Faculty of Science and Technology
Department of Computer Science

Seadrive

Remote file synchronization for offshore fleets

—

Peter Haro

INF-3981 Master's Thesis in computer Science - May 2016

“Sometimes a scream is better than a thesis”
–Ralph Waldo Emerson

Abstract

File synchronization- and hosting services is not only an integrated service in everyday life, but also a powerful tool to support business and organizational activities. In order to provide users with a transparent experience, the systems relies on sophisticated mechanisms to create a seamless integration. The problem with these systems is that they are designed for stable network connections with a low variety in latency, throughput and loss-rate. The systems optimized for low bandwidth networks are implemented to work on a small set of small text-based files, and assumes no prior knowledge of the contents on the receiver.

Offshore vessels outside the range cellular networks employ a variety of satellite based communication suites and accommodating physical hardware. These networks are notorious for having poor upload- and download speed, high loss rate, poor latency with high variability and are subject to frequent dropped connections. Furthermore, the fiscal cost associated by using these connections are high, as the highest performing networks charge per kilobit transferred. These connections are unsuitable for modern file hosting services, and file synchronization frameworks, as they never complete synchronizing, often due to the assignment of new IPs.

Therefore providing the naval fleet with a reliable file-synchronization protocol, and small in transmission overhead is of the utmost importance. In order to facilitate the needs for file hosting services, we created a file synchronization framework, which allows for different deduplication, file-synchronization and file transportation schemes. The idea was to support a computationally inexpensive method emphasizing speed over reliability on Local Area Networks, and a robust but slower methodology for Wide Area Networks.

This thesis presents Seadrive- a new file synchronization framework that targets offshore-based fleets and their land-based counterparts. By utilizing a file synchronization methodology inspired by binary patch distributions, and creating a novel reliable application level transport protocol, we are able to successfully synchronize large files through simulated satellite-based network topologies. In order to assess the capabilities of our framework, we performed various experiments on the artifacts in the form of micro- and macro benchmarks, comparing them to both Rsync and Rdiff based protocols.

Our results show that Seadrive is able to produce smaller patches than both Rsync and Rdiff based protocols, with fewer TCP and application layer requests necessary, saving up to 10 hours on the slowest network connection and is able to reliably transfer data through unreliable network topologies.

Acknowledgements

I would like to express my first and foremost gratitude to my advisors Professor Otto Anshus, Svein Bertheussen, Vidar Berg and Asbjørn Pettersen, for your guidance, support and valuable insights. I would also express my thanks to Dualog, for creating this project and providing an office for a measly student.

On a more personal level I would like to thank my colleges at SINTEF Nord, especially Bård Hanssen for providing good humor and being my personal scapegoat, and of course the coffee machine providing me with necessary life support.

Finally, I would like to thank my fiancée Maria Brattfjell and my family, for showing loving support throughout my madness.

Contents

List of Figures

List of Tables

List of Listings

List of Code Snippets

List of Abbreviations

ACM	Association for Computing Machinery
AFS	Andrew File System
API	application programming interface
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
D3	Data-Driven Documents
DAL	Data Abstraction Layer
DLL	Dynamic Link Library
GUID	Globally Unique Identifier
HTML5	version 5 of the HyperText Markup Language standard
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
IOCP	Input/Output Completion Port
IP	Internet Protocol
KB	Kilobyte
kb	Kilobit
LAN	Local Area Network

LIFO	Last In First Out
LSP	Local Synchronization point
MB	Megabyte
MS	Milliseconds
NFS	Network File System
OS	Operating System
RAM	Random Access Memory
RPC	Remote Procedure Call
RSP	Remote synchronization point
RTT	Round trip time
SQL	Structured Query Language
TCP	Transmission Control Protocol/Internet Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UiT	University of Tromsø
UUID	Universally Unique Identifier
VLC	Variable Length Chunking
WAN	Wide Area Network
WLAN	Wireless Local Area Network

1 Introduction

Modern naval- and fishing fleets utilize a multitude of various information systems from different sources when planning and executing offshore operations. They are equipped with several sensors and instruments, which provide a constant stream of information regarding various on-board systems, of which some are readily available to the crew and actively employed during an operation. The governing cooperation of large naval-, fishing-, oil-, etc. fleets requires information to flow from their management system(s) to their fleet in a robust manner, likewise the fleets have information required by the governing entity. The information necessitates the need to differentiate between different recipients; the captain requires some documents, while crewmembers have differing needs.

Important legislative rules, regulation certificates and other documents have real-time constraints for reaching the fleet, and must therefore reach their intendent destination before a given deadline. In addition, skippers have the same real-time demands to deliver documents to the governing body.

The information flow in today's systems consists primarily of e-mail exchange, however these systems have been deemed unsuccessful, because end users report them as unsatisfactory for the following reasons:

- Important information is lost in the copious amount of other e-mails
- Documents arrived during shift A are often not read by shift B

- The systems are slow
- The files sent through these systems are constrained not only by file-type, but also their size

When important information is lost, the results can vary from small trifles to disasters, such as overfishing, not following updated safety regulations or monetary losses. Therefore, entities that manage large fleets have experimented using cloud-based file-synchronization frameworks to deliver data to the end-users. Although there exists a myriad of file synchronizations frameworks and file hosting services such as "Dropbox", "Seagate", "Box", "Wuala", and "Google drive", to name a few, the systems do not function correctly for ships connected to the internet through satellite based network-links. The connections are plagued with high packet loss and frequent dropped connections, which causes most file-synchronization services simply to restart the transmission from the start. This causes files to never be synchronized from and to vessels offshore. Furthermore, some systems simply just stalls, getting stuck at certain points and never synchronizing themselves, possibly due to new Internet Protocol (IP) addresses being assigned.

To provide a useable framework for laymen to dispatch, read and update various data files, in order to accurately disseminate the required information to end-users, we propose a file-synchronization application and framework: Seadrive. The primary objective of the Seadrive project is to establish a robust file synchronization framework to disseminate data in a many to one, and one to many relationships. In order to achieve the objective, the application must support stable transmission over unstable network connections, allow for retransmissions of a subset of a file, and be able to rebuild the file regardless of type after transmission.

In this thesis we will explore a subset of the functionality provided by Seadrive, we will show how the system can manage file-synchronization on unstable network links with low bandwidth, with emphasis on the file synchronization mechanisms used to provide reliable data transfer. We outline this functionality as the transport protocol, and in chapter 7.1, we will provide the lessons we learned during this process.

1.1 Problem definition

In this thesis, we consider the problems of designing and implementing a file synchronization framework for creating the Seadrive application. The framework consists of several interconnected modules, such as file synchronizers, transport protocols and methodologies for facilitating file hosting services. For this thesis, we delimit ourselves to focus primarily on the file-synchronization mechanisms required for offshore data communication through satellite based network bearers. Although usability through accessible interfaces in order to serve data to consumers is of the utmost importance, it is not an exact measurable metric. Due to this nature, we do not perform usability tests nor do we consider the look and feel of the application. However, we do examine the underlying mechanisms required to support data synchronization, both local and remote, therefore our thesis is that:

> *An effective framework for synchronizing files between vessels and ship owning companies can be implemented automating file system management for usage in real-time applications and operations.*

To support our thesis, we have built a prototype allowing us to perform experiments to determine the system characteristics, performance and problems. The prototype allows us to study the communication protocol in micro and macro scale. We measure the latency in clock-time to perform small operations using micro-benchmarks, and perform macro-benchmarks to determine Central Processing Unit (CPU)-time, Random Access Memory (RAM) utilization and latency in clock-time to synchronize files. We also evaluate our methods by comparing them to two modern Rsync and Rdiff based protocols in order to determine the necessary data size required to synchronize files.

1.2 Targeted Applications

We do not believe it is tractable to build a feature-complete File Hosting Service application with file synchronization as the main objective, while achieving full interoperability across different files, file-systems and Operating System (OS)s, while having optimal traffic patterns for each case. However, we believe it possible to create a Windows-based file synchronization framework to support the most frequent usages of these frameworks with significantly less effort. In this thesis, we do not aspire to recreate modern frameworks such as Dropbox, Seafile, etc. Instead, our objective is to create a file synchronization framework optimized for offshore vessels, with emphasis on the transportation of data between sender and recipient.

The core functionality of the framework, and transport protocol, by themselves imposes no limitation of which algorithms are used, or functionality which may be implemented in the future. However, the server and client communicator(s) are restricted to windows as they utilize Input/Output Completion Port (IOCP)[1] to leverage the capabilities of multiprocessor systems. Furthermore, the algorithms are not designed for speed as every file is treated as binary chunks of data, and subsequently not exposed to any optimizations based on file-type.

We have in this thesis focused exclusively on achieving full compatibility with all file types on windows systems after 9X versions, as it is the most used system in our focus domain. We target compatibility for our entire framework using x64[1] architectures.

1.3 Contributions

The contributions of this work are:

- Principles
  - A file hosting service can reduce the amount of file synchronizations by utilizing a set of trusted servers in a centralized Approach. This reduces the problem size by $O(1 - (\frac{N_{lsp}}{N_c}))$
    - N_{lsp} - Number of Local Synchronization points
    - N_c - Number of clients requiring the synchronized file
  - Synchronize as much as possible locally, short traveled data and computations.
- Models for
  - File synchronization – Parallel synchronization based on signature systems and Delta-Differential
  - Remote data transport protocol for robustness
  - Designing a rudimentary file-synchronization framework with emphasis on the remote transport protocol
- Artifacts
  - Seadrive Framework

1. Note that we will use the term x64 throughout the entirety of this thesis to denote both the Intel-86-64 and AMD64 platforms collectively

    * A business layer to manage cross cutting concerns
    * A Data Abstraction Layer for data management
    * A implementation and concretization of the framework
    * An experimental project to perform experimental evaluation
  - An implementation of the remote and local synchronization points in order to realize the system architecture

- Lessons learned
  - New approaches to file synchronization might be more suited for the offshore domain
  - Input/Output (I/O), memory and CPU overhead is often negligible in the context of very poor network connections
  - Established frameworks are not always the optimal solution

1.4 Methods and materials

1.4.1 Methodology applied for this thesis

Computer science is one of the youngest science disciplines, having evolved for just over 60 years into what it is today. In 1989 the *Task Force of the Core of Computer Science*, formed by the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) Computer Society; provided us with a definition of computer- science and engineering: "Computer science and engineering is the systematic study of algorithmic processes-their theory, analysis, design, efficiency, implementation, and application that describe and transform information. . . " [5]. They conveyed this definition in their final report, which deduced an intellectual framework for the disciplines of research within the computing field. The report also identified three paradigms, which forms the basis of computer science research: *theory*, *abstraction*, and *design*

Theory is an iterative process rooted in mathematics, which is based on the idea of characterizing the objects of the study to create a definition and hypothesizing among their possible relationships to provide a theorem. The relationships provided in the theorem are thus analyzed to be proven or disproven and the results are evaluated.

Abstraction outlines an experimental scientific method aiming to examine phenomenon's using an iterative method. The method forms hypotheses to construct models, which are challenged by experiments in order to make a prediction. The data collected from experiments is vital in this stage, as the hypothesis are not mathematically proven.

Design is the final paradigm and has its roots in engineering, where system requirements and specifications are defined, the systems designed, implemented and tested. Like the other paradigms, it is an iterative process, which lasts until the system fulfills the requirements.

This thesis is rooted in the area of Information Systems Research[26], which covers certain aspects of all three paradigms. The initial stages of this thesis were composed of compiling existing understanding and theory, in order to provide a solid theoretical fundament. By using this knowledge, we were able to devise requirements for our system, and design components to complete the specification devised by the requirements. Aided by theory and abstraction we were able to implement our framework for file synchronization, and by following the iterative process, we successively increased our knowledge in this domain. We respectively obtained more knowledge on the intricacies of file synchronization frameworks; we were subsequently able to discover new requirements and refine existing ones, thus allowing us to implement functionality to satisfy these. Finally, we evaluate our work experimentally showing its capabilities and assessing its efficiency, using a quantitative analysis.

1.4.2 Procedures

We performed two micro benchmarks on the file-synchronization methodology (some refer to these methodologies as deduplication as well), provided by the artifacts, with each benchmark being run 100 times to generate consistent results. The experiments were conducted on a newly rebooted Windows 7 installation, booting in selective startup with only the system services loading using the original boot configuration for the machine. For the micro-benchmarks, we measured the latency, i.e. the real time an operation took to complete, and the file-size generated by these commands. The primary dataset used was the files located in the folder DATA/testFiles.

We also performed macro-benchmarks in order to evaluate operations integral to the file synchronization framework, with metrics such as Packets per second, RAM and CPU usage and the effectiveness of the remote transport protocol. We used the dataset located in the folder "DATA/SYNCFILES". We tested the synchronization of these files and breaking the connection once in order to determine the effectiveness of retransmission.

A more complete description of the procedures can be found in chapter 5 & 6.

1.5 Context

This thesis is written as a part of Dualog's project Seadrive in collaboration with my advisors Otto Anshus (University of Tromsø, Department of Computer Science), Svein Bertheussen (Dualog A/S), Vidar Berg (Dualog A/S) and Asbjørn Pettersen (Dualog A/S). This thesis primarily focuses on creating an application for file synchronization between fishing fleets and the fisheries, both inland and offshore, thus providing a common access interface for files across a multitude of platforms, users and use-cases.

1.6 Assumptions and Limitations

The following assumptions and limitations were contrived during the work with this thesis. Hopefully all limitations of the artifacts, and the assumptions made will be clear after reading the details in the upcoming chapters.

- Three machines has been used in this work, all with 64-bit operating systems and 64 Structured Query Language (SQL) server enterprise version, we assume comparable results across all SQL-server installations as long as named pipes are available.
- We assume network bandwidth and capacities based on real data plans available for offshore entities the spring of 2016 in Norway.
- We will not examine methods for patching new files, i.e. creating a patch for a file new to the system from another existing file.
- We will delimit ourselves to focus on the methods provided for file-synchronization in the overall system
- We do not handle automatic merge-conflicts which may arise due to conflicting file-synchronization
- We do not consider security aspects of our file-synchronization framework

as it is considered outside the scope of the thesis

1.7 Structure of the Thesis

This thesis is structured in 9 chapters including the introduction.

Chapter 2 provides an introduction to related literature in the field of file synchronization. It is divided into three main parts, Data deduplication in which most system base their technology around. Data differencing which is an interesting technique usually found in binary patch distribution and conflict resolution for file synchronizers.

Chapter 3 provides an review of "state of the art" technologies, showing what techniques these systems use.

Chapter 4 presents the Architecture of Seadrive and provides an introduction to the concepts of local- and remote synchronization points

Chapter 5 describes the design the Seadrive framework

Chapter 6 describes and detail the implementation of the Seadrive framework

Chapter 7 outlines the experimental design and setup, including datasets employed

Chapter 8 provides an thorough analysis of Seadrive as a framework and how the remote transport protocol perform

Chapter 9 summarizes the results, and prupose proposes future work to extend this thesis.

2 Review of related literature

In this chapter, we provide an overview over related literature relevant for understanding the underlying mechanisms required for effective transportation of data, with emphasis on methodologies used for file-synchronization. We will primarily focus on Data deduplication and applicable techniques in the domain of data synchronization.

2.1 Data Deduplication

Data deduplication (henceforth deduplication), is the process of eliminating copies of repeating data, thus reducing both the intra-file- and inter-file data redundancies [29]. "*By identifying common chunks of data both within and between files and storing them only once, deduplication can yield cost savings by increasing the utility of a given amount of storage[43].*"

The effectiveness of deduplication varies widely across the different deduplication algorithms and different data sets. Although deduplication can provide great savings in terms of space savings, it is a data intensive application and comes with higher resource overheads on existing storage infrastructure.

2.1.1 Taxonomy

According to [29] deduplication solutions differ along three key dimensions; Placement of the deduplication functionality, Timing of deduplication with regards to the foreground I/O operations and Algorithm used to find data redundancies.

Placement

Deduplication can be performed at different locations, depending on the particular needs of the targeted application; it can be performed either on the client, storage array or on an appliance [29]. Appliance deduplication is utilized with specialized hardware and is therefore not subject to further discussion as we do not have access to these appliances and thus not applicable within the scope of this thesis.

In client deduplication, the duplicated data is removed before transmitting data to the server, thus reducing the data required to transmit the file. Therefore, it is often denoted as transmission deduplication. The reduction in bandwidth is a tradeoff between information sent, and processing capabilities at the client side, as it is required to process the files before exchanging metadata. The storage array deduplication is also referred to server-side deduplication i.e. the recipient of the data performs the deduplication process. This removes any types of content-aware deduplication algorithms that operate by understanding the details of data.

Timing

Deduplication can either be performed as the data is transferred to the recipient (Synchronous/in-band) or asynchronously in pre-defined intervals (out-of-band). In a synchronous operation, every attempt to write to stable storage goes through the deduplication process, i.e. data written to stable storage is deduplicated. This makes the process amendable to client-side placement because the data store metadata synchronously reflects its contents and can be queried immediately by clients, eliminating duplicate network traffic [29]. This method of deduplication can add a significant amount of latency to the system.

In the asynchronous operations the data is first written to stable storage before performing the deduplication process. This causes the deduplication process to happen after writes, and requires the system to purge duplicated data. These properties makes the placement of out-of-band operations amendable to the server side, however, this removes the benefits of reduces network traffic of synchronous operations. Placing an out-of-band operation at clients, causes the clients to not have up to date files, which is not beneficial. Although the out-of-band operations solve the bottleneck in terms of throughput at the server, it greatly increases the amount of I/O operations required. Thus making the choice of timing a decision based on the application *Service Level Objectives*.

2.1.2 Methodologies

There exists a myriad of deduplication methodologies, which involves invoking several processes to both chunk and restore the files. Irrespective of framework, application or algorithm, deduplication can be categorized into four major steps [28]:

1. Identifying the unit of comparison
2. Creating smaller unique identifier of these units to be compared
3. Match for duplicates
4. Saving unique data blocks

Therefore, the deduplication process itself can be divided into three generic steps as seen in figure 2.1 [4]:

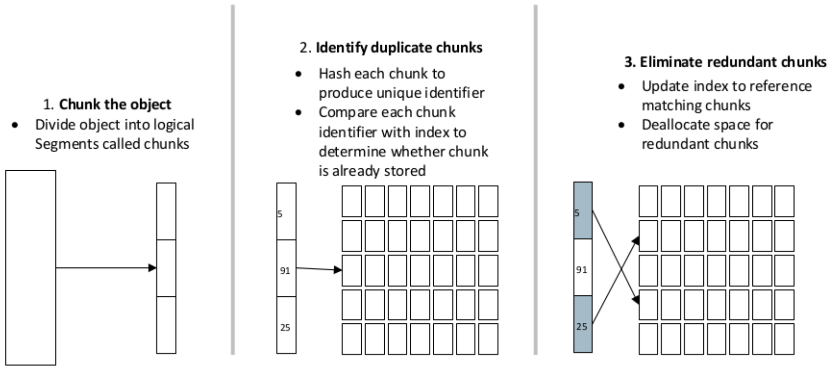
1. Chunk the object
• Divide object into logical Segments called chunks
2. Identify duplicate chunks
• Hash each chunk to produce unique identifier
• Compare each chunk identifier with index to determine whether chunk is already stored
3. Eliminate redundant chunks
• Update index to reference matching chunks
• Deallocate space for redundant chunks
5
91
25
5
91
25

Figure 2.1: The generic deduplication process according to [4]

In step one; a given file is divided into individual chunks of either fixed or variable size. In step two, each chunk is hashed to produce a unique identifier, which we will denote as the *checksum* for that chunk. The checksums are subsequently compared with an index to determine whether that chunk is already stored in the system. In step three, the actual deduplication takes place i.e. where redundant chunks are eliminated by updating the indexes referencing matching chunks, and de-allocating space/deleting for the redundant ones.

2.1.3 Deduplication methodologies

Deduplication is a process often specialized to solve specific *Service Level Objectives*, therefore based on the objective of the application and its needs; the methodology of the frameworks varies in terms of how they achieve the deduplication. The methods for deduplication are a tradeoff between I/O usage, processing time and storage needs. The process can be dived into two main categories based on how they manage their files, Course- and Fine-grained chunking.

Course-grained chunking creates larger blocks of data, and is therefore less resource intensive in regards to I/O usage, CPU utilization as opposed to fine-grained chunking. Because of the larger blocks of data, the index of chunks has fewer entries thus reducing the time spent looking for a specific chunk and creates few checksums.

We present the three most commonly methodologies for deduplication, and outline their advantages and disadvantages.

Whole file hashing/Single Instance storage

Whole file hashing does not break the file into several smaller chunks, but rather generate a hash value for the binary contents of the file [29][28]. Files with equal hash values, and optionally a byte-by-byte comparison, will be eliminated as duplicates from the system. Therefore, whole file hashing can only detect and remove redundant files, thus storing only a *single instance* of a given file.

Whole file hashing does not need to keep a complex index to manage blocks, nor is it computationally expensive, it creates minimal metadata therefore it does not create high I/O overhead. However, whole file hashing stores an abundance of redundant data compared to other methods of deduplication, and is therefore not very eligible to reduce network bandwidth.

2.1.4 Fixed block hashing/Fixed-size Chunking

Fixed block hashing divides a file into smaller entities called *chunks* at a fixed interval, regardless of content-type. For each block, a hash-function is used to generate a signature for the binary data, which uniquely identifies its contents. If a chunk's checksum corresponds to another checksum stored in the system, the chunk is redundant, and therefore not stored.

This methodology appears to overcome the problems that occur under whole file hashing. If a file is modified from N bytes into the file, only the remaining chunks including N needs to be transmitted to the recipient in order to update the given file. By utilizing a more fine-grained approach than whole file hashing, fixed-size chunking is able to further remove redundancies in the system. Although fixed-size chunking can greatly reduce the storage required for several files, it cost more in terms of CPU-utilization, I/O overhead and metadata storage. Furthermore, it does not handle prepending changes to files very well; one prepended byte would invalidate all chunks in the file. This happens because all blocks have new binary contents, and thus new checksums to identify themselves.

2.1.5 Variable Block hashing/Variable-size chunking

In order to avoid rebuilding entire files as a result of prepending bytes to a file, variable block hashing does not divide a file into fixed-sized chunks. Rather, this methodology utilizes a technique known as *Content-based chunking* which is a way of breaking a file into a sequence of chunks so that chunk boundaries are determined by the local contents of the file[13].

This is achieved through the utilization of a sliding window algorithm that works as follows: There is a pair of predetermined integers D and $r, r < D$. A fixed width sliding window protocol of width W is moved across the file and at every position k in the file, the fingerprint, F_k, of the contents of this window is computed, and k is s a chunk boundary if $F_k modD = r$. The fingerprinting algorithm must in this case be both fast and efficient, because of the many fingerprints required. In order words, the algorithm creates chunks when a pre-determined condition has been satisfied, and at that breakpoint, a chunk is created.

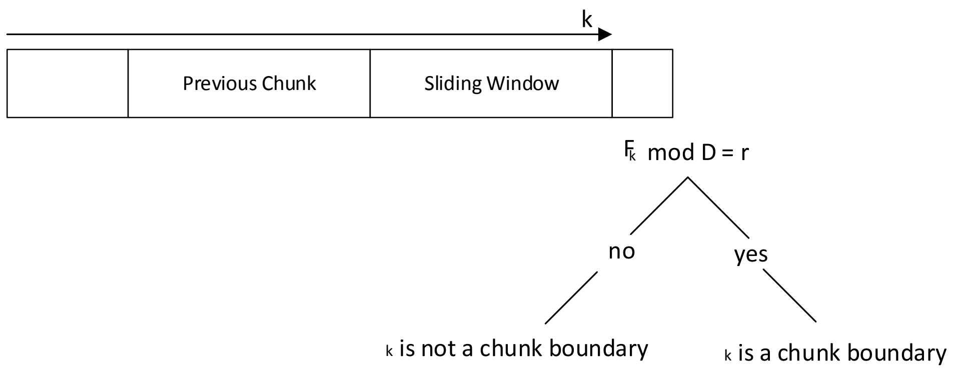

Figure 2.2: The sliding window algorithm from [13]

Due to the nature of the sliding window algorithm, variable-sized chunking methodologies applies specialized fingerprinting algorithms optimized for speed. Although regular hash-functions could be used to compute the fingerprints, they are too slow for this use-case. Therefore, variable block hashing utilizes rolling hash algorithms, because these hashes can be computed quickly based on earlier calculated checksums, which makes them ideal for sliding window algorithms. Instead of computing the entire checksum for each chunk, they can utilize the old checksum as input and transform it using the new data at the current position. This solves the major problem of fixed block hashing, because inserted or deleted bytes moves the boundaries of all chunks according to their amount of modification, resulting in fewer chunks modified. However, this method is not only the most computational expensive, it also creates the most overhead in I/O operations. The biggest contributing factor to the performance of variable block hashing is the rolling hash algorithm, and as such, we will outline three well-documented applicable methodologies for rolling hashes.

Rabin fingerprints are calculated using randomly chosen polynomials $p(t) \epsilon Z_2[t]$ over a finite field to calculate the hash for a given sequence of bytes [37]. These fingerprints are calculated over a sliding window protocol; as a result, the new fingerprints can be calculated based on the old ones, thus making it an efficient algorithm. The Rabin fingerprints are used in many systems ranging from file systems to search algorithms [7][44][31][13].

PLAIN The Pretty Light And INtuitive (PLAIN) fingerprint algorithm was created as a part of the Low Bandwidth File System [42]. Alex Spiridonov et Al. argues that the randomization in the Rabin fingerprints is redundant;

because of this, they replace the randomization with a summation of the underlying data. This allows them to use a very efficient add-operation in order to greatly increase the efficiency of the algorithm, causing it to outperform the Rabin fingerprinting scheme in terms of speed.

Adler32 is a checksum algorithm developed by Mark Adler in 1995, which is designed to increase speed over reliability [30]. During his PHD dissertation, Andrew Tridgell modified Adler32 into a rolling fingerprint algorithm, which he used for Rsync [47]. The algorithm works by concatenating two separately calculated 16-bit checksums, which is based on an efficient summation process as with PLAIN.

The problem with variable block hashing, ignoring the I/O overhead, is the fact that they are based upon the basic sliding window protocol. The algorithm requires the rolling hash method to determine the chunk boundaries, and will only split the file if the pre-determined condition is met. This shows that the chunk boundaries are determined by the probability of the condition occurring, which means that there is no minimum- or maximum block size. Even when dealing with non-random data, it is possible for the break condition to never be true, thus causing chunks to grow in size until infinity.

Opposed to the basic sliding window protocol, other variable block hashing methods aims to resolve these shortcomings caused by not having minimum- and maximum block sizes.

Two Threshold Two Divisor is a method for creating variable sized blocks that also assures that a block is not smaller than a given threshold, and not larger than a maximum size [11]. In order to maintain the maximum size for a given block, the algorithm two divisors D and D', where $D > D'$. Because D' is strictly smaller than D it has a higher probability of finding a set cut-point, however if neither D' or D finds a cut-point, it simply creates one at the maximum block size. If D does not find one, but D' does, it reverts to the cut-point found by D', because D' works as a backup divisor [11].

Bimodal Content defined Chunking utilizes blocks of two different sizes, in which the algorithm creates large blocks unless the file is in a "...limited region of transition from duplicate to non-duplicate data" [23]. Kruus et.al claims that this method increase the average chunk size, while maintaining a reasonable deduplication elimination ratio, without any special purpose metadata. This is achieved by maximizing the probability of long binary sequences in currently unknown data to appear in later sequences and that breaking these large chunks into smaller chunks around what the authors define as "change regions", will benefit the application.

2.2 Data differencing

"*Data differencing is the process of computing a compact and invertible encoding of a "target file" given a "source file*""[22]. The output comes in the form of a patch file, which allows the source file to transform into the target file. Data compression is considered a special case of data differencing and we will not further expand upon this as we expect the reader to have some familiarity with data compression.

The problem domain in data differencing is managing memory constraints and processing time when compared to usability. In order to create optimal patches (smallest), the processing time might near infinite and the memory requirements skyrocket. Therefore, usability refers to whether creating the patches can be applicable in the target domain within a certain time frame and memory constraints. We can further divide the data differencing approaches into two categories:

Known data differencing: This methodology knows what the storage format is, and is optimized to work specifically on the target format

Unknown data differencing or Generic data differencing: This methodology is designed to work on any storage targeted formats, and subsequently cannot be optimized to target a specific format.

Irrespective of the algorithmic approach applied by the data differencing software, the patch generation methodology always relies on delta encoding. From a pure mathematical standpoint, the delta encoding aims to create a patch for any given file ideally within the absolute entropy, but in reality the patches are subject to the Kullback–Leibler divergence [24].

The delta encoding algorithms therefore aims to record the changes between two files using the smallest amount of data. These algorithms are defined in two ways, *Symmetric deltas*, and *directed deltas*, where a directed delta is the change-set required to transform a version v_1 into version v_2. The symmetric deltas are denoted as deltas where $\Delta(v_1, v_2) = (v_1 \setminus v_2) \cup (v_2 \setminus v_1)$

There exists a multitude of delta encoding algorithms for data differencing, some working on entire files, some on fixed block sizes and others on variable sized chunks. We will later in this thesis explore one case and how it can be applicable to the domain of file synchronization. However, first we address the problem space for these methodologies, without examining how the problems are solved.

2.2.1 Mathematical fundament

The theoretical fundament is the process of counting matches with mismatches with respect to the edit distance, i.e. Given two strings, S T of lengths n, m over an alphabet $\Sigma, n > m$, to find all substrings S' of S such that S' can be transformed into T via a sequence of most k substitutions, insertions and deletions[34].

Therefore, the problem with mismatches is in actuality a number of problems. Taking: $\delta\Sigma \times \Sigma \longrightarrow \mathbb{R}$ as a function that identifies how close two symbols match, and defining:

$V_i = \sum_{j=0}^{m-1} \delta(S_{i+j}, T_j)$,

The following problem arises as shown in [34]

1. Compute V_i for all $0 \leq i \leq n - m$

2. Given some $k \epsilon \mathbb{R}$, find all integers $i \epsilon [0, n - m]$ satisfying $V_i > k$

3. Given some $t \epsilon \mathbb{N}$, find values $x_1 ... x_t$ such that V_x takes on the t largest possible values

4. Let $t \epsilon \mathbb{N}$ be given a set $X = x_1 .. x_t$ be fixed but unknown, and suppose that S and T are generated by random process in such a manner that $E(\delta(S_i, T_j)|i - j \epsilon X)\ X > \tilde{X} = E(\delta(S_i, T_j)|i - j \notin X)$ for some constants $X, \tilde{X} \epsilon \mathbb{R}$ Find X with high probability

2.3 Conflict resolution in file synchronizers

Detecting and resolving merge conflicts in file synchronizers can be difficult to understand and the validity of conflict resolution policy can differ, depending on the targeted domain[38]. Tao et Al. formalized the problem in [38], as first finding the set S of all operations that have been performed in order to compute the subset of S such that within the subset, "all global orderings that are consistent with the local orderings have the same effect"[38]. This subset can be used to compute the sequences of commands S_i^* to be applied at each replica. They summarize this into the following three steps:

1. Update detection examines each replica to determine the update sequence of Si that have been executed at the replica

2. Reconciliation takes as many commands as possible from the sequences S_i and computes the sequences S_i^* to be executed at each replica

3. Conflict resolution takes the leftover, "conflicting", commands and figures out what to do with them

Although this formalization is made for file synchronization in distributed filesystems, they are applicable to geo distributed file hosting services, as they face the same issues. Once connections take place over inter- networks, geo-distributed networks needs to make trade-offs between consistency and the validity of their services, which has been formalized in the CAP-theorem[14]. Conflict resolution in geo-distributed file systems are classified into two groups: operations- and state-based [46].

Operation-based approaches log the file system operations on each site and then propagate the log to the other sites on which these operations may be replayed to keep the replicas consistent. However, this approach usually require global synchronizations, which causes all sites to stop receiving more updates and exchange their logs to define new sequences of operations to be applied on each site; this is not practical in real-world geo-distributed file systems[46]

State-based approaches keep track of the state of each file and directory, then the final states or deltas of the changed files and directories are propagated to the other sites to be merged there. Although this approach is feasible, it is hard to implement as modeling a filesystem incorrectly can lead to erroneous behavior.

3 Review of related Technologies

There exists a multitude of both file synchronization software and frameworks, as well as file hosting services. Understanding the theoretical foundation and the “state of the art” in technological advancement is a vital prerequisite in order to progress. In relation to our thesis we will present some of the most widely used file synchronization frameworks and two distributed file system as their technologies aims to accomplish the same objectives as Seadrive, albeit with different constraints.

Data synchronization is the process of establishing consistency among data from a source to target data storage and vice versa, as well as the continuous harmonization of data over time. File synchronization is a subset of data synchronization, in which data synchronization is the fundamental process allowing files to be shared between differing applications and users. File synchronization is the process of ensuring computer files in two or more locations are updated via a set of rules defined in the system, and there exists several file-based solutions for data synchronization. In this thesis we delimit ourselves only to the strict interpretation of file synchronization, and purposefully disregard version control systems and content mirroring, however we will explore distributed file systems. Data synchronization in the context of files can coarsely be divided into three categories.

- *One-way file synchronization (mirroring)* where files are copied or updated from a single source location and disseminated to one or many recipients.

- *Two-way file synchronization* where files are transferred and updated in a bi-directional manner, with the purpose of keeping two locations identical to each other.

- *Many-to-Many file synchronization*, where multiple entities updates and transfer files in an overlay network, commonly used in distributed applications.

A file synchronizer is the process that makes files consistent, while preserving changes made in a system where more than one entity requires the given file(s). When changes are made to a file in the system, the replicas no longer contain the same information, thus facilitating the need for synchronization. The process of synchronization is not tautological, as the different set of replicas can contain different, conflicting information. A simple file containing the string "Hello, my name is abc", and the same file at a different location containing "Hello, my name is bcd", it is not obvious which file should be kept over the other. In cases like these, the file synchronizer requires a policy for conflict resolution.

3.1 File synchronization protocols

The purpose of file synchronization frameworks differs in what they aspire to accomplish, and therefore they attempt to solve the file synchronization problem[47][38][45]differently. In some systems, immediate updates are required, while others simply populate updates periodically. The algorithms used to achieve the updates, whether periodically, immediate or on demand, can be divided into either single-round or multi-round algorithms [16].

Single-round protocols utilizes fixed or variable sized chunking mechanisms to compare file contents and generating binary patches in an iterative manner. These protocols are preferable in scenarios involving small files and large network latencies due to protocol complexity, computing and I/O overheads [16]. Multi-round algorithms will often use recursive partitioning of unmatched blocks, mostly in a breath manner first. The divide and conquer algorithm are subsequently used to send hashes between the server and client to detect changes in remote files[20]. The multi-round algorithms are preferable to the single-round algorithms in the case of large collections over slow networks, because of the many rounds of compression [16]. However, these advantages can be lost in very slow wide-area networks[51].

3.1.1 Widely used remote file synchronization algorithms

There exists are myriad of remote file synchronization algorithms in the wild, however for this thesis we will outline what we believe to be the most influential methodologies.

We will try to give a simplified formalization of the file synchronization problem: Given two files f_{new} and $f_{old} \epsilon \Sigma^*$ over a given alphabet Σ, in this case bytes, and two computers A and B, connected by a communication link [44], where A holds f_{new}.

The content of f_{new} is denoted as a_i and of fold as b_i, the aim of the algorithm is for B to receive a copy or updated version of f from A. The basis of a remote algorithm can thus be denoted as [47]:

1. B sends some data S based on b_i to A
2. A matches this against a_i and sends some data D to B
3. B constructs f_{new} using b_i, S and D

As Andrew Tridgell put it in his PHD dissertation [47]:

> "The questions [sic] then is what form S will take, how A uses S to match on f_{new} and how B reconstructs f_{new} "(modified for the formalization in this chapter).

3.1.2 Rsync

Rsync is the best-known single-round protocol for file synchronization [16] and is bundled in several Linux distributions. Rsync is fundamentally an improved version of the fixed-size chunking approach into variable-sized, and in similar fashion breaks a file into chunks, which are transmitted from A to B. Rsync specifically splits files into disjoint chunks of a fixed size b and utilizes hash functions to calculate the fingerprint of each chunk, and sends the fingerprint to the receiver. Due to the possible misalignments between the files, it is necessary for the recipient to consider every window size of b in the new file for a possible match with a chunk in fold [48]. The formal definition of the algorithm is as follows:

Given the same denotations used in the preceding section approach with R and H as hash algorithms, one fast to calculate all byte offsets, and one slow to ensure no data collision

1. B divides b_i into N equally sized blocks b'_j and computes signatures R_j and H_j on each block. These signatures are sent to A.

2. For each byte offset i in a_i A computes R'_i on the block starting at i.

3. A compares r'_i to each R_j received from B

4. For each j where R'_i matches $R_j A$ computes H'_i and compares it to H_j

5. If H'_i matches H_j then A sends a token to B indicating a block match and which block matches. Otherwise A sends a literal byte to B.

6. B receives literal bytes and tokens from A and uses these to construct a_i

As we can see from the Rsync method for synchronizing files, it can efficiently group changes in blocks and compress the data transfer to speed up transmission. However, it requires both the sender and recipient to actively gather and generate chunks, and generate checksums for each round in its run. These characteristics of Rsync make the algorithm unsuitable for frequent changes in large repositories [16] [20].

3.1.3 Unison

Unison is a well-known multi-round file synchronizer that works in similar fashion to Rsync [36] [35] [3]. It chunks the files into disjoint blocks, compares these blocks before merging changes, and utilizes a rolling checksum algorithm to detect changes. Unlike Rsync, Unison utilizes a two-way file synchronization algorithm. This causes Unison to split the synchronization into two phases: Update detection and reconciliation. During the update detection phase, it monitors for file changes based on modification time, the cryptographic fingerprints and other metadata [36]. If changes are detected the file is marked as dirty. During the reconciliation phase it merges the updates into what they call a task list, and based on its recursive multi-round method, they solve for merge-conflicts and recreate the files.

The similarity of Unison and Rsync's algorithmic approach means that Unison shares the drawbacks in regards to frequent updates of small files. Although unisons update phase is significantly shorter because of their usage of metadata rather than checksums.

3.1.4 Dropbox

Unlike the systems we have shown so far, which are on-demand file synchronization frameworks, Dropbox is a near real-time file synchronizer. Dropbox in itself is a file hosting service, however for the thesis we will focus on the file synchronization mechanisms known as of spring 2016.

Dropbox as a file synchronizer offers *live synchronization* or *continuous reconciliation* [3], to achieve this functionality it employs file watchers. File watchers are mechanisms, which react to operating system events whenever a file is changed, renamed, updated or deleted locally, and these events are registered in the Dropbox application. This allows Dropbox to do automatic recording of file versions. However, the inner workings of Dropbox is unknown as the system is undergoing continuous changes, and based on proprietary software. We will outline the information we know that has not changed since written about.

The data deduplication in Dropbox is proprietary but we do know they use a form of either statically or Variable Length Chunking (VLC), but the sources that have analyzed Dropbox differs. For instance Jin San Kong ET. Al. claims Dropbox to use VLC [40] while [9] [8] [50] claims Dropbox is using static length chunking. Regardless of the deduplication scheme employed, we know that they no longer utilize global deduplication, which can be validated by the reader by simply uploading two identical files on different repositories. We cannot know why they removed this feature, but it is speculated to be because of privacy concerns or the high cost of retrieving files.

The Dropbox client control flow can be divided into three partitions[8]:

1. Notification
2. Metadata administration
3. System logging which we ignore as it is considered irrelevant for the thesis

The Dropbox client keeps a continuously open Transmission Control Protocol (TCP) connection used for receiving changes remotely where remote changes are periodically polled on 60 seconds intervals, unlike local changes, which are updated instantly. Upon local changes, a synchronization transaction starts by sending messages sent to the metadata servers. Once the metadata exchange protocol is completed, the remote data storage protocol manage the actual exchange of data.

In [27] Z Li. Et Al. Measures the amount of network traffic generated by adding new files to the Dropbox sync folder, and they observe the amount of metadata to remain near constant for files in the range of 1 byte to 100 Megabyte (MB). The metadata is on average 33 Kilobyte (KB) +- 5Kilobit (kb). However, the traffic sent to the synchronization varies greatly; the ratio of data to upload is 38200 times larger for the 1-byte file, but only 1.1266 times larger for the 100 MB file. This is mostly caused by the close to fixed sized metadata information sent which causes the Dropbox overhead to amortize over large file sizes.

3.2 Distributed file systems

Distributed file systems is a file system that supports the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet [6]. We will not outline the functionality and requirements of a distributed file systems as we consider this to be outside the scope of this thesis. However, we will present two case studies to examine their architecture. We do this in order to exemplify architectural solutions for solving the file synchronization problem, as it is a subset of distributed file system demands.

3.2.1 Sun Network Filesystem

The Sun Network File System (NFS) follows an abstract model where all implementations of the NFS support the NFS protocol – which is a set of remote produce calls that provide the means for clients to perform operations on a remote file store. Although the NFS protocol is operating system independent, we will outline the UNIX implementation as it was initially developed for UNIX.

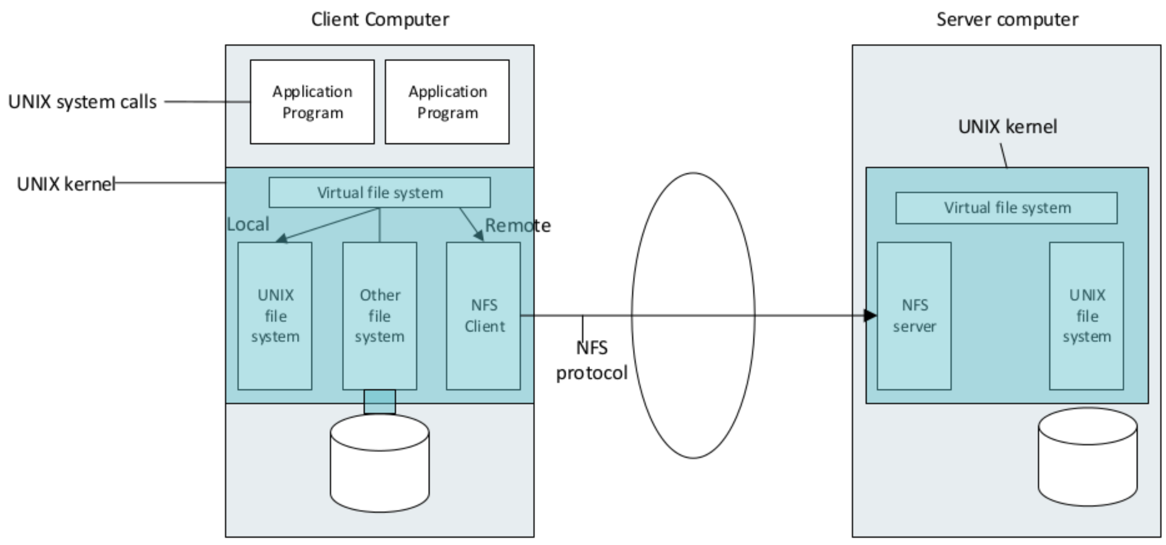

Figure 3.1: NFS architecture as outlined in [6]

The NFS Server module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module as the computer holding the relevant file system [41].

The NFS client and server modules communicate using Remote Procedure Call (RPC). Sun's RPC system developed for use in NFS is built upon the Open Networking Computing RPC. It can be configured to use either User Datagram Protocol (UDP) or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

The NFS server implementation is stateless, enabling clients and servers to resume execution after a failure without the need for any recovery procedures. Migration of files or filesystem is not supported, except at the level of manual intervention to reconfigure mount directives after the movement of a filesystem to a new location.

The caching of file blocks at each client computer enhances the performance of NFS. This is important for the achievement of satisfactory performance but results in some deviation from strict UNIX one-copy file update semantics.

3.2.2 Andrew File System

In similar fashion to NFS, the AFS, provides transparent access to remote shared files for UNIX programs running on workstations. Access to AFS files is via the normal UNIX file primitives, enabling existing UNIX programs to access AFS files without modification or recompilation. AFS is compatible with NFS. AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS based, so files are referenced by NFS-style file handles rather than i-node numbers, and the files may be remotely accessed via NFS[18].

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

- *Whole-file serving*: AFS servers (in AFS-3, files larger than 64 kb are transferred in 64-kb chunks) transmit the entire contents of directories and files to client computers.

- *Whole-file caching*: Once a copy of a file or a chunk has been transferred to a client computer, it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible.

The four following steps illustrates an operation running on AFS:

1. When a user process in a client computer issues an open system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy is then opened and the resulting UNIX file descriptor is returned to the client.

3. Subsequent read, write and other operations on the file by the process in the client computer are applied to the local copy.

4. When the process in thee client issues a close system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.

AFS is implemented as two software components that exist as UNIX processes called Vice and Venus. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer.

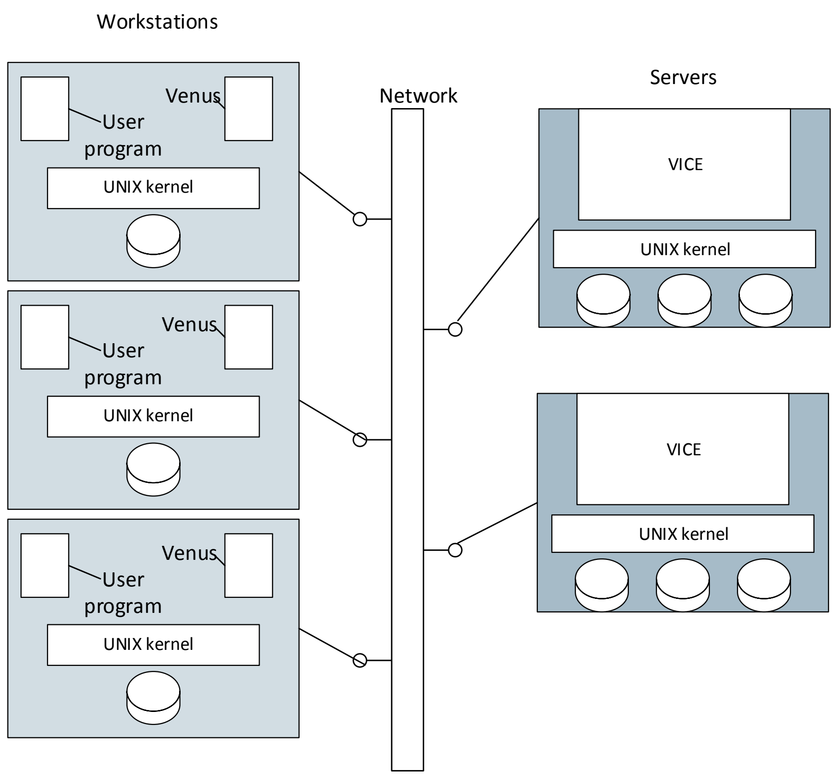

Figure 3.2: AFS process distribution as outlined in [6]

The files available to user processes running on workstations are either local or shared. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local disks of workstations. In order to support the caching mechanisms one of the file partitions on the local disk of each workstation is used as the cache, holding the cached copies of files from the shared space. Venus manages the cache, removing the least recently used files when a new file is acquired from a server to make the required space if the partition is full.

AFS utilizes a weak consistency model supported by the local cache. Once a read or write operation have completed, and the file has become modified the local copy are copied back to the file server maintained by callbacks.

4 Architecture

Seadrive provides a framework that consists of several different components, where each component encapsulates a particular functionality. These components are interconnected to work in unison in order to create the client- and server side software. The architecture will show the inter- and intra-connection between the different file synchronizers and how these components communicate in order to minimize traffic of the high latency network connections subjected to frequent drops. In order to complete the primary objectives defined for the Seadrive application, the architecture must accommodate for physical constrains such as hardware, network bandwidth, loss ratio, and network topology. The overall architecture concerning communication and dataflow is outlined in figure 4.1.

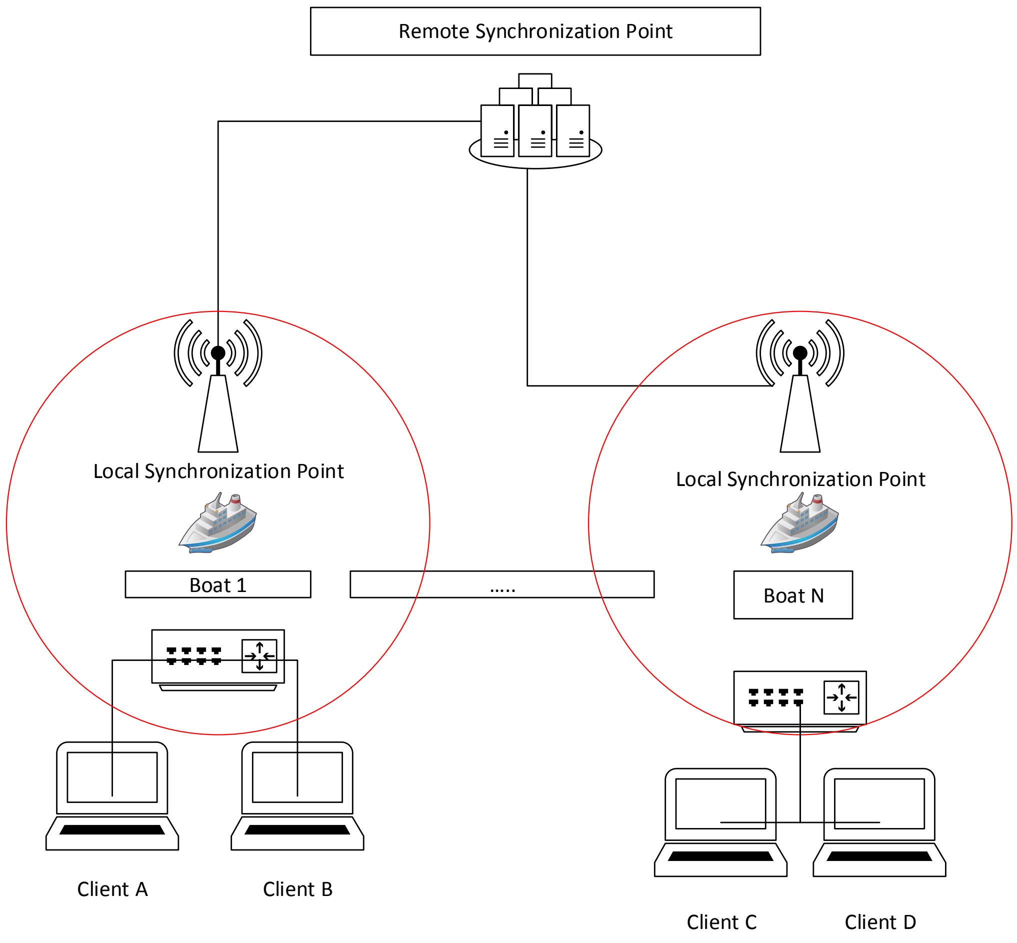
Remote Synchronization Point
Local Synchronization Point
Boat 1
.....
Local Synchronization Point
Boat N
Client A
Client B
Client C
Client D

Figure 4.1: Birds eye architecture of Seadrive. Clients are reciprocally synchronized within the LSP, and is continuously synchronizing with the RSP whenever possible. Red rings indicate the LSP.

Synchronization Point. In this thesis we will delimit ourselves to focus on the following components outlined in Figure 2, and we'll not discuss the intricacies of the software required to actually use satellite based network links, such as Dualog Connection Suite[1] or Inmarsat launch pad[2], nor will we examine how these network suits affects the application itself. We will also disregard how land based partners affect the server- and remote file repositories on vessels. Rather, we will focus on the components required to synchronize, transfer and update files among all clients on a vessel, and to the main land-based storage

1. http://dualog.com/services/overview/how-it-works
2. http://www.inmarsat.com/support/bgan-firmware/bgan-launchpad/

facility and vice versa.

The *logical architecture* can be roughly divided into three parts: The Client, the LSP and the RSP. The clients are the end-users of the application that interacts with files in a Dropbox like manner. Files are dragged and dropped to a folder denoted as the Seadrive sync folder, and these files are immediately uploaded to the LSP. Likewise, the client application continuously downloads or updates new files from the LSP. The LSP synchronizes all clients with each other and with the RSP. Its other responsibilities contain managing possible merge conflicts and facilitate all remote and local communication. The RSP simply store all files from all clients and disseminates these accordingly.

The architectural choice of having LSPs on each vessel for several clients was inspired by AFS [18][19]. The local server can leverage the advantages of Local Area Network (LAN) networks with low latency, high bandwidth and minimal packet loss, or Wireless Local Area Network (WLAN) with 4G or 5G. Therefore, changes done locally at a vessel can use a simplistic transport protocol to transfer changes within the vessel. Furthermore, this architectural choice allows us to have a single point of entry for all data communication both in- and outbound from a vessel. This also alleviates the RSPs resources, because it does not have to send the same information to each client, i.e. if a vessel has 20 users on average, the load is reduced by 20 * number of vessels.

Similar to AFS, Seadrive utilizes a weak consistency model on the client side. Read and write operations on an open file are not updated remotely until the file is closed. During modification, the file is marked as dirty, to indicate that the file has changed contents, and not until Windows release the file lock are changes merged into the LSPs' copy.

In order to detect modification, deletion or creation of files, Seadrive utilizes a callback centric event driven daemon. On remote changes not local to the vessel, the local and RSPs initiates a negotiation protocol in order to determine the changes between the differing versions. Once the change set has been identified, the changes are transferred in statically sized chunks. New files added to the RSP from a land-based client takes priority over the vessel based files, because management entities have a higher priority to disseminate files.

The Seadrive framework uses an N-tier architecture[21], and we currently employ four layers. The N-tier architecture was selected for its ability to allow developers to create more flexible and reusable applications [10]. We employ what we denote as the Data Abstraction Layer (DAL) [APPENDIX 1], the business logic layer, the application layer and the presentation layer (implementations).

The quintessential aspects of the framework is the division of intra- and inter communication. The intra communication between clients and the local synchronization uses a lightweight transport protocol, which is not designed for robustness nor speed, but for simplicity. The remote transport protocol between local and RSPs is designed for robustness and to use minimal bandwidth.

5 Design

framework on the windows platform. We explore both the inter- and intra-communication required to perform remote file synchronization, how data is managed in the system, and the mechanisms necessary to facilitate real-time file synchronization within the local network on the boat. We focus on the methods we employ in order to facilitate the requisites of file synchronization over satellite-based connections. Although the requirements to provide file hosting services, file synchronization and deduplication differs, they rely on many of the same mechanisms in order to accomplish their task, and at a high level of abstraction, the application can be presented as seen in figure 5.1 :

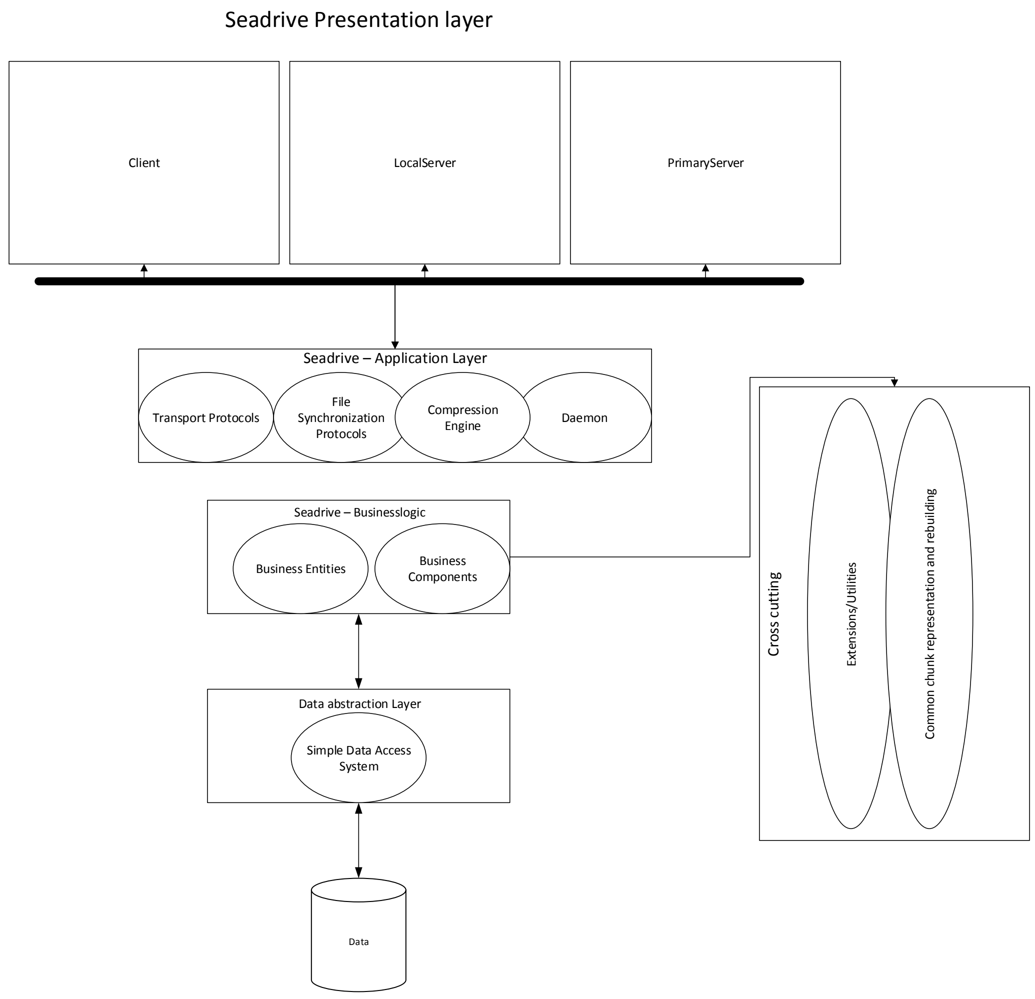

Figure 5.1: Shows a simplified model of the entire application stack

The presentation/Application layer are formed from the architectural demands to reduce the traffic over the high latency, low bandwidth network connections. We will discuss each section of the framework in the following subchapters.

5.1 The Data Abstraction Layer – I/O management

The DAL was created in the project "eSushi" which was developed by the author in an earlier project, and what is denoted as the simple data access model described in Appendix A is re-used for Seadrive. We store data in a MSSQL database in a parallel environment, thus we require the ability to handle concurrency issues in a transparent manner, which alleviates the burden for developers.

> "The DAL's primary purpose is to provide unified access to all data available in the system. Subsidiary is it required to lessen the burden of managing data, resolving issues such as weak typing of the business data, centralizing data-related policies and apply a domain model to simplify the business logic. We also want to have a resolution of concurrency problems which may arise with conflicting requests in the same time window." [Appendix A]

The Read and Write I/O operations regarding the deduplication, preparing data for transportation and maintaining a serializable in-memory cache in this file synchronization framework does not deal with large file (gigabyte+), manipulation due to the nature of the application. The application also requires simple data access throughout the entire stack and is modeled after the classical Data Access Pattern. The design of the DAL is fully described in Appendix A and is denoted as the "Simple Data Model, or Simple Data Access".

5.2 Business Logic Layer – Core functionality

The business Logic Layer aims to reduce complexity by separating tasks into different areas of concern. Applications can have many cross-cutting concerns such as authentication, logging, validation, coupling and cohesion, and a common representation of data. The Business layer tries to resolve these problems by centralizing the rules in order to enforce correct usage.

The core functionality of "Seadrive" is simply all functionality required throughout the entire stack of the application itself, thus ranging from simple extensions i.e. checking whether an item exists, or access to the functionality implemented over several instances. The core functionality thus functions as a library providing convenience functionality to developers and users, while abstracting away underlying implementation, such as inversion of control [49]. It also holds all object data models in use by the system, allowing all applications developed to have a unified understanding of the data in the system.

This methodology facilities data sharing on every level in the framework allowing applications to be developed, while ignoring the DAL. Utilizing these methods, flexible components can be registered in the system, implemented as a module, and mounted wherever needed, while maintaining data interoperability.

5.3 Application Layer – Seadrive

By the virtue of N-tiered applications, Seadrive is detached into several components with different responsibilities. Each component encapsulates a particular functionality and interacts with other components in order to implement the applications. The application layer is thusly responsible for the concretization of the high-level components in the file synchronization framework, which is required for data deduplication, transport protocols, compression and the interconnection of these components.

This library is then able to form the basis for Seadrive and its functionality at a high-level abstraction allowing the representation of complex operations in a unified manner, where implementations from the business layer can be changed without affecting development. In the following sections, we will outline the application layers' primary responsibilities, on which the framework places its fundament.

5.4 Data Deduplication

Data deduplication is a core asset to Seadrive as both a file synchronization framework and as a file hosting service. In order to reduce both the bandwidth and the number of messages required to perform file synchronization, data duplication is a powerful ally as it can lower both. The deduplication schemes can reduce both the data sent by identifying redundant byte sequence and thus not sending them, and local file storage can be reduced immensely by only storing a single instance of a byte sequence. The data deduplication process in Seadrive is generic to facilitate supporting the optimal methodology beneficial to the users' needs.

The deduplication process consists of three components; the chunker, the compression engine, and the rebuilder.

- *Chunker*: In order for the system to perform data deduplication at a finer level than file-based deduplication, a key component is the chunker. This component divides an OS file into several smaller sequences of bytes denoted as chunks (singular: chunk). The chunks are interchangeable regardless of static or variable sized chunking, as they contain metadata fields to support this. The chunks have an overhead of 44 bytes to support integrity checking of data, owned files and to report its own size.

- *Compression Engine*: The compression engine is simply an engine which allows chunks to be compressed or decompressed using different compression algorithms. This allows the de-duplicated data to be reduced further in size before storage or transmission.

- *Rebuilder*: The rebuilder is simply the entity that rebuilds a file from given chunks.

These components work in unison with the DAL to perform Data Deduplication in Seadrive. A given file is divided into chunks by the chunker, the chunks are subsequently compressed through the compression engine, and the non-redundant chunks are stored locally in an MSSQL database. The compressed chunks can also be used for transmission by identifying non-redundant chunks both locally and remotely and therefore only transmitting necessary chunks in order to rebuild the files by the rebuilder.

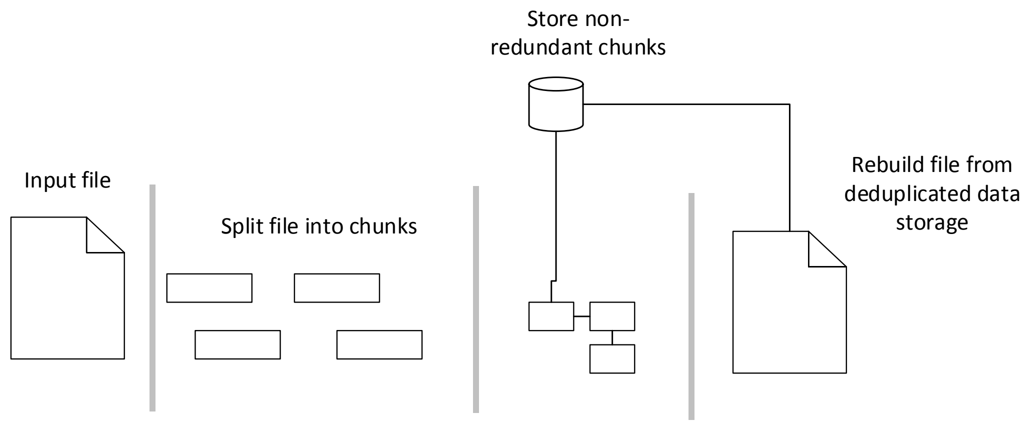
Store non-redundant chunks
Input file
Split file into chunks
Rebuild file from deduplicated data storage

Figure 5.2: Shows the generic Seadrive data deduplication process

5.4.1 Delta difference data deduplication

The data deduplication process shines when dealing with variable- or fixed-size chunks, however a comparable process should exist for systems based on delta-difference file synchronization. Therefore, data deduplication based on delta differences differs in their design. These methodologies are also designed around the DAL in order to perform data deduplication, but instead of chunks, patches for each file version is stored. For any given file whom at one point has been changed, patches has been created. These patches must be symmetric in order to change the file between different versions. In this manner, content revision is supported, and only patches needs to be stored.

5.5 Filesystem monitor, change detection and the application facade

The Seadrive Synchronization relies on repositories, which is a subscribable entity. Repository is a windows folder in which the entire file-tree within the folder is replicated at all targeted applications, at clients, the LSP and the RSP. To reliably detect changes to the file-tree we employ file system watchers (also known as file alteration monitors), these services watches the local replica of the repository as a folder in windows. Whenever a change occurs to the folder, be it added files, deletion, renaming files or folders or files are changed, events fire which are handled by an event driven application façade.

The façade is the primary Seadrive daemon that handles everything the framework require to manage local changes, and disseminating the changes to other replicas. The events from the filesystem watcher is filtered at the daemon to avoid multiple events caused by the same action to propagate, and these events determines whether a file should be marked as dirty. The façade determines when change sets will be propagated further in the system. The dirty files are subsequently prepped for transportation by feeding the changes to the de-duplicator to re-calculate the change sets, which are then disseminated to recipients.

5.6 File synchronization and Transport Protocol

In order to support bi-directional file synchronization among multiple users and with file hosting capabilities at a land-based operational center, the framework has different strategies for resolving the needs at different applications.

By virtue of the architecture, the data flow in Seadrive is straightforward; it can logically be divided into two parts, local and remote changes. Local changes consist of all changes done at the client side to the LSP. While remote synchronization is the process between the RSP and the local. This division of the models is a deliberate design choice, based on the problems they attempt to solve. The differing methodologies these models apply will be further expanded upon in the following subchapters.

The primary concern for a file synchronization framework is synchronizing files between different locations and applications without corrupting the data. The synchronization process should happen unbeknownst to the user in a non-blocking manner to provide the illusion of always reaching your files regardless of data locality in a file hosting service and should be simple to use. However, a comparable concern, especially with large files, is speed. Data should be delivered within a user-specific definition of reasonable time, thus having differing methodologies for local and remote storage allows for tradeoffs between I/O complexity, memory utilization and transfer speed. We will show the design for the two different approaches separately and compare them to each other in the experiments chapter. For both methods, we will show the design from the bottom up until the entire data path has been realized.

5.7 Local synchronization protocol

The local synchronization protocol is the process of synchronizing all the data in a given repository within a vessel. It is the primary method of communication between each client interconnected in the LSP, its responsibilities include managing uploads and downloads within all entities in the LSP.

Once the daemon in conjunction with the deduplication process has calculated change sets and stored local compressed copies of the file-tree as a snapshot, the data is transferred from either client to the LSP or vice versa. The local synchronization protocol is designed to be a lightweight process both computationally and in memory, as the process takes place on reliable high throughput networks. The protocol is not designed as a rigorous algorithm, rather allowing for a high degree of flexibility, but some rules of conduct is necessary.

The data transfer in the local synchronization protocol is a two-step process, which starts by negotiating changes. A set of metadata concerning the changed file(s) is transmitted from sender to receiver in order to validate that the files have actually changed. The receiver responds by requesting changes from the file(s) in one of two methods. The simplest methodology is to request the entire file in order to rebuild it from scratch, or request the missing chunks in order to patch the file. For LSPs, once it has received a new file, the changes disseminated to all clients used a broadcast mechanism. To ensure that the data has been transferred completely, the receiving entity compares the checksum of the rebuilt file with the checksum(s) received in the negotiation.

The protocol is initiated in two different manners, differing slightly in their respective behaviors. On the client's application startup, the local synchronization protocol assumes the LSP to contain the newest file version if conflicts occur. This is a deliberate design choice in order to avoid automatic merge conflicts management, as this process is immensely difficult where the worst-case scenario can completely ruin files. We chose this approach because when a previously disconnected client connects to the system; the client computer has most likely been powered off or suspended and is therefore most likely out of date with the LSP replica.

During continuous use, i.e. both the client and LSP online, once changes are registered for transportation, the last entity to update a given replica in the system will have the prevailing change. This means that when two or more entities updates a given replica, the last to be registered in the system is the final version, and propagate throughout all replicas. Based on surveys on the stakeholders in this project, we have determined this to be an unlikely scenario and is thusly not handled because of the mentioned complexity and risks of automatic merge conflict management.

5.8 Remote Synchronization protocol

Remote synchronization is the process of synchronizing files between the LSP to the RSP, and vice versa. This synchronization is the critical mechanism that makes offshore-based file synchronization possible. Like the clients, the LSP monitors changes within itself, which creates a changed-set, consisting of dirty files and the given deduplication and synchronization mechanism. Once the changed set is prepared and ready for transportation, the LSP initiate contact with the RSP to negotiate the transmission.

Unlike the Local Synchronization protocol, the remote protocol utilizes a rigid algorithm for the transmission of data focusing on robustness. The remote synchronization protocol is designed to be a stateful algorithm, in which every operation must correspond to a pre-determined state. This means data sent out-of-order between expected states must retransmit from a previous state known to both the sender and recipient.

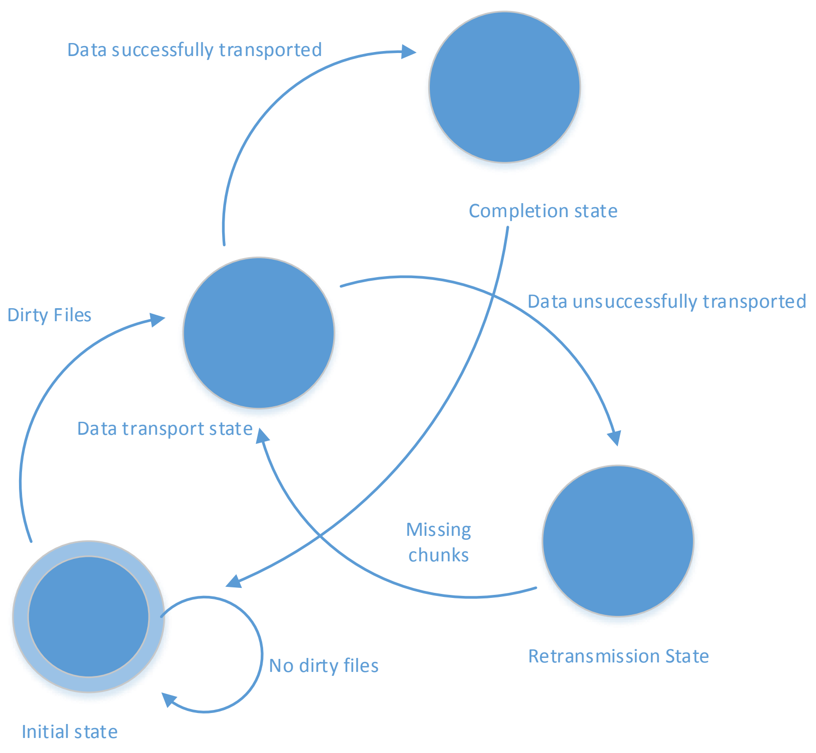

Figure 5.3: State diagram of the sender in the remote transport protocol

The protocol is initiated by the sender sending an *initial* packet, which contains the set of files the sender believes to be changed, and corresponding metadata for each file. This causes the recipient to mirror the state to *initial* for this connection. Subsequently the recipient parses the initial metadata, and produces an initial response, which contains the given file(s) that it does not have up to date, and negotiates the data transfer protocol in use for the remaining file synchronization. Consequently, both the recipient and sender advances in the context of the current connection to the *data transport state*.

To support reliable backtracking, the sender dispatches a metadata packet, which contains all metadata, required to synchronize the upcoming file. After the metadata packet has been successfully sent, the sender begins sending the data necessary to recreate the file, based on the currently employed synchronization scheme. Irrespective of the selected synchronization scheme, the data is split into fixed-length chunks of binary data and transmitted to the recipient.

Once all chunks of the data required to synchronize the file has been transferred, the recipient applies the changes to a copy of the replica, calculate a checksum for its binary contents and validates if it has been successfully rebuilt. If the data transfer was incomplete, the recipient begins the *retransmission state*.

Given successful transportation and patching of all files marked dirty from the sender, both entities begins the completion state. The completion state contains of the recipient sending a metadata packet equal to the initial packet containing metadata of replicas the recipient believes the sender not to have. If the sender responds with files it needs synchronized, the algorithm reinstates the data *transport state*, making the recipient the sender, and vice versa.

As previously specified, if a given data transfer is incomplete, or a connection has been dropped, the retransmission protocol begins. The retransmission state differs depending on what caused it. If data transfer was unsuccessful, the recipient ensures all checksums of all chunks corresponds to the expected value and demands the retransmission of the corrupted/missing chunks. Once completed, both parties involved switches over to the *data transport state*.

In order to support retransmission of data for dropped connections, the primary synchronization point contains an in memory cache of all connected clients, clients meaning connected entities, most likely LSPs. The cache contains a Globally Unique Identifier (GUID), of the connected entity which corresponds to their connection. If a connection has been dropped, the GUID is used to locate the exact state of the previous connection. It is important to note that IP-addresses cannot be used as identifiers in the system, as several vessels can utilize identical IP-addresses and they are subjected to frequent changes.

The retransmission protocol for dropped connections is thusly initiated by reconnecting entities, which begins by dispatching a *retransmission* packet, which sets the state of both the sender and recipient to the *retransmission state*. Once the recipient receives the retransmission packet, which must contain the Universally Unique Identifier (UUID) of the sender, it determines which chunks of the current file in transit it is missing based on the earlier received metadata file. If no metadata file is found, it signals the sender to begin transmission from the last file not transferred. Otherwise, it sends a list of integers corresponding to missing chunks, and the sender parses these, to send only missing chunks. Once the sender calculates which chunks to send, both entities reverts to the data transportation state.

The remote synchronization protocol is comparable to a serial sequential algorithm, although the physical the transportation of data happens in parallel; files are sent one at a time in order. The following pseudo code outlines the transport of data from both the sender and recipient's perspective.

Sender very simplified which unlike a real scenario assumes synchronous connections, and does not indicate the states.

Algorithm 5.1: Sender pseudocode

```
initialResponse = SendInitialPacket(intialMetadata);
dirtyFiles = ParseInitialResponseToDetermineDirtyFiles
RequiredToSyncRepository(initialResponse);
...

SendLoop():
    for dirtyFile in dirtyFiles:
        metadata = CreateMetadata(dirtyFile);
        await SendMetadata(metadata);
        TransportData(dirtyFile);

TransportData(dirtyFile);
    dataChunks = Chunker.CreateChunks(dirtyFile);
    response = Paralell.Send(dataChunks);
    if not response.ok:
        SendMissingChunks(response.MissingChunks);

Timeout():
    response = RegisterRetransmission(myGUID);
    SendMissingChunks(response.missingChunks);

SendMissingChunks(missingChunks);
    response = Parallel.Send(missingChunks);
    if not ok:
        SendMissingChunks(response.MissingChunks);
    else:
        RemoveCurrentDirtyFile();
        SendLoop();
```

Simplified recipient with the same assumptions and limitations:

Algorithm 5.2: Receiver pseudocode

```
OnPacketReceived(packet):
    switch(packet.packetType) {
        case initialData:
            CreateInitialDataResponse(packet);
        case metadata:
            CreateMetadataResponse(packet);
        case transmission:
            AddData(packet);
        case retransmission:
            HandleRetransmission(packet);
        ....
        case default:
            SendInvalidPacketResponse(Sender);
    }

AddData(packet):
        AddDataChunkToExistingChunks(packet.Chunk)
        if ChunksComplete:
            file = RebuildFile()
            if file.Checksum != metadata.file.Checksum:
                BeginRetransmission()
            else:
                SendPositiveResponse()

HandleRetransmission(packet):
    RegisterClient(packet.Client);
    oldConnection = UpdateClientInformationAndRemoveOld(
        packet.Client);
    ResumeFromState(oldConnection)

ResumeFromState(connection):
    OurChunks = RetrieveChunks(connection.File);
    missingChunks = OurChunks.Where(
        ourChunks.Id not in ListOfAllChunks);
    SendMissingChunks(missingChunks);
```

6 Implementation

Because of the different methodologies applied in the design of each layer and the selection of an N-tiered architecture, each layer must be discussed separately in order to uncover their different characteristics. We will disclose a set of methods showing how the design can be realized and cover our implementation specific details.

6.1 Data Abstraction Layer

The DAL has been implemented in C#.net with dependencies to the System Configuration, SQL server runtime, and the Business Logic Layer. The DAL project references the Microsoft Data Annotations library to allow models the flexibility of being pure code, database tables and other entities simultaneously. It also has the Entity Framework package installed to ensure interoperability of the models invoked by the Entity Framework and depends on the Business logic layer in order to access the system entities.

We've implemented the DAL to hold important state information regarding the system such as updated files, files which needs to be synchronized and the entire repository replica as a serializable in-memory filesystem tree.

The implementation details are outlined in Appendix B denoted as the simple data access pattern.

6.2 Business Logic Layer

The business layer has been implemented in C#.net utilizing the .NET runtime. In addition to the same references used in the DAL, it also references the Microsoft Core Libraries in order to utilize IOCP. It also has the entity framework package installed in order to achieve system interoperability.

As designed, the business logic layer contains all entities that the system requires throughout the entire application stack. This include things such as concurrency entities, chunks, extensions and utility. We utilized the underlying .NET implementation for Transmission Control Protocol/Internet Protocol (TCP) communication, but we must note that we implemented our own servers, clients and socket in order to harness the power of IOCP. We based aspects of our IOCP implementation of EPServer[1].

We went with the IOCP programming model because if effectively resolves the "one thread per client problem" while sharing the load between multiple threads. The threads in an IOCP model are placed in an Last In First Out (LIFO)-queue [1], which means that there is a high probability for an awoken thread to still have data in cache. Furthermore, this alleviates developers the burden of creating high performance server/clients by advanced thread handlers and event buses, as the asynchronous I/O are provided "out of the box" by the .NET framework.

6.3 Application layer - Seadrive

The application layer contains the interconnection of all entities in the system to create the complex structures required by the application, thus forming the concretization of the framework. We will discuss these entities separately in order to detail the implementation technicalities and provide a thorough explanation of their usage.

6.4 Deduplication

The deduplication process has been implemented as designed, i.e. partitioned into several domains of concern, where a respective entity resolves a single issue. The chunker is a generic interface in which the user requests a concrete instantiation based on the preferred chunking methodology. The chunker emits

1. http://www.codeproject.com/Articles/10330/A-simple-IOCP-Server-Client-Class

the file divided into several smaller sequences of binary data with 44 bytes of metadata fields as the code listing 6.1 shows:

```
public class Chunk
{
        private readonly byte[] _checksum;
        private readonly byte[] _contents;
        private readonly byte[] _size;
        private readonly byte[] _fileChecksum;
        ...
}
```

Listing 6.1: private variables of the chunk class

The checksum algorithm used to verify the integrity the binary contents of a file is the SHA1 algorithm while the checksum for the contents of a given chunk is user-selectable. We do this in order to support multiple chunking methodologies, as some algorithm relies on a rolling checksum. We selected the SHA1 algorithm for files, because of the low frequency for collisions, while performing reasonably fast. The chunker emits a list of chunks or compressed chunks on completion.

The compression engine has been implemented to either provide data compression on either chunks local to the chunker, or on the resulting uncompressed chunks. We utilize this scheme, so that users have choice on zip algorithms and data locality. The compressed chunks are subsequently stored in the system by utilizing the Unit of Work pattern from the DAL. This allows us to keep a thread-safe serializable in memory cache in the form of a SQL database.

6.5 Seadrive artifacts implementation

To actualize the architecture of Seadrive and implement the design of the different component in the framework, we have implemented the system as a tripartite solution. The LSPs are realized as TCP servers and denoted as local servers, and the RSP is a cluster of machines communicating with the LSP's over TCP, through a server implementation denoted as the primary server. The clients are implemented as windows applications utilizing the Seadrive framework as a Dynamic Link Library (DLL).

Though the following subchapters we'll explore the implications of the design, specifically how the deduplication process, the file synchronization and the real-time constrains of file hosting services affect the application framework and how the design resolve the constrains set forth in 1.2.

6.6 Clients

Users interacts with a desktop/laptop frontend known as the Seadrive client, which displays how the system manage the synchronizable folder, the status of the files and progress report. The Seadrive client is modeled as a daemon or service depending on the terminology and this service will handle deduplication, synchronization and reconciliation of the file(s). The user selects a folder that functions as a local replica of the remote synchronization repository and from that point on the service is live.

The client has two primary states, which is indicative of its behavior, startup and runtime. It is important to differentiate between these two states, because if the computer, or the service, has been suspended, the application assumes the remote changes since the last execution will be newer and the file-tree should be updated before continuation. Furthermore, it is equally important to detect whether local changes has occurred while the service been suspended.

During the *startup phase*, the service immediately resumes communication with the Windows file system watcher on the local repository replica and registers the event filter to the system. The next step is to update the system based on the previous known state of the system, i.e. which files are and are not synchronized.

The algorithm for change detection is straightforward and implemented as follows:

1. Request existing known replica filesystem tree locally (if one is known at all)
    (a) If the system is not known create an empty tree
2. Scan all files in the repository
3. If we detect files with a different checksum to what the file previously had, this indicates local changes
    (a) Mark the file as dirty
    (b) Pass the new chunks through the de-duplicator to store the new file in its entirety as compressed chunks without redundancy
4. If we detect a new file to the system, dispatch the file for deduplication and subsequently mark the file as dirty

5. Store changes in the local database and leave the startup phase and enter the runtime phase

The runtime phase is a continuous cycle of which primarily communicate with the local server to support continuous reconciliation and detect local changes. As the startup phase registered the file system watcher, the runtime phase filters these events as the OS fires them. Parallel to application usage, such as synchronization and reconciliation, the event bus is continuously listening for file system events concerning our local replica of the repository, and updates entities accordingly.

We have decided to prioritize different file system events individually in order to optimize our synchronization process and to avoid merge resolving, (conflict resolution). If a file or subdirectory is deleted these entities are immediately handled and detached from the in-memory file system tree before being stored to disk. Renaming a file simply updates the relative path and marks the entity as dirty and informs the local server of the change. Changing the binary contents of a file requires deduplication at multiple locations and are therefore not immediately managed, rather we mark the file as dirty and await until we prepare data for dissemination.

At regular intervals, the runtime phase polls the local server to determine whether any remote changes to the repository has materialized. Simultaneously it polls the local database to request dirty files if any, and if either parties respond positively it initiates file synchronization with the local server. If we detect local dirty files we create a snapshot of the current file as of this time, de-duplicate the file, and dispatch the updates to the local server. We await transportation before creating the snapshot of dirty files because Windows create hidden files called “lock files” which accommodate open files; in order to avoid forcibly taking control over these files. The snapshot can thus be uploaded safely, without damaging the integrity of the local file.

Once the daemon has processed all local files which needs to be transported, the data is transferred in memory to the local client, which is a IOCP Sender and Receiver. The synchronization protocol from client to the local server is based on simplicity rather than the ideal solution. The dirty files are sent as tuples of file GUID and the checksum of the file contents. The server responds with a list of GUIDs which are not identical or missing. The client thus sends the invalid files in its eternity. Once the final file is transferred it requests files that the client itself does not have, or have been updated, if any the transmission happens identical just from the local server to client.

6.7 Remote Transport protocol

The remote transport protocol is the methodology applied to conduct file synchronization from shore-based vessels to a land based storage facility or vice versa. The implementation of the protocol makes use of TCP as the underlying transport protocol in order to have reliable connection oriented network communication[12]. This choice was not taken lightly, however due to the volatile networks this application targets, the reliability and data flow control options favors TCP over UDP. We utilized an abstract base class to represent all communication structures transmitted through our IOCP network implementation. This class is simply our application layer [52] container which contains the current data, the type of data and the state of the sender.

Although the remote transport protocol is a stateful algorithm it has not been implemented as a finite state machine due to the fact we're working in a parallel environment in which we could find no suitable model applicable to be both computationally effective while achieving good resource management and control logic [15][32]. Therefore, we elected to use a binary sequence detailing the state of the current sender of our packet abstraction.

We denote a binary sequence packet into a 16-bit integer as the transport flags, where we currently employ 13 of the 16 bits available. This is comparable to the hardware implementation of structures where each binary digit is equivalent to a value assigned by humans. This allows us to interpret the binary sequence as a collection of flags, utilized to indicate the senders state, from whom and to whom the packet is ment and the current expected operation. For example, the sequence *1001000001100* means that the operation is currently sent from the LSP to the primary server, requesting to upload a response which contains missing data. This allows the recipient to not only respond appropriately to a given package, it allows both parties to validate that they are in fact in the same state, so that data is not misinterpreted in the system. The following example is an extremely simplified version of the receiver in the primary server daemon showing how this binary sequence encode the order of operations.

```
RemoteTransportFlags flag = remoteMessage.GetMessageType();
if (flag.HasFlag(RemoteTransportFlags.ClientToServer))
{
    if(flag.HasFlag(RemoteTransportFlags.InitialPacket))
    {
        ...
    }
    ...

    //Local server is transmitting file data to primary server
    if(flag.HasFlag(RemoteTransportFlags.FileUpload))
    {
        if(flag.HasFlag(RemoteTransportFlags.TransportMetadata))
        {
        ...
        }
        if(flag.HasFlag(RemoteTransportFlags.TransportData))
        {
        ...
        }
        ...
    }
}
```

Listing 6.2: Shows the usage of the transport flags

In this manner, we can build trees where the current packet received can only drop to its correct container, and subsequently interpreted, ignoring TCP corruption because is the only way to corrupts the flags while maintaining a valid checksum. Although the transmission of these packages are at the lowest level dispatched by the .NET provided sockets in System.Net.Sockets, we encapsulate these sockets into our own interface to achieve a higher level of abstraction. Combined with the sender registering itself with its GUID, this abstraction allows us to have a unified understanding of whom we are communicating with, and which state both participants are in.

The protocol is initiated by the sender establishing a connection with the recipient, once established, the recipient stores the UUID and the connection metadata. Each packet used in this process are serialized and compressed to minimize the traffic demands through specialized serialization protocols. The sender dispatches an initial packet to start the synchronization negotiation consisting of a list of dirty files in the form of tuples, which contains the GUID, and checksum of the local file(s). The recipient compares these values against their own copy of the replicated file(s), and responds in a specialized initial response packet.

The initial response packet consists of the GUID of the file, and the missing chunks required to rebuild the file(s). However, the response packet might be different depending on the current deduplication scheme and whether or not a delta-differential methodology is applied, it might simply respond with the GUID of the missing file, thus requesting a patch.

Once the parties have established the data required to keep the file(s) in sync, the sender begins the data transfer. The data is divided into configurable fixed size chunks and dispatched to the recipient. To accommodate the variable, but always high latency of the connections, the receiving side sockets will never time out. Therefore, the recipient cannot identify whether the connection has been truly broken during the data transfer phase. To accommodate for this predicament, once a client has a network connection again, it transfers a specialized packet encoded with a retransmission flag.

This causes the server to look through its existing connections, validate that the client has in fact started transportation of data at a prior time, and resumes the transfer at the point the connection was lost. This implies that the recipient must send a response depicting exactly how much data he received, which chunks he has and receive the remaining ones.

To indicate the final packet from the sender, it is encoded with a terminate connection packet, however this does not necessarily end the connection. If the recipient has new data the sender does not have, it is sent to the sender in using the same protocol, but switching the roles from recipient to sender and vice versa.

6.8 Remote file synchronizer

The primary methodology applied for remote file synchronization in Seadrive has been implemented as a binary difference file synchronizer. The fundament is based on Colin Percivals BSdiff in [34]. However, as the reference implementation in C[2] utilizes qfsusort [25] for the generation of sorted suffix arrays, it does not work with all files, and performs very slowly with high memory footprint. Therefore we ported the C code to C# and replaced the suffix array generation with SA-IS[33] to improve both the performance and reliability.

This allows file synchronization to utilize only one step to generate the patch, as the sender simply calculates the patch from the previous known version local to the recipient and transmits the patch directly, without needing to negotiate the results. For every dirty file in the system, both forward and backwards deltas are created in order to support synchronization from multiple version of the file. The preceding operations allows for content revision, as we can restore any file to a previous version by simply patching the file N steps in either direction.

2. http://www.daemonology.net/bsdiff/

6.9 Local Server – Local Synchronization point

The local server is a TCP based server located at each vessel that utilizes Seadrive. It serves two purposes, to synchronize the repository for each client on the vessel, and to provide a single access point for incoming and outgoing data with primary server. The local server consists of a sender, a receiver and the event driven service for managing data. Therefore, it functions as a server, a client and a service. Due to the duality of its nature, we will discuss the incorporation of new information versus the dissemination of known information separately.

6.9.1 Sending and receiving data

The appropriation of new data to the local server causes a chain of events to happen, once the file(s) has been successfully transferred. The completion raises an event that new files are ready for dissemination, and in similar fashion to the client, they are marked as dirty and de-duplicated to reduce storage. If the data originates from the primary server, the local server simply broadcast the new files to all clients connected using the transfer protocol outlined in 5.7. If the files originate from the client after deduplication takes place, it attempts to contact the primary server, and keeps contacting the remote server until a connection is established. Once a connection is established, the remote transport protocol negotiation is initiated following 6.7.

6.9.2 Local server deduplication for variable-sized chunking synchronization

The local server handles significantly more data simultaneously than the clients themselves do. In order to accommodate the increased I/O operations the files are fed into Communicating Sequential Processes (CSP) based channels [39][17], which are de-duplicated by parallel consumers which creates files prepared for transportation. These consumers receive the event from the IOCP sockets, which begin the process for consuming files. The broadcast simply fetches the resulting files from the channels whenever they are completed for dissemination.

6.9.3 Local server deduplication for binary difference synchronization

This deduplication process is simpler and can simply be managed by a single thread, which computes both reverse and regular delta patches in order to simultaneously support updating and reverting files. Although this methodology alleviates I/O resources it is computationally more expensive.

6.10 Primary Server

The primary server is a land-based computing cluster which holds all the data from all the different local-servers in use. It stores all data in the system as deduplicated compressed chunks of data, or as a list of reverse and regular delta patches. The primary server is simply a web server which can function as both a recipient and sender. Whenever it communicates with a local server it utilizes the remote transport protocol, but it also has the capability to broadcast data to each local server currently connected. It should be noted that it's a software level broadcast which disseminate a message to all participants connected to a "room".

Contradictory to the local server, the events registered in the system when receiving data is not equally worth. When a land-based entity dispatches data to the primary server, this is indicative of a high priority event, and is therefore disseminated immediately to its intended recipient.

Data deduplication in the primary server is identical to the local server, depending on the file synchronization configured for the connected LSP.

7 Experimental design and setup

This chapter outlines the experimental setup employed to evaluate the deduplication scheme for data transmission and the synchronization protocol. We examine both the design and methodology for the experiments and the datasets used. The experiments are designed to cover the responsibilities for file synchronization in Seadrive, ranging from selecting deduplication scheme to integral operations supported by the application. We do this by creating micro- and macro-benchmarks.

7.1 Datasets

The datasets used as a part of this evaluation are a folder consisting of various files, a sub-directory and a file in the sub-directory. The files are deliberately in various file-formats in order to evaluate how the system performs on differing formats. The files consist of random and non-random data for the sake of properly evaluating the different deduplication methodologies. All experiments make use of the same set of files.

All files can be viewed in the folder /Data/sync_files/", and within the set of files containing "non-random_english" and "..caesar_commentary.." are fetched from

Project Gutenberg[1]. The file "Oversized_pdf_testo" is retrieved from Princeton University[2], while the author produced the rest. In the final subset provided by the author; the random files prefixed with "random_...", are generated from a simple python script as seen in listing 7.1

Code Snippet 7.1: Python script to generate *size* random bytes

```python
import os
size = 512000000
with open('large_random_file', 'wb') as fout:
    fout.write(os.urandom(size))
```

To complement the files, we created a set of equal size to the original, which contains randomly modified files, where sections are both removed and added. The only exception is the file " random_large_file_base_modified", as we tried our hardest to create a worst-case scenario for deduplication. All modified files are suffixed with the sequence "_modified".

7.2 Experimental design

We employ two different systems for evaluating Seadrive because of the nature of the application. Micro-benchmarks are used in order to evaluate the different file synchronization methodologies and does not require more than a single computer. However, to evaluate the synchronization protocol a more sophisticated setup is required to provide a thorough analysis of its characteristics.

Therefore, we provide two different methodologies for evaluating the systems independently.

7.2.1 Micro-Benchmarks

The micro-benchmarks evaluate the performance of various file-synchronization schemes in terms of data required to patch a file, thus affecting time for transportation and the time required to create the patches. The experiments are designed in order to identify which mechanism Seadrive utilizes for file-synchronization, thus we will compare their relative performance to simulated ones.

1. https://www.gutenberg.org/
2. http://scholar.princeton.edu/sites/default/files/oversize_pdf_test_0.pdf

We have conducted micro-benchmarks on a reference implementation Octodiff[3], which is an optimized version of Rdiff, implemented in C#.net for usage in Octopus Deploy[4]. Furthermore, we utilize our own version of [5], based on the .net port publicly available on Github[6]. The original .net implementation was to slow for our needs. Finally, we compare these results to the Seadrive Binary patch utility, although their methodologies differ.

To validate the result, all micro-benchmarks were run 100 times, with no data in neither memory nor the CPU cache, and the application terminates once completed. We also manually applied the patches to the files to ensure that the processes worked as intended. We will also examine the memory and CPU-utilization to identify potential bottlenecks.

7.2.2 Macro-Benchmarks

The macro-benchmarks has been designed in order to uncover the characteristics of the Seadrive file synchronization protocol. In order to do so, we performed remote file synchronization from the Local- to the Primary server.

The taxonomy of file synchronization[2] services postulates that the most important metrics are the amount of data exchanged between synchronizing devices, computations, network size, robustness and memory. Therefore, we measured the performance of Seadrive on several different configurations. Normal land-based Wide Area Network (WAN) RTT with patches in 256 and 1024 bytes chunk size. For RTT variability, we simulated the network to have from 1 to 3 second RTT, and we tried configurations with loss-rate from 0 to 15

To validate the result, all macro-benchmarks were run 30 times, with no data in neither memory nor the CPU cache, and between each run, we truncated the database tables. We also manually applied the patches to the files to ensure that the processes worked as intended. We will also examine the memory and CPU-utilization to identify bottlenecks, and try to determine when the protocol break.

3. https://github.com/OctopusDeploy/Octodiff
4. https://octopus.com/
5. http://librsync.sourcefrog.net/
6. https://github.com/braddodson/librsync.net

7.2.3 Experimental setups

All experiments have been carried out on 64-bit Microsoft Windows based operating systems, using a Microsoft Enterprise SQL Server 64-bit for logical storage. All micro-benchmark experiments used a single machine implementation of the applications in order to be evaluated by the examinators, and were run on a DELL Latitude E7440 using Microsoft Windows 7 64-bit OS with the following specifications: Intel Core i7-4600U @ 2.10 GHz with four cores, Intel Graphics, 16GB RAM @ 2100 GHz, and a LITONIT-LMT-256 SCSI disk. We denote this machine as "reference-laptop".

The reference-laptop is used to simulate the client, and client activities in the macro-benchmarks, with two other machines for the local- and primary server. The local server is a laptop with identical specifications to the reference-laptop. The primary server is a custom built server using Windows Server 2016 64-bit Standard Edition OS with the following specifications: Intel Xeon E3-1231 V3, Socket-LGA1150 CPU @ 3.4 GHz with 4 physical cores and 32 logical, ASUS Radeon HD 5450 1GB DDR3 Silent, 32 GB ECC-RAM @ 1600 MHz, Crucial SSD(unused), and a crucial Barracuda 3TB disk @9800 RPM.

The benchmarks are not pinned to a single CPU-core despite the fact that it might provide unhindered results. Doing so would prevent the operating system thread schedulers from moving threads between cores for balancing load once interference from system calls occur. We observed this to happen frequently, most likely due to two of the test machines being laptops and Windows does this motivated by power saving benefits. However, the algorithms employed in all experiments are parallel in nature, pinning the benchmark to a single core would completely ruin their performance and not be equivalent to live systems which employs the Seadrive framework.

8

Evaluation and results

Although the main objective for Seadrive was ensuring data could be synchronized correctly between land-based systems to offshore vessels with a focus on correctness of the functionality. A tantamount facet to the usefulness of the file synchronization framework is performance and scalability, as we aim to deliver large quantities of data across several consumers in a structured manner.

In this chapter, we evaluate the experiments performed on the file synchronization framework and analyze what the results indicate for future usage.

8.1 Micro-Benchmarks

The first micro-benchmark measures the average size in bytes the methodology needs to send over the network in order to synchronize the file(s). The Rsync inspired procedures creates a signature for each file, before computing the delta file containing the required changes. They do this because the algorithm assumes no prior knowledge to the recipient files. Therefore, the total content they need to send is the signature file in addition to the delta file, although in actuality they receive the signature from the intended recipient. We will address this issue in depth later in the evaluation.

The following table 8.1 presents the data and compares it to the number of bytes changed in each file. The field "byte-differential" shows the number of bytes changed from the original to the modified file. The changes are at different locations within the files, sometimes additions, sometimes removal of data and some are restructured. It's important to note that a single change can change the entire binary content of a file, for instance appending a byte to the start of a file will cause all subsequent bytes to be "off by one", causing a large binary difference, even if the files have very similar contents.

Table 8.1: Displays the compression rate on the test-set in bytes

Filename	Byte-differential	LibRsync-Signature	LibRsync-DELTA	LibRsync-Complete	Octodiff-Signature	Octodiff-DELTA	Octodiff-Complete	Seadrive-BinaryDiff
asdf.txt	51	48	61	109	50	108	158	185
non-random_ceasar_commentary_i_v.txt	139143	3000	12614	15614	2182	12751	14933	194
non-random_english.txt	5453131	114096	11847	125943	82418	11964	94382	695
non-random_english_in_word.docx	2955660	52248	3195130	3247378	37750	3195181	3232931	3200710
oversize_pdf_test_0.pdf	85935769	1787520	28100	1815620	1291002	28331	1319333	6833
PeterCapstone.docx	1164096	22836	1092278	1115114	16508	1092518	1109026	1040333
random_large_file_base	509971950	9000012	14605	9014617	6500024	14725	6514749	39154
random_small_file_base	34280	5412	7151	12563	3924	7238	11162	161
repetetive_medium_sized.txt	4606944	5088	19946	25034	3690	53386	57076	575
smalltxt.txt	29	48	51	99	50	93	143	138
smallword.docx	9954	264	11325	11589	206	11365	11571	9040
Thesis.pdf	639867	38748	26099	64847	28000	25024	53024	311
Oversize_pdf_additionTest.pdf	159207227	1787520	60611514	62399034	1261000	59266518	60527518	40897843
Average	59239853.92307692	985910.76923076925	5002363.153846154	5988273.923076923	709754.15384615387	4901477.076923077	5611231.230769231	3476628.6153846155
% of original file size	0	0	0	10.108522432976809	0	0	9.4720544686950561	5.868732593262342
Compression-Rate %	0	0	0	89.891477567023188	0	0	90.527945531304937	94.131267406737663

The reason the algorithms performs so well is because they are all based on finding equal patterns for usage in updating the files to a new version. When comparing the differing methodologies, we see that the Rysnc-based methodologies perform very similar in compression, both in the signature and delta file. Only differing a total of ∼0.9% on their average size required to transport in order to update a given file. The noticeable entry is the binary-diff mechanism; it produces on average 4% smaller patch sizes, although it massively outperforms the remaining methods on the large binary file with random data and the oversize_pdf with additions, accounting for this, the difference is still 4%. We also note that on small files, we observe some overhead caused by the approaches, which in some cases are larger than the changes itself.

We also measured the time it took for these operations to finish, i.e. the time it took for signature generation, delta generation or to generate the binary diffs. To measure the time, we use the Microsoft Stopwatch Library built into the .NET runtime to measure the latency of each operation. We measure only the latency of the operation itself; specifically, we measure only the generation of signatures, delta files or the binary diff, not the initialization of the application, nor the clean up before finishing. E.G:

Code Snippet 8.1: C# code to measure time

```
Stopwatch sw = new Stopwatch();
sw.Start();
SeadriveRsync.ComputeSignature(path, outputFullPath);
sw.Stop();
```

In order to accurately measure the aforementioned procedures, we mitigated them into their own applications, where we ran a different program to clear out the CPU-cache, writing and reading 8-MB of garbage data to ensure no cache hits would interfere with the measurements of the code in question. We do this because data might reside in virtual memory of the application or the CPU-cache, furthermore since we run the programs in execution scripts some might have been cached to the surrounding application.

Table 8.2: Shows the average run time in order to create delta-differences in milli- and regular seconds

Filename	LibRsync-Signature	LibRsync-DELTA	LibRsync-Complete	Octodiff-Signature	Octodiff-DELTA	Octodiff-Complete	Seadrive-BinaryDiff	
asdf.txt	29	11	40	4	105	109	53	
non-random_ceasar_commentary_i_v.txt	5	45	50	7	125	132	738	
non-random_english.txt	164	269	433	104	381	485	8105	
non-random_english_in_word.docx	60	3762	3822	52	287	339	22228	
oversize_pdf_test_o.pdf	1831	3179	5010	1546	3958	5504	220915	
PeterCapstone.docx	23	1399	1422	21	171	192	6501	
random_large_file_base	8709	18339	27048	9	22482	22491	1153983	
random_small_file_base	7	17	24	8	102	110	211	
repetetive_medium_sized.txt	7	202	209	4	297	301	3107	
smalltxt.txt	1	2	3	4	108	112	9	
smallword.docx	1	16	17	4	109	113	48	
Thesis.pdf	52	67	119	41	124	165	2240	
Oversize_pdf_additionTest.pdf	1969	78611	80580	1272	85100	86372	586557	
Average	989.07692307692309	8147.6153846153848	9136.6923076923085	236.61538461538461	8719.1538461538457	8955.7692307692305	154207.30769230769	
average in Seconds	0.98907692307692308	8.1476153846153849	9.136692307692309	0.23661538461538462	8.719153846153846	8.9557692307692314	154.20730769230769	

Table 8.3: Shows the time to transfer the delta-files over various dataplans in hours

	libRsync	Octodiff	Seadrive-binary-diff
8 kb/s	1.66 H	1.55 H	0.96573 H
4 kb/s	3.32 H	3.11 H	1.93146 H
512 bits/s	25.990 H	24.35 H	15.089 H

The latency-measurements unsurprisingly shows that sliding window protocols are significantly faster than an algorithm based on suffix-array sorting. They run in an almost negligible time in our use case, with Octodiff as the fastest method clocking in at ~8.7 seconds. We observe a significant overhead of running the Seadrive binary diff protocol, it is ~17 times slower than Octodiff at producing the patch-files and uses on average 154 seconds to produce the patch.

Combining the two tables, we can see that in order to select algorithms for synchronizing files one must look at the surrounding system and its intricacies. In Seadrive, we are specifically dealing with two types of networks, Geo-Sync network with 500-600 Milliseconds (MS) RTT and a low-orbit mesh-system with 500-5000 MS RTT. These networks provide a maximum bandwidth available to the application at 8KB/s, 4KB/s and 512 bits/s, where the maximum throughput is dependent on the data plan. For the following results in table 8.3, we measure the time in hours required to transfer the average data required to successfully synchronize the files. We ignore the costs of multiple transmission required by the Rsync algorithms, as they are negligible in the time windows presented.

We observe that all protocols create files so large that the time spent constructing the various files have almost no impact on the overall time required for synchronizing the files. We note that the time saved by the Seadrive-binary diff methodology is significant, by a 1.5 order of magnitude. This causes the time spent to generate the binary diffs insignificant; furthermore, it saves a huge fiscal cost, as each byte sent is equivalent to monetary values. By the virtue of these results, the binary diffs are clearly the optimal method to save both time and monetary values.

8.2 Macro-benchmarks

Due to the volatile nature of satellite-based networks, we were sadly not able to test a real-life implementation on the networks set up for experimental evaluation. The network we had access to, a geo synchronized Inmarsat network, never successfully synchronized for data transfer. Therefore, we set up a configuration on a LAN network to test the resource consumption on the primary server during active usage. Simulation technology could show some parts of how the system works on low-throughput high latency networks but will not detail how it actually performs. However, dropping the connection can easily be simulated through killing the connection during transfer. To measure the intricacies of the network connection such as actual TCP packets and packet types, we used Wireshark[1] to listen on the exact connection filtering only application usage. We used Perfmon.exe to measure the total CPU utilization of the execution and RedGate ANTS[2] Performance profiler for virtual memory, CPU-time within the application, thread counting and I/O operations.

8.2.1 Full application usage – Window size 256 bytes

The first session reported a total of 21825 TCP packets with an average length of 1053 bytes and no packages in the range of 0-39 and > 2560 bytes. Because we were stress-testing on a high throughput network we were able to observe whether the server was able to properly respond to all packages, which it was. This indicates that the server at even such a small window size on the chunks is capable of parsing the data from a client at much higher speeds than it would in a real-case scenario. The whole session lasted about 5.6 seconds on average.

1. https://www.wireshark.org/
2. http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/

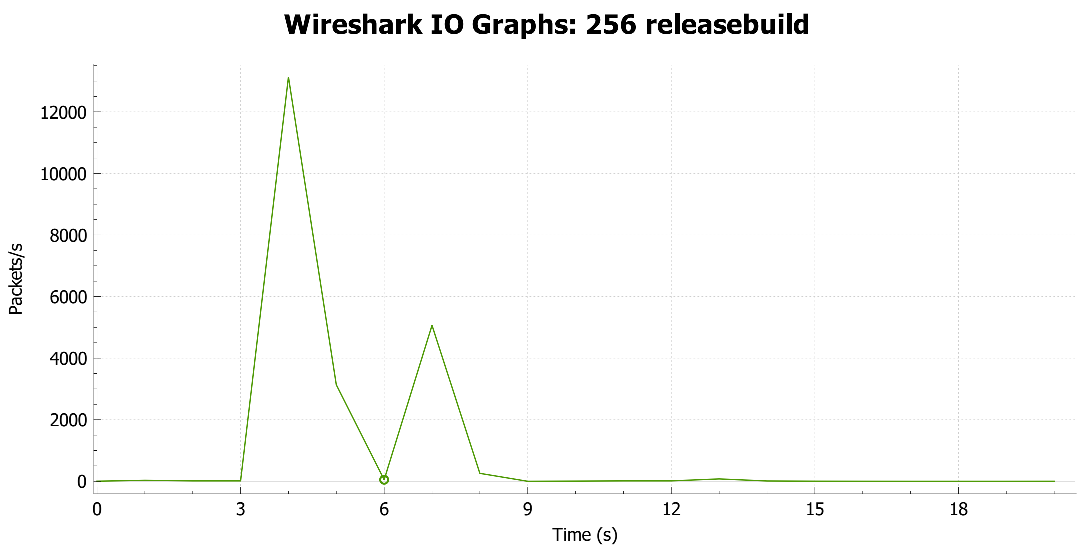

Figure 8.1: Shows the IO graph for the network communication between the local server and primary server. The green ring indicates where we killed the connection

We see from the graph in figure 8.1that a stateful implementation for both client and server does not prohibit the client from sending the data in large burst, although a real-life satellite network would limit this to 8kb/s or lower, worth of packages. Interesting to note is that the algorithm is able to quickly increase in speed once the local and primary achieve re-transmission. However, due to this being a simulation it is advisable to be skeptical to the results.

The primary server reports a total of 6% CPU-load distributed across 8 logical cores. We define these 6% of CPU-load as 100% of the application cumulative load. We observed the application to utilize 34 threads to receive and disseminate the incoming data from the network card, three for applying the binary patches in combination with the application main thread. This results in a total of 38 threads. Most of the CPU-time was spent retrieving and pushing data from the network card, accumulated at 81.642 % of the total cumulative load. The second most CPU-intensive task was patching the files, accounting for 17.013% of the application total. This implies that the synchronization protocol, i.e. the protocol which decide where disseminated packages are placed and interpreted only required 1% of the application total. We observed that we spent on average about 0.45% time in the garbage collector and utilized a maximum of 632 MB RAM. This is caused by the fact that the .NET virtual machine does not perform garbage collection during CPU -intensive tasks nor does it so until required.

8.2.2 Full application 1024 byte window size

Increasing the window size sped the application up by 0.6 seconds on average, but used more TCP packages reporting a total of 22428 packages with an average size of 1110.5 bytes. Similarly, to the previous result, we observed no irregularities in the packages, and that the protocol worked as intended.

Perfmon reported on average 5% CPU utilization, 1 % less than on 256-byte window size. Just like the previous experiment, most of the application time was spent waiting for and disseminating packages, totaling of 84% of the overall time. We observed the number of threads to be identical, probably due to the data size staying the same. However, we noticed an increased amount of data stored in RAM, peaking at 750 MB RAM, and we spent zero time in the garbage collector until the final file was transferred. During the execution, we achieved a total of 456 MB in I/O data bytes during the patching of large files, which roughly translated into 74.239 data operations per second. Based on the trends we saw, it indicates that larger chunk sizes trades CPU-time for memory load, thus making the application faster.

8.2.3 Simulated delay sessions

All simulated delay sessions used the configuration that had 1024 byte chunk sizes, as it seemed to run slightly faster than the 256 bytes. We used the clumsy[3] 0.2 software to simulate the network environment from the local server to the primary server. We simulated the connection first using a realistic RTT of about 1 second, with no additional packet loss. This did not affect the program in any significant manner as the application was only 1 second slower and the packet graph in 8.2 looked almost identical to the configuration without delay in 8.1.

3. https://jagt.github.io/clumsy/

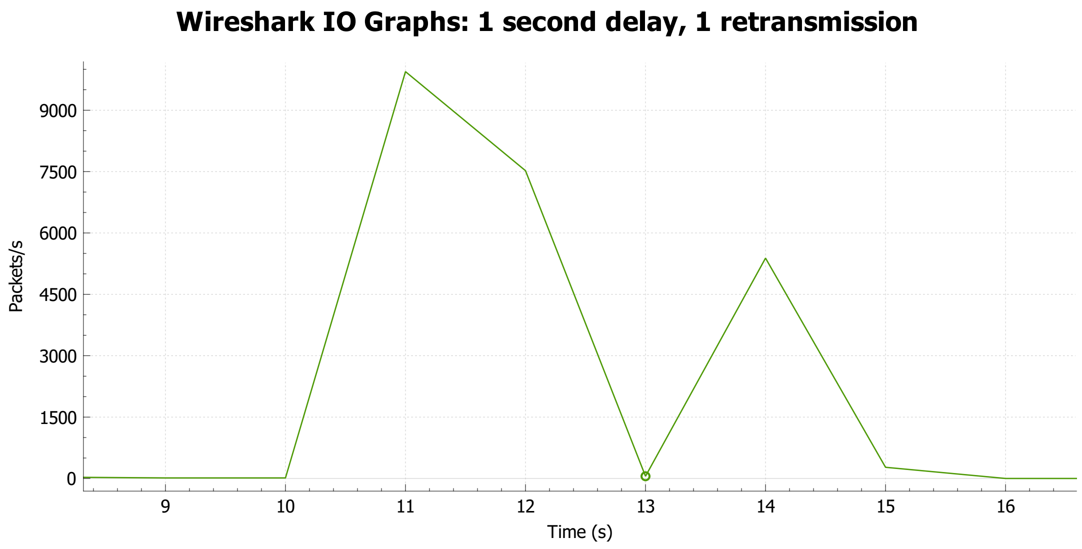

Figure 8.2: Shows the IO graph for the network communication between the local server and primary server with 1 second RTT. The green ring indicates where we killed the connection

However, increasing the delay to 3000 MS even without packet lossage, the system grinds close to a halt, as shown in the following figure:

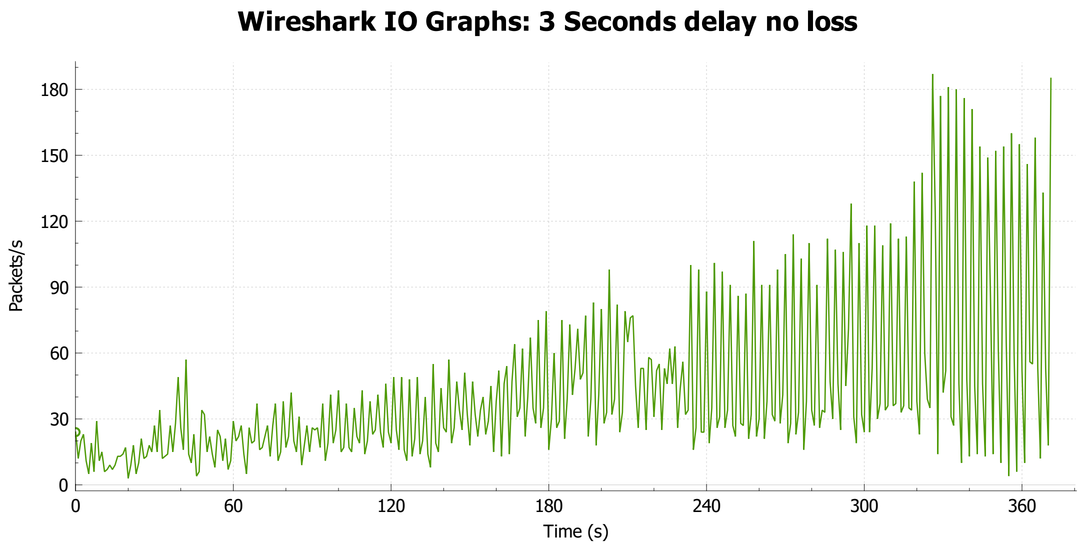

Figure 8.3: Shows the IO graph for the network communication between the local server and primary server with 3 second RTT, no retransmissions

The system will eventually complete the operation, although at a significantly slower rate. We also simulated packet loss over the network and at about ∼10% packet loss or higher the application grinds to a halt. This shows that the application does not cause significant overhead, which affects the transmission of data, rather that we must abide by the rules of TCP. The network traffic data shows that we simply follow regular transfer of data over TCP.

8.3 Analysis

We refer to the taxonomy of synchronization protocols outlined in section 7.2.2 when we evaluate the performance of the application itself, with primary focus on the file transfer protocol. The most critical aspect of a file synchronization protocol is the amount of data exchanged. The metric is particularly important both for scalability and because it directly affects the time required in order to complete the synchronization.

In Seadrive, we propose to utilize a binary-delta differential approach in order to minimize the data transmitted over WAN, however we must know how much data we actually send on average compared to the ideal amount of 8MB. The 256 chunk size configuration sent on average 21825 TCP packets with Ethernet encapsulation. This means the underlying transmission protocol had 54 additional bytes of metadata per packet (14 from Ethernet, 20 from IPv4 and 20 from TCP). This means the total overhead for aforementioned session is 1.17855 MB. The 1024-byte configuration which used on average 22428 TCP packets had an overhead of 1.211 MB.

The fact that Seadrive utilizes a high-level serialization scheme for the different types of packages in order to alleviate the developers and the application level state implementation with negotiations creates a significant amount of overhead. The entire conversation from start to finish, utilized a total of 25 MB, an overhead of ∼15MB. In light of this information, we need to either reduce the scheme down to simply sending bytes and interpret it based on hard coded variables, or modify the remote transport protocol to send less information. Although these luxuries can be afforded on land-based systems, this will have significant fiscal cost on an offshore based system.

The system does not require as much computation as expected when considering the binary delta differential approach. Furthermore, it is possible to simply mark the files as dirty during transmission allowing the system to perform this computation should it ever interfere with transmission of data. Even at large loads the system is able to process the files without any problems and does not affect the running time of the application, nor use substantial amount of resources.

The system network size is a fundamental aspect of the scalability of the overall system, and Seadrive performs well in this category by basing its design on AFS. This allows us to communicate with $N/boats$ rather than N clients, thus increasing scalability. Because we utilize patch files for synchronization there is no need to lock and important system files even when patching, as the operation can take place in memory. The only drawback is that the primary server serves as a single point of failure and can disable the entire system, where a distributed protocol could allow for more connections.

When a client loses its connection to the primary server, the system allows for retransmission of data without significant overhead, only $32+(4*successfullytransferredchunks)$ bytes. This shows that the synchronization protocol offers a form of robust data transfer, however due to the usage of a primary server it also contains a single point of failure. A distributed methodology could reduce the risk of synchronization failure, but could be significantly more expensive in monetary resources. This would require several entities in the system to perform data transfer which for a company could be very expensive as more recipients would be involved.

The memory overhead of the application is smaller than expected; the protocol itself contains very little overhead. However, doing the binary patches in memory instead of through buffers on disk causes us to use more memory than necessary. Because we have mitigated the response for the remote file synchronization algorithm to work between two powerful computing entities (personal pc to powerful server and vice versa), we have sufficient memory in all use cases. The clients connected to the local server does not require much memory as files are streamed, thus only buffering the required data on demand.

8.4 Discussion

The Seadrive file synchronization framework are able to synchronize files to a remote location without losing data. It provides a robust interface which allows lost connections to continue the synchronization where it left off, thus resulting in less overhead rather than retransmitting all missing content of that file. We observed the remote protocol to function well on good network connections. As we do not modify TCP nor implement reliable connections over UDP, we are at the behest of the underlying transport algorithm during high delay and/or loss rate.

Contrasted to traditional file synchronization- and file hosting services, we observed that using a binary delta-differential methodology as the synchronization protocol were able to reduce the data size required to patch a file, thus lowering the demands for the underlying network. Due to the controversies surrounding the use of global deduplication concerning privacy, and the cost of rebuilding these files, we have not applied such a scheme to our system, even if it can reduce network communication.

We demonstrated that the protocol works close to normal on 1-second delay on both incoming and outbound packages. We further observe that the few responses generated by the recipient in the algorithm allows the sender to fully utilize the connection for data transmission, thus only being limited by the underlying connection while still maintaining the context required to patch a file. The primary servers' statefulnes neither does not interfere with the transportation of data, nor does it hinder the robustness of the application, rather it aids it.

We observed several drawbacks and limitations of the framework, especially concerning the file synchronization protocol. Primarily, the system is required to manage old file versions, either store the binary contents or as we do, keep the old checksum stored. Using our methodology, the old patches must also be kept in storage so that entities with N different versions can patch the file. In order to provide robustness if not all file versions are stored, we must fall back to Rsync/Rdiff based algorithms in order to successfully synchronize the file(s).

Our choice to keep programmable models easy, relying on the binary formatting and serialization through the .NET virtual machine to use high level classes comes with overhead in data transmission. This makes the system have a higher cost in monetary, time and processing resources. The new binary artifact for the primary server is also lacking in a standard deduplication scheme, which requires further research.

The application provided insights into file synchronization frameworks at several levels, and creating a robust software level transport protocol was necessary in order to support offshore systems. We have observed that binary delta differential algorithms might have a future for high latency low bandwidth networks and that the overhead in clock-time required for such methods are negligible for very slow internet connections.

8.4.1 Lessons learned

In the process of developing this application, we discovered several lessons to be learned, both in engineering practices and commonly applied knowledge. We learned that most traditional out of the box software does not work on offshore vessels due to a high chance of connection drop. This causes the program to run on infinite loops as the IP addresses might change. Furthermore, we observe that the traditional methodologies does not prove optimal in such environments because minimizing transfer size is more important than calculating the file differences fast. Finally, we learned that processing time is not an important factor, in fact, it might be considered the least important factor as the time spent transporting data is so large.

9 Concluding remarks

File- synchronization and hosting services has become an integrated part in in people's lives, such as Dropbox, Unison and Rsync. In order for the aforementioned type of services to accommodate offshore users, it is essential to understand why the existing systems breaks, and how we can provide reliable synchronization.

In this thesis, we have described and analyzed key issues when performing file synchronization and some of the most popular frameworks. We have developed Seadrive, a novel file-synchronization framework to solve the file synchronization problem in the domain of offshore entities, and compared the output size and speed to generate differential files. Seadrive outperform traditional methods for generating file differentials, but are slower to do so. We analyzed the remote transport protocol and provided insights into the scalability of the architecture in order to identify possible bottlenecks. We observed the protocol to function as well as TCP does during remote synchronization and that retransmission where you continue from whence the connection broke, is possible. The Primary server is the most critical entity in the system and its resource consumption is modest.

9.1 Future work

The Seadrive framework and application requires thorough investigation in deduplication techniques that works in combination with binary delta generation. Methodologies employed such as ZFS Deduplucation[1] are of interest, however, as we have discovered throughout this thesis, delving into new schemes might be beneficiary.

The system does not use global deduplication as previously outlined. Therefore, small repositories does seldom have the chunks required to de-duplicate a file, thus the file must be sent in its entirety. In order to avoid such cases, we want to investigate the usage of similarity searches in order to locate files close to the entity in question, thus creating a patch, which can create the new file from an existing one. As suffix array generation in SA-IS is closely related to similarity search, we believe this to be an interesting route worth examining.

The Seadrive file synchronization framework has been implemented and evaluated as part of this thesis in the context of offshore-based file synchronization. However, delta-differential algorithms as part of file distribution is applicable to many fields, such as data distribution or one-way file synchronization. Analyzing the binary delta approach for green computing could provide further insight into the advantages and disadvantages for this algorithm usability on embedded devices.

For the application, Seadrive needs to reduce the metadata overhead required for serialization, although having a high level class interpreted directly from transferred data is beneficiary for developers, this will increase fiscal costs for customers. Therefore, the implementation should mitigate the usage of high-level classes down to binary arrays. This should be straightforward due to all operations encoded with a preceding binary field denoting the transport packet.

For robustness and real practical results, the system needs to be evaluated on a real satellite based network and not only on simulated networks. This will reveal the real characteristics of the application and the most critical areas to improve upon and optimize. Furthermore, the transport protocol requires further investigation in order to reduce the amount of messages sent to combat the varying rate of packet loss.

Having specialized algorithms and protocols based on the different file formats and also incorporate database systems are part of future work and was out of scope for this thesis

1. https://blogs.oracle.com/bonwick/entry/zfs\dedup

9.2 Conclusion

As part of this thesis, we have successfully designed and implemented a novel file-synchronization framework and we have placed special emphasis on the remote transport protocol. Our contribution delivers a new methodology to remotely synchronize files, which seems to work satisfactory for their current usage. The files are harmlessly deliver to the remote recipient, which can process the files in parallel to its other responsibilities.

We were initially surprised to see the reduced size a binary-delta approach produces compared to the Rsync-based protocols, however we understand why academia and most software utilize this methodology. There is seldom cases where reducing the file size compared to producing fast patches benefits the application thus causing the process to speed up.

Based on the findings in this thesis the Seadrive file-synchronization framework works satisfactory for its purpose, but future work can provide additional benefits at almost every layer of the application.

Bibliography

[1] 2016. URL `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365198(v=vs.85).aspx`.

[2] Sachin Agarwal, David Starobinski, and Ari Trachtenberg. On the scalability of data synchronization protocols for pdas and mobile devices. *Network, IEEE*, 16(4):22–28, 2002.

[3] Sundar Balasubramaniam and Benjamin C Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108. ACM, 1998.

[4] Dave Cannon. Data deduplication and tivoli storage manager. *Tivoli Storage, IBM Software Group (September 2007)*, 2009.

[5] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[6] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[7] Landon P Cox, Christopher D Murray, and Brian D Noble. Pastiche: Making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, 36(SI): 285–298, 2002.

[8] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[9] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 205–212. ACM, 2013.

[10] Intel e Business Center. N-tier architecture improves scalability, availability and ease of integration, 2001. URL `http://cadeiras.iscte.pt/CDSI/fich/N-tier%20Architectures-Intel.pdf`.

[11] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30:2005, 2005.

[12] Kevin R Fall and W Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[13] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 394–400. ACM, 2005.

[14] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[15] A. Girault, Bilung Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, Jun 1999. ISSN 0278-0070. doi: 10.1109/43.766725.

[16] Deepak Gupta and Kalpana Sagar. Remote file synchronization single-round algorithms. *International Journal of Computer Applications*, 4(1): 32–36, 2010.

[17] Charles Antony R Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[18] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[19] John H Howard et al. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center, 1988.

[20] Utku Irmak, Svilen Mihaylov, and Torsden Suel. Improved single-round protocols for remote file synchronization. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1665–1676. IEEE, 2005.

[21] Adrian Kearns. 5-layer architecture, November 2010. URL `https://morphological.files.wordpress.com/2011/08/5-layer-architecture-draft.pdf`.

[22] David Korn, J MacDonald, J Mogul, and K Vo. The vcdiff generic differencing and compression data format. Technical report, 2002.

[23] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *FAST*, pages 239–252, 2010.

[24] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[25] N Jesper Larsson and Kunihiko Sadakane. *Faster suffix sorting*. Citeseer, 1999.

[26] Allen S Lee and Richard L Baskerville. Generalizing generalizability in information systems research. *Information systems research*, 14(3):221–243, 2003.

[27] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *Middleware 2013*, pages 307–327. Springer, 2013.

[28] Ravindra Mahabaleshwar. Effective data deduplication implementation, 2011. URL `http://www.tcs.com/SiteCollectionDocuments/White%20Papers/HiTech_Whitepaper_Effective_Data_Deduplication_Implementation_05_2011.pdf`.

[29] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008.

[30] Theresa C Maxino and Philip J Koopman. The effectiveness of checksums for embedded control networks. *Dependable and Secure Computing, IEEE Transactions on*, 6(1):59–72, 2009.

[31] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 174–187. ACM, 2001.

[32] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-

parallel finite-state machines. *ACM SIGPLAN Notices*, 49(4):529–542, 2014.

[33] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE, 2009.

[34] Colin Percival. *Matching with mismatches and assorted applications*. University of Oxford, 2006.

[35] Benjamin C Pierce. Foundations for bidirectional programming. In *Theory and Practice of Model Transformations*, pages 1–3. Springer, 2009.

[36] Benjamin C Pierce and Jérôme Vouillon. What's in unison? a formal specification and reference implementation of a file synchronizer. 2004.

[37] Michael O Rabin et al. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[38] Norman Ramsey, El Csirmaz, et al. An algebraic approach to file synchronization. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 175–185. ACM, 2001.

[39] Bill Roscoe. The theory and practice of concurrency. 1998.

[40] Jin San Kong, Min Ja Kim, Wan Yeon Lee, Chuck Yoo, and Young Woong Ko. Multi-level metadata management scheme for cloud storage system. *International Journal of Multimedia and Ubiquitous Engineering*, 9(1):231–240, 2014.

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

[42] Alex Spiridonov, Sahil Thaker, and Sourabh Patwardhan. Sharing and bandwidth consumption in the low bandwidth file system. Technical report, Citeseer, 2005.

[43] Mark W Storer, Kevin Greenan, Darrell DE Long, and Ethan L Miller. Secure data deduplication. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10. ACM, 2008.

[44] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization, 2002.

[45] Torsten Suel, Patrick Noel, and Dimitre Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 153–164. IEEE, 2004.

[46] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging semantics for conflict updates in geo-distributed file systems. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 10. ACM, 2015.

[47] Andrew Tridgell. *Efficient algorithms for sorting and synchronization*. Australian National University Canberra, 1999.

[48] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[49] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[50] Haiyang Wang, Ryan Shea, Feng Wang, and Jiangchuan Liu. On the impact of virtualization on dropbox-like cloud file storage/synchronization services. In *Proceedings of the 2012 IEEE 20th international workshop on quality of service*, page 11. IEEE Press, 2012.

[51] Hao Yan, Utku Irmak, and Torsten Suel. Low-latency file synchronization in distributed systems.

[52] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980.

/A

Appendix 1

Appendix 1: Data abstraction layer design

3.2 The data abstraction layer

The Data Abstraction Layers primary purpose is to provide unified access all data available to the system. Subsidiary is it required to lessen the burden of managing data, resolving issues such as weak typing of the business data, centralizing data-related policies and applies a domain model to simply business logic. We also want to have a resolution of concurrency problems which may arise with conflicting requests in the same time window.

The data in eSushi is stored in two different formats, Microsoft Enterprise SQL Server databases and NetCDF/HDF-5 [8] [9] files. All files added to the system, are either in, or converted to these formats. This choice was made to create simple and unified data access patterns, thus alleviating the common problems of dealing with large quantities of heterogeneous data, also known as a high degree of variety. However data used internally by the decision support system in the calculation cluster might use intermediate data-sets of different formats, however they are believed to be outside the scope of this thesis.

At the lowest level data access in eSushi is provided through pure SQL and file read/write. The choice of supporting .SQL scripts and NetCDF/HDF-5 files is for time-critical applications which require schema/array-based data as quickly as possible. On top of the raw access, the system divides into two different subsystems taking advantage of the speed of raw access to populate models of the users' choice and high level abstractions to interact with the user. In this submission we will show the complete cycle of both methods, but only on MSSQL as data storage, as the code artifacts for NetCDF/HDF-5 contains trade secrets.

3.3 Data Models and access patterns

Through the course of this thesis we will use the terms simple- and complex models interchangeably with simple- and complex access patterns to denote the same mechanisms. We denote a data model, or data access pattern as a method of managing external and remote data sources. The data models provided in our decision support framework can be divided into two parts; simple and complex models. This division of the models is a deliberate design choice, based on the problems they attempt to solve. These models provide data access in differing methods which will be further expanded upon in the following subchapters.

The primary concern for us as developers is simple access to data throughout the complete application stack, regardless of placement. The model should also be simple to use and require little programming to accomplish their requests. However a comparable concern, especially when dealing with large data sets, is speed. Data should be delivered within a user-specific definition of reasonable time, thus creating a more complex method of retrieving data, as there's a tradeoff between speed and complexity. Allowing simple access at less speed also reduces the time to develop interfaces and applications based on existing data, thus alleviating the finances of further software engineering on the project [10]. We will show the design for the two different approaches separately and compare them to each other in the experiments chapter. For both methods we will show the design from the bottom up until the full flow has been described. An overview of a subset of the DAL describing the data access patterns can be seen in the following figure:

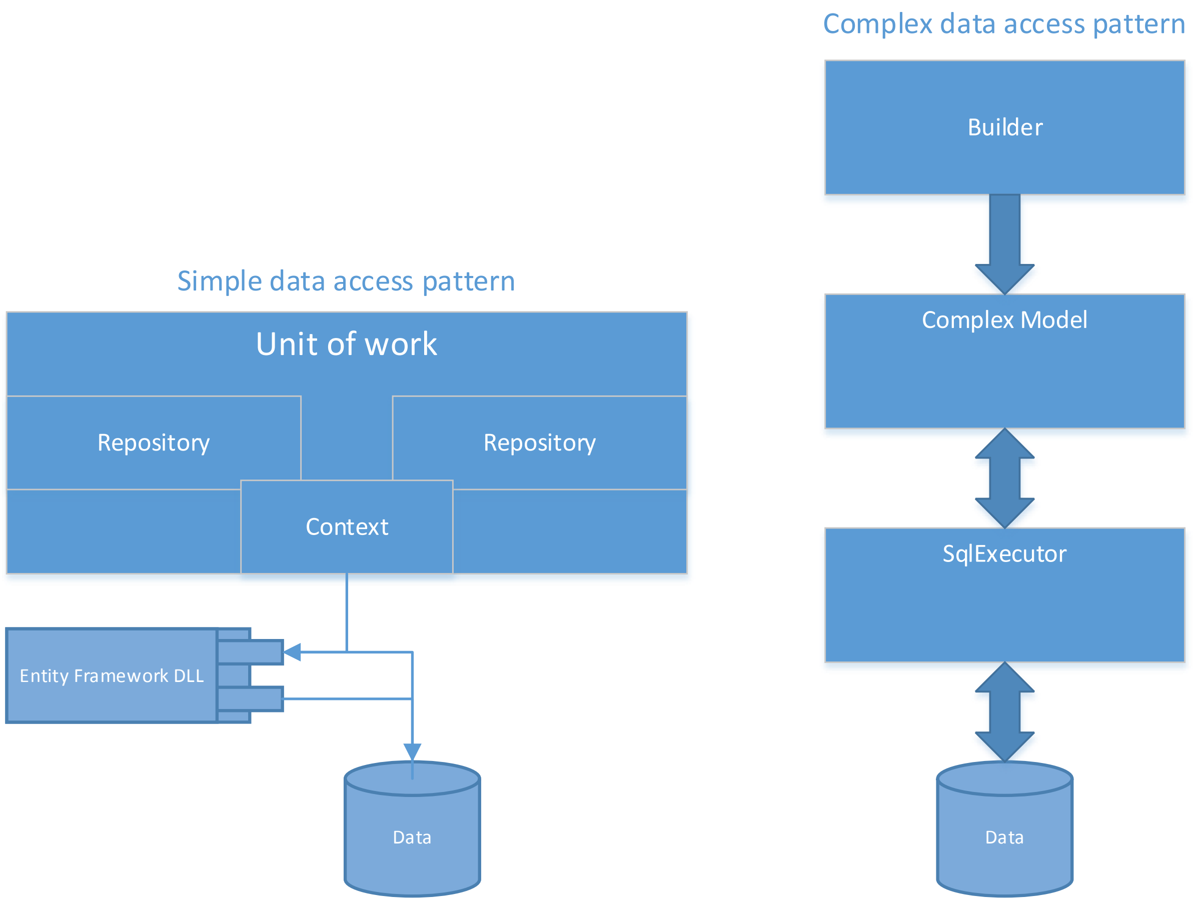

Figure 4: Simplified design of the data access patterns

3.3.1 Simple data models

The simple models bare many similarities to the classical Data Access Pattern entities (also known as data access components in literature and throughout MDSN), meaning that the model correspond to exactly one entity in the system. In this context an entity is defined as either a database object or a NetCDF/HDF-5 object, connoting to a simple mapping with a one-to-one correspondence with a table schema or the metadata description object described in the NETCDF/HDF-5 file. This allows users a complete overview of all available data stores in the system at a higher abstraction level, namely a class in the selected programming language, without any consideration to how the data is retrieved. [11]

The simple data models in eSushi can easily be compared to the data models in the Entity Framework [12] without the excessive boilerplate which arises when creating Ado.Net data models. The simple data model usage is exactly the same without 400 lines+ generated for each model, thus alleviating the code base of this burden, simultaneously lessening the technical debt which can arise with auto-generated code.

3.3.2 Generic repository

In applications with many different clients retrieving data, writing data, and data management consists of many cross-cutting concerns. In order to manage these concerns, especially when they have complex domain models, it's often beneficiary to form a layer which isolates domain objects from details of database code. The repository pattern was popularized in the book "Patterns of Enterprise Application Architecture", by Martin Fowler Et al, and the best description of the pattern can be found in the book itself:

"A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers." [13]

Therefore a repository strategy was selected for the simple models when providing easy access as it simplifies data-related policies such as caching. However the repository method for providing data access can in itself easily overburden the programmers with tons of code if each model provided its own data rules. Therefore for eSushi we created a generic repository which works with any entity in the system with all the described properties, the repositories exposes a simple CRUD interface with supports for filters. However in order to consolidate access rules for specific data store, they have their own generic repository to ensure proper usage across applications.

3.3.3 Unit of Work

The generic repository removes redundant code and the effects of partial updates. For instance two different entities within the same transaction might utilize different data stores as their contexts (two different databases/netCDF-HDF-5 files or a combination therein), one might fail silently where the other succeeds. In order to ensure all repositories runs within the same data store context, thus coordinating all transactions, is to use a Unit of Work.

In practicalities this means that the unit of work maintains lists of objects affected by a business transaction and coordinates the writing out of changes the resolution of concurrency problems. The unit of work for eSushi has been designed to avoid changing the databases for each change in the object models, because it ends up becoming very slow. Furthermore, interactions spanning for a long time or across different data stores can cause inconsistent reads if you're required to keep track of all objects. The key functionality provided by the Unit of Work is deciding what happens at commit time. It opens the transactions, does any concurrency checking, and writes the changes; thus alleviating the need for applications to explicitly call updates on the data-stores.

3.4 Complex data models

The complex models are complex in the amount of code needed to utilize their functionality, not in their methodology. The complex models are simple objects containing the resulting data one entity at a time after retrieving and performing calculations on the data storages. This allows the user both to tune exactly what output is expected, while simultaneously populating a high level abstraction. However, this abstraction contains a burden in code artifacts as they also contains a Data Abstraction Layer Query which defines the means for communicating with the underlying store, the method for retrieving data, and the calculations required for their data sets. The DAL queries utilized by the complex models are created through the usage of what we call builders, which allows the users to quickly create advanced queries which can use several data stores in their operations.

3.4.1 Execution engine and builders

The builders are a set of specific objects used to create DAL queries. They have a base which is shared amongst all builders to simplify the operation and are exposed to a generic interface for datatypes. This ensures the resulting dataset to be interoperable with the complex model. They utilize a builder pattern [7] to manage the parameters required and lessen the probability of errors.

The execution engine is simply a generic engine to execute the resulting command contained in the DAL Query. It starts by opening a connection to the necessary data stores, once the store is connected; it performs the specified commands, and the resulting set of data is disseminated back into the model.

/B

Appendix 2

Appendix 2: Data abstraction Layer implementation

4.2 Data Abstraction Layer

The data abstraction layer has been implemented in C# with dependencies to the System Configuration, SQL server runtime, and the Business Logic Layer. It also has the Entity Framework package installed. Here we will outline the implementation of both the simple and complex data access functionality and give a detailed exposition of their usage.

4.2.1 Simple data access

The basic entity in the simple data access pattern is the object mapped model (1 to 1 mapping between code class and data-store entity), which is exposes to the developer as a common class. This is the core entity in which data is populated, and the developer receives either a single entity or a collection of entities. The generic repositories are implemented supporting a simple Create, Read, and Update API (CRUD), with additional support for filtering and reordering, done by user specific methods. Figure 4 shows the relationship between the different entities which constitutes the simple data access model.

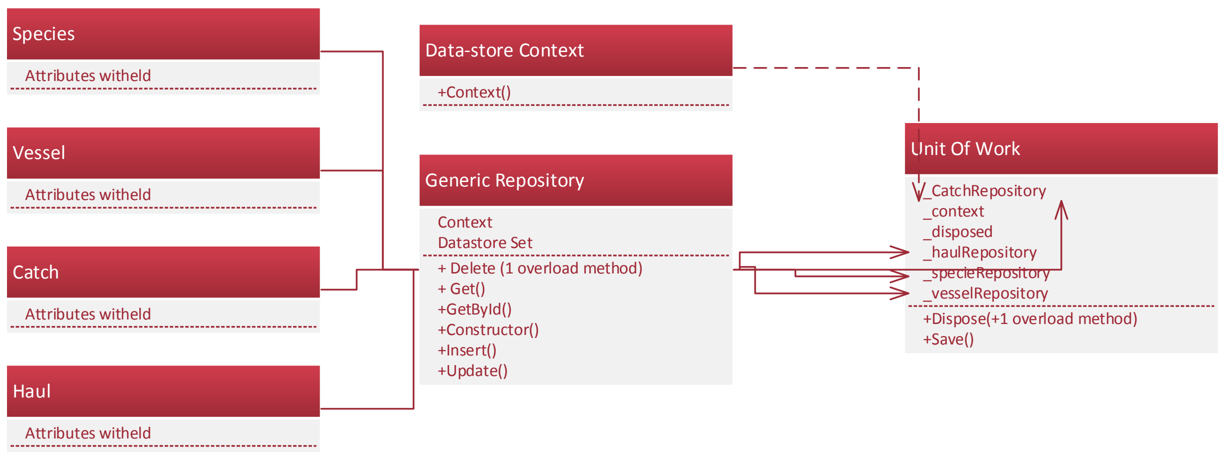

Figure 5: Shows the relationships within the Simple Data Access pattern

The generic is repository functions as a container for an entity within the framework and concrete instantiations are contained in the Unit of Work, once for each entity in the system. By combining the repositories with the unit of work, we can cover the desired aspects described in the design, while keeping code artifacts minimal.

Combining the repositories with the Unit of Work allows for new entities to be added to the system with just two lines of code, while simultaneously enforcing the data access rules. The data flow interacting with the Unit of Work can be seen the Figure. 5, where the mechanism of the repository layer will we expanded upon later.

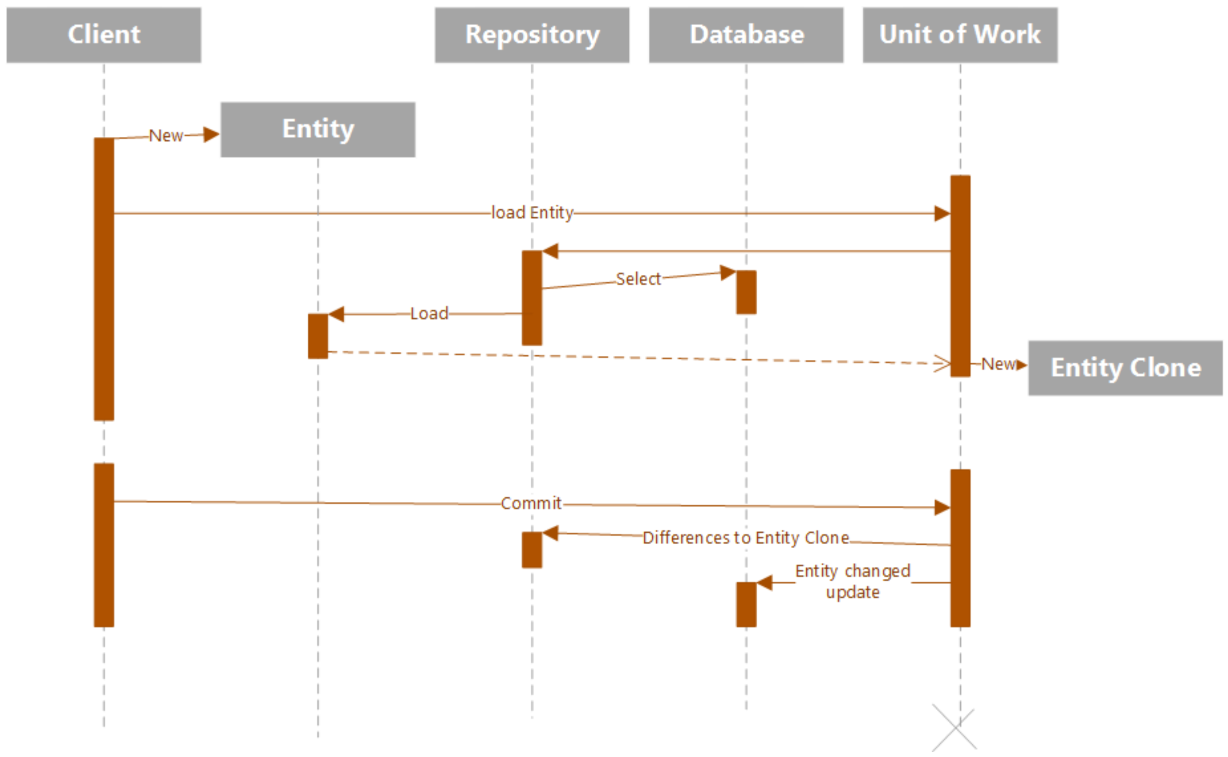

Figure 6: Displays an overview of the data flow in the Simple Data Access Model, without the details of the repositories

To add a new entity to the system, the developer adds a property to the Unit of Work, let's call it _*specieRepository* and thus create a getter for said repository.

```
        public GenericRepository<Species> SpecieRepository
        {
            get { return _specieRepository ?? (_specieRepository = new
GenericRepository<Species>(_context)); }
        }
```
Code 2: Shows a getter for a concrete instance of the generic repository

The execution flow of the simple data model can be segregated into two parts, input and output. In this definition output is requesting stored data in the system; we will show its usage through an example, in this instance the developer requests the Norwegian name of all fish species known today:

```
var species = _unitOfWork.SpecieRepository.Get().Select(specie => specie.NName).ToList();
```

Code 3: Complete code required to generate a list of all species filtered by their Norwegian name

This line of codes starts a chain of events, first we request that the unit of work should collect all the data from the repository of fish species, secondly we request that the output data should be filtered to only return the Norwegian names. The specie repository connects to the data-store using the context provided by the Unit of Work. The data-store retrieves the values requested and filters them based on the function inputted through the usage of LINQ. Lastly it converts it into a list for convenience. The data flow interacting with the repository layer is outlined in the following model:

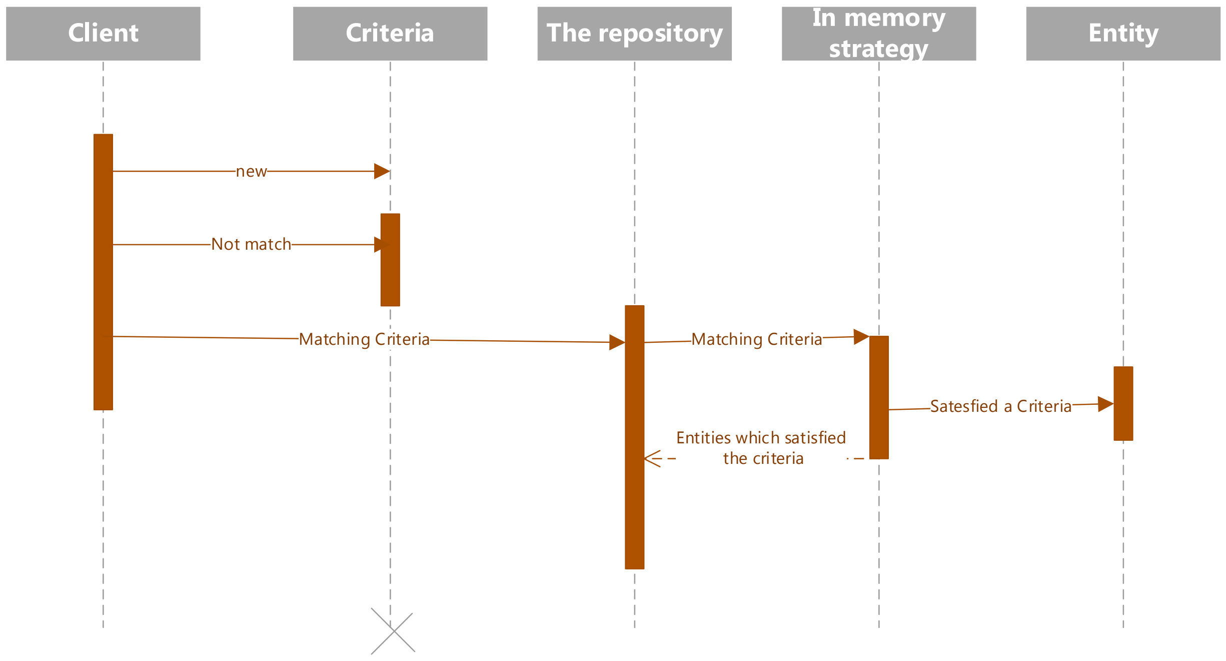

Figure 7: An equal model can be found in [13] and our pattern is implemented in the same manner. Displays the characteristics of the repository.

The Repository allows functions to be submitted in either the Get method or you can do it while selecting data as in the example. The get method is parameterized by `Expression<Func<TEntity, bool>> filter = null,` (Code 4) which are simply added to the generated query. It also supports adding other properties by mapping the properties inputted, reducing them, and finally transforming them to the query format. Furthermore data is retrieved using eager loading, which means when an entity is retrieved, all related entities are also collected, thus allowing faster subsequent access.

```
query = includeProperties.Split(new[] {','},
StringSplitOptions.RemoveEmptyEntries).Aggregate(query, (current, includeProperty) =>
current.Include(includeProperty));
```
Code 5: Shows how the system maps, reduces and transforms properties to the query format

After this has happened the data resides in memory until the Unit of Work determines it is no longer needed. In order to achieve control over what the Unit of Work puts into memory it inherits from the C# *IDisposable* interface. This allows the developer to simply override the dispose method wherever the Unit of Work is used to have complete control over the memory lifecycle.

Submitting data to the system is done through the simple data access interface. For the developer it is a very simple operation consisting of these two lines, as the Unit of Work handles both the transactions and the context of the data store.

```
            _unitOfWork.CatchRepository.Insert(new Catch());
            _unitOfWork.Save();
```
Code 6: Displays how new entities are stored in the system

4.2.2 Complex data access

When performing complex data access not already existing in the system, the developer starts by creating a new model containing the expected output data, which extends the capabilities of the store(s) it needs to contact. In this submission he would extend the capabilities of the SQL execution engine, where all output models are covariant. He would go on to create a builder for the DAL query, which inherits from the base builder and extend an interface which denotes the possible features provided by the system. Finally he would supply the model to the execution engine which would retrieve the data back to the user.

We will show a simple example for generating a heatmap of the positions off all fishing activity by Norwegian ships, weighted by the quantity found.

```
            HeatmapQueryBuilder queryBuilder = new HeatmapQueryBuilder();
            SqlExecutor executor = new SqlExecutor();
            var initializer = new HeatmapData(queryBuilder
                .SetVessels(boatNames)
                .SetSpecies(speciesNames)
                .SetFromDate(fromDate)
                .SetToDate(toDate)
                .Build()
                , allVessels);
            return executor.ExecuteSql(initializer);
        }
```

Code 7: Complete code required to generate a heatmap of all fishing activies in the system

/C SQL scripts

Algorithm C.1: Primary Server schema

```
USE [master]
GO
/****** Object:  Database [SeadrivePrimaryServer]    Script Date: 01.06.2016 20:35:45 ******/
CREATE DATABASE [SeadrivePrimaryServer]
 CONTAINMENT = NONE
 ON  PRIMARY
( NAME = N'SeadrivePrimaryServer', FILENAME = N'C:\Program␣Files\Microsoft␣SQL␣Server\MSSQL12.S
 LOG ON
( NAME = N'SeadrivePrimaryServer_log', FILENAME = N'C:\Program␣Files\Microsoft␣SQL␣Server\MSSQL
GO
ALTER DATABASE [SeadrivePrimaryServer] SET COMPATIBILITY_LEVEL = 120
GO
IF (1 = FULLTEXTSERVICEPROPERTY('IsFullTextInstalled'))
begin
EXEC [SeadrivePrimaryServer].[dbo].[sp_fulltext_database] @action = 'enable'
end
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ANSI_NULL_DEFAULT OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ANSI_NULLS OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ANSI_PADDING OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ANSI_WARNINGS OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ARITHABORT OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET AUTO_CLOSE OFF
```

```
GO
ALTER DATABASE [SeadrivePrimaryServer] SET AUTO_SHRINK OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET AUTO_UPDATE_STATISTICS ON
GO
ALTER DATABASE [SeadrivePrimaryServer] SET CURSOR_CLOSE_ON_COMMIT OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET CURSOR_DEFAULT  GLOBAL
GO
ALTER DATABASE [SeadrivePrimaryServer] SET CONCAT_NULL_YIELDS_NULL OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET NUMERIC_ROUNDABORT OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET QUOTED_IDENTIFIER OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET RECURSIVE_TRIGGERS OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET  DISABLE_BROKER
GO
ALTER DATABASE [SeadrivePrimaryServer] SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET DATE_CORRELATION_OPTIMIZATION OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET TRUSTWORTHY OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET ALLOW_SNAPSHOT_ISOLATION OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET PARAMETERIZATION SIMPLE
GO
ALTER DATABASE [SeadrivePrimaryServer] SET READ_COMMITTED_SNAPSHOT OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET HONOR_BROKER_PRIORITY OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET RECOVERY SIMPLE
GO
ALTER DATABASE [SeadrivePrimaryServer] SET  MULTI_USER
GO
ALTER DATABASE [SeadrivePrimaryServer] SET PAGE_VERIFY CHECKSUM
GO
ALTER DATABASE [SeadrivePrimaryServer] SET DB_CHAINING OFF
GO
ALTER DATABASE [SeadrivePrimaryServer] SET FILESTREAM( NON_TRANSACTED_ACCESS = OFF )
GO
ALTER DATABASE [SeadrivePrimaryServer] SET TARGET_RECOVERY_TIME = 0 SECONDS
GO
ALTER DATABASE [SeadrivePrimaryServer] SET DELAYED_DURABILITY = DISABLED
GO
EXEC sys.sp_db_vardecimal_storage_format N'SeadrivePrimaryServer', N'ON'
GO
USE [SeadrivePrimaryServer]
GO
/****** Object:  Table [dbo].[ChunksReceived]    Script Date: 01.06.2016 20:35:45 ****
SET ANSI_NULLS ON
GO
```

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[ChunksReceived](
        [LocalServerId] [nvarchar](50) NOT NULL,
        [FileChecksum] [varbinary](20) NOT NULL,
        [ChunkNum] [int] NOT NULL,
        [Size] [int] NOT NULL,
        [Id] [int] IDENTITY(1,1) NOT NULL
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[localServerIdSession]    Script Date: 01.06.2016 20:35:45 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[localServerIdSession](
        [LocalServerId] [nvarchar](50) NOT NULL,
        [CurrentFileInTransit] [varbinary](20) NULL
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[VirtualFile]    Script Date: 01.06.2016 20:35:45 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[VirtualFile](
        [Guid] [nvarchar](200) NOT NULL,
        [RelativePath] [nvarchar](200) NOT NULL,
        [FileChecksum] [varbinary](20) NOT NULL,
        [PreviousVersionChecksum] [varbinary](20) NULL
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
USE [master]
GO
ALTER DATABASE [SeadrivePrimaryServer] SET  READ_WRITE
GO
```

Algorithm C.2: localserver schema

```
USE [master]
GO
/****** Object:  Database [seadriveServer]    Script Date: 01.06.2016 20:37:14 ******/
CREATE DATABASE [seadriveServer]
 CONTAINMENT = NONE
 ON  PRIMARY
( NAME = N'seadriveServer', FILENAME = N'C:\Program␣Files\Microsoft␣SQL␣Server\MSSQL12
 LOG ON
( NAME = N'seadriveServer_log', FILENAME = N'C:\Program␣Files\Microsoft␣SQL␣Server\MSS
GO
ALTER DATABASE [seadriveServer] SET COMPATIBILITY_LEVEL = 120
GO
IF (1 = FULLTEXTSERVICEPROPERTY('IsFullTextInstalled'))
begin
EXEC [seadriveServer].[dbo].[sp_fulltext_database] @action = 'enable'
end
GO
ALTER DATABASE [seadriveServer] SET ANSI_NULL_DEFAULT OFF
GO
ALTER DATABASE [seadriveServer] SET ANSI_NULLS OFF
GO
ALTER DATABASE [seadriveServer] SET ANSI_PADDING OFF
GO
ALTER DATABASE [seadriveServer] SET ANSI_WARNINGS OFF
GO
ALTER DATABASE [seadriveServer] SET ARITHABORT OFF
GO
ALTER DATABASE [seadriveServer] SET AUTO_CLOSE OFF
GO
ALTER DATABASE [seadriveServer] SET AUTO_SHRINK OFF
GO
ALTER DATABASE [seadriveServer] SET AUTO_UPDATE_STATISTICS ON
GO
ALTER DATABASE [seadriveServer] SET CURSOR_CLOSE_ON_COMMIT OFF
GO
ALTER DATABASE [seadriveServer] SET CURSOR_DEFAULT  GLOBAL
GO
ALTER DATABASE [seadriveServer] SET CONCAT_NULL_YIELDS_NULL OFF
GO
ALTER DATABASE [seadriveServer] SET NUMERIC_ROUNDABORT OFF
GO
ALTER DATABASE [seadriveServer] SET QUOTED_IDENTIFIER OFF
GO
ALTER DATABASE [seadriveServer] SET RECURSIVE_TRIGGERS OFF
GO
ALTER DATABASE [seadriveServer] SET  DISABLE_BROKER
GO
ALTER DATABASE [seadriveServer] SET AUTO_UPDATE_STATISTICS_ASYNC OFF
GO
ALTER DATABASE [seadriveServer] SET DATE_CORRELATION_OPTIMIZATION OFF
GO
ALTER DATABASE [seadriveServer] SET TRUSTWORTHY OFF
GO
```

```
ALTER DATABASE [seadriveServer] SET ALLOW_SNAPSHOT_ISOLATION OFF
GO
ALTER DATABASE [seadriveServer] SET PARAMETERIZATION SIMPLE
GO
ALTER DATABASE [seadriveServer] SET READ_COMMITTED_SNAPSHOT OFF
GO
ALTER DATABASE [seadriveServer] SET HONOR_BROKER_PRIORITY OFF
GO
ALTER DATABASE [seadriveServer] SET RECOVERY SIMPLE
GO
ALTER DATABASE [seadriveServer] SET  MULTI_USER
GO
ALTER DATABASE [seadriveServer] SET PAGE_VERIFY CHECKSUM
GO
ALTER DATABASE [seadriveServer] SET DB_CHAINING OFF
GO
ALTER DATABASE [seadriveServer] SET FILESTREAM( NON_TRANSACTED_ACCESS = OFF )
GO
ALTER DATABASE [seadriveServer] SET TARGET_RECOVERY_TIME = 0 SECONDS
GO
ALTER DATABASE [seadriveServer] SET DELAYED_DURABILITY = DISABLED
GO
EXEC sys.sp_db_vardecimal_storage_format N'seadriveServer', N'ON'
GO
USE [seadriveServer]
GO
/****** Object:  Table [dbo].[CacheEntries]    Script Date: 01.06.2016 20:37:14 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[CacheEntries](
        [Id] [int] IDENTITY(1,1) NOT NULL,
        [FileChecksum] [varbinary](20) NOT NULL,
        [BlockNum] [int] NOT NULL,
        [ZippedData] [varbinary](1024) NOT NULL,
        [BlockChecksum] [varbinary](20) NOT NULL,
 CONSTRAINT [PK_CacheEntries] PRIMARY KEY CLUSTERED
(
        [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = O
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
/****** Object:  Table [dbo].[FileDirectory]    Script Date: 01.06.2016 20:37:14 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
```

```
GO
CREATE TABLE [dbo].[FileDirectory](
        [Filename] [varchar](250) NOT NULL,
        [IsDirectory] [bit] NOT NULL,
        [Checksum] [varbinary](20) NOT NULL,
        [IsDirty] [bit] NOT NULL
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
USE [master]
GO
ALTER DATABASE [seadriveServer] SET  READ_WRITE
GO
```