České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Timr Marek

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Replikovatelné BLOB úložiště v distribuovaném prostředí

Pokyny pro vypracování:

Nastudujte důsledně články popisující existující BLOB storage systémy. Analyzujte možnosti replikace dat uložených do BLOB storage databází. Navrhněte a implementujte vlastní BLOB Storage databázový systém, který bude využívat distribuce dat mezi jednotlivé služby s důrazem na horizontální škálovatelnost systému. U implementovaného úložiště dbejte na replikaci uložených dat. Pro systém implementujte jednoduché webové grafické rozhraní informující o základních metrikách systému (zdraví, orientační stav replikace apod.). Klíčové části systému důsledně otestujte a dokumentujte.

Seznam odborné literatury:

[1] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in Haystack: facebook's photo storage. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, Berkeley, CA, USA, 47-60.
[2] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY, USA, 143-157. DOI=http://dx.doi.org/10.1145/2043556.2043571
[3] Asynchronous Object Storage with QoS for Scientic and Commercial Big Data Michael J. Brim, David A. Dillow, Sarp Oral, Bradley W. Settlemyer and Feiyi Wang Oak Ridge National Laboratory SC13 November, 2013
[4] RESTfull .NET, Jon Flanders, ISBN: 978-0-596-51920-9, O"Reilly 2009
[5] Troelsen, Andrew, 2012, Pro C# 5.0 and the .NET 4.5 Framework. Apress, ISBN: 1430242337

Vedoucí: Ing. Martin Mudra

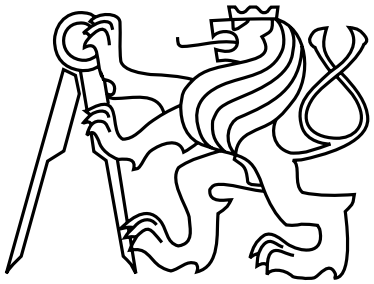
Platnost zadání do konce letního semestru 2017/2018

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 6.2.2017

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Replicated BLOB storage in distributed environment

Bc. Marek Timr

Supervisor: Ing. Martin Mudra

Study Programme: Open Informatics

Field of Study: Software Engineering

May 25, 2017

Aknowledgements

I would like to thank my supervisor, Ing. Martin Mudra, for giving me the opportunity to work on this project, for his guidance and advice. I would also like to thank my family for their support and encouragement throughout my whole studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 25, 2017 .....

Abstract

This thesis deals with the storage of data in distributed data stores. The work studies the possibilities of storing blob data and their representation in storage systems. The thesis deals with object storage design, which distributes its data between individual system members. Emphasis is on the horizontal scalability of the system. The design studies synchronization problems and ensuring of a consistent state after a failure of the system. The work contains an implementation of the proposed system and tests its functionality.

Abstrakt

Tato práce se zabývá ukládáním dat do distribuovaných datových úložišť. Studuje možnosti ukládání blobových dat a jejich reprezentaci v úložných systémech. Práce se zabývá návrhem objektového úložiště, které distribuuje svá data mezi jednotlivé členy systému. Důraz je kladen na horizontální škálovatelnost systému. Návrh studuje problémy synchronizace a zajištění konzistentního stavu při selhání systému. Práce obsahuje implementaci navrženého systému a testuje jeho funkčnost.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the present times, applications produce increasing amounts of data that has to be stored. The nature of data might be various and change over the time. The variability prevents usage of conventional database technologies requiring the static structure.

We also want to conveniently and quickly access any of the stored data. Loss of data may result in a decrease of performance of even cause serious financial damage. We have to deal hardware failures. The more data we store, the higher the chance is some data will get corrupted of lost.

A distributed data storage might satisfy all the needs. Distributed data store is a network of interconnected computers where data are usually stored on multiple nodes. Data store ensures high availability and consistency of stored data. We expect the system to be scalable, therefore adding computational resources should appropriately increase the performance of the whole system.

Several established companies are providing paid services to store virtually any amount of data. Although cloud services allow storing user data with ease, it usually comes with a significant monetary cost. The client is charged for stored volume and used traffic. Naturally, there are open source variants of such storage systems, each providing slightly different features. The downside is that the user has to manage his private infrastructure of machines. Then it depends on the application if it is worth for the user to acquire and maintain the infrastructure. Since the systems are expected to be deployed in large data centers, to work efficiently, their hardware requirements might be significantly high.

1.2 Goals of the thesis

The goal of the work is to design and implement simple BLOB (abbreviation stands for binary large object) storage. This system should run in a distributed network environment. That means it should handle problems associated with distribution and synchronization of shared state. The user of that system must be able to store and read files. The system should ensure that user data would be available even in the event of failure of a part of the system.

The great emphasis of this work will be on a selection of open source tools used for implementation. The system should have as little constrained as possible. That means it should be able to operate on common modern operating systems. Additionally, the user should be able to deploy the system on the majority of relevant hardware available to him.

Chapter 2

Analysis

In this chapter, the term object storage is defined. We specify functional and non-functional requirements put on a simple BLOB storage.

A significant part of the analysis is dedicated to studying of current object storages. Since there is a plenty of different systems, a few systems are selected for the deeper study. The analysis describes strategies of the systems to store, replicate and serve data to clients.

The analysis then describes some principles of file systems and management of state in a distributed environment. The end of the analysis looks into capabilities of current computer technologies. This breakdown will come into use during the implementation of a proof of concept BLOB storage.

2.1 Block and object storage difference

Object storage is a general term that refers to way data are organized and how are manipulated. The data are stored in form of objects, which constist of three components[1, 2]:

Data – The content of the object in an arbitrary format

Metadata – List of attributes which describes stored data. The content of metadata dependes on application. It may contain information about what is stored, what is the format of the data, the owner of the data or some security related information.

Global identification – An unique identifier that allows the data to be found in the distributed system. The location of the object may be transparent for the user of the system.

Block storage divides data into blocks of fixed length without any accompanied metadata. The data are easier to modify unlike on object storage. Modifying file stored in an object storage requires the object is modifies as a whole and then rewritten back. Block storage requires access only to blocks, that needs to be modified. Some block storages can be mount as regular file systems.[2]

Block storage is more suited for applications that require virtual disk storage and the data are modified frequently. The size of the files is smaller and access to them is random[2].

Object storage is more suitable for archiving, backuping and preserving data[3]. The content of files has static character and they are accessed sequentialy. Individual object may be large. Object storage is cheaper to scale.

In this document the term object will refer to a file, that an user deposits into storage. By constrast, the term file will be used to describe physical representation of object on a disk.

2.1.1 BLOB

The abbreviation BLOB stands for Binary Large OBject[4]. Blob is an amorphous piece of binary data, which internal structure is not known externally[4]. Another interpretation refers to BLOB as to *binary data type*[4]. In the context of block storage,BLOB is another term for an object. Therefore terms BLOB and object are interchangeable in this work.

2.2 Analysis of requirements

In this section are listed functional and non-functional requirements put on designed system. The requirements are set by the assignment of this thesis.

2.2.1 Functional requirements

1. The system will store user data in the form of BLOB objects
2. User won't be able to modify stored data, only to erase them
3. The system will expose RESTful (Representational State Transfer) interface for client applications
4. The system will utilize the most of internet connection bandwith between system and client by writing and reading from disk in parallel
5. The system will enable to monitor the state of the storage with a web application

2.2.2 Non Funcitonal requirements

1. The storage must provide protection against loss of data in the event of technical failure
2. The storage must be highly horizontally scalable
3. Adding new storage nodes to cluster must be easy for the user
4. The system must be platform independent
5. The system must be able to restore after unexpected system crash
6. The system must be operational on relevant hardware with low performance
7. The system should prefer use of open source frameworks and libraries

2.3 Related work

There are existing object data storages that offer features similar to requirements of this work and are widely used in the cloud environment[5, 6]. The following section will study some of those data stores in order to examine advantages and drawbacks of features of the systems.

2.3.1 Haystack

Haystack is an object storage optimized for the efficient storage and retrieval of billions of photos, developed by the Facebook company. Its goal is to provide high throughput and low latency[7]. The system also aims to be fault tolerant, simple and cost-effective. The implementation of Haystack is not open to the public, however the paper [5] describes the general architecture.

The Haystack consists of several subsystems. The most important are Haystack Directory and Haystack Store. Haystack Directory manages general metadata about photos and their location in the Haystack Store.

The previous system stored each photo separately on NFS (Network File System) in directories consisting thousands of files. Reading an image lead to an excessive number of disk operations due to how NFS manages directory metadata[5]. Haystack Store composes individual photos sequentially to files of sizes roughly of 100GB. The Store holds the specific location of an image in the memory. It maps image identity to the physical file, its length and position in that file. The underlying file system assigns no other metadata that would be ignored regardless. Each image stored in the sequence file is stored in a structure called needle[5]. The needle consists a header and footer with metadata related to file and image data. During read, Haystack Store retrieves file location and offset from memory, calculates the length of the whole needle and reads it. Usually, only one disk operation is enough to fetch the whole needle.

To speed up the start of the system, Haystack Store manages an index file matching the structure of the volume file but containing only the metadata. The systems load the index file to the memory and therefore does not have to traverse the volumes[5].

Haystack Store distinguishes physical and logical volumes. The logical volume consists of several physical volumes on different machines. When Haystack stores an image on a logical volume, the image is written to all physical volumes[5].

The figure 2.1 describes the architecture of the system and sequence of requests made when an image is requested. A client browser requests the web server a photo over HTTP (Hypertext Transfer Protocol). The web server retrieves metadata of the file and returns an address of a CDN (Content Delivery Network) and a image identifier. The CDN stores only photos that are popular at that time. If CDN does not own the photo, the client approaches the Haystack Store. The Store searches its cache, and if there is a miss, it loads the photo from a physical volume[7].

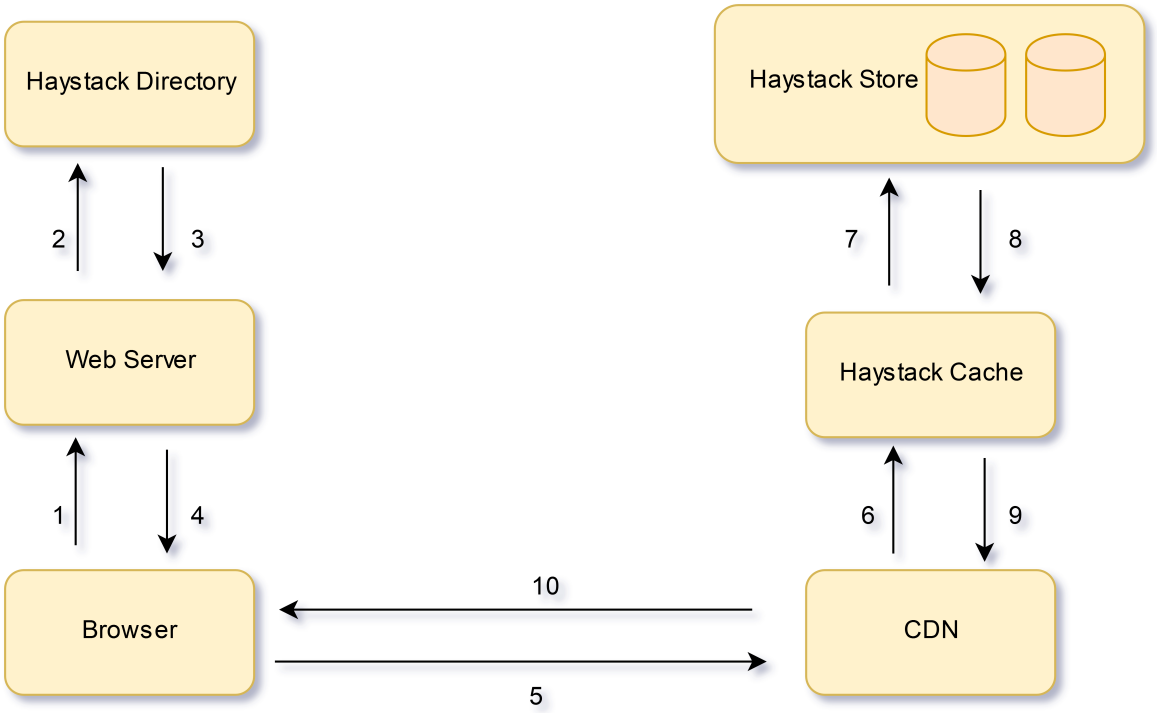


Figure 2.1: Schema of a sequence of requests necessary for retrieval of an image from the Haystack store

2.3.2 Windows Azure Storage

Windows Azure Storage (abbreviated to WAS) is cloud storage that enables clients to storage structured and non-structured data at arbitrary volume[6]. WAS is a member of numerous group of services offered on the Azure platform[8]. The client is allowed to store files in the form of BLOBs, structured data to tabular service called Azure Table and messages to Azure Queue service. These services provide different RESTful APIs (application programming interface) to access the data. However, the services share the same internal data representation. The client is charged for stored data and the traffic.

WAS is a very complex system that ensures strong consistency of data, their availability and partition tolerance [6], even though it is in the contradiction with CAP theorem (explained in the section 2.4).

Storage Stamp (abbreviated to Stamp) is a single data center of the WAS system. The capacity of one Stamp is up to 30PB. The WAS assigns the client a name of an account that matches one Storage Stamp. The Stamp encapsulates a front-end serving client requests and components named Partition Layer and Stream Layer. Figure 2.2 shows general architecture.

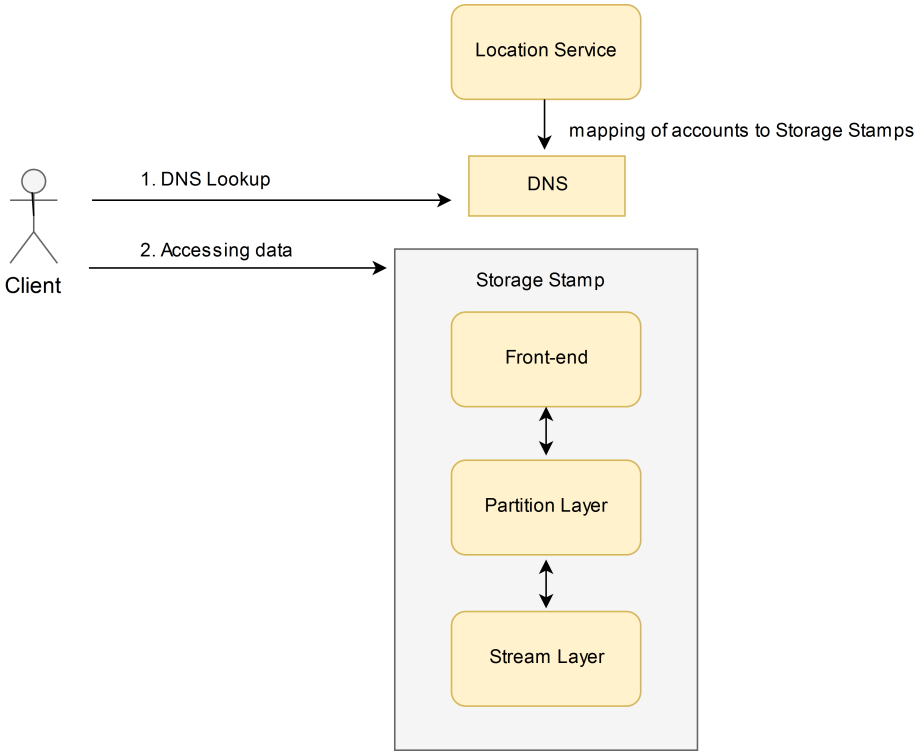


Figure 2.2: Schema of high-level architekture of Windows Azure Storage

2.3.2.1 Partition Layer

Partition Layer (abbreviated to PL) provides an abstraction of stored data (BLOB, Table, Queue). It ensures transactional operations on objects and their strong consistency. PL also contains cache to reduce I/O (input/output) operations on Stream Layer[6].

PL is responsible for replication of data on the object level between individual Stamp data centers. This replication takes places asynchronously after writing of an object[6].

2.3.2.2 Stream Layer

Steam Layer (abbreviated to SL) stores data in the form of Streams, which are read by PL. SL, unlike PL, does not know the meaning of stored data. The Stream is a list of pointers to files names extents. In the context of WAS, an extent contains blocks of different length. Each block represents the smallest unit of data, that WAS manages. The figure 2.3 describes the structure of a Stream. SL comprises two principal components, the Stream Manager and series of Extent Nodes. Extent Node stores extent files and Stream Manager monitors their state and location. SL also replicate data, but on the level of extent files and in the context Stamp data center. During block write, SL synchronously writes a block to three different Extent Nodes. Large objects are broken up over many extents by Partition Layer. For each object, the PL keeps track of what Stream, extents and byte offsets in these extents contain the object data.

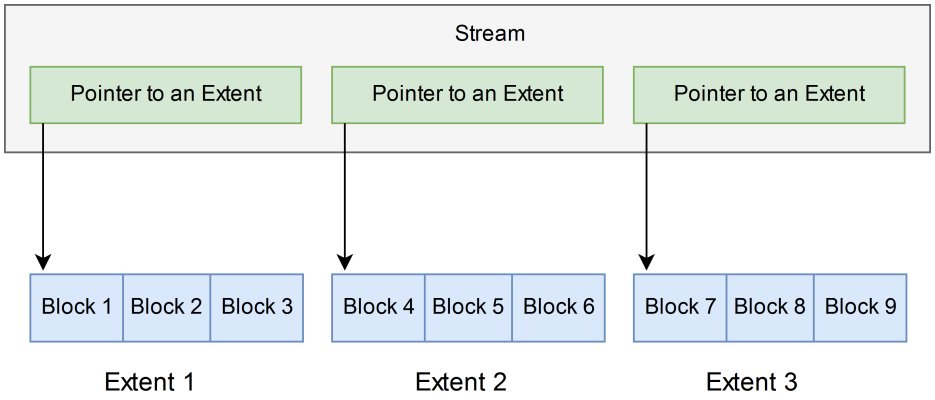


Figure 2.3: Structure of a Stream stored on the Stream Layer of Windows Azure Storage

2.3.3 SeaweedFS

SeaweedFS is an open source scalable distributed file system. SeaweedFS aims to be able to store billions of files and serve then fast [9]. The design of the system draws inspiration from the Haystack store. The file system is not POSIX (Portable Operating System Interface) compliant, but with the help of additional software, SeaweedFS could be mount by FUSE (Filesystem in Userspace). System exposes a HTTP API and serves responses in JSON (JavaScript Object Notation) format [9].

Each member of the cluster takes up to two roles. Member can be a *master server* and a *volume server*.

2.3.3.1 Volume server

Volume file is a file of the maximum size of 32GB that contains stored objects in a similar way as Haystack does. Volume servers manage volumes files. Each stored object is associated with a unique object id. The volume server stores mapping from an object id to a volume, an offset in the volume and the length of the object in a key-value database or memory. Volume server accepts, stores and server client data. The server replicates the volume files to other volume files in the system [9].

2.3.3.2 Master Server

The master server only maps unique volume ids to volume servers. This approach is different other mentioned storage systems because the server does not know anything about stored files. Although there may be multiple servers in the cluster with the master role, only one at a time is a leader. Delegating storage of metadata to volume servers lowers load on the master [9].

2.3.3.3 Write of an object

The upload of a file consists of two steps. First, a master server must assign volume id for storage of the file. Master allocates file ids sequentially. For security reasons, the master

server generates a random file cookie for each object. The file cookie prevents the unauthorized clients from accessing random files based on guessing. The client receives JSON response from the master containing a location of a server volume, a file id, a file cookie and a volume id.

In the second step of the upload, the client sends the file to the given volume server and attach all identifiers to the request.

The SeaweedFS is suitable for only small and medium size files such as images for websites. If a client wants to upload a larger file, it must be split into chunks by master server. Upon client request, the master returns a list of ids where each maps a chunk to object in storage[9].

2.3.3.4 Process of failover and election of a leader

Despite multiple master servers running in the cluster, only one at a time is holds mapping of volumes. If a client requests an object write on a master server that is not currently the leader, the server will redirect the client to the current one (schema shown on figure 2.4). The present leader of the cluster periodically sends heartbeats to others. If the server fails, a new leader is elected. Therefore the master server is not a single point of failure of the storage system.

The election is accomplished using Raft[10] consensus algorithm. The consensus is achieved by selecting a leader. The leader will replicate shared state to the followers. During and after the election, there is time window, in which system does not accept client requests. Each master must send its volume mapping to the newly elected leader.

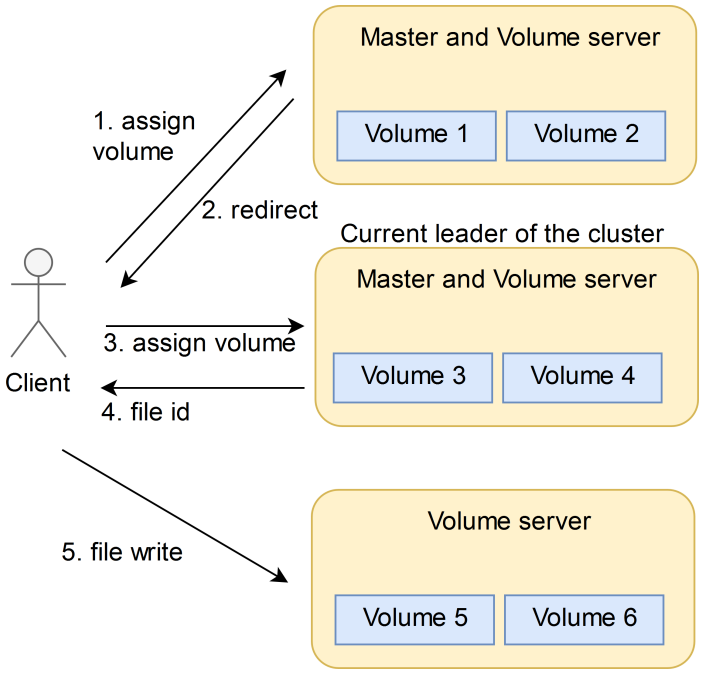


Figure 2.4: Diagram of align assignment of a volume and file ids for write by current leading master server

1. Client requests assignment of a volume
2. Master redirects the client to the current leader of the cluster
3. Client requests assignment of a volume
4. Master generates file and cookie id and assignes a volume for write
5. Client writes the file to node managing assigned volume

2.4 Shared state in distributed environment

The sharing of some common state in distributed network of computers is a significant problem. The system must deal with concurrent updates issued on different members of the system. The data must remain in the consistent state.

CAP theorem states that a distributed data store may only provide only two out of three following guarantees[11]:

Consistency – A read operation is guaranteed to return the most recent written value.

Availability – A non-failing node will return a reasonable non-error response within a reasonable amount of time. The system remains functional even if there are failed nodes.

Partition Tolerance – The system will continue to function when network partitions occur. Network partition is a failure of a network device that causes a network to be split.

Regular relation databases usually ensure strong consistency of data. The strong consistency means that after an update is committed and acknowledged by the system, all clients must see the last committed value. In contrast, a system that provides eventual consistency of data does not guarantee, that subsequent accesses to data will return the last updated value[12]. However eventual consistency guarantees that if no new updates are made on an object, eventually all members of the system will return the most up to date value.

2.4.1 Concensus on a value

Clients of distributes storage system may update a value on different nodes simultaneously. The systems must ensure consistency of the data and must propagate the correct recent written value to the rest of the nodes.

When distributes systems synchronize data across the cluster, they may use an implementation of a Paxos[13]. Paxos is a family of consensus algorithms. The consensus is a process of agreeing on some value among of group of participants. The general principle of the process is following[14]:

1. A node called leader prepares a proposal of a value to a group of acceptors
2. The acceptors may promise that they accept the new value and reject all previous updates
3. If the majority of acceptors accept the proposal, the leader proposes the value.
4. If the acceptor did not receive any new proposal, it accepts new value

Implementation of a consensus protocol is complicated as it is prone to errors. Systems often rely on good synchronization of clocks. Any clock skew may result in data loss. The system may also end up in a split state. Network partition divides cluster into two separate networks. Each network creates a quorum and elects a leader. The system remains functional and continues to accept client writes to maintain availability. When the partition is resolved,

the state of both networks has to be merged. The storage rollbacks the writes on one of the sub-networks. Consistency is sacrificed in favor of availability [15].

It is important to state that this is only relevant to data stores that allow modification of stored data. Storage of immutable data does not suffer from problems mentioned above.

2.5 Analysis of file systems

The main functionality of the designed system is the storage of data. It is crucial to understand how files systems manage the data to build an efficient solution.

2.5.1 Journaling file systems

Journaling file system is a system that records changes to be made in the files [16]. Changes are registered in a journal. The purpose of journaling file system is to ensure consistency of data in the event of system crash or power loss. In that situations, changes recorded in the journal are used for reconstruction of files. Journaling reduces the chances of file damage.

Operations on file systems may consist of a series of smaller operations. In case of interruption between sub-operations, the system may end up in a non-consistent state. Journaling FS (File System) closes these operations in transactions. The transactional approach ensures atomicity of each operation. After system failure data are reverted to the state before transaction beginning, or the transaction is finished.

The transaction consists of these steps[16]:

1. Writing of planned changes on file to the journal
2. Making of changes to the file system
3. Recording successful completion of operation to the journal
4. Removal of record in the journal

Storing a journal decreases the performance of FS during file write because each piece of data has to be written twice. Some file systems record only metadata (eg. NTFS [17]). Recording only metadata increases performance but it involves a risk, that inconsistency happen between metadata and data.

Journal is a special file stored on designated place in the file system.

2.5.2 Ext4

Ext4 is journaling file system for Linux. On ext4 FS, each file is associated with an *inode*. The *inode* stores file metadata and a list of data blocks. Metadata are attributes partly standardized by *POSIX*. The attributes contain for example file size, user ID, group ID, file access mode, timestamp[18].

The blocks in the list are either direct and point to the disk sector, or are indirect and point to another list of blocks. In a case of large continuous files, ext4 can map file using

an extent tree. Allocating file using indirect mapping requires mapping each block. Leaves of the extent tree record number of covered blocks. This approach reduces the number of metadata blocks used moreover, brings some disk performance improvement.

Number of *inodes* is set during file system creation. The number is based on the heuristic how many files will be stored on FS. Storing a lot of small files can lead to exhaustion of *inodes* prohibiting storing another file.

Directories on ext4 are files that map names to *inodes*. [18] Previous versions of ext4 stored listings as a linear array. The performance was not great when directory contained many files. Ext4 stores mapping in *htree*, a special version of *btree* with the constant depth of one or two levels. The system hashes filename and uses the hash to traverse the tree to find a file. A leaf node contains a list of files which are then sequentially searched.

Maximum file size on ext4 is 16TiB

2.5.3 NTFS

NTFS (New Technology File System) is a proprietary file system developed by Microsoft[19]. Each file stored on NTFS is recorded in a special file Master File Table (MFT). The MFT contains metadata about itself, directory listings and may also store file data. There is a mirror copy of the MFT in case the original got damaged[20].

The file record contains file metadata named *resident attributes*. If the attributes take too much space, additional clusters outside the MFT are allocated to store *nonresident attributes*. A cluster is the smallest amount of disk space allocated to store file data, usually of a size of 4KB[20].

One of the file attributes contains extent list with file data. To access data faster, data of small files may reside in the MFT. The architecture allows the maximum size of a file to be 16 exabytes. However, the implementation permits 16 terabytes[20].

Directory entries are specific records that contain an index to files. Large directories are organized in a B-tree structure, that contains pointers to other directory entries stored in clusters outside the MFT[20].

2.6 Analysis of software tools

This section analyses options for tools for implementation of the storage system. The most affecting choice is the selection of a programming language. Languagea come with an ecosystem of available libraries and frameworks with different stability and performance. Following list captures needed functionality based on requirements put on this thesis:

1. Deploy on multiple platforms
2. Database access
3. Serving of HTTP content
4. Fast file access

2.6.1 Deploy on multiple platforms

The system is required to be deployable on common operating systems which are Windows, Linux, and macOS. There are two major high-level programming languages with rich ecosystems and good overall performace[21] which are available to all three operating systems. Java and C#. They both allow building an application which is runnable on any system. C++ is omitted from the selection of tools despite being arguably more performant due to the fact, that C++ is compiled and not interpreted language. The reason is that build of the system would require multiple compilations for every architecture, and it is more complicated to create fully portable code than using the above languages.

In the beginning, C# was exclusively bound to Windows platform. That is no longer true with the existence of Mono. Mono[1] is cross platform and most importantly open-source implementation of Microsoft's .NET Framework. The system is Ecma standard-compliant[22] of *Common Language Infrastructure*, which describes executable code and runtime environment of C# and other languages. That allows the code to run on different computers and architectures. The Mono project mirrors the development of C# language done by Microsoft.

In comparison, there is Java with OpenJDK. OpenJDK is an open-source implementation of Standard Edition of Java Platform[23]. It is a common Java runtime environment on many UNIX distributions. OpenJDK is also a reference implementation of Java platform[24]. In contrast, Mono developed by Microsoftś subsidiary Xamarin does not guarantee the uniformity with C# on Windows.

Both mentioned platforms require a runtime environment to be installed on the target machine and both are full featured languages with many similarities from the programmer point of view.

2.6.2 Database access

The system will have to store information about user data in some way. The system will also have to record the state of the storage persistently. Any loss or corruption of data about state directly affects users. Naturally, such storage would often be accessed and therefore it must not slow down the system.

Currently, there are several NoSQL databases available, that provide high performance. Namely Cassandra[2] or for example MongoDB [3]. Each commonly used solution has good client support for both Java and C#. NoSQL databases are usually very well horizontally scalable and distributed. They are schema-less. The latter means querying data with some relation between them is harder or not possible in case of key-value stores. Distributed NoSQL databases may sacrifice consistency to availability. Read and write operations are not necessary *ACID*. The lower consistency could be manifested by the risk of loss of data due to system crash of network partition.

The problem could be solved as a compromise between both requirements by selecting a relevant relational database. Relational databases provide operations with *ACID* (Atomicity,

[1] `http://www.mono-project.com/`
[2] `https://cassandra.apache.org/`
[3] `https://www.mongodb.com/`

Consistency, Isolation, Durability) properties. Some databases could be scaled by sharding. The sharding is technique splitting up a large table of data row-wise. A large table can be split into multiple smaller ones that will be placed on a separate server.

To access the database conveniently Java offers Hibernate[4] ORM framework compliant with Java Persistent API. For the C# there is complementary framework NHibernate[5].

It will not be necessary for each member of the distributed system to run a large relational database. However, members of the system will have to store at least a minimum amount of file metadata. These would contain some file identification and its location based on internal representation. There may not be a reason to store such data in a database and instead hold it in memory for very fast access. However, there are two reasons for the use of a database.

Despite each record could be as little as a few tens of bytes, it could add up to tens of megabytes after surpassing millions of stored files. The records would occupy the memory of the machine which resources could be constrained as required in 2.2. The storage service will be running in the background and should not limit other applications from running when not used.

The latter reason is the need to persist the state when the server is shutdown. There should be a representation of stored objects from which the system would read the state to memory. Also, the system should anticipate unexpected crashes which could corrupt the saved state. For that reason, servers could use an embedded journaling database to persist the storage state.

There are several options to choose from in Java ecosystem. Commonly used embedded databases contains SQLite[6], H2[7] and Apache Derby[8]. In some cases they can produce more performace than PostgreSQL[25].

For Mono, there is only SQLite as an embedded open-source relational database. The drawback of SQLite is the lack of support for transactional access.

2.6.3 Serving of HTTP content

The requirements specify the way the clients will access the data store. The system is required to serve clients using a RESTful API. Most requests are expected to require a significant amount of time to complete due to transfers of large data streams. Also, serving requests of a large quantity of concurrently connected clients may be problematic[26]. It is important to choose a framework capable of handling such pressure.

Beside traditional container based Java servers, there is low level I/O framework Netty[9]. The main idea of the framework is to be event driven and spend as little time as possible in user space. Despite running on JVM with garbage collector, Netty manages pools of native memory. That approach allows in some cases to effectively work with data without loading them into heap memory. There are dozens of large open source systems based on Netty[27], namely Apache Cassandra, Elastic Search or Play Framework.

[4] `http://hibernate.org/`
[5] `http://nhibernate.info/`
[6] `http://www.sqlite.org/`
[7] `http://www.h2database.com/html/main.html`
[8] `https://db.apache.org/derby/`
[9] `http://netty.io/`

The counterpart for Netty in C# is Kestrel[10], cross-platform HTTP server base on asynchronous I/O library *libuv* [11]. Kestrel is used as a permormant async server for *ASP.NET Core* application. If one would want to get more low-level access, the *libuv* could be used directly. Since *libuv* is a framework written in C, there are C# wrappers[12] [13].

The performance of Kestrel and Netty is very high in comparison with other platforms. When comparing the two discused frameworks, Netty is rather faster accoriding to benchmarks[28].

2.6.4 Fast file access

The system is expected to write and read large portions of user data from disks. There is not much of proccessing needed. To increase performance, the system should utilize zero-copy data transfers to save CPU cycles and memory bandwidth[26]. Netty incorporates these principles in it is design[29].

The approach for C# application is to either invoke native library code, which is not very suitable for designed multiplatform system, or to utilize class *UnmanagedMemoryStream*[30].

[10] `https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel`
[11] `https://github.com/libuv/libuv`
[12] `https://github.com/StormHub/NetUV`
[13] `https://github.com/txdv/LibuvSharp`

Chapter 3

Design

This chapter describes proposed design of storage system in detail. The design is based on gathered knowledge from analysis. This chapter will describe the in-depth architecture of the whole system. It will cover competencies of each member of the system. The design describes operations and internal processes of the system and handling of failures.

3.1 System architecture

The system will consist of two main components, a master server (in the text mentioned simply as a master or abbreviated to MS) and a series of storage nodes (simply nodes or abbreviated to SN). The figure 3.1 presents an organization of a storage cluster. The client of the system is meant to be an application communicating with the exposed interface.

3.2 Master Server

The master server will manage the running of the whole cluster. Its responsibility will be storage of object metadata and object locations on different storage nodes. The metadata will be served on demand to clients and individual nodes. All metadata will be stored in a relational database.

Analysis showed that sharing a state between nodes of a distributed system is problematic. Therefore the master server will run in only one instance in each cluster. The server will be a unified access point for all clients.

Responsibilities of the server will include generation of object identifiers and assignment of suitable nodes for storage of a new object.

The master server will keep a connection with storage nodes to monitor their health. Through this communication channel, the nodes will receive commands to replicate or delete object data.

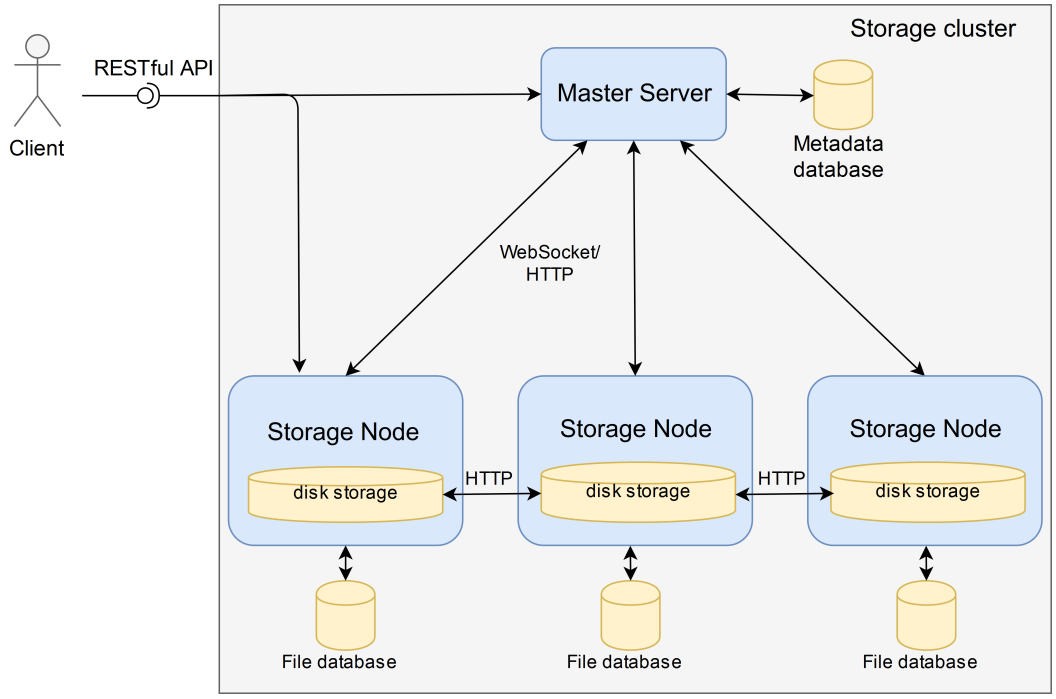


Figure 3.1: Diagram of designed architecture of the storage cluster with communication channels between components

3.3 Storage Node

The storage node will be the fundamental building block of the system. Its main purpose will be to accept, store and serve client data. The node will manage an internal database mapping stored objects. The database will contain only the minimum metadata necessary for storage.

3.4 Operations on object

There are a few required operations, that client will be able to execute on an object. These operations are:

- Object write
- Object read
- Object delete

Each operation will be a continuous process that will be composed of smaller subtasks client will have to carry out. Subtasks differ on what component of the storage will be applied.

Master Server will offer following functions to the client:

- Create new object and assign it to nodes
- Get locations of nodes for finishing storing
- Get location of stored object
- Delete stored object

Functions on a Storage Node mainly follows processes started on the Master Server. The following features will be available to a client:

- Upload partial data of a new object
- Finish data upload
- Read stored object

3.4.1 Object write process

The process of storage of a new object will be the most complicated of general operations. It will consist three steps. Detailed description of each will follow

1. Client request creation of a new object on Master Server
2. Client uploads data to a storage nodes
3. Client makes finishing request on a storage node to end the upload

Figure 3.2 demonstrates sample process of storing of an object.

3.4.1.1 Client request for object creation

Client will request creation of a new object on Master Server. The request will contain some metadata about stored file. This design proposes to store the original name of the file and its length. MS will record a creation timestamp. The new object will be given a unique identificator described in section 3.4.1.4. The client also specifies mandatory replication factor in his request. The replication factor is desired number of nodes, on which a replica of the object should be stored. If there will be enough connected storage nodes that could satisfy requested replication, MS will select a group of them of the same size as the replication factor. These nodes will be assigned to collect and store the object. Server's response to the client will contain a new id and a list of assigned storage nodes. The list will contain IP addresses and corresponding ports at which the servers will be accessible.

3.4.1.2 Upload of client data to a storage nodes

The client will be able to choose any given storage node to upload object data. The request will contain the unique id and the length of the content.

If the object is large, it does not have to be sent using single HTTP request. Instead, client will be advised to include *Range* header[31] in the request and send only partial content. There are multiple possible usages and formatting of *Range* header. The SN will not implement all of them. The design considers a simple case where the value of the header is in the form of `bytes=firstBytePos-lastBytePos` . An example of the header for request containing first 1000 bytes of a file would be `bytes=0-999`. The implementation will not limit the functionality.

There are couple of reasons to split data into multiple requests

- Better recovery in case of network failure
- To achieve better throughput
- To allow seeking in the stored data

Network connection should not be considered stable. A large file transfer might take a significant amount of time and the connection interruption is possible. Therefore in case of failure, client does not have to start all over, but instead only send the last partial content again.

When dealing with higher latency, the bandwidth-delay product should be taken into consideration. It measures the maximum amount of data transmitted over the network before being acknowledged.[32] Opening multiple connections should increase the throughput to utilize available bandwidth fully.

The client will be allowed to send data to any of given group of storage nodes. Each storage node will synchronously share the acquired data with other peers in the group. To prevent buffering of data on a node leading to crash, SN will hate to put a back pressure on the client to lower the incoming traffic.

Storage Node will have to verify the existence and the length of the send object presented to him. When SN receives a client request, it will try to fetch the object metadata from its database. If there is no record, object either does not exist, or SN is missing object metadata. Therefore before reading the content of the request, SN will make a request on internal API of the master server to get the metadata and a list of peers. This scenario could lead to wasteful requests to the master server in a case when client sends simultaneously multiple requests at once. For each of the request, SN would connect to the MS. Not only the same metadata would be sent redundantly, but each request would open a new TCP connection. for that reason, the design in 3.1 proposes an open WebSocket connection between MS and each SN. After client creates a new object, MS preemptively pushes the object metadata over the open TCP connection to all nodes in the group.

The group of storage nodes will be a measure to prevent loss of the tenant data. In case of the failure of the node to which client is connected, a client should continue sending data to other members of the group. Because the original node will resend the client data to the peers, other nodes most likely will have the previously sent parts.

Considering uniform chance of a failure of any given SN, the larger the node group will be, the higher the chance of failure of any member. Therefore when SN cannot synchronize data to his peer because of the peer failure, client data transfer should not be interrupted with the server error. The purpose of the group is to maximize the availability of the system for the client for the duration of the write of the data.

When a failed node from the group is reconnected, the system will not make any effort to sychronize missing client content. This would require that each SN would have to monitor the health of the peers for every object that is beeing assigned to it and also it would have to synchronize what is the state of the object on other peers. This overhead would significantly reduce the performance and scalability of the storage. The other approach would require transferring the responsibility to the master server. Each SN would have to commit collected parts. This would bring massive overhead to MS and decreased scalability even more because the architecture takes into account only one instance of MS. With increasing number of SN connected to cluster would increase the load on MS.

3.4.1.3 Client finishing request

Since the client will be allowed to send the object data to any available node from the given group and nodes will not share a common state of the upload, the client will have to notify that the upload is over. The client will mae a finishing request on any node from the group. The node will respond to the client with an acceptation of the request. Then the NS will transform the uploaded data into an internal representation described in the section 3.5. After a successful finalization, node will notify the MS using the WebSocket channel. Master server will save, that the object is stored on the node, and change the state of the object from writable to readable (depicted in the figure 3.5). Master Server then uses the WebSocket connection to notify other nodes from the group to also finalize the object. MS will give them the location of the living replica in case any node is missing a part. The client will not be part of the process.

Figure 3.2 shows a situation in which a node crashes during the upload. The Storage Node 2 is missing part of client data. A similar scenario could contain crashed node that restarted, and now the client is issuing finishing request on it. In both cases the node is missing data and cannot transform the object for permanent storage. It will refuse client request client and respond with a list of all missing data ranges. The client will be obliged to add all absent parts and make the finishing request again.

In a situation, an SN is missing some parts and MS is issuing a file finalization, there will be no client to resend the data. The node will hate to connect to the peer that already has a complete replica of the object and request all missing data ranges from it. The location should be requested on the master server. After successful finalization, the node will notify the MS. The master then records the location of the replica in the database.

3.4.1.4 Object identity

Design of the storage system does not consider any user roles or user permission. For that reason each object is given a statistically unique identifier. This approach limits the clients to guess the ids of other objects in an attempt to damage or access them without any

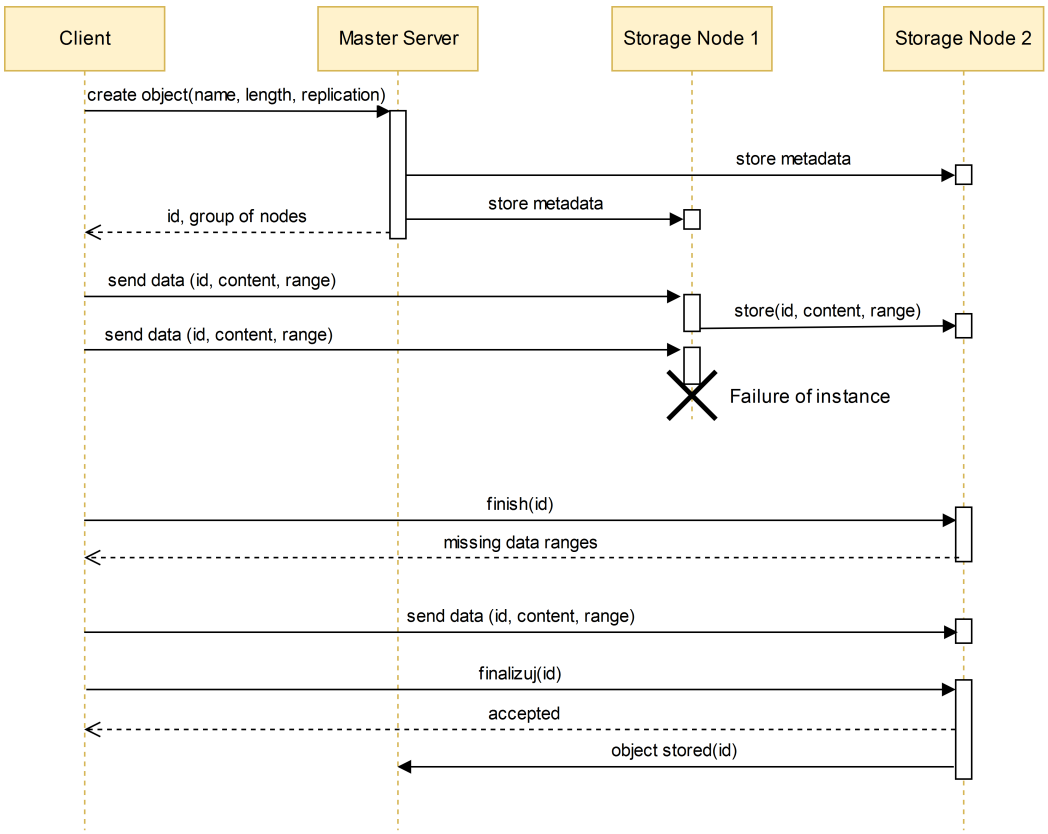


Figure 3.2: Diagram of an example situation of a write of an object with replication factor 2 with failure of one of the nodes

authorization. The length of the id will be 64 bits. It provides more than enough large namespace for objects and allows it to be internally stored as *long* data type. Clients will be presented with an alphanumerical representation which will be used in URI (Uniform Resource Identifier) of client HTTP requests.

Generation of the id will take place on the master server during client's request to create a new object. The generation will be based on UUID (universally unique identifier)[33].

3.4.2 Object read process

Reading an object will be less complicated than writing. The procedure will take two steps:

1. Retrieve the object location from the master server
2. Request data from a storage node

The master servers will be able to tell, which running and connected storage nodes keep the object. The server will return a list of these nodes together with their IP addresses. The client may assume that the data will not migrate in the cluster often and instead directly approach a storage node from the process of object write. If the node no longer stores the object, it will simply respond with an error status.

In order to speed up the transfer, the client may connect to multiple nodes at once. In that case, the request should contain *Range* header specifying what part of the object should be returned from the particular node.

The whole sequence with optional request on master is shown in diagram 3.3

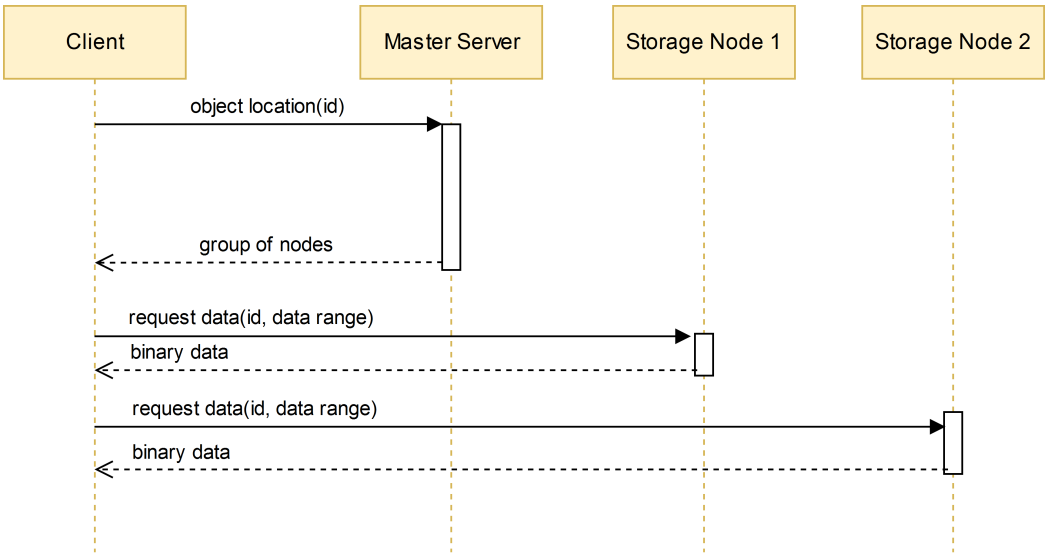


Figure 3.3: Diagram of sequence of client requests necessary for retrieval of object data

3.4.3 Object deletion process

To delete an object, the client will have to make a request on the master server. The server handles removal of the data from each storage node. A general process is shown in the figure 3.4. Some storage nodes holding the object may not be connected to the cluster at that moment. The master server must ensure that the deletion will be executed after the nodes reconnect.

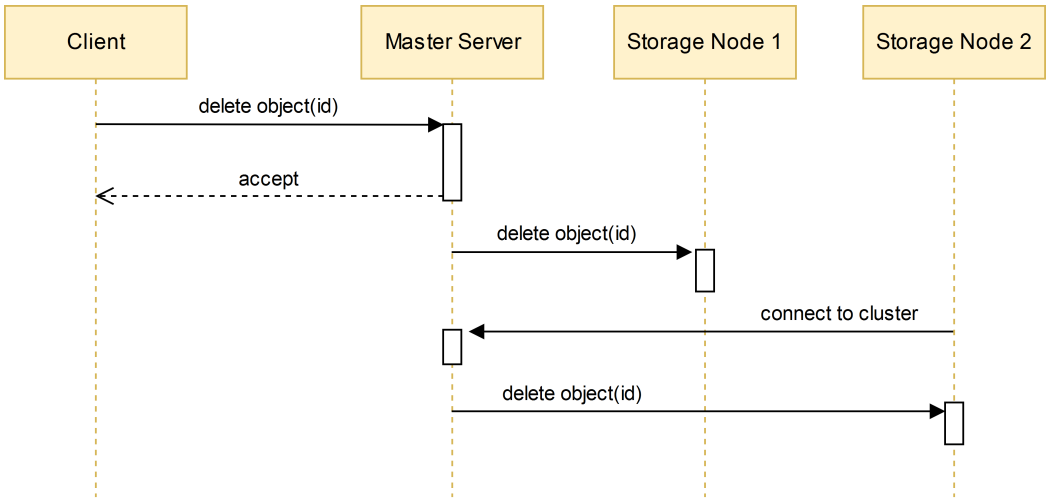


Figure 3.4: Process of asynchronous object deletion after client request

3.4.4 Object lifecycle

The figure 3.5 shows the main states of the object. The states are divided with respect to writability and readability.

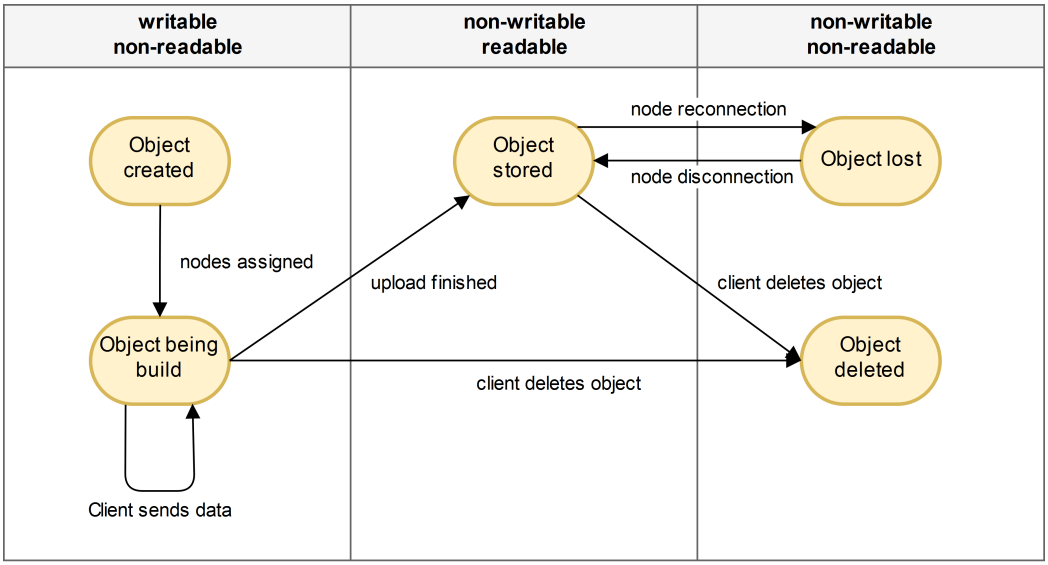


Figure 3.5: Diagram of the main states of an object and the transitions between them

3.5 Object Representation

The analysis has shown that storing large amount files may slow files system and increase the number of disk operations to access a file. The file system also stores metadata that the system would not utilize. For that reason, the system will store objects in files of large length. This work will refer to these files as monoliths. The figure 3.6 describes the internal organization of data.

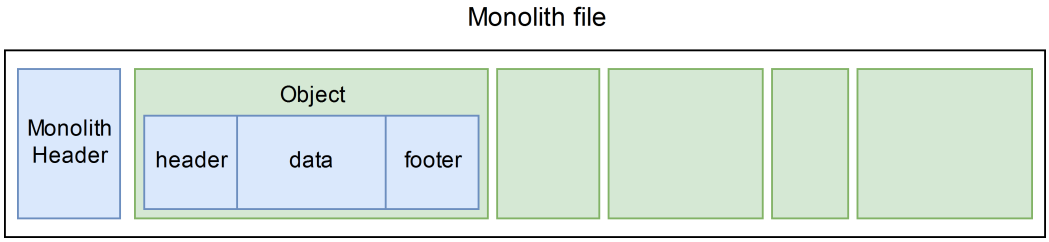


Figure 3.6: Diagram of the internal structure of a monolith file

The monolith file will start with a header that contains a unique constant identifying that this is a monolith. The purpose of the constant will be to prevent reading and interpreting files, which are not monoliths. The constant will a sequence of 8 bytes encoding the word *monolith* in ASCII. After the constant, a 4-byte flag follows indicating the state of the monolith. The flag can mark the monolith as deleted. This is described more in the section 3.5.5. The rest of the content of the file is a sequence of individual objects. Each object consist of three components:

- Object header
- Binary data content
- Object footer

Figure 3.7 shows a layout of an object. The header will identify the content of the object. It will include a 64-bit identifier, the length of the following data block, the flag indicating the state and the 32-bit check number.

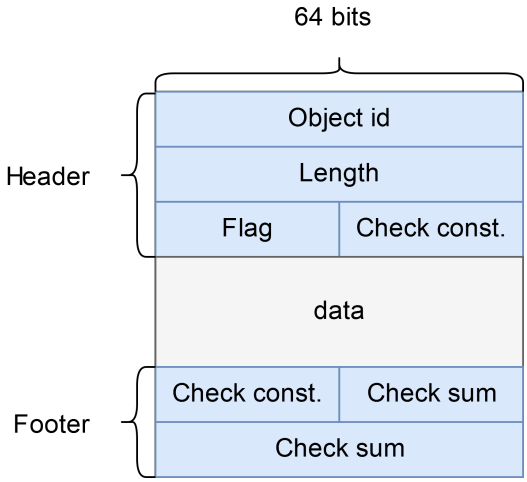


Figure 3.7: Diagram of the structure of an object stored in a single monolith file

The header will be followed by the data block that contains the whole object. The footer after data block should contain two numbers. The checksum number calculated from data content and the check number from the header. System will randomly generate the check number for each object during their write. When the system will read the content of the file or scan the stored objects in the monolith, matching numbers in header and footer will signify, that the reader is not misinterpreting some random data as a header. It will also means that the monolith contains the whole object and not only a part.

3.5.1 Maximum object size

By design the system will no not split stored object. The advantage of this approach is that less metadata is necessary to be stored on a SN about a location of an object. To know the position of an object, only the file and offset in that file will be necessary for access. If the system stored the objects as a list of blocks, it would be much harder to reconstruct the state of SN after a failure. Reading a split object would require more disk operations, possibly require opening of multiple monolith files. Conventional magnetic disks are more efficient when reading data sequentially.

The downside of this approach is that the system must limit the maximum size of a stored object. The limit is the maximum length of a monolith. This length should be configurable, and our design suggests the size be 100GB. The size depends on the used file system. However, modern file systems usually permit this size.

3.5.2 Partial Content

Although the idea of monolith files draws from the design of Haystack volumes (described in section 2.3.1), there is a fundamental difference in use. Since the system allows clients to upload large files, they could not be synchronously appended to the corresponding monoliths. Clients will also be allowed to send the partial content of their data, and they may send it in an arbitrary order.

Therefore an object is appended to a monolith only after the content is stored on the SN. When the client uploads a part of the content, SN stores the content into a separate file. The name of the file will contain the object identifier and the position and the range of the data in the original uploaded file. When SN processes client request to finalize upload, it will check, if there are missing parts of the object. If not, SN will prepare an appropriate header and a footer and appends the objects to the monolith file. After that, SN will erase the temporary upload data.

3.5.2.1 Expiration of temporary data

Storage Node deletes the temporary data during four events.

Client finished object upload – After an object is fully written to a monolith, SN deletes all temporary files.

Client deletes unfinished object – The delete is issued indirectly by MS.

Temporary files are checked at SN startup – SN checks temporary files during startup and deletes the malformed.

Temporary files expires – The client is obliged to finish upload of an object in 24 hours. If the condition is not satisfied, SN deletes the abandoned temporary files.

3.5.3 States of a monolith file

Storage Nodes will record two indices for each monolith. The former is readability of the file. The meaning is obvious. SN should not use non-readable monoliths when serving clients. The indication comes into use when SN will have to manipulate which would interrupt readers serving requests.

The latter indication will be writability of the monolith. Monolith may be sealed when its size approaches the maximum allowed limit. Although no objects will be permitted to be appended to the file, SN will be able to delete objects from the non-writable monolith.

The combination of these two indices determines four states of a monolith.

3.5.4 Selection of a monolith to store an object

An appender will be a unit capable of writing data to a monolith file. The storage node should manage an appender for each writable monolith. The appender is the only one permitted to write data to the specific monolith. Therefore objects will be appended sequentially. An appender will hold open file handler for writing.

A machine hosting a SN may have mounted multiple different disks. On the start of the SN, a configuration will specify which disks should be used for storage of the monoliths. SN will track the amount of used space in the memory.

When SN needs to append a new object, an appropriate appender will be selected. The selection consists of these steps:

1. Storage Node will choose a location location of the least used disk by the storage
2. SN searches for a writable monolith which size does not exceed the maximum limit after the object is appended
3. If there is no such file, SN creates new monolith file
4. SN issues appropriate appender to write the object
5. If the size of the monolith after write exceeds 90% of the maximum size, the monolith state will be set to non-writable, and appender will be closed.

By spreading files equally on each disk, SN distributes IO load and utilizes better disk capabilities. This distribution strategy is a heuristic that will fail if there is a disk of significantly lower capacity. This disk will be probably filled first preventing any storage of new objects.

3.5.5 Deletion of an object

By the nature of object storage, objects are expected not to be changed, respectively deleted, often. However, when SN is issued to remove the object, it must be deleted from each monolith file. Such operation would require a significant amount of disk operations if the SN had to shift all stored data after deleted object and would be unwise to be done. Instead, SN will use the flag in the header of the object introduced in section 3.5. The flag from will be appropriately set to indicate, that the following object is deleted. After SN sets the flag, it will increase counter associated with monolith recording the size of deleted objects in the file. This operation is trivial and allows adding new objects to the file at the same time.

3.5.6 Compaction of an monolith

When deleted objects occupy more than 25% of the size of the monolith file, SN should compact the monolith. Monolith compaction is an operation during which living objects are transferred to a new location, and the monolith is deleted. Example compaction is depicted on figure 3.8

SN should delete such monolith only if it satisfies three conditions:

1. Monolith is not writable
2. Deleted files occupy more than 25% of the size of the file
3. There are no readers of the file

It is important that the monolith should be in the non-writable state for two reasons. Deletion of a couple of objects when the monolith size is short would trigger unnecessary copy operations and would prevent the monolith to grow. The latter reason is obvious since there is no need to delete the file when new data are appended.

The situation is more complicated with readers. SN checks the amount of space retained by deleted object after it removes an object. At that time, there may be several clients reading data from the file. Therefore the compaction should proceed as follows:

1. SN traverses the file and collects each offset and header of each non-deleted object.
2. SN selects a new location for living objects. If there is no space in an existing monolith, a new one is created.
3. SN appends living objects to selected monolith and makes records in the database
4. SN sets the state of the source monolith to non-readable to prevent new readers to start new reads
5. SN removes records in database pointing to the file.
6. SN sets the flag in the header of the monolith to mark the whole file as deleted.
7. After all readers are finished, SN deletes the physical file

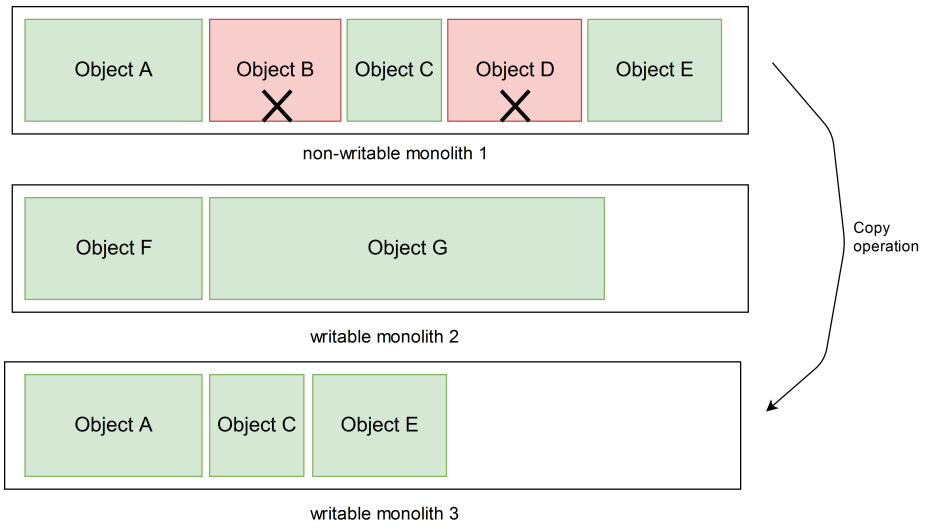


Figure 3.8: Process of compaction of a monolith and selection of a new location for the living objects

Living objects are transferred to a monolith which current size could accommodate copied data. Monolith 1 is deleted afterward.

To know when the monolith is not being read anymore, SN will have to record the number of readers for each monolith. Storage Node should add deletion flag to the header of the monolith in a case of system crash. The file will be skipped during startup scan and immediately deleted.

3.6 Communitaction protocols

The master server and the storage nodes cannot operate entirely independently in the cluster. Servers will need a communication channel to exchange information about stored objects and peers in the cluster. The design distinguishes two types of message exchanges. Unidirectional and bidirectional messages.

3.6.0.1 Unidirectional messages

Following unidirectional messages will be sent between a master server and a storage node:

- MS sends essential metadata to SN
- MS issues deletion of an object
- MS issues acquisition of a replica
- SN sends notification about successful storage of a file
- SN sends heartbeat message

None of these messages require the accepting side to respond with a value. Also, request contained in each message is idempotent, and it is safe when such message is sent multiple times. Therefore these messages will be transferred between MS and nodes over a WebSocket connection. A WebSocket connection provides a full-duplex communication channel over a single TCP connection. This approach brings two advantages. TCP protocol provides reliable message delivery, and servers will not wastefully open a new connection for each message sent.

3.6.0.2 Messages requiring a reponse

There is no case in which a master would need a direct response from a node. Following requests will be forwarded from a storage node to the master:

- SN will need to check existence and length of an object
- SN will need to acquire a list of running nodes that shares assigned group for uploaded object

The master server will expose a RESTful API providing this data. This interface will be used only internally and will be accessed solely by the storage nodes.

3.6.0.3 Joining a new storage node to the existing cluster

The master server will have to record a current number of connected storage nodes. For each storage node, the master must know its address and ports on which the node listens. The server will also track statistics about used disk space to distribute objects across the whole cluster better. A storage node must go through two stages to be fully functional. The figure 3.9 describes the process of a node joining the cluster.

1. SN makes a join request on the MS using internal REST API. The request encloses local and public IP addresses and used TCP ports. If the node was a part of the cluster in the past, it adds its unique identification.
2. The master may generate a new unique identifier for the new node. The master may also require a list of all objects currently stored on the node.
3. If the node is required to send the list, the procedure continues with the following request on master containing a list of ids.
4. The master accepts the request and returns the assigned identifier of the node. From that moment, the master considers that the node joined the cluster but does not recognize the node as functional.
5. In the next stage, connecting storage node must open a WebSocket connection. The node must send a message containing node identity so that master can pair the connected channel with the node.
6. The master receives the identification message on the newly connected WebSocket channel. If the enclosed id matches a node that is not connected and has already joined the cluster in the previous stage, the master sends a message over the channel containing confirmation of connection.
7. In this stage, the master acknowledges the node is connected and functional. It will accept messages send over the WebSocket and assign a newly created object to this node.
8. After receiving confirmation message over WebSocket, the node may start serving client requests as long as there is an open WebSocket connection.

3.6.0.4 Communication between peers

During client uploads and object replications, the storage nodes will need to exchange object data. Each transfer will have a different recipient. It would be unnecessarily complicated for each node to maintain a persistent connection with every other node in the cluster. For that reason, each data transfer will be carried out over HTTP protocol. HTTP methods are well suitable for requesting parts of data and posting data to other peers. Storage nodes will have to expose RESTful API intended for other peers. The clients will not have access to this interface.

3.6.0.5 Data transfer protocol

Transfer of object data between different nodes will not require any additional protocol on top of the HTTP. Nodes will specify sent data using HTTP URI and appropriate headers. The content of the message will be plain binary data. However, no object data will be transferred over masters internal API and WebSocket connection. The data in the message bodies will be serialized using Google Protocol Buffers. Protocol Buffers are platform neutral data interchange format [1]. Protocol Buffers serialize structured data to efficient binary represen-

[1] `https://developers.google.com/protocol-buffers/`

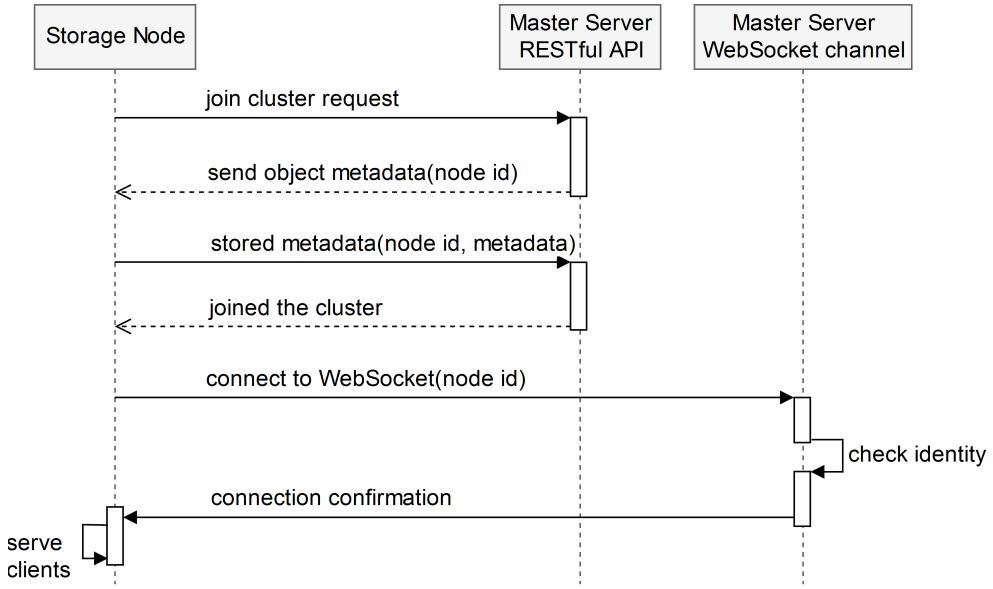


Figure 3.9: Protocol of the communication during the connection of a node to the cluster

tations suitable for data transfer. Serialization and deserialization speed is comparable to widely used JSON format[34]. The design favors the use of Protocol Buffers due to good integration with Java platform and opportunity to establish a fixed structure of messages.

3.7 Heartbeat

The master server will require periodic reports from each connected node to know the current status of the cluster.

SN will send a heartbeat every five seconds. The time period is chosen to keep the master up-to-date but not to bring a significant overhead to the nodes to prepare the message. The node will transmit the message over the open WebSocket channel. The report in a heartbeat will contain statistics about disk usage. The node will send the amount of disk space used by the monolith files and temporary files and space used occupied by temporary upload data. The node will send these metrics, although the master will know every stored object on the node and could calculate the sum of the taken space. Thhe storage node will have an internal policy of how many local copies of each object will store. When a client makes a delete request on the master, the master will not know exactly when the physical data of the object will be removed. As designed in the section 3.5, deleted data may remain stored on the node after the associated object is deleted. The master server will use the disk statistics to spread stored objects across the cluster better.

The figure 3.10 shows a heartbeat check. The master server will set a grace period of one minute for each node to send a heartbeat. If the period the time elapses, and the server will not receive any heartbeat, the master server will mark the node as disconnected and will close the WebSocket connection. The period is set to respect possible peak loads on the node and network instability.

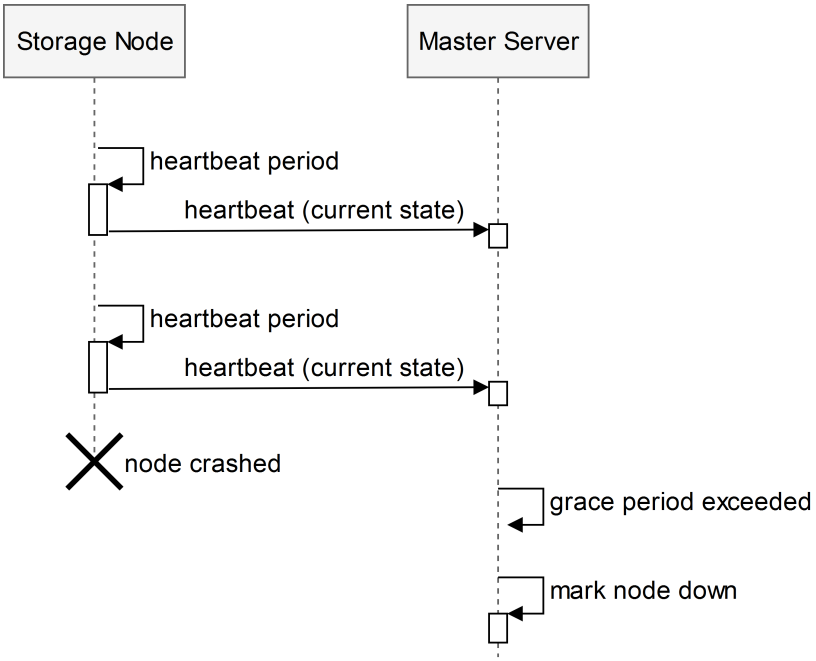


Figure 3.10: Sequence diagram of heartbeats sent from a storage node to the master server

3.8 Replication of data

The crucial requirement of the system is to protect the stored data against a loss. The system will achieve this by replicating data across different machines in the cluster. The data will be available to the client even during failure of a node. During the creation of an object, the client will define the value of replication factor. Replication factor defines how many times the object is stored in the cluster.

The system will distinguish five states of an object.

Newly assigned object – This is the initial state of an object after client issues creation request on the master. No storage holds a replica and the object is not readable.

Under replicated object – After finalization of an object or after a failure of a node, the number of replicas will most likely not match the desired replication factor. The master server must command a sufficient number of nodes to acquire a replica.

Fully replicated object – For requested replication factor, there is the same number of live replicas in the cluster. This is desired state for each object.

Over replicated object – There are more live replicas in the cluster than the set replication factor. The master server should delete the excessive.

Lost object – No SN holds the object data. Unless a node with an existing replica starts and joins the cluster, the object is lost and not recoverable.

3.8.1 Storage state check

The master server is responsible for the correct replication. The server will periodically search for incorrectly stored objects. Every five minutes, the MS will assemble lists of those objects and sends a message using the WebSocket channel to each affected node to either acquire a replica or delete it. The time period balances the involved operation overhead and the time the system may be in the incorrect state.

3.8.2 Assigment of missing replicas

The master server should not assign an object to a random node. The size of the object may vary between kilobytes to gigabytes. This may lead to a situation in which a node is tasked to acquire a series of large objects while other may collect only small one. The system should distribute the work and data evenly.

Since storage nodes will periodically send heartbeats containing used disk space. The master also will know about objects assigned to the node but not yet stored. The combination of these two metrics gives the master an estimate of what amount of disk space will be used. The under replicated object is assigned to the least filled up node. An example situation with an under replicated object is shown in figure 3.11.

The side effect of this strategy emerges during an introduction of a new empty node to the existing cluster. The master will direct most of the work to it until the node is filled similarly as others.

Each object can be assigned to a node only once. That means that if there will be fewer nodes in the cluster than replication factor, the object will never get into fully replicated state.

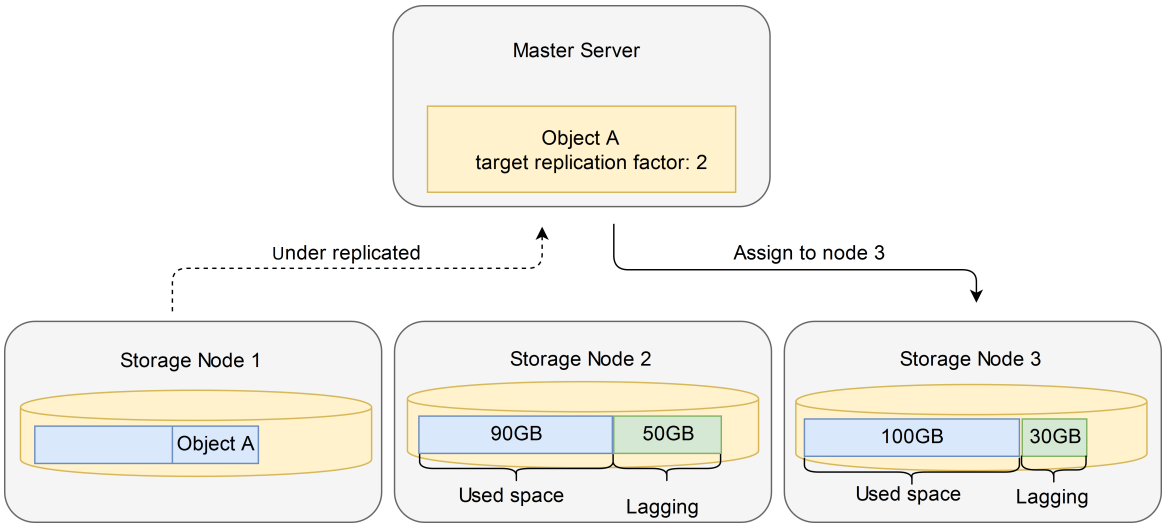


Figure 3.11: Diagram of the strategy for assignment of an under replicated object to a storage node

The object is assigned to a node that is currently not storing its replica. The node with the least amount of disk space used is selected

3.8.3 Acquisition priority

When the master server sends a node a command to acquire a replica, it will determine the priority of this operation. The goal is to lower the chance that the system will lose an object. The priority will be a ratio between replication factor and number of living replicas.

$$\frac{\text{replication factor}}{\text{readable replicas}}$$

Objects with lower replication factor will usually receive a higher priority because there is a greater chance that disconnected node will make object unavailable.

3.8.4 Deletion of redundant object replicas

During the creation of an object, the master server designates a group of nodes of size matching replication factor. When a node crashes, the master server will expand the group appropriately to match the replication factor. After crashed node restarts, there will be redundant replica stored. The master will randomly select a node that will have to delete its replica.

3.8.5 Deletion of expired objects

During the storage check, the master server will list objects that expired. An expired object is an object abandoned by the client. The client may have sent a part of the object content but never issued a finalization request on any node. After a period of 24 hours passes from the time of object creation, the master will delete the object. The master notifies every associated group to delete internally stored metadata and to delete temporary files.

3.8.5.1 End of storage check

At the end of the storage check, the master server sends a list of objects to each node. These will be assigned objects that the node did not yet acquire. Although each node may receive the same information repetitively after each check, the list will contain updated priorities. If the node failed to get a replica after the previous check, this would trigger a new attempt to get the object.

3.9 Scalability

During the design of architecture, it was important to consider scalability options. There are two scaling methods[35]:

Horizontal scaling – The scaling is done by adding new nodes to the distributed system. The system must manage nodes efficiently to aggregate the computational power.

Vertical scaling – Vertical scaling means to add additional resources to a single member of a system. It may involve an increase of CPU power, disk space and speed, or increase memory size. A scalable application should appropriately utilize additional resources and proportionally improve its performance.

3.9.1 Storage Node scalability

The current design anticipates that the system will be scaled through storage nodes. Both scaling methods can be applied to achieve a similar outcome. To increase a storage space, either a new disk could be mounted and given to use to a node, or a new instance of a node could be added to the cluster. The same principle applies to disk speed. With more disks, a node can write and read faster. Similarly, an additional instance could be deployed using the disks. The node itself is not affected much by the existence of other nodes in the cluster, therefore adding new nodes should nearly linearly increase the performance of the storage.

The number of stored objects on a single node should not significantly affect performance. However, the disk space is expected to be exhausted long before the database slows down.

3.9.2 Master Server scalability

The current design of the system prohibits the horizontal scaling of the master server as there is only one instance running in the cluster. However, the high load is put on the nodes. The only purpose of the master is to assign objects to nodes and serve metadata, occasionally sending lists of objects to nodes. A potential bottleneck of the system is the database. As the number of stored objects will increase, the queries may take more time to complete. If the system ends up slowed down by the master, the database of the objects could be sharded between several instances. The only option for the master is the vertical scalability.

3.10 Startup of a node

When a storage node starts, it must make sure, that the state stored in the database of the node is matching the content of the files on the disk. This check must precede an attempt to join the cluster and start of content serving. Traversing the whole file content may be time intensive operation. For that reason, the systen design distinguishes two possible situations:

1. The node successfully terminated during the last run
2. Previous run of the node ended up in a crash

3.10.0.1 Storage node startup after successfull termination

At startup, a node will need to know the result of the previous run. Therefore during program termination, after finishing all pending writes, the node will create a file marking the successful end. The node will search for the file during startup. If such file is found, the node will remove it and proceed with a fast scan of the disks. The procedure is following:

1. The node retrieves list of all monoliths stored in the internal database.
2. For each monolith file record, the SN searches the file system for associated file.

3. If the file is found and its length matches the stored value, the node will assume the content is identical to the records in the database. This approach is primitive. Substituting the original file with a counterfeit with the same length and file path would cause the node to serve invalid data. However, that would happen only when someone intentionally tried to damage the run of the program. The presented approach is a sufficient solution for regular use.

4. If the node does not find matching file for the record in the database, it will remove records of all files previously stored in the missing file.

5. The node searches for files that are not recorded in the database.

6. The node scans the content of each unknown file and adds found objects to the database.

7. The content of the database now matches the content of the files on the disks.

3.10.0.2 Storage node startup after a failure

After a failure of a node, the content of the database may not match the content of the monolith files. The node may have written data to the file, but the database may be missing a record about it. The node may have been interrupted during the write, and the files may be corrupted. When the gravestone file is not found during startup, a full scan of the disk is done.

1. The node deletes the content of the internal database.

2. The node searches for all monolith files on all assigned file paths.

3. The node scans the content of each monolith. Scanning requires a sequential reading of headers and skipping data content to verify constant in the footer (structure of file described in the section 3.5). If any encountered object data are malformed, the node will assume that previous write to the file was interrupted. The node will truncate the file at the position of the last successfully scanned object.

4. The node puts a record for each found monolith and each object to internal database.

5. The node will ignore and immediately delete monolith files that will contain delete flag in the monolith header. These monoliths were marked in the previous run as deleted, but because there were clients reading data from them, the node could not remove them at that time.

6. The node will inspect the temporary files and delete all with length mismatching the value coded in their name. These are the files to which the node have been writing client uploaded data during the crash.

7. The content of the database now matches the content of the files on the disks.

3.10.0.3 Missing checksum of a monolith file

The node will not compute any checksum value for stored files, although comparison of a checksum during startup would be more reliable. The computation of a checksum brings major drawbacks. Any change made in the file would invalidate the currently computed checksum. For example, during the deletion process of an object, the node effectively changes a single bit in the file to save disk operations. The second reason is, the computation of a checksum during startup would require reading the whole files.

Chapter 4

Implementation

4.1 Platform

The system is required to be platform independent and should rely only on libraries and tools which are available under an open source license. After the analysis, the Java platform was selected to implement all parts of the system. PostgreSQL[1] was selected as an object-relational database for the master server. Each node uses the H2[2] embedded database. The five major components were developed:

- Master server
- Storage node
- Common library
- Storage client library
- Console client application
- Graphical web interface

4.2 Netty I/O framework

The Netty framework was playing the crucial role during the implementation. Entire network communication is carried by the Netty. Both the master and storage nodes use the framework for decoding HTTP requests and encoding the responses.

Netty was also used for client requests. The storage nodes connect to other peers using Netty. The client library uses the framework as well.

Store nodes utilize the ability to send a part of a file without directly reading the data and loading them into the heap of JVM (Java Virtual Machine). Netty automatically transfer a file region to the connected socket using the zero-copy technique when the socket is available for writing[36].

[1] `https://www.postgresql.org/`
[2] `http://www.h2database.com/`

4.3 Master server API

The master server exposes several different APIs.

- Internal API with object and node metadata for storage nodes
- API for GUI web page AJAX requests
- Client API for the access to the storage

The table 4.1 lists methods allowed to the clients.

Method	URI	Mandatory parameters	Status	Reponse body
POST	`/object`	`length=:file-length` `replication=:rep-factor` `filename=:filename`	200 OK	JSON encoding assigned object id and storage nodes. Example shown in listing 4.1
GET	`/object/:id`	`op=finalize`	200 OK	JSON encoding storage nodes for finalization.
GET	`/object/:id`	none	200 OK	JSON encoding list of connected nodes with replica
DELETE	`/object/:id`	none	200 OK	empty

Table 4.1: Table of methods available to client exposed by the master server

Listing 4.1: Sample response from the master to a client containing assigned group of nodes to an object in JSON format

```
{
  "objectId": "1c7387fc4d229d6e",
  "availableNodes": [
    {
      "address": "203.0.113.1",
      "port": 10201
    },
    {
      "address": "203.0.113.2",
      "port": 10001
    }
  ]
}
```

4.4 Storage Node API

The storage nodes also have interfaces for both internal and external use, but they lack methods for web monitoring. The methods expected to be called by the clients are listed in the table 4.2.

Method	URI	Headers	Body	Status	Reponse body
PATCH	`/object/:id`	`content-length`, optionally `range`	file data	200	empty
PUT	`/object/:id`		empty	202 or 404	empty or missing ranges. Example in listing 4.2
GET	`/object/:id`	optionally `range`	empty	200	object data

Table 4.2: Table of methods available to client exposed by a storage node

Listing 4.2: Sample response from a storage node containing missing parts of an object in JSON format

```
{
  "rangeList": [
    {
      "from": 0,
      "to": 30
    },
    {
      "from": 40,
      "to": 1024
    }
  ]
}
```

4.5 Common library

The master server and storage node share some parts of functionality. A Java `jar` library was created to avoid duplicate code. The library also contains definitions of all Protocol Buffer messages and associated utility classes for convenient message serialization and deserialization.

4.6 Client library

Both master and nodes expose RESTful API to clients. If the response is not containing data, it is in the JSON format. The format is platform and language independent, and every major

programming language ecosystem provides several frameworks and libraries to communicate with such APIs. However, the implemented system may require multiple requests to upload a large file. For that reason, a Java client library capable of upload, download, and deletion of an object was implemented. The library splits large files during an upload and transfers them in parallel. The library handles server errors and retries an upload on another node of the cluster.

4.7 Console application

A small console application for convenient access to the storage was implemented. The application internally uses the client library. The console application has two modes of operation. The user may start an interactive session or invoke a single command from the command line providing sufficient program arguments. The usage of the application is shown in the figure 4.1.

```
Scalable Storage Console Client
Available commands

upload          <filename>  [replication-factor]   upload file
download        <object-id> ["destination"]       downloads object with optional custom destination
delete          <object-id>                       delete object
help|?|:h                                         show help
exit|quit|:q                                      exit

Command line flags
-i --interactive                                  interactive mode
-m --master     <ip>                              address of master server
-p --port       <port>                            port of master server
```

Figure 4.1: Screenshot of the printed help of the console client application

4.8 Graphical interface

One of the system requirements was that a user must be able to monitor the state of the storage using a web interface. For that reason, a single page application listing stored objects and connected nodes was implemented. The application uses *React*[3] framework. The components of the application fetch data using *AJAX* calls from the master server. The master server offers a couple of methods to fetch *JSON* data about particular objects and nodes. The *React* application is assembled to only three files which the master server sends to the web browsers:

- HTML index file
- Javascript file with the application logic
- CSS stylesheet file

The user can see the current location and replication state of each object in the graphical interface. The figures 4.2, 4.3 and 4.4 show a sample listing. The web page can be accessed from the browser on a default address `http://masterIP:9001/monitoring`

[3] `https://facebook.github.io/react/`

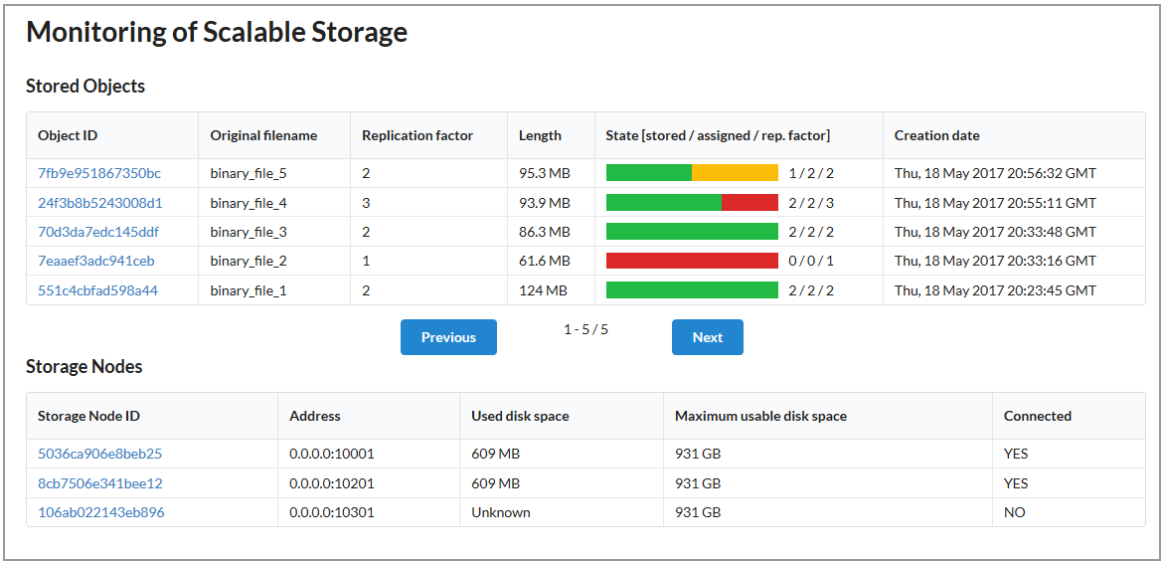


Figure 4.2: Screenshot of GUI with a sample listing of currently stored objects

The state column shows the current status of a object.

- The object with the name `binary_file_5` is currently assigned to two nodes and one of them is already holding a replica.
- Since there are only 2 connected nodes, the object `binary_file_4` with replication factor 3 cannot be fully replicated.
- The node storing object with the name `binary_file_2` was disconnected and replication factor was set to 1, therefore the replica is lost.

4.9 Dependency injection

Both master and node rely on *Guice*[4] dependency injection framework. The framework is lightweight and configuration is done programmatically. The master server utilizes Guice's support for Java Persistence API. The Guice takes care of commits of transactions[37].

4.10 Configuration

Both master and storage node are configured using *Typesafe Config*[5] library. The configuration files are written using *HOCON* [38] syntax, which stands for *Human-Optimized Config Object Notation*. The language is a superset of JSON notation. The library provides two benefits. During startup, the configuration file is mapped in a type-safe manner to a Java class. If there is a missing value, the program gracefully terminates itself. The latter advantage is that any configuration value may be replaced during launch using Java system properties passed as a command line argument of the program.

[4] `https://github.com/google/guice`
[5] `https://github.com/typesafehub/config`

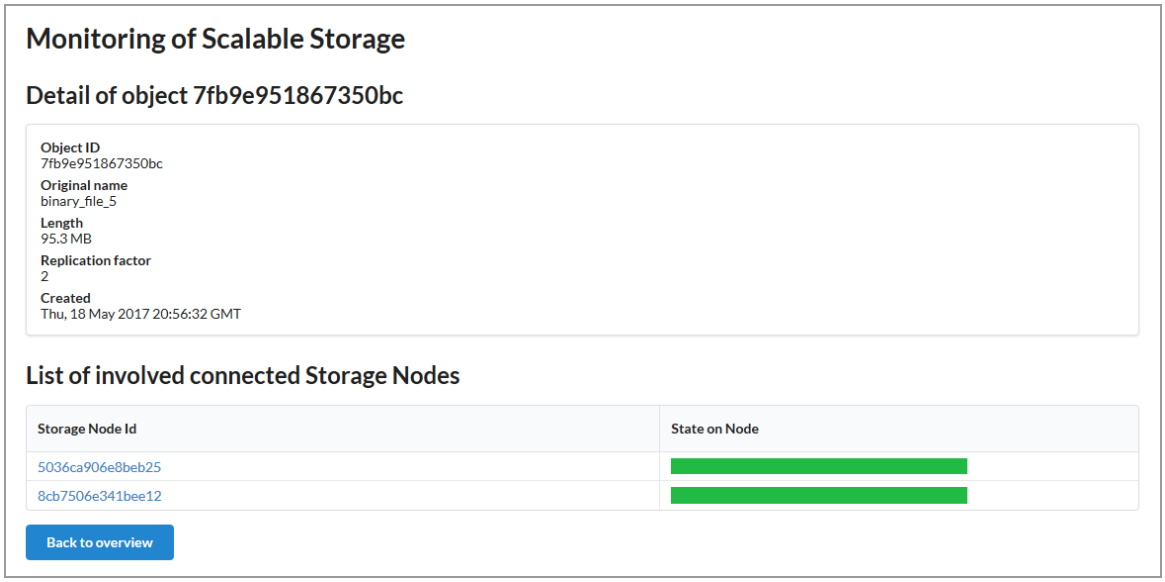


Figure 4.3: Screenshot of GUI showing details about stored object

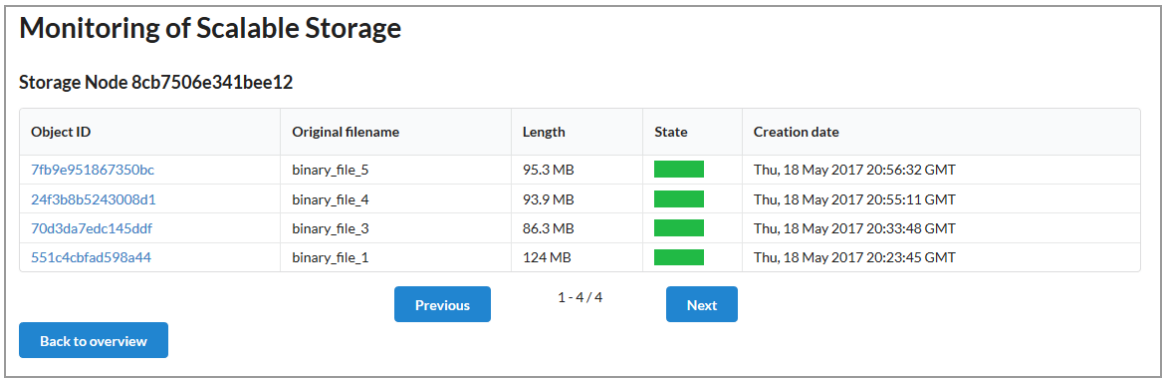


Figure 4.4: Screenshot of GUI with a listing of stored objects on a single node

4.11 Logging

It is difficult to observe what is happening in the distributed system while clients concurrently access it and individual members of the system communicate with each other. To see better what is happening, all servers log their activity using *logback*[6] framework. The *logback* implements the SLF4J API[7]. The server created logs and logs produced by used libraries using the same common API are gathered into one place. However, the JDBC driver used during implementation for connection to PostgreSQL database was still using the legacy *java.util.logging* API and had to be bridged[39] to the *Logback*.

[6] `https://logback.qos.ch/`
[7] `https://www.slf4j.org/`

Chapter 5

Testing and results

The important part of the development of the distributed storage was testing. The system should reliably handle user data. Therefore verification was needed, that the designed behavior was correctly implemented. Two different approaches to testing were selected. In a sample distributed environment was tested the overall performance of the system. Next, several test cases were prepared to inspect behavior of both master server and individual storage nodes.

5.1 Performance testing

The testing was performed in a distributed environment simulating a possible real deployment. The cluster was composed of four credit card-sized computers. There was one computer with the role of master and three separate storage nodes each equipped with a regular magnetic disk. The client application was running on a desktop computer. The computers were located in a private network. The table 5.1 captures the relevant hardware specifications of the used components. The figure 5.1 describes the topology of the computers in the network.

For each test, the network traffic was measured on the client computer. On each storage node was measured disk activity. The measurement procedures of the monitoring tools were different. The traffic was measured with Linux tool *ifstat* showing the amount of incoming and outcoming data. The disk statistics were collected with tool *iotop* sampling actual disk read and write speed. For that reason, the measured disk statistics does not completely match the network speed.

	Master / Stogage node	Test client PC
Processor	Dual-core 1GHz	Intel i5-6500 3.2GHz
Memory	1GB DDR3	8GB DDR4
Disk	1TB 5400 RPM	—
Network interface	1Gbit/s	1Gbit/s
OS	Debian 8 based Linux	Windows 10
Environment	Java 8, PostgreSQL 9.6	Java 8

Table 5.1: Table of hardware specification of the computers used during the testing

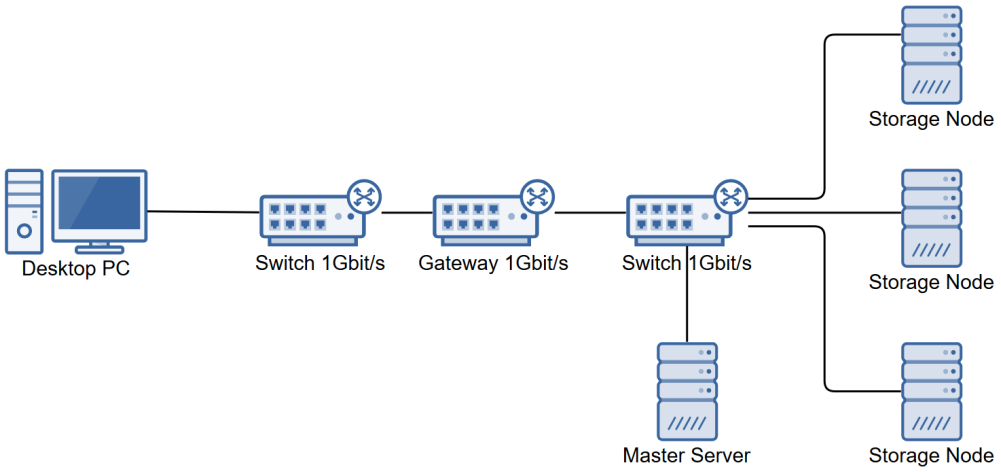


Figure 5.1: Diagram of the network connection configuration of the computers used during the testing

5.1.1 Write tests

The first performance tests aimed to measure a write throughput of the test configuration. Several clients were dispatched to upload simultaneously a larger amount of files. The test was parametrized followingly:

- 10 concurrently operating clients
- 1000 files with a total size of 2GB
- Replication factor set to 3

The results of the test are listed in the table 5.2. It took a significant time to make all objects readable after their upload. Despite the fact that clients uploaded data to three different disks, quite low performance was observed. The behavior is attributed to the nature of magnetic disks and the size of the selected uploaded file. A regular disk is capable of write and read speed of roughly 120MB/s in case of sequential access. However, during random write and read access of small files, the performance drops bellow 1MB/s [40]. During this test, the nodes had to write and access a significant number of smaller files causing many seeks on disks.

Upload duration	250s
Average disk write	11,406KB/s per node
Average upload speed	8,174KB/s
All objects readable	420s
All objects replicated	454s

Table 5.2: Results of the first write performance test

After the test, a second test was run with a different replication factor. The factor was set to 1 to eliminate data transfers between individual nodes. The results are listed in the table 5.3. Since the user data did not have to be shared between nodes, the write throughput was higher than in the previous test.

Upload duration	110s
Average disk write	6,360KB/s per node
Average upload speed	18,588KB/s
All objects readable	110s

Table 5.3: Results of the second write performance test

5.1.2 Read tests

The second group of tests measured the speed at which storage can serve the objects. The configuration was similar to previous tests.

- 10 concurrently operating clients
- 1000 files with a total size of 2GB

The results are captured in the table 5.4. In the following test, the size of objects was reduced. The clients were tasked to download 5000 files each of size of 200kB. The results of the second read test are in the table 5.5. The decrease in performance is credited both to greater overhead needed for request serving and a larger amount of disk seeks.

Test duration	24s
Average disk read	22,307KB/s per node
Average download speed	88,711KB/s

Table 5.4: Results of the first read performance test

Test duration	28s
Average disk read	18,952KB/s per node
Average download speed	37,862KB/s

Table 5.5: Results of the second read performance test

5.2 Functional testing

Client actions trigger series of events in the system. Testing each of them would involve a creation of a simulated context with mock requests and responses from other members of the cluster. However, such test would not verify, if the other participants would send such messages. For that reason, a series of behavioral tests were created. The system was tested

as a black box from the perspective of a client with a minimal understanding of the internal structure.

The main advantage of implemented tests is that they are fully automated. After a change is introduced in the source code, these test can be simply launched from a command line to show, if there is a regression in the behavior of the system. The scripts are able to start multiple instances of servers. After a test is finished, the scripts clean up files created by the servers. The databases of storage nodes and stored monolith files must not interfere with future runs of the tests.

A couple of simulated scenarios covering the most of the functionality of the distributed storage were created.

5.2.1 Create, read and delete scenario

The goal of the test is to verify that a client can upload data to storage, data are replicated and can be read and deleted. The test starts with an instance of the master server and three storage nodes. The tasks are following:

1. Upload a file with the replication factor 3.
2. Download and compare content from each node.
3. Issue delete of the object on the master server
4. Try to read the deleted object directly from each node. No data should be served.

5.2.1.1 Delete after node shutdown scenario

The aim of the tests is to verify that a node synchronizes with the changes of state of the storage made during its absence. In the beginning, a master server and single node are started. Tasks are following:

1. Upload two files to the storage.
2. Shutdown the storage node.
3. Issue delete of the first object on the master server.
4. Restart the shutdown storage node.
5. Try to download the first object directly on the node. No data should be served.
6. Try to download the second object. The node should return correct data.

5.2.1.2 Replication balancing scenario

In this scenario, the master server should correctly fulfill desired replication factor of the stored object. The test starts with an instance of the master server and two storage nodes.

1. Upload a file with replication factor 2.
2. Launch third instance of a storage node.
3. Terminate one node holding a replica of the object.
4. Await cluster rebalancing.
5. Tyr to download the object from the new node. The node should return newly acquired data.
6. Restart the killed node
7. Await cluster rebalancing
8. Try to download the object from each node. Only two of them should return object data.

5.2.2 Manual testing

There are cases, which are quite difficult to reliably script for automatic evaluation. These situations involve failures of individual members of the cluster. These cases were tested manually using the console client application and the client library. During the tests, server logs and presented state in web GUI were observed. The servers were manually interrupted during their operations. The goal was to observe following behaviors:

1. After a failure and a restart of the master server, each storage node can reconnect.
2. The master server notices when a node is disconnected.
3. A client can successfully switch upload server after the first one fails.
4. A storage node sharing currently uploaded data to other nodes can handle a disconnection of a peer.
5. After a restart, a storage node should delete uploaded temporary data of interrupted client requests.
6. After a restart, a storage node should discover and correctly truncate damaged monolith files.

5.2.3 Test results discussion

The behavioral tests verified that a client could upload, download and delete data from the storage system and the data are correctly replicated. The performance tests showed satisfactory performance during reads. The write tests revealed a flaw in the current implementation. The storage node does not throttle the throughput of incoming data. The more data are uploaded, the longer it takes to make them readable. Ideally, the node should limit incoming traffic after a queue of finalization operations surpasses a certain limit. The test setup amplified the flaw. If the nodes had a separate disk for temporary data and monolith files, the queue could be emptied much faster.

Chapter 6

Conclusion

In this theses, it was designed and implemented a simple storage system that can store and serve binary data in the form of the objects. A client communicates with the system using a RESTful API.

The architecture divided work of the storage between a central master server that manages object metadata, and a set of store nodes keeping the data itself. The implemented system achieved horizontal scalability with increasing the number of storage nodes in the cluster. Implemented system is platform independent due to selected technologies.

Both master server and individual storage nodes can deal with the lost connection between each other. Servers automatically reconnect and resume their operation.

The system is capable of replicating client data across different storage nodes to prevent a data loss. The system maintains the target number of replicas which could be different for each object. The missing replicas are replaced and the redundant deleted. After a storage node failure, the server deals with malformed data and synchronize with the current state of the storage cluster.

System exposes a graphical interface served as a web page to show the current state of the storage. For testing purposes, it was implemented a sample application that utilizes RESTful API of the system and can upload, download and delete data from the data store.

The behavior of the system was verified by a set of scripts and manual tests. In an experimental distributed environment performance was measured and the results are included in this work.

The functionality of the system covers all goals and requirements of this thesis.

6.1 Future work

During the design and implementation, some options for improving the system have emerged. However, all improvements would be beyond the scope of this work.

6.2 Security

The current implementation does not address any form of security of the system. All network communication is transferred using non-secured HTTP. The system could be improved to

use HTTP over SSL (Secure Sockets Layer). The designed system does not manage any client accounts to impose access restrictions on client data.

In the situation the whole system is residing on a private network, an SSL termination proxy may expose the API of the storage to the wide internet. Therefore internal communication between storage components could remain unencrypted.

6.3 File compression

The current design does not include data compression. Although the compression would save disk space, it would also be more CPU intensive. The system is required to run on computers with low performance. The compression would also make reading data more difficult. The client is allowed to request any part of a stored object. This partial read would require starting reading the object from the beginning to decompress it.

6.4 Periodical file checks

Future improvement of the system could add periodical scans of the store. Stored files could be regularly checked to find damaged files or to identify malfunctioning disks. A probe would calculate a checksum for each found object and compare it with a stored value.

Bibliography

[1] Michael Factor et al. "Object storage: The future building block for storage systems". In: *Local to Global Data Interoperability-Challenges and Technologies, 2005*. IEEE. 2005, pp. 119–123.

[2] Tony Piscopo Yadin Porter de León. *Object Storage versus Block Storage: Understanding the Technology Differences*. URL: `https://www.druva.com/blog/object-storage-versus-block-storage-understanding-technology-differences/` (visited on 05/20/2017).

[3] Michael J. Brim et al. "Asynchronous Object Storage with QoS for Scientific and Commercial Big Data". In: *Proceedings of the 8th Parallel Data Storage Workshop*. 2013, pp. 7–13. ISBN: 978-1-4503-2505-9.

[4] Wikipedia contributors. *Binary large object*. URL: `https://en.wikipedia.org/wiki/Binary_large_object` (visited on 05/20/2017).

[5] Doug Beaver et al. "Finding a Needle in Haystack: Facebook's Photo Storage". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. 2010, pp. 47–60.

[6] Brad Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 143–157. ISBN: 978-1-4503-0977-6.

[7] Qi Huang et al. "An analysis of Facebook photo caching". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 167–181.

[8] *Azure Products*. Microsoft. URL: `https://azure.microsoft.com/en-us/services/` (visited on 05/20/2017).

[9] Chris Lu. *SeaweedFS*. URL: `https://github.com/chrislusf/seaweedfs` (visited on 05/15/2017).

[10] Diego Ongaro and John K Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[11] Robert Greiner. *CAP Theorem: Revisited*. URL: `http://robertgreiner.com/2014/08/cap-theorem-revisited/` (visited on 05/15/2017).

[12] Werner Vogels. "Eventually Consistent". In: *Commun. ACM* 52 (2009), pp. 40–44.

[13] Jonathan Ellis. *Lightweight transactions in Cassandra 2.0*. URL: `http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0` (visited on 05/15/2017).

[14] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[15] Kyle Kingsbury. *Jepsen: MongoDB*. URL: https://aphyr.com/posts/284-call-me-maybe-mongodb (visited on 05/15/2017).

[16] The Linux Information Project. *Journaling Filesystem Definition*. URL: http://www.linfo.org/journaling_filesystem.html (visited on 01/20/2017).

[17] *NTFS Journaling*. URL: http://ntfs.com/transaction.htm (visited on 01/20/2017).

[18] Wiki contributors. *Ext4 Disk Layout*. URL: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout (visited on 05/13/2017).

[19] Wikipedia contributors. *NTFS*. URL: https://en.wikipedia.org/wiki/NTFS (visited on 05/20/2017).

[20] *How NTFS Works*. Microsoft. URL: https://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx (visited on 05/20/2017).

[21] *Which programs are fast?* URL: http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html (visited on 05/07/2017).

[22] *ECMA-335-Part-I-IV - ECMA-335, 1st edition, December 2001*. URL: http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-335,%201st%20edition,%20December%202001.pdf (visited on 05/07/2017).

[23] *OpenJDK*. Oracle. URL: http://openjdk.java.net/ (visited on 05/21/2017).

[24] *Java Platform, Standard Edition 8 Reference Implementations*. URL: http://jdk.java.net/java-se-ri/8 (visited on 05/07/2017).

[25] *Performance Comparison*. URL: http://www.h2database.com/html/performance.html (visited on 05/07/2017).

[26] Dan Kegel. *The C10K problem*. URL: http://www.kegel.com/c10k.html (visited on 05/07/2017).

[27] *Netty, Related projects*. URL: https://netty.io/wiki/related-projects.html (visited on 05/07/2017).

[28] *Web Framework Benchmarks*. URL: https://www.techempower.com/benchmarks/#section=data-r13&hw=ph&test=plaintext (visited on 05/07/2017).

[29] Daniel Bimschas. *Zero-Copy Event-Driven Servers with Netty*. URL: https://www.slideshare.net/danbim/zerocopy-eventdriven-servers-with-netty (visited on 05/07/2017).

[30] *UnmanagedMemoryStream Class*. Microsoft. URL: https://msdn.microsoft.com/en-us/library/system.io.unmanagedmemorystream(v=vs.110).aspx (visited on 05/07/2017).

[31] R Fielding, Yves Lafon, and J Reschke. *Hypertext transfer protocol (HTTP/1.1): Range requests*. Tech. rep. 2014.

[32] Kai Chen et al. "Understanding bandwidth-delay product in mobile ad hoc networks". In: *Computer Communications* 27.10 (2004), pp. 923–934.

[33] Paul J Leach, Michael Mealling, and Rich Salz. "A universally unique identifier (uuid) urn namespace". In: (2005).

[34] Bruno Krebs. *Beating JSON performance with Protobuf*. URL: `https://auth0.com/blog/beating-json-performance-with-protobuf/` (visited on 05/15/2017).

[35] Wikipedia contributors. *Scalability*. URL: `https://en.wikipedia.org/wiki/Scalability` (visited on 05/15/2017).

[36] Norman Maurer. *Interface FileRegion*. URL: `https://netty.io/4.1/api/io/netty/channel/FileRegion.html` (visited on 05/18/2017).

[37] Sam Berlin. *Using JPA with Guice Persist*. `https://github.com/google/guice/wiki/JPA`. 2014. (Visited on 05/18/2017).

[38] Havoc Pennington. *HOCON – Informal Specification*. `https://github.com/typesafehub/config/blob/master/HOCON.md`. 2017. (Visited on 05/18/2017).

[39] *Bridging legacy APIs*. Quality Open Software. URL: `https://www.slf4j.org/legacy.html` (visited on 05/18/2017).

[40] *WD Blue 1TB (2015) - Benchmark*. Microsoft. URL: `http://hdd.userbenchmark.com/WD-Blue-1TB-2015/Rating/3520` (visited on 05/20/2017).

Appendix A

Nomenclature

ACID Atomicity, Consistency, Isolation, Durability

API Application programming interface

ASCII American Standard Code for Information Interchange

BLOB Binary Large Object

CDN Content delivery network

FS File System

FUSE Filesystem in Userspace

GUI Graphical user interface

HTTP Hypertext Transfer Protocol

IP Internet Protocol

JDBC Java Database Connectivity

JPA Java Persistence API

JSON JavaScript Object Notation

JVM Java virtual machine

MFT Master File Table

NFS Network File System

POSIX Portable Operating System Interface

REST Representational state transfer

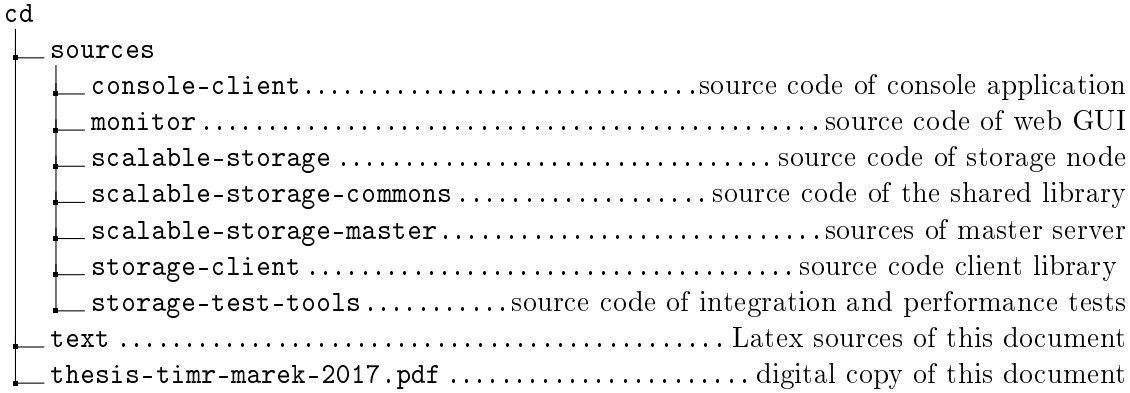
SSL Secure Sockets Layer

TCP Transmission Control Protocol

UUID Universally unique identifier

Appendix B

Content of the attached CD



B.1 Installation manuals

Each directory with the source codes includes a `README.md` file. The file contains a list of required software and steps for build and run of the software.

Appendix C

Open-source licenses

Library	License
Apache Commons IO	Apache License 2.0
Apache Commons Lang	Apache License 2.0
Guice	Apache License 2.0
Hibernate	LGPL 2.1
HikariCP	Apache License 2.0
H2	dual licensed under MPL 2.0 and EPL 1.0
Jackson Databind	Apache License 2.0
Logback	dual licensed under EPL v1.0 and LGPL 2.1
Netty	Apache License 2.0
PostgreSQL	The PostgreSQL Licence similar to MIT license
PostgreSQL JDBC driver	BSD 2-clause
Protocol Buffers	custom permissive license
React	BSD 3-clause
SLF4J	MIT License
sql2o	MIT License
Typesafe Config	Apache License 2.0

Table C.1: Table of open-source licenses of used libraries and programs