goal here is to preserve existing DataDomain data organization and add additional data placement policy that would provide scalability. DDG Client machine cuts data stream into regular chunks and combines them together into a *super chunk* of 1 MB size. Next, the client machine computes the *feature* that is the SHA-1 hash of the first 64 bytes of the super-chunk. The value of the feature statically determines the node that stores all chunks of the given super chunk. This is a very simple solution, but it has several disadvantages: the failure of any node results in data loss; according to simulation results published in [4] systems with more than 2 nodes have uneven capacity utilization which requires ongoing data migration between the nodes; chunks are deduplicated only within single nodes, thus many chunks may not be deduplicated, because they are stored according to a feature computed based on one chunk in the super chunk, not based on the locations of chunks stored earlier. Commercial DataDomain solution scales only up to 2 nodes. HYDRAstor makes use of DHT extended with supernodes to implement data placement policy that avoids the disadvantages DDG experiences.

EMC Centera [3] if one of the first (if not the first) commercial systems to use content-addressed storage for archiving. To provide resiliency it's architecture is based on redundant arrays of independent nodes (RAIN) [13], which is a kind of RAID parity protection applied to system nodes. Additionally, two copies of each data object are stored on disks on separate nodes. The system consists of storage nodes and access nodes handling read, write and delete operations. According to available information [37] chunks have constant and large 100 MB size. This approach results in lower deduplication than the much smaller variable block size used in HYDRAstor. The architecture of EMC Centera is not publicly available. However, according to information gathered by intra-box analysis of EMC Centera in [37] we conclude that the hash table is replicated over all access nodes. Write or delete operations result in broadcast messages that update that the hash table on each access node. This solution limits scalability. Since it is an archival solution its performance is not the main objective. Centera's designers focused more on configurable retention settings ensuring information is not erased prior to the expiration of its defined retention period (non-eraseability feature).

HYDRAstor's CAS-based data organization allows ANs to deduplicate data before sending it to SNs. There are other commercial solutions that deduplicate data at the source i.e. on client machines. Such solutions decreases network traffic between client machines and a global repository. Clients filter out redundant data before sending backup data over networks, making it possible to protect systems even if they are over congested LANs or WANs. A global repository provides deduplication for all clients. Such a solution decreases network traffic for backup, but may suffer when all backed up data has to be transferred for recovery. Thus remote offices can configure their local repository. Clients backup data to the local repository which in turn replicates data back to the central data repository. Such a feature is not supported by HYDRAstor's data organization. A local repository provides fast recovery. EMC Avamar [2], Symantec PureDisk [22, 87], Symantec NetBackup 5020 [88] and ExaGrid [1] are examples of this type of solution. Avamar uses 24 KB variable size blocks. The hash of a block determines which node and disk stores the block in the global repository. This is similar to HYDRAstor where the block hash determines block location based on component location that handles that hash. However, Avamar scalability of global repository is limited to 16-node configurations. PureDisk together with Symantec backup media servers can be used to create a global repository providing deduplication. From the available information it is not clear what is the exact data organization of such configuration. We suspect that media servers balance load between PureDisks and divide hash key space between the PureDisks or statically assigns backup streams to PureDisks. The latter case would mean that the deduplication is not global among all PureDisks. However, even if the deduplication is global such repository has limited scalability - it can scale up to 96 TB. Symantec NetBackup 5020 provides better scalability - it scales up to 192 TB. ExaGrid is an offline deduplication grid solution providing incremental scalability, but can only configure up to ten servers into a single grid configuration of up to 320 TB raw capacity with maximum full backup size limited to 130 TB. To support fast data restore it maintains full, non-deduplicated copies of the most recent backups on the client site and only sends delta changes to the offsite global repository during post-process deduplication. In addition to using fast LAN instead of slower WAN network, such a solution ensures fast recovery, because data does not have to be reassembled from many deduplicated fragments. The solution addresses problems related HYDRAstor's data organization which has limited performance of read operations. However, this fast recovery is achieved at the expense of worse efficiency of storage usage - more or less half of the capacity has to keep non-deduplicated latest backup on the client site and the other half deduplicated backups in the global repository.

RepStore [97] units LAN tailored DHT (similar to FPN) with so called *smart storage bricks* to archive low storage cost and high performance balance. RepStore's data organization automatically employs replication for active write-intensive data and erasure-coding for archive data. HYDRAstor's data organization does not support such automatic and dynamic change of redundancy type for better performance. RepStore uses content-based addressing, but does not provide deduplication. Since RepStore's data organization is based on DHT it is the only system in this group that is highly scalable.

FAB [31] is a another system that makes use of smart storage bricks to create a distributed disk array. To provide resiliency FAB uses a *Seggroups*. Seggroup defines the distribution of user blocks over smart bricks. The location of bricks hosting a seggroup is changed by an agreement protocol, a technique similar to the one used in HYDRAstor. Seggroups are data organization similar to HYDRAstor's supernodes. As opposed to HYDRAstor, seggroups' metadata has to be replicated by all bricks in the system to support read and write operations. This may limit FAB's scalability.

RADOS [95] harnesses smart storage intelligence and autonomy to distribute complexity surrounding redundant storage, failure detection and failure recovery. Data objects are organized within *placement groups*, a logical collection of objects that are replicated by the same set of devices. Thus they resemble HYDRAstor's supernodes. The functionality of devices acting within the placement group are similar to the functionality of HYDRAstor's components.

Ursa Minor [6] prototype extends the idea of the flexible redundancy policy implemented by RepStore to versatile cluster-based storage. Ursa Minor data is organized in such a way that for each stored block users can choose a redundancy policy (replication or erasure coding), storage-node fault type (fail-stop or Byzantine), number of storage-node faults to tolerate, data location and timing model (synchronous or asynchronous). This flexibility exceeds HYDRAstor's, RepStore's and FAB's data organization. However, it comes with disadvantage of data managed by a single object manager that limits system scalability.

RepStore, FAB, RADOS and Ursa Minor are all research projects.

FalconStor [30] is the market-leading Virtual Tape Library(VTL) solution provider. FalconStor has two pools of storage - a VTL pool (RAID-5 with hot spare) managed by VTL nodes and a *Single Instance Repository* (SIR) pool (RAID-6) managed by SIR nodes. The VTL pool can scale up to 8 VTL nodes, the SIR pool can scale up to 4 SIR nodes. Client data is sent to the VTL pool in a backup tape format that is stored in disks. The incoming data is stored on disk in a self-describing format that is optimized for sequential reads and writes. To avoid data defragmentation on disks and to reach high performance FalconStor does not use a file system. Instead, FalconStor allocates raw, consecutive disk sectors to each virtual tape at the full size of the emulated tape. Data deduplication is done later on by a background process that scans the VTL data pool and determines whether data is unique or has already been copied to the SIR pool. The process then passes only single instances of unique data to the SIR pool. The original data from the virtual tape is replaced by direct references of where the data is stored on the SIR node [90]. This allows data to be directly read from appropriate locations. However, such solution suggests that data within a SIR pool cannot change location and the system cannot quickly balance itself after new nodes are added. FalconStor tries to decipher virtual tape format prior to the deduplication process to increase the deduplication ratio by identifying data from different parts of the backup stream multiplexed by a backup application. FalconStor uses SHA-1 over various size chunks of data. The hash table is distributed amongst all SIR nodes. SIR nodes keep counters of instances of hashes and deletes data when counters drop to zero. FalconStor solves the problem of lack of locality of the hash table by storing the entire hash table in RAM. However, this brute force solution comes with cost - SIR nodes themselves (not including RAM of VTL nodes) require five times more RAM for each TB of storage than HYDRAstor does.

Highly-scalable systems In this group there are archival systems that store data in wide area networks built with untrusted nodes. Often the nodes are commodity PCs with with unused, free disk capacity. Such systems deal with failures (quite often byzantine ones), malicious users, and nodes continually enter and exit the network, often without warning. Some of them address anonymity of users who publish data. Such systems provide very high scalability, however due to the abundance of disks the storage capacity is consumed inefficiently - these systems often do not provide deduplication, store data in many replicas and may keep stored data forever (no delete operation). These systems use some kind of distributed hash tables, commonly Chord [85], and are self-reorganizing and self-repairing. All the systems in this group are results of research projects and are not commercial solutions.

OceanStore [75] is a globally scalable persistent storage utility with a file system interface and strong data consistency. Users can store and update data objects. Objects are kept forever. As with HYDRAstor blocks, OceanStore blocks are immutable. Update operations result in the creation of new versions of the objects - there is no in-place update. OceanStore employs two-tier based data organization. The first is the *inner-ring* hosted by servers maintained by trusted providers. The inner-ring is responsible for maintenance of primary replica of objects and provides consistent and autonomic operations on the replicas. All operations are executed under the Byzantine agreement protocol. HYDRAstor supernodes' functionality and organization is similar to the inner-ring. The second tier, the archival-tier, is responsible for replication by the creation of additional replicas and archiving by the creation of an erasure coding version of the primary replica. The second tier utilizes desktop computers as nodes that may not be powerful, well connected and trusted. OceanStore utilizes Tapestry [98], a distributed hash table, to locate objects in the system. Objects are divided into blocks. OceanStore identifies blocks by the hash of their content, thus with help of Tapestry the same block can be part of many data objects. Thus OceanStore finds duplicates, but its positive impact on efficient storage usage is limited by keeping many replicas and erasure coded versions of the deduplicated blocks.

PAST [25] resembles a simpler version of OceanStore. It provides a simple storage abstraction for persistent, immutable files. It makes use of the Pastry [76] distributed hash table as location and routing schema. Stored files are identified by an unique id returned to the users. There is no filesystem interface. To provide resiliency many replicas of each file are created. Whole files are stored.

CFS [21] is a better version of PAST. It stores blocks, rather the whole files and balances the nodes by spreading blocks evenly over the nodes. It uses Chord [85] to locate data. IVY [55] is a peer-to-peer read-write file system built on top the CFS. Unlike OceanStore, it does not provide strong data consistency.

Glacier [39] uses desktop PCs that have abundant but unreliable storage space to build highly durable, decentralized storage. It assumes large scale correlated and unpredictable Byzantine failures. Glacier data organization trades storage usage efficiency for durability. For example, to ensure an objects survives failure of 60 percent of the nodes with a probability of 0.999999, the storage overhead is about eleven-fold. To minimize the cost of the storage overhead Glacier uses erasure codes instead of replicas.

SafeStore [49] is a concept that uses multiple autonomous storage service providers (SSPs) to ensure data safety by spreading RAID-ed or erasure coded data among them. It uses SSPs audit mechanism to quickly detect data loss and trigger data recovery before additional faults. The idea of an erasure code system dedicated to the environment with multiple data centers protecting one another data is also described in Myriad [15].

Chapter 7

Conclusions

7.1 Summary

In this thesis we have studied the impact of data organization on distributed storage systems for backup and archival data.

We have identified and described system requirements whose fulfillment depends on the organization of the data stored in the system. Additionally, we have shown that organizing data to fully satisfy each requirement is impossible. This is because there are trade-offs among indirectly conflicting distributed storage system requirements. For many pairs of requirements, organizing data in a way that improves the fulfillment of one requirement reduces level of fulfillment of the other requirement. We have identified and described such conflicting requirements.

Next, we have proposed a novel data organization that resolves the trade-offs among requirements in a reasonable way. This is verified by using the proposed organization in a commercial DSS system called HYDRAstor. In this system, data is organized around a distributed hash table with virtual supernodes spanned over physical nodes. Data resiliency is provided with erasure codes, with fragments of erasure-coded blocks distributed among supernode components. Fragments are stored in containers that are organized into chains to allow fast storage of data streams and enable efficient data consistency management, data health verification and data reconstruction.

We have analyzed the requirements' fulfillment and described trade-offs resolution of proposed data organization. The analysis showed that the proposed data organization satisfied almost all storage system requirements. However, based on experience gathered during the work on HYDRAstor, we have identified improvement opportunities described in the next section.

7.2 Future Work

The analysis of system requirements fulfillment in sections 5.4 on page 79 and 5.5 on page 107 shows the following improvement opportunities:

Limited scalability of read operation due to deduplication As described in section 5.5.3 on page 111, deduplication degrades the locality of data streams stored on disks. The first, initial backup is stored by the system as stream of sequential data located on successive sectors of disks. However, each subsequent backup stream has some fraction of its data changed. Such streams are deduplicated based on all previously stored backups streams. Thus, a backup stream may be scattered over random sectors of disks. This leads to a situation in which the reading of the first backup is fast, but reading the most recent one is slow, because the system has to gather backup data from many disk locations. This is a problem, because users expect the opposite behaviour. That is, reading the most recent backups should be fast, because they are usually used for data recovery, whereas earlier backups are commonly used for archival purposes so their retrieval is not performance critical. One possible approach to address this problem is to change the write operation so that it will not deduplicate data if it leads to a high level of stream degradation. With this approach, the stream of the most recent backup will not be degraded, but at the reduced deduplication effectiveness.

Limited scalability of read operation due to high number of concurrent writes The number of users concurrently writing data to the system is proportional to the system size. In a very big system this leads to another problem with the performance of the read operation as described in section 5.4.6 on page 103. Many users writing data simultaneously cause many fragments from different streams to be interwoven in each SCC. The stream sorting process merges the interwoven fragments within each SCC into consecutive disk sectors. However, with a high number of users, streams may be still fragmented even after stream sorting. As a result, stream prefetch is not effective which leads to random disk access. The source of the problem is the distributed hash table that uniformly distributes streams over all nodes. Fixing the problem requires an increase in streams' temporal locality on nodes of big systems while preserving effective deduplication.

A physical node overloaded by data fragment reconstruction As described in section 5.5.5 on page 112 a physical node with one or several disks failed or a quick replacement of a failed physical node result in all reconstructed data fragments being written to one physical node. Such physical node may become a hot spot and a bottleneck. To avoid this problem reconstructed data should be initially written to other physical nodes to reach desired resiliency level. Later on, the fragments should be transferred to the target physical node as a low priority task not to overload the physical node.

Constant size of supernodes The size of supernodes and their failure resiliency stays at the same level with increased system size and increased number of supernodes. Since the reliability of each supernode is constant, with very large systems the reliability of the entire system can be significantly decreased. The solution here is to allow supernode cardinality to change dynamically with the growth of the system. Such solution will certainly require changes of the system implementation, however general idea of the proposed data organization will not change.

Bibliography

[1] ExaGrid. http://www.exagrid.com.

[2] EMC Avamar: Backup and recovery with global deduplication, 2008. http://www.emc.com/avamar.

[3] EMC Centera: content addressed storage system, January 2008. http://www.emc.com/centera.

[4] Data Domain Global Deduplication Array, 2011. http://www.datadomain.com/products/global-deduplication-array.html.

[5] United States of America 107th Congress. Public Law 107-204: "Sarbanes-Oxley Act of 2002". July 2002.

[6] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael P. Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[7] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment, 2002.

[8] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Olph Y. Wang. Serverless network file systems. In *ACM Transactions on Computer Systems*, pages 109–126, 1995.

[9] Rekha Bachwani, Leszek Gryz, Ricardo Bianchini, and Cezary Dubnicki. Dynamically quantifying and improving the reliability of distributed storage systems. In *SRDS*, pages 85–94. IEEE, 2008.

[10] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The sdsc storage resource broker. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '98, pages 5–. IBM Press, 1998.

[11] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[12] Johannes Blömer, Malik Kalfane, Marek Karpinski, Richard Karp, Michael Luby, and David Zuckerman. An xor-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[13] Vasken Bohossian, Chenggong C. Fan, Paul S. LeMahieu, Marc D. Riedel, Jehoshua Bruck, and Lihao Xu. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Trans. Parallel Distrib. Syst.*, 12:99–114, February 2001.

[14] Peter J. Braam. The Lustre Storage Architecture, 2004.

[15] F. Chang, M. Ji, S. Leung, J. MacCormick, S. Perl, and L. Zhang. Myriad: cost-effective disaster tolerance, 2002.

[16] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *JOURNAL OF NETWORK AND COMPUTER APPLICATIONS*, 23:187–200, 1999.

[17] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009, 2001.

[18] Rick Cook. How to estimate the lifespan of LTO tapes. *SearchDataBackup.com*, May 2007.

[19] EMC Corporation. EMC Centera. Content Addressable Storage. Product Description Guide.

[20] Hitachi Data Systems Corporation. Virtual Tape Library Solutions Brief Improving the Cost, Reliability, and Responsiveness of Your Backup and Recovery Infrastructure . *Hitachi Data Systems Corporation*, January 2006.

[21] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.*, 35(5):202–215, 2001.

[22] Mayur Dewaikar. Symantec netbackup puredisk. June 2009.

[23] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. *Lecture Notes in Computer Science*, 2009, 2001.

[24] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[25] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, pages 75–80, Schloss Elmau, Germany, May 2001.

[26] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 197–210, Berkeley, CA, USA, 2009. USENIX Association.

[27] Cezary Dubnicki, Cristian Ungureanu, and Wojciech Kilian. FPN: A Distributed Hash Table for Commercial Applications. In *Proceedings of the Thirteenth International Symposium on High-Performance Distributed Computing (HPDC-13 2004)*, pages 120–128, Honolulu, Hawaii, June 2004.

[28] Cezary Dubnicki, Cristian Ungureanu, and Wojciech Kilian. Fpn: A distributed hash table for commercial applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 120–128, Washington, DC, USA, 2004. IEEE Computer Society.

[29] Donald E. Eastlake and Paul E. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), September 2001.

[30] Inc. FalconStor Software. Virtual Tape Library (VTL), May 2010. http://www.falconstor.com/products/virtual-tape-library.

[31] Svend Frølund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair C. Veitch. Fab: Enterprise storage systems on a shoestring. In *HotOS*, pages 169–174, 2003.

[32] John Gantz and David Reinse. The Digital Universe Decade - Are You Ready? *IEEE Trans. Parallel Distrib. Syst.*, May 2010. Sponsored by EMC Corporation.

[33] Gartner. Market Share Analysis: Enterprise Distributed System Backup/Recovery Market, Worldwide, 2009. June 2010.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[35] Jim Gray. Why do computers stop and what can be done about it?, 1985. Tandem Computers.

[36] Jim Gray. A Census of Tandem System Availability Between 1985 and 1990, January 1990. Tandem Computers.

[37] Haryadi S. Gunawi, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiri Schindler. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, WI, June 2005.

[38] Ibrahim F. Haddad. Pvfs: A parallel virtual file system for linux clusters. *Linux J.*, 2000, November 2000.

[39] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

[40] John Hantz, Christopher Chute, Alex Manfredi, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: an updated forecast of worldwide information growth through 2011. In *An IDC White Paper sponsored by EMC*, March 2008.

[41] John H. Hartman and John K. Ousterhout. The zebra striped network file system. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 1993. ACM.

[42] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 205–213, Washington, DC, USA, 2005. IEEE Computer Society.

[43] Quantum Hewlett-Packard, IBM. LTO Program: The First Year. 1990.

[44] James V. Huber, Jr., Andrew A. Chien, Christopher L. Elford, David S. Blumenthal, and Daniel A. Reed. Ppfs: a high performance portable parallel file system. In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 385–394, New York, NY, USA, 1995. ACM.

[45] IBM. Autonomic computing: Ibm's perspective on the state of information technology. IBM.

[46] IBM. IBM ProtecTIER Deduplication Solution. http://www-03.ibm.com/systems/storage/tape/protectier/.

[47] IEEE/ANSI STD. 1003.1. Portable operating system interface (posix) — part 1: System application program interface (api) [c language], 1996.

[48] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, 2003.

[49] Ramakrishna Kotla, Lorenzo Alvisi, and Michael Dahlin. Safestore: A durable and practical storage system. In *USENIX Annual Technical Conference*, pages 129–142, 2007.

[50] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[51] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *In Proceedings of IPTPS02*, Cambridge, USA, March 2002.

[52] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[53] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Commun. ACM*, 29:184–201, March 1986.

[54] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[55] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36:31–44, December 2002.

[56] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[57] Andy Oram. *Peer-to-Peer : Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Sebastopol, CA, March 2001.

[58] Alex Osuna, Eva Balogh, Alexandre Ramos Galante de Carvalho, Rucel F. Javier, and Zohar Mann. *Implementing IBM Storage Data Deduplication Solutions.* IBM Redbooks, March 2011.

[59] Alex Osuna, Reimar Pflieger, Lothar Weinert, Xu X Yan, and Erwin Zwemmer. *IBM System Storage TS7650 and TS7650G with ProtecTIER* . IBM Redbooks, 2010.

[60] European Parliament. Directive 2006/24/EC "On the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communication networks". March 2006.

[61] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[62] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[63] Martin Placek and Rajkumar Buyy. A taxonomy of distributed storage systems, technical report. In *GRIDS-TR-2006-11*, Australia, July 2006. Grid Computing and Distributed Systems Laboratory, The University of Melbourne.

[64] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 211–249, London, UK, 1995. Springer-Verlag.

[65] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[66] W. Curtis Preston. Use disk as a primary backup target and overcome shoeshining. *SearchDataBackup.com*, October 2006.

[67] W. Curtis Preston. *Backup & recovery. Inexpensive Backup Solutions for Open Systems*. O'Reilly Media, January 2007.

[68] W. Curtis Preston. Target deduplication appliance performance comparison. http://www.backupcentral.com/mr-backup-blog-mainmenu-47/13-mr-backup-blog/348-target-deduplication-appliance-performance-comparison.html, October 2010.

[69] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[70] Lelii Sonia R. Quantum adds data verification for tape with StorNext archiving software. *SearchDataBackup.com*, April 2011.

[71] Arcot Rajasekar, Michael Wan, and Reagan Moore. Mysrb & srb: Components of a data grid. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 301–, Washington, DC, USA, 2002. IEEE Computer Society.

[72] Chalfant Randy. Tape: A Collapsing Star. *www.mainframezone.com*, March 2010.

[73] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[74] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association.

[75] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *FAST'03: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2003. USENIX Association.

[76] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[77] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem, 1985.

[78] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39:447–459, April 1990.

[79] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[80] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you?, 2007.

[81] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 409–418, Washington, DC, USA, 2004. IEEE Computer Society.

[82] SNIA. Common Internet File System (CIFS), Technical Reference. Technical Report 1.0, January 2002. Storage Networking Industry Association (SNIA).

[83] SNIA. *Common RAID Disk Data Format Specication.* SNIA Storage Networking Industry Association., March 2009. Version 2.0, Revision 19.

[84] Steffen Staab, Francis Heylighen, Carlos Gershenson, Gary William Flake, David M. Pennock, Daniel C. Fain, David De Roure, Karl Aberer, Wei-Min Shen, Olivier Dousse, and Patrick Thiran. Neurons, viscose fluids, freshwater polyp hydra-and self-organizing information systems. *IEEE Intelligent Systems*, 18:72–86, July 2003.

[85] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[86] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[87] Symantec. Netbackup puredisk 6.6 data sheet.

[88] Symantec. Symantec netbackup appliances. http://www.symantec.com/business/ theme.jsp?themeid=nbu-appliance.

[89] Asaro Tony. iSCSI Report. The State of iSCSI SANs. *Enterprise Strategy Group, Inc.*, January 2005.

[90] TruthInIT.com. Product Analysis Report FalconStor SIR VTL, August 2009. http://www.falconstor.com/dmdocuments/TruthINIT-FS-VTL.pdf.

[91] C. Ungureanu, A. Aranya, S. Gokhale, S. Rago, B. Atkin, A. Bohra, C. Dubnicki, and G. Calkowski. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *FAST '10: Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 225–239, Berkeley, CA, USA, 2010. USENIX Association.

[92] Marc Waldman, Aviel D. Rubin, and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[93] Chonggang Wang and Bo Li. Peer-to-Peer Overlay Networks: A Survey, April 2003.

[94] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, 2002. Springer-Verlag.

[95] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Garth A. Gibson, editor, *PDSW*, pages 35–44. ACM Press, 2007.

[96] Bryce Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 104–110, London, UK, 2002. Springer-Verlag.

[97] Zheng Zhang, Shiding Lin, Qiao Lian, and Chao Jin. Repstore: A self-managing and self-tuning storage backend with smart bricks. In *ICAC*, pages 122–129, 2004.

[98] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53, 2004.

[99] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

Appendices

Appendix A

Appendix

A.1 Standard deviation from the mean size of a supernode

Let:

- p be the number of supernodes,
- n be the number of user blocks stored in the system,
- b be the block size,
- $\frac{1}{p}$ be probability that a block is stored by supernode S,
- X_i be a random variable whose value is change in supernode S size after storing ith block in the system.
- XN be a random variable whose value is size of supernode S after n blocks are stored in the system.

The block i with probability of $1 - \frac{1}{p}$ is stored by some other supernode than S and with probability of $\frac{1}{p}$ is stored by supernode S. Thus X_i is equal to b with probability $\frac{1}{p}$ and is equal to 0 with probability $1 - \frac{1}{p}$. Thus:

$$E(X_i) = \frac{b}{p}$$

$$VAR(X_i) = E((X_i - EX_i)^2) = (1 - \frac{1}{p})(0 - \frac{b}{p})^2 + \frac{1}{p}(b - \frac{b}{p})^2 = b^2(\frac{1}{p} - \frac{1}{p^2})$$

Since $XN = \sum_{i=1}^{n} X_i$ then

$$E(XN) = n * E(X_i) = n * \frac{b}{p}$$

Since X_i and X_j are uncorrelated random variables then

$$VAR(XN) = n * VAR(X_i) = nb^2(\frac{1}{p} - \frac{1}{p^2})$$

Standard deviation is

$$stdev(XN) = \sqrt{VAR(XN)} = b * \sqrt{n(\frac{1}{p} - \frac{1}{p^2})}$$

Coefficient of variation(CV) of XN (used in section 5.4.1 on page 79) is

$$CV(XN) = \frac{stdev(XN)}{E(XN)} = \sqrt{\frac{p-1}{n}}$$

List of Figures

List of Tables