BoxBroker: A Policy-Driven Framework for Optimizing Storage Service Federation

Rene Heinsen[1], Cindy Lopez[1] and Eui-Nam Huh[1]

[1] Department of Computer Engineering and Science, Kyung Hee University, 1732, Deogyeong-daero,
Giheung-gu, Yongin-si, Gyeonggi-do, 17104, Republic of Korea

[e-mail : reneheinsen@khu.ac.kr, cindylopez@khu.ac.kr, johnhuh@khu.ac.kr]

*Corresponding author: Eui-Nam Huh

Received April 19, 2017; revised June 23, 2017; accepted September 3, 2017;
published January 31, 2018

Abstract

Storage services integration can be done for achieving high availability, improving data access performance and scalability while preventing vendor lock-in. However, multiple services environment management and interoperability have become a critical issue as a result of service architectures and communication interfaces heterogeneity. Storage federation model provides the integration of multiple heterogeneous and self-sufficient storage systems with a single control point and automated decision making about data distribution. In order to integrate diverse heterogeneous storage services into a single storage pool, we are proposing a storage service federation framework named BoxBroker. Moreover, an automated decision model based on a policy-driven data distribution algorithm and a service evaluation method is proposed enabling BoxBroker to make optimal decisions. Finally, a demonstration of our proposal capabilities is presented and discussed.

Keywords: Storage federation, Data distribution, Policy-driven, Log analysis, Parallel data processing, Service evaluation

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (2015-0-00240,Cloud Storage Brokering Technology for Data-Centric Computing Standardization) and the Energy Efficiency Technology Program of the Korea Institute of Energy Technology Evaluation and Planning(KETEP) granted financial resource from the Ministry of Trade, Industry & Energy, Republic of Korea (No. 20152020106310)

http://doi.org/10.3837/tiis.2018.01.017 **ISSN : 1976-7277**

1. Introduction

IT industry is facing enormous challenges regarding data storage and management, as a consequence of the incredible volume of data that is being generated from all kind of sources and formats. Moreover, every year data generation speed is exponentially increasing. Approximately every two years, world's generated data is doubled [1]. For example, in 2013 the world's generated data was around four zettabytes, by 2015 increased to nearly nine zettabytes, and is expected to reach 40 zettabytes in 2020 [2]. In order to address this phenomenon, storage vendors are developing innovative solutions, for example, cloud storage [3, 4, 5], distribute file systems [6, 7, 8], enterprise storage [9, 10, 11], among others. Nevertheless, enterprises often are forced to implement multiple storage solutions so they can keep with the demand. As a result of heterogeneous architectures and management interfaces on those storage solutions, having multiple implementations increases management cost and complexity, as well as, preventing the interoperability between them.

Multiple storage solutions integration can be useful for improving data availability, security, reliability, durability, and storage scalability to the expenses of increasing management complexity. Storage Federation model purpose is to address management complexity in multiple storage solution environments by providing a common management interface and integrating rule-based decisions. In 2010, Vellante [12] defined *Federated Storage* as "*the collection of autonomous storage resources governed by a common management system that provides rules about how data is stored, managed, and migrated throughout the storage network*". In 2011, *3PAR Peer Motion* software was released by Hewlett Packard Enterprise in order to federate multiple *3PAR StoreServ* systems [13]. The main benefits introduced by *3PAR Peer Motion* where load balancing, non-disruptive data migration and lifecycle management costs reduction. However, this software is limited to the use of *3PAR StoreServ* systems.

In this paper, we aim to face the challenges of file-based heterogeneous storage service federation and the automated rule-based decision making. Our contributions can be summarized as follows:

- We propose a storage service federation framework in order to integrate multiple file-based heterogeneous storage services and provide a single control point with API for fast application development.
- We define a policy model to easily create rules about how data is distributed and replicated for optimal service selection based on predefined policies is proposed.
- A storage service evaluation method based on log analysis is proposed, with the purpose of creating dynamic services profiles.
- A file striping mechanism is implemented that along with our framework parallel data processing are able to improve performance and data throughput.

The rest of this paper is organized as follows; Section 2 analyses and discusses related work. Section 3 introduces BoxBroker's architecture and implementation details. Section 4 defines BoxBroker's decision algorithm. Section 5 describes our services evaluation method. Section 6 shows the implemented file striping mechanism. Section 7 presents conducted experiments and results. Section 8 summarizes our contributions and recognizes future work.

2. Related Work

There are numerous storage federation approaches, most of them are focused on cloud storage federation for preventing vendor lock-in and simplifying multiple accounts management. Yang and Ren [14] provide a framework for federating open cloud storage services and the selection model is purely based on service performance and available space. Vernik et al. [15] present an on-boarding federation mechanism for adding a special layer on cloud storage services allowing them to import data from other services. This approach requires certain adaptability level from the cloud service provider becoming hard to implement. Libardi et al. [16] propose a Multiple-cloud Storage Selection Framework modeled as a knapsack problem, allowing them to select the best subset of providers in order to fulfill users requirements. Its main limitation is the requirement of user input parameter selection for each file upload. Janviriya et al. [17] approach consist in a Multiple Cloud Storage Integration Systems based on RAID (Redundant array of independent disks) 0 stripping technology which improves performance, storage capacity, and security. However, no fault tolerance mechanism is proposed. Zhao et al. [18] propose a middleware that enables any end-user application to automatically and securely stores files in multiples cloud storage accounts. Shah et al. [19] implement a multi-cloud storage integration service for personal cloud storage users. Their approach presents multiple storage accounts in a single interface but lacks integration as a single storage pool. Malensek et al. [20] propose a private and public cloud federation method in order to improve queries throughput in large datasets.

In addition, several approaches referred to Service Selection and Data Distribution were studied. Esposito et al. [21] propose three different methods for selecting the best service set in order to maximize the quality of service and minimize cost. The three methods utilized are based on Fuzzy Logic, Theory of Evidence and Game Theory. Chang et al. [22] use a probability-based algorithm in order to select the best coordination of services that achieved the maximum availability based on a given budget. Other studied approaches regarding data allocation are [23, 24, 25], focusing on load balancing in heterogeneous storage environments. In most of the mentioned related works, only one or two aspects of storage are considered. For example, [21] and [22] mainly focus on availability and cost, while [23, 24, 25] focus on capacity, access frequency, and performance.

Other investigated works were focused on log and text sentiment analysis. Log files have been an important source of information to organizations in order to quickly identify issues [26]. Mho et al. [27] propose a multi-stage analysis architecture using pattern matching, machine learning, and log analysis in order to effectively detect SQL-injection attacks. Grech and Clough [28] propose a method based on Jaccard coefficient, with the purpose of evaluating log clustering analysis stability. Chen et al. [29] propose a cloud security framework based on log analysis using security audit system standards. On the other hand, sentiment analysis has been used to analyze text's polarity in social media in order to define whether the text is positive or negative based on specifics topics [30]. Hamzehei et al. [30] propose a Semantic Scoring Sentiment Analysis Service, which evaluates the scalability and efficiency of multiple sentiment analysis methods using a large volume of microblogs.

Our work proposes a new added value to storage federation consisting in the possibility of federating different kinds of file-based storage services including; cloud storage, distributed file systems, object storage, ftp, among others. In addition, our work relies on a policy-based data distribution algorithm that allows the definition of any number of storage aspects in order to select the best services to efficiently fulfill clients requirements. Moreover, a service analysis method based on logs is provided for periodic service evaluation and maintaining dynamic service profile. In addition, our work relies on a policy-based data distribution algorithm that allows the definition of any number of storage aspects in order to select the best services to efficiently fulfill clients requirements. Moreover, a service analysis method based on logs is provided for periodic service evaluation and maintaining dynamic service profile. **Table 1** shows a comparison between BoxBroker and other approaches.

Table 1. Related Work Comparison Table

		Approaches				
		[14]	**[16]**	**[17]**	**[18]**	**BoxBroker**
Storage aspects used for selection	Availability	x	o	x	x	o
	Performance	o	o	x	x	o
	Cost	x	o	x	x	o
	Available space	o	x	o	o	o
	Allows definition of other aspects?	x	o	x	x	o
Allowed storage types	Cloud storage	o	o	o	o	o
	Non-cloud storage	x	x	x	x	o
Type of policies	(none/static/dynamic)	none	static	none	none	dynamic
Service analysis	Service analysis	x	x	x	x	o
Data protection	Striping	x	x	o	o	o
	Encryption	x	o	x	o	x

3. BoxBroker: Storage Service Federation Framework

This section describes our proposed framework. BoxBroker allows clients to work across multiple file-based storage services like a single one. It converts distinct storage services in a seamless pool and offers a single control point. On the other hand, it integrates an intelligent policy-driven data distribution algorithm and a log-based service analysis, in order to make optimal and automated decisions. Moreover, it enables clients to modify its normal behavior creating their own policies.

The main goal of our framework is to provide a federated storage layer for fast application development on top of it while giving enough customization power. BoxBroker can work with any file-based storage service through *Storage Service Interface* implementation. This allows clients to integrate our framework with their desired services, instead of restricting them to a fixed number of implemented services. Furthermore, it supports a single service multiple accounts integration. BoxBroker accepts services with three type of authentication: none, user-password and OAuth 2.0 [31]. Some examples of supported services are: cloud storage, distribute file systems, network file systems, object storage (parent-child file storage), ftp, among others.

BoxBroker can be configured to store or not the location of the files. If storing file location is set off, means that it acts like a middleware between clients and storage services. Storing file hierarchy is delegated to clients, giving them more flexibility in order to choose the suitable environment for the task. Nevertheless, file location storage is not required.

We also propose a REST API wrapper for our framework. it provides a token-based authentication process, allowing clients to register, authenticate and request authorization token to call the API endpoints. More details in Section 3.7.

BoxBroker's architecture is shown in Fig. 1. The framework is composed of six main components: Policy Engine, OAuth Manager, Metadata Manager, Storage Manager, Storage Service Analyzer, and Storage Service Interface, which are described in sections 3.1 to 3.6 respectively.

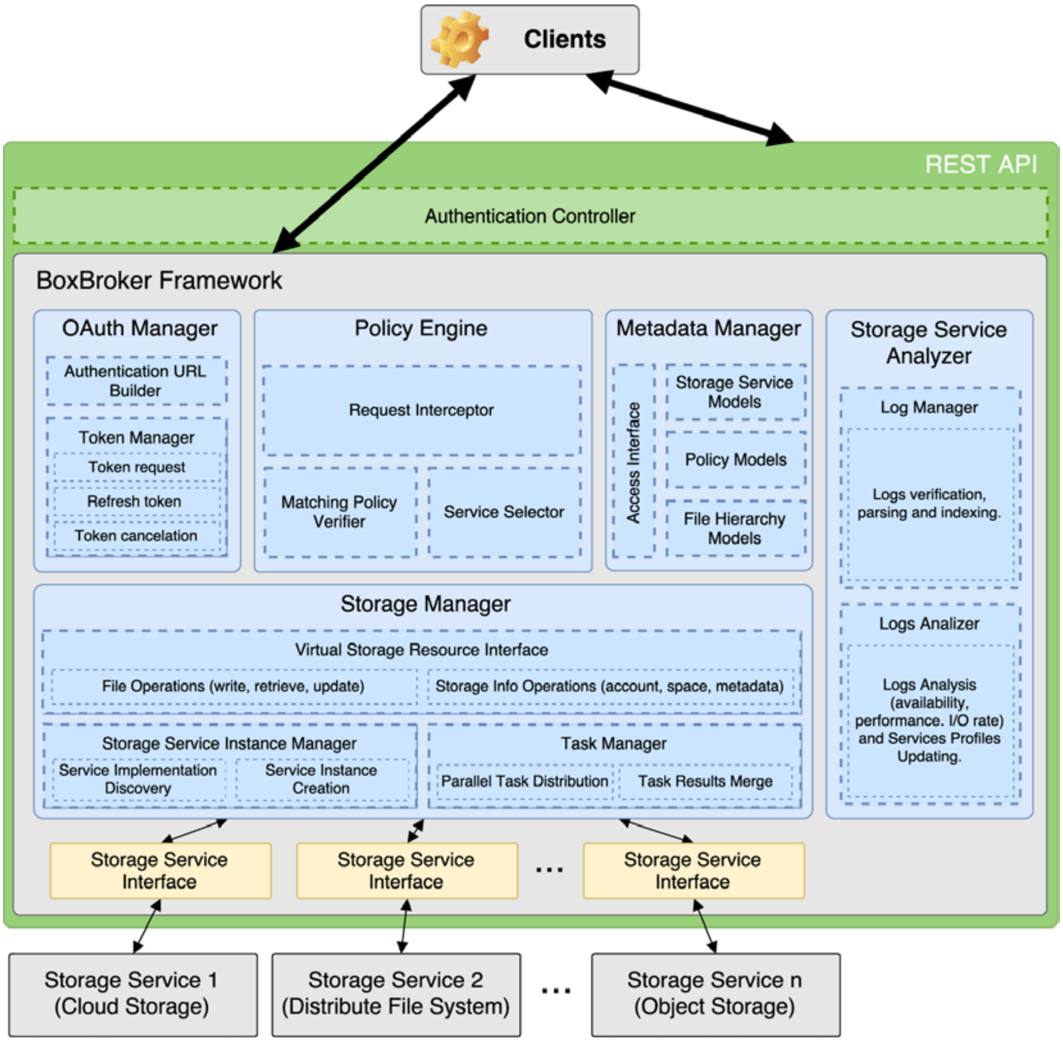

Clients
REST API
Authentication Controller
BoxBroker Framework
OAuth Manager
Authentication URL Builder
Token Manager
Token request
Refresh token
Token cancelation
Policy Engine
Request Interceptor
Matching Policy Verifier
Service Selector
Metadata Manager
Access Interface
Storage Service Models
Policy Models
File Hierarchy Models
Storage Service Analyzer
Log Manager
Logs verification, parsing and indexing.
Logs Analizer
Logs Analysis (availability, performance. I/O rate) and Services Profiles Updating.
Storage Manager
Virtual Storage Resource Interface
File Operations (write, retrieve, update)
Storage Info Operations (account, space, metadata)
Storage Service Instance Manager
Service Implementation Discovery
Service Instance Creation
Task Manager
Parallel Task Distribution
Task Results Merge
Storage Service Interface
Storage Service Interface
...
Storage Service Interface
Storage Service 1 (Cloud Storage)
Storage Service 2 (Distribute File System)
...
Storage Service n (Object Storage)

Fig. 1. BoxBroker Framework's Architecture

3.1 Policy Engine

This component can be considered as BoxBroker's brain. Therefore, it is responsible for decision making, services selection, request verification, and request complementation. In order to make automated decisions a policy-based data distribution algorithm is implemented (refer to Section 4 for more details). Three different policy models are implemented. The first model, which is specified in section 4.2.2, defines the importance order for services profiles metrics. The second one specifies the number of replicas for a given condition. This second model allows clients to define a different number of replicas for distinct data. The third model indicates if the file must be striped and specifies the blocks size. *Policy Engine* is composed of three sub-components:

3.1.1 Request Interceptor

Is responsible for verifying and complementing each request before sending it to the Storage Manager. In order to complement a request, services credentials must be included. Therefore, all request must be verified for this component. Request Interceptor's obligations can be split into two blocks:

- *Obtaining Credentials*, in a case of policy-applicable request this process gets the services credentials from *Metadata Manager* for the selected services by *Service Selector* and *Matching Policies Verifier*. For example, writing a file or retrieving a replicated file, in both a decision must be taken. On the other hand, for a not policy-applicable request, meaning the request must be sent to all services or a specific service, no decision is required. For example, reading a not replicated file or getting the available space.
- *Complementing Request*, integrates services credentials with the original request and sends it to Storage Manager. Furthermore, in the case of OAuth 2.0 credentials, these are sent to *OAuth Manager* for verification before integrating them with the request.

3.1.2 Matching Policy Verifier

Is in charge of verifying if any policy condition matches with the current file. First, active policies are requested to *Metadata Manager*. Each policy condition is stored as string expression, for example, (File.Size >= 12500000 AND File.Size <= 62500000) OR File.TypeMatch("(audio|video)/\S+"), **Table 2** shows policy's condition allowed operators. The second step is to parse those conditions into binary expression trees [32], as shown in **Fig. 2**. Moreover, expression trees are compiled in order to create executable rules [33]. Then, policies, where their rule returns a match, are selected. Because each policy is independent of the others, a parallel processing technique is used for policies verification. As we mentioned before, there are three kinds of policies, importance order, replication, and file striping policies. Matching multiple importance order policies is allowed. However, in the case of matching multiple replication or file striping policies, the last created policy is selected. Finally, matching policies are given to *Service Selector*.

Table 2. Policy Condition Allowed Operators

Type	**Operator**	**Description**
Relational	>, <, >=, <=	112 Greater than, Less than, Greater than or equal to, Less than or equal to [34]. Allowed for `File.Size` only.
Equality	==, !=	Equal to, Not equal to [34]. Allowed for all properties.
Conditional	AND, OR, &&, ||	Conditional AND, Conditional OR [34]
Unary	!	Logical negation [34].
Special	Contains("xxx")	Verify if a specified substring occurs withing the property. Allowed for `File.Name` and `File.Type` [34].
	TypeIn("xxx,yyy")	Verify if the type of the file is equal to any of the specified types. Types are specified in a comma separated string. It must be applied to `File` instead of `File.Type`.

	TypeMatch("xxx")	Verify if the type of the file matches a regular expression. It must be applied to `File` instead of `File.Type`.
	NameMatch("xxx")	Verify if the name of the file matches a regular expression. It must be applied to `File` instead of `File.Name`.

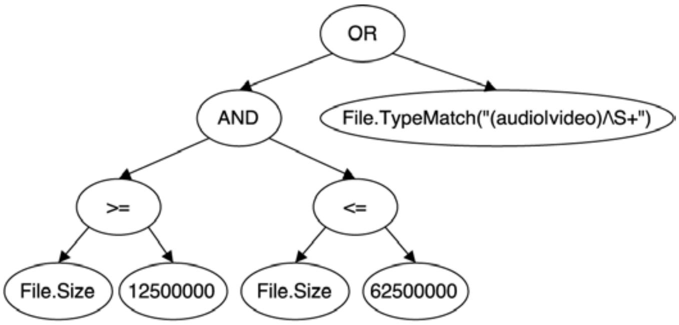


Fig. 2. Parsed Expression Tree

3.1.3 Service Selector

Is responsible for selecting the best group of services according to a set of policies and a replication factor. *Service Selector* implements a policy-based service selection algorithm widely described in Section 4. If a file is set to use striping technique the blocks are distributed and replicated across all services, see Section 6 for more details. In the case of file storing request, an extra request is generated for getting services available space. Services without enough available space are discarded from the selection process. Furthermore, if no policies are provided service selection is purely based on the available space. On the other hand, for replicated file retrieving request and no policies provided service selection is random.

3.2 OAuth Manager

This component is designed for allowing the implementation of services that used OAuth 2.0 authentication protocol, mostly Cloud Storage [31]. Service profiles using OAuth 2.0 protocol must specify *ClientId*, *ClientSecret*, *Scope*, *OAuth Url*, *OAuth Url extra parameters*, *Token Url*, and *Support Refresh Token* parameters in order to enable *OAuth Manager* to follow the authentication flow. *OAuth Manager* has two sub-components:

- *Authentication Url Builder*, is responsible for constructing an OAuth authentication Url based on a given service's profile and the desired redirect Url. Moreover, it verifies if the constructed Url is valid.
- *Token Manager*, is the component responsible for requesting, refreshing and canceling services access tokens.

3.3 Metadata Manager

Defines storage services, policies, and file hierarchy domain models [35]. Moreover, implements an access interface to allow other components retrieve, create and modify models information. The component uses Microsoft Entity Framework code-first approach, which is a domain driven design technology focused in domain classes creation instead of database design [36]. Using Entity Framework enables client applications the possibility of easily extending our domain models by creating new domain classes or overwriting the existing ones.

3.4 Storage Manager

It is a centralized storage resource management component. It exposes an interface to access multiple storage service resources from a single point. Moreover, it implements parallel processing technique to access multiple services concurrently and improve the execution time. *Storage Manager* uses reflection-oriented programming in order to dynamically create and access services implementation classes [37]. It is composed of three sub-components:

- *Virtual Storage Resource Interface*, exposes a single access point converting all the distinct storages services in a seamless storage pool. The methods are divided into two main sub-classes:
  - *Storage Data Operations*, defines all data access operation. For example, writing, retrieving, and updating files.
  - *Storage Information Operations*, consists of accessing services statistics information. For example, available space, service account information, and storage metadata.
- *Storage Service Instance Manager*, is responsible for service implementation classes discovery and instance creation. This component uses reflection-oriented programming approach to find the implemented class for each service according to a unique name defined in storage service domain model and the *Storage Service Interface* inheritance.
- Task Manager, is in charge of parallelly accessing multiple storage services and combined their responses. It uses *.Net Task Parallel Library* in order to create independent tasks that run concurrently. This technology provides an efficient and scalable system resources use. Moreover, it has been enhanced for load balancing and throughput maximization making relatively lightweight tasks. As well as provide a rich set of APIs for waiting, cancellation, exception handling, status reporting and scheduling [38]. Additionally, *Task Manager* asynchronously reports all storage services response stats to *Storage Service Analyzer* for logging and analysis.

3.5 Storage Service Analyzer

This component is designed for maintaining dynamic services profiles based on log analysis. *Storage Service Analyzer* collects services response stats (e.g. service and client identification, request type, response time, errors, file attributes), then parse and store them for further analysis. The analysis extracts relevant information like service availability, performance, and load. Finally, in order to make better decisions, services profiles are constantly updated with the analysis results. It is divided into two sub-components:

- Log Manager, is composed of *Elastic Open Source Stack*. *Elastic Stack* is a set of software useful for collecting, parsing, searching, storing, analyzing and visualizing data from different sources and formats in real time [39]. Used components from *Elastic Stack* are: *Logstash* for logs collection, verification, and parsing; *Elasticsearch* for logs storage and fast searching; *Kibana* for real time logs data visualization and service monitoring.
- Log Analyzer, this is a log analysis service which usee logs stored in *Log Manager* to extract relevant information about the federated storage services. It takes advantage of

Elastic Stack real time processing and fast searching to keep up-to-date storage services profiles. Section 5 presents the proposed metrics extracted from service logs and the employed methods.

3.6 Storage Service Interface

Defines the methods that each storage service must implement in order to enable BoxBroker to access them. Moreover, the interface implementation makes possible to dynamically discover and instantiate services classes using reflection-oriented programming approach. Fig. 3 shows how BoxBroker's components and subcomponents interact between each othe. Furthermore, it describes the general process for a client request.

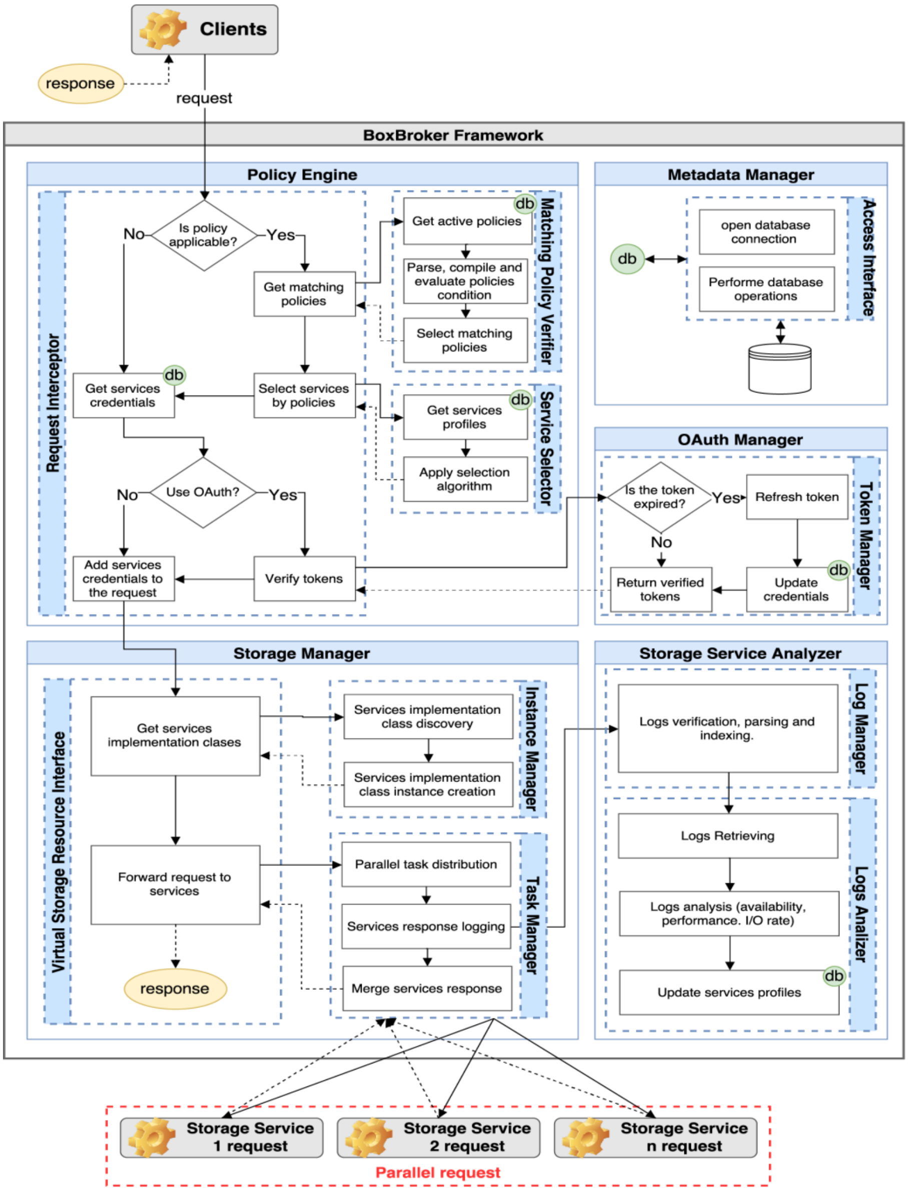


Fig. 3. BoxBroker Framework’s Components Interaction.

3.7 BoxBroker API

The implemented REST API transforms BoxBroker in a standalone service solution. Moreover, it extends BoxBroker's architecture and domain models. An *Authentication Controller* component is integrated in order to provide a token-based authentication process. This extra component allows clients to register, authenticate and get an authorization token to access the API endpoints. On the other hand, domain models were extended in order to allow clients to define their own service credentials, policies, and associated files.

4. Policy-based Storage Service Selection

4.1 Problem Description

We want to distribute and replicate data across multiple storage services, based on dynamic and personalized policies. Each storage service is defined by a group of numerical metrics. A policy specifies the importance of each metric to store a specific data. Finally, a number of desired replicas is provided.

A storage service has a finite number of metrics which define it. There is a vast range of heterogeneous metrics useful for analyzing the best match to store the data. Nevertheless, their values have not only different ranges but also opposite meaning, causing difficulties in their comparison. For example, availability and cost, obviously they have a different range of values and also opposite meaning. In the case of availability, you will choose the highest possible. On the contrary, you would like to minimize the cost. Such heterogeneity requires normalization in order to have all the characteristics lying between the same range [0, 1] [40]. Moreover, we assume that all the metrics are defined as; the smaller the value the better the service. To illustrate, the availability can be defined as the probability of failure. As the probability of failure decreases the availability increases.

On the other hand, we have a set of policies which help us to choose the correct group of storage services for our data. Each policy is defined by two properties. First, a condition to match a policy with the data. Second, a selection factor order, which defines a weight for each service's metric based on importance order, where the first order has the highest weight and so on.

4.2 Mathematical Definition

Suppose we have a set of storage services $S = \{s_1, s_2, \dots, s_n\}$, a set of policies $P = \{p_1, p_2, \dots, p_n\}$ and a replication factor r . We want to select a subset R of size r from S, where the distances between P and R are minimized.

4.2.1 Storage Service Model

Each service s_i is represented by a vector $K_{s_i} = \{\phi_1^i, \phi_2^i, \dots, \phi_k^i\}$, where ϕ_j^i denote the j-th metric's value for service s_i.

$$\phi_j^i \in [0, 1]\#(1)$$

4.2.2 Policy Model

Similarly, a policy p_i is defined by a vector $K_{P_i} = \{\varphi_1^i, \varphi_2^i, \dots, \varphi_k^i\}$. The j-th metric's importance order based on policy p_i is denoted by φ_j^i.

$$\varphi_j^i \in \{1, 2, \dots, k\}\#(2)$$

4.2.3 Metrics Weight

The metrics weight is exponentially distributed according to its importance order. The weight of metric j based on policy p_i is defined by the equation 3, where λ is named weight factor, see section 7.2.2 for more details.

$$f_W(p_i, j) = e^{-\lambda\varphi_j^i}\#(3)$$

Total weight of a metric j according to all policies P is expressed as:

$$f_W(P, j) = \sum_{i=1}^{m} f_W(p_i, j)\ \#(4)$$

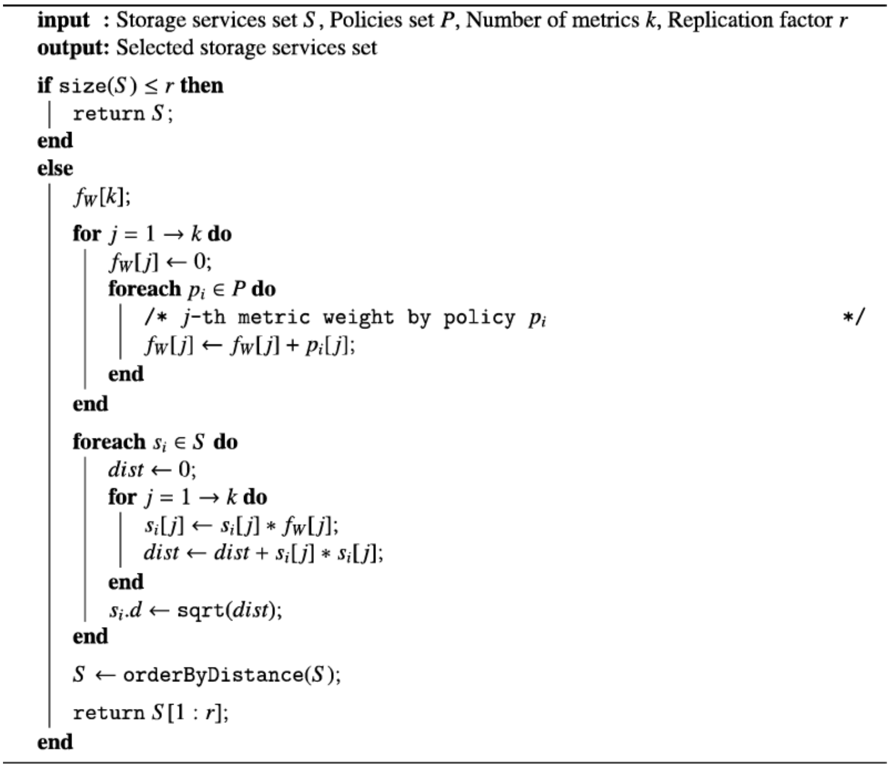
4.3 Proposed solution

To solve the services selection problem, we modeled it as a k-Nearest Neighbors problem. Usually, k-NN is used to categorize a query point depending on its neighbors [41]. For our purpose, the query point Q is the center of the space, where all the storage services dimensional metrics are maximized. k (number of neighbors) is determined by r (replication factor). For distance measurement, we used a weighted Euclidean distance [40], where each feature is scaled by the corresponding weight based on the policies.

$$D(s_i) = \sqrt{\sum_{j=1}^{k}\left(\phi_j^i \times f_W(P, j)\right)^2}\ \#(5)$$

Finally, the algorithm order by distance and takes r nearest services. **Algorithm 1** shows policy-based storage service selection algorithm implementation.

Algorithm 1. Policy-Based Storage Service Selection



```
input  : Storage services set $S$, Policies set $P$, Number of metrics $k$, Replication factor $r$
output: Selected storage services set

if size($S$) $\leq r$ then
    return $S$;
end
else
    $f_W[k]$;
    for $j = 1 \rightarrow k$ do
        $f_W[j] \leftarrow 0$;
        foreach $p_i \in P$ do
            /* $j$-th metric weight by policy $p_i$                    */
            $f_W[j] \leftarrow f_W[j] + p_i[j]$;
        end
    end
    foreach $s_i \in S$ do
        $dist \leftarrow 0$;
        for $j = 1 \rightarrow k$ do
            $s_i[j] \leftarrow s_i[j] * f_W[j]$;
            $dist \leftarrow dist + s_i[j] * s_i[j]$;
        end
        $s_i.d \leftarrow$ sqrt($dist$);
    end
    $S \leftarrow$ orderByDistance($S$);
    return $S[1 : r]$;
end
```

5. Log Analysis

Logs have been used as an important source of information for organizations in order to quickly identify and resolve issues [26]. Other approaches about log analysis benefits were discussed in section 2. We are proposing the integration of log analysis in *Storage Service Federation Environment* with the purpose of evaluating services availability, performance, and workload. The proposed analysis transforms our static services profiles into dynamic profiles that change in time, helping our selection algorithm to take up-to-date decisions. This analysis is done periodically and the number of log records used on it depends on a pre-configured time interval, e.g. daily, weekly, monthly, etc. **Table 3** describes the relevant log attributes for our analysis.

Table 3. Relevant Log Attributes

Attribute	Description
`LogTime`	Date and time when the log record was created.
`LogLevel`	Logs are tagged with ”ERROR” if an error occurred or ”INFO” if not.
`ServiceId`	Services unique identification
`RequestType`	The requested operation type.
`RequestTime`	Date and time when the request was sent to the service.
`ResponseTime`	Date and time when the response of the service was received.
`FileSize`	The size(bytes) of the file involved in the requested.
`ServiceProtocol`	The Protocol utilized to establish a communication with the service (e.g. HTTP, FTP, WebDAV, ...).
`ErrorCode`	An error code if it exists.
`ErrorMessage`	An error message if it exists.

5.1 Service Availability Analysis

In order to calculate the service’s availability score, we used the equation 6, which was described by Marcus and Hal in [42]. $MTBF$ represents the mean time between failures and $MTTR$ represents the mean time to repair or recover from a failure.

$$A = \frac{MTBF}{MTBF + MTTR} \#(6)$$

Our environment can generate errors that we do not want to take into consideration for availability score computation. Per instance, some services have a limit on the file size that can be written through their APIs/interfaces, or the credentials are not correct/expired. These errors and others should not impact service availability because those errors are caused by our framework or the clients. In order to minimize this scenario, service limitations must be specified in its profile. We are proposing two discrimination methods for selecting the errors that affect service availability. The First method is based on the combination of *ServiceProtocol* and *ErrorCode*, a dictionary of protocols and error codes can be configured in BoxBroker to specify the errors that must be considered in availability calculation.

Table 4 shows some proposed error codes. The second method is based on text sentiment analysis which is applied to *ErrorMessage* in the case of *ServiceProtocol* and *ErrorCode* absence. For text sentiment analysis we are using Naive Bayes classifier to measure error message polarity [43] where negative *ErrorMessage* are considered in availability scores. **Table 5** shows some useful training phrases. Errors in **Table 4** were also used for training tagged as negative. Basically, we want to select errors associated to off-line service and avoid errors caused by badly formed requests.

Table 4. Error Codes Related with Availability

Protocol	**Code**	**Description**
HTTP / WebDAV	423	Resource Locked
	429	Too Many Requests
	500	Internal server error
	501	Not Implemented
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Timeout
FTP	421	Service Not Available
	425	Can't Open Data Connection
	426	Connection Closed; Transfer Aborted.
	434	Requested Host Unavailable.
	451	Requested Action Aborted. Local Error in Processing.
	534	Could Not Connect to Server.
	10060	Cannot Connect to Remote Server.
	10068	Too Many Users, Server is Full.

Table 5. Sentiment Training Set Sample

Phrase	**Polarity**
Network Authentication Required	Positive
Internal issue	Negative
Bad Request	Positive
Insufficient Storage	Positive
Unauthorized	Positive
Failed to write file to disk	Negative
Method Not Allowed	Positive
Unsupported Media Type	Positive
Unable to save or download files	Negative
Server error	Negative
Bandwidth Limit Exceeded	Positive
User is over storage quota	Positive
Invalid username or password	Positive
Requested action not taken. File name not allowed	Positive

5.2 Service Performance Analysis

Read and write operations are used to measure service performance, which is defined as the average number of megabytes processed by a service in a second (MB/s). Equation 7 is applied for computing read and write performance based on a set of logs.

$$Perf = \frac{\sum_{i=1}^{n} \frac{FileSize_i}{ResponseTime_i - RequestTime_i}}{n} \#(7)$$

5.3 Service Workload Analysis

Service workload is also measured utilizing read and write operations, defined as the percentage of bytes processed for a service based on total bytes processed for the system in a period of time. Equation 8 is applied for computing read and write workload based on a set of logs.

$$W_{s_i} = \frac{\sum_{File\, \in\, s_i} FileSize}{\sum_{File\, \in\, S} FileSize} \#(8)$$

6. File Blocks Allocation Method

BoxBroker offers the option of using a data striping technique in order to improve performance and throughput. Moreover, segmenting a file in blocks and distributing them across multiple services can be used as a security approach, as the file is spread across multiple locations it can not be read without all blocks and the correct order. BoxBroker implements a RAID 0 like striping algorithm, where consecutive blocks are stored in different services. Since RAID 0 does not provide fault tolerance mechanism, we extended it adding a replication factor r, which indicates how many copies of each block we would like to store. This mechanism allows BoxBroker to recover a file even when $(r-1)$ services have failed.

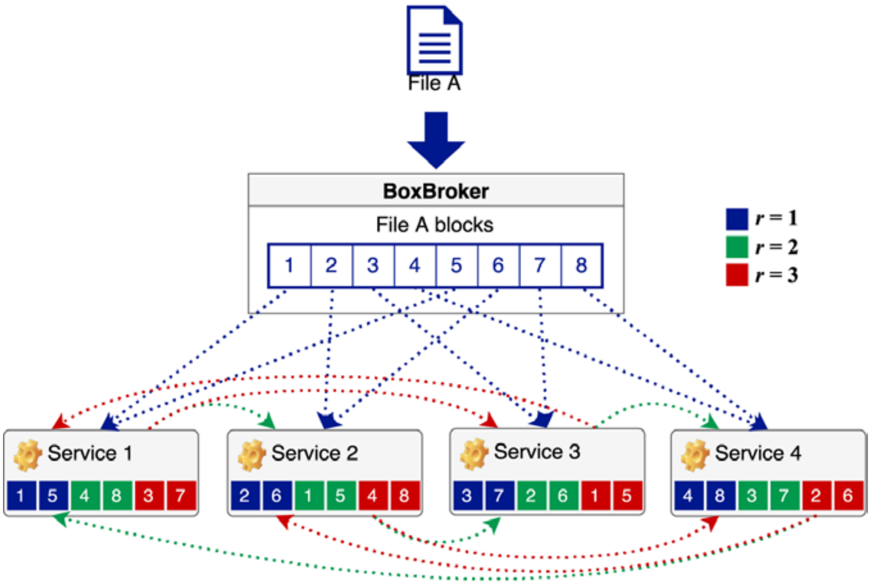


Fig. 4. File Blocks Allocation Method

Fig. 4 shows a graphical representation of our proposed file blocks allocation method. File A is divided in eight blocks, blue blocks represent the original blocks, green ones represent the first replica, and the red ones the second replica. The set of blocks B_{s_i} allocated on a service s_i can be expressed with the equation 9, where S represents the set of disposable services, B file blocks set, and b block number. Furthermore, B_{s_i} is replicated in$(r-1)$ different services. B_{s_i} is replicated in s_{i+j} if $i+j \leq |S|$ or $s_{i+j-|S|}$ if $i+j > |S|$ for all $j \in \{1,2,\dots,r-1\}$ and $2 \leq r \leq |S|$.

$$B_{s_i} = \left\{b \in B: i = b - \left\lfloor \frac{b}{|S|} \right\rfloor * |S| \right\} \#(9)$$

7. Experiments & Results

7.1 Log Analysis

In order to evaluate the functionality of the *Storage Service Analyzer*, we developed a client application that generates requests through our framework to four storage services providers Box, Dropbox, GoogleDrive, and OneDrive denoted as S_1, S_2, S_3 and S_4 respectively. Box and GoogleDrive have a object storage architecture while Dropbox and OneDrive use a filesystem architecture. The generated requests were made of multiples reading and writing transactions with randomly selected files of various sizes. Collected logs were used to calculate services availability, performance, and workload. Moreover, an overall analysis and comparison between services are provided. Furthermore, the results shown in this section are used to evaluate our selection model in section 7.2. The experiments were conducted for a period of 20 days and the results were divided into four sub-periods of five days each, $T_1(1-5)$, $T_2(6-10)$, $T_3(11-15)$ and $T_4(16-20)$.

7.1.1 Availability Analysis

Availability analysis is done according to equation 6. The results scores are shown in **Fig. 5**. We can see that all the services have at least one period with a score equal or greater than 99.9%. Service S_2 has availability average over 99.9%, it means approximately eight hours or less of downtime every year. S_3 has an average of 98.8%, meaning at least 4.38 days of downtime per year. S_4 and S_1 have an average around 99.7% for 24 hours or less of downtime per year. However, S_4 achieved more than 99.99% in periods T_1 andT_4 transformed as 52.5 minutes of downtime each year. According to the results, the difference on availability score seems very small, nevertheless, the downtime is quite significant. It should be noted that these results are based on randomly generated errors.

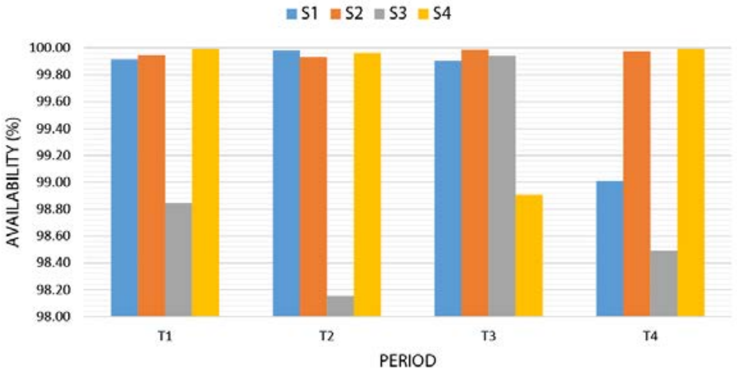


Fig. 5. Availability Analysis Results

7.1.2 Performance Analysis

Fig. 6 and **Fig. 7** show services read and write performance results respectively. In the results can be appreciated that the performance keeps the trend in all periods. Moreover, we can see the fact that if a service has the best write performance doest not necessarily means that it would have the best read performance, or vice-versa. Despite that S_3 and $S_4$are battling for the best read performance, S_4 clearly has the best write performance.

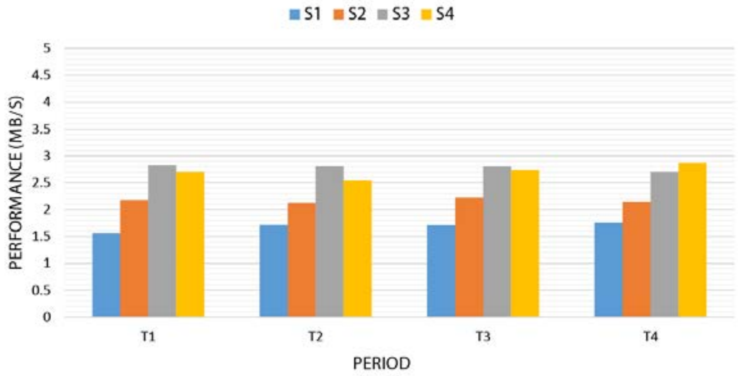


Fig. 6. Read Performance Analysis Results

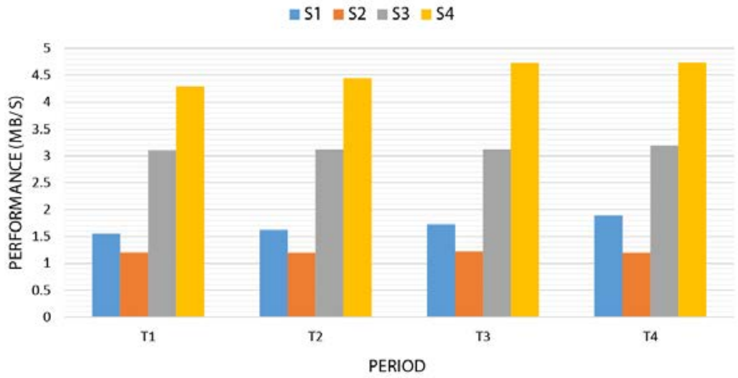


Fig. 7. Write Performance Analysis Results

7.1.3 Workload Analysis

As our transaction simulator sent requests to all services the workload is almost balanced. Read and write workload results are shown in **Fig. 8** and **Fig. 9** respectively. For both cases, the workload percentage oscillate between 24.5 and 25.5.

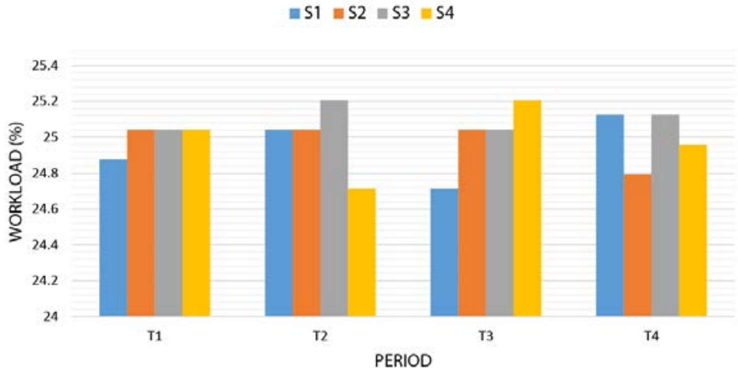


Fig. 8. Read Workload Analysis Results

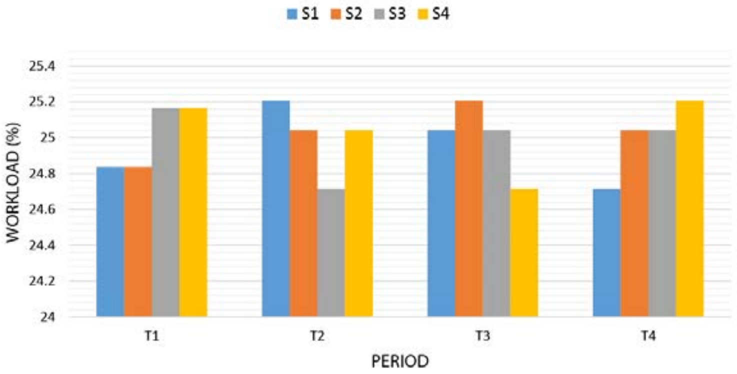


Fig. 9. Write Workload Analysis Results

7.2 Policy-based Storage Service Selection

In this section, we are going to demonstrate how our selection algorithm works using T_4 availability and performance results from section 7.1, together with other metrics such as cost and available space. However, as we defined in section 4.1, all metrics must be represented as; the lower the value the better the service. Therefore, we redefined the metrics as:

- *Availability*, is used as the percentage of time that the service was unavailable. It is calculated as $100 - Availability$. The lower the time the better the service's availability.
- *Write* and *read performance*, are represented as the average time in seconds for processing 1MB of data. They are calculated as $1 / (R|W\ Performance)$. The lower the time the better the service's performance.
- *Cost*, the price of storing 1GB data. The lower the price the cheaper the service.
- *Used space*, the percentage of space that is currently used.

Table 6 lists four services and their metrics values. Consider measuring the distances of each service using this metrics values. The contribution of cost and space in the distance calculation is minimum, we can say that the distance is practically determined by performance and availability. **Fig. 10** illustrates graphically the disparity between metrics. Seeing that each metric has its own range $[a, b]$, some form of normalization is required in order to balance out the contributions in the distance calculation [40].

Table 6. Storage Services Profiles

	Services			
	S$_1$	**S$_2$**	**S$_3$**	**S$_4$**
Availability (%)	0.992	0.025	1.511	0.007
Read Perf. (s/MB)	0.569	0.466	0.370	0.348
Write Perf. (s/MB)	0.530	0.833	0.313	0.211
Cost (USD/GB)	0.011	0.010	0.009	0.007
Used Space (%)	0.022	0.037	0.054	0.090

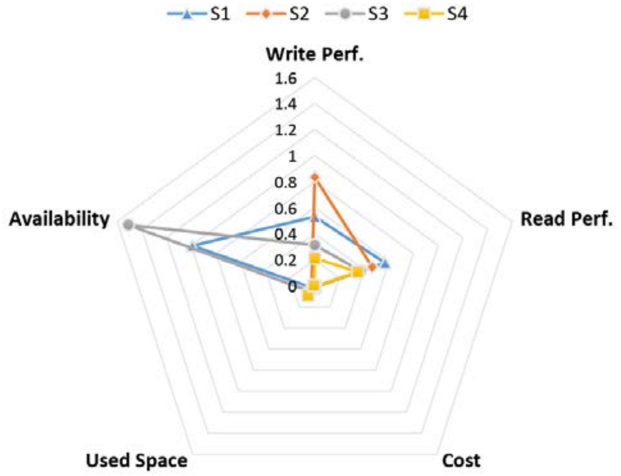


Fig. 10. Not Normalized Service Profiles

7.2.1 Metrics Normalization

In order to balance metrics contribution in distance calculation, all metrics are scaled to a common range $[0, 1]$ [40]. We assumed that for all metric the lower bound $a = 0$ and the upper bound $b = max(MetricValues)$ respectively. The normalized metric's value z is determined by the equation:

$$z = \frac{(x-a) \times (1-0)}{b-a} = \frac{x}{b} \#(10)$$

Table 7 and Fig. 11 show the services profiles after applying the normalization. Comparing Fig. 10 and Fig. 11, we can say that each metric has equal contribution in order to measure the service's distance.

Table 7. Normalized Storage Services Profiles

	Services			
	S_1	S_2	S_3	S_4
Availability (%)	0.656	0.017	1.000	0.005
Read Perf. (s/MB)	1.000	0.818	0.650	0.611
Write Perf. (s/MB)	0.636	1.000	0.376	0.253
Cost (USD/GB)	1.000	0.876	0.864	0.604
Used Space (%)	0.244	0.411	0.600	1.000

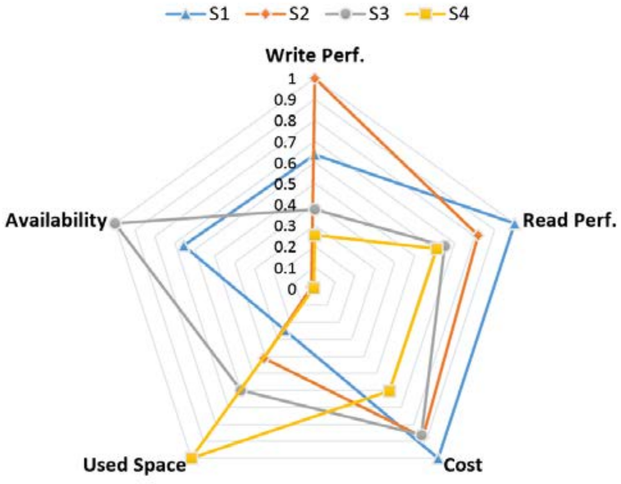


Fig. 11. Normalized Service Profiles

7.2.2 Weight Factor Selection

After defining our services profiles, the next step is to select how metrics weights are going to be distributed. In equation 3 we defined an exponential formula for distributing the weights. This subsection shows the effect of different weight factor values. ↑ and ↓ denote increment and decrement respectively.

Table 8. Weight Factor Effect

	Weight factor	
	Increase	Decrease
Order’s weight	↓	↑
Weights distribution	↓	↑
Weights difference	↑	↓

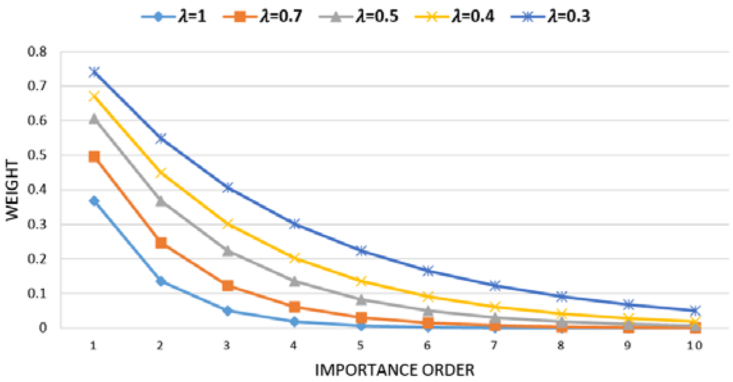


Fig. 12. Weight Distribution for Distinct Wight Factors

The effect of weight factor is described in **Table 8**. As λ increase, the weight of each order decrease. In the same way, the weights are less distributed, this means that weights are more concentrated in the first orders. Nevertheless, weights difference increase as λ increase. **Fig. 12** shows the weight distribution for $\lambda \in \{1, 0.7, 0.5, 0.4, 0.3\}$. In **Fig. 12** we also can graphically see what **Table 8** expressed. Let's take $\lambda = 1$ and $\lambda = 0.5$ as an example:

- The first order weights are 0.37 and 0.61 for $\lambda = 1$ and $\lambda = 0.5$ respectively, so as λ decrease, the orders weights increase.
- For $\lambda = 1$ only the first 2 orders have a weight bigger than 10%, however, for $\lambda = 0.5$ the first 4 order have a weight bigger than 10%, so as λ decrease, the weights are more distributed.
- The weight for order j is a percentage of the weight for order $j - 1$. These percentages are approximately 37% and 61% for $\lambda = 1$ and $\lambda = 0.5$ respectively. In the same way, we can say that the difference between two orders weights is 63% for $\lambda = 1$ and 39% for $\lambda = 0.5$. So as λ decrease, the weights difference also decrease.

In summary, weight factor selection depends on the number of metrics we want to take into consideration. Then, the difference you want between weights. Finally, the weights magnitude you want. Based on the description above, for our experiments we selected a weight factor:

$$\lambda = 0.4\#(11)$$

7.2.3 Policies Definition

At this point, we have defined our services profiles, which are shown in **Table 7**, and the weight factor we are going to use (equation 11). The next step is to define some policies to apply in our experiments. Therefore, four policies are defined to match next conditions:

- P_1 → data that is usually modified and read, where write/read performance and availability are the priority. For example, work in progress documents (text, code, presentation, ...).
- P_2 → one-time or rarely write but usually read data. As a result, the most important metrics are read performance and availability. For example, media files (music, videos, pictures) and some PDF documents could fix in this category.
- P_3 → one-time or rarely write and read data. For example, data backups. Therefore, more importance is given to available space and cost.
- P_4 → less important data. Data that maybe never will be used again but you have to keep it. For this data category, the cost is the most important metric.

Table 9 defines our set of policies and its metric's importance order. Metrics weights based on each policy are shown in **Fig. 13**.

Table 9. Policies Metrics Order Definition

	Policies			
	P_1	**P_2**	**P_3**	**P_4**
Availability	1	1	3	5
Read Performance	3	2	4	4
Write Performance	2	4	5	3
Cost	4	3	2	1
Used Space	5	5	1	2

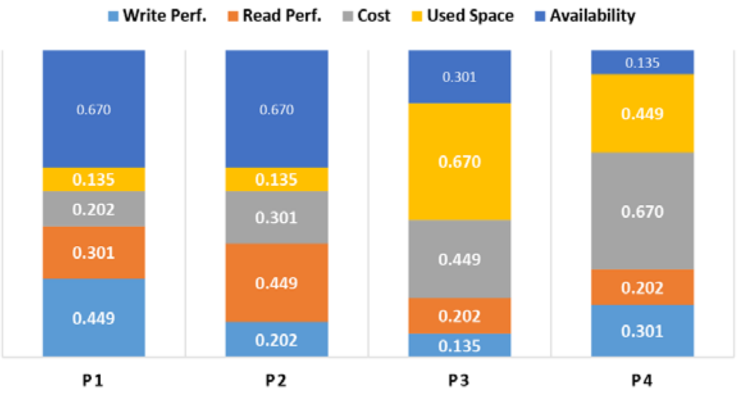


Fig. 13. Policies Metrics Weights

7.2.4 Storage Service Selection

Finally, we have all the pieces to apply our algorithm and select the best storage services. With the services profiles in Table 8 and policies weights in Fig. 15 we can compute each service's distance using equation 5. Then, services are ordered by distance and select the desired number of replicas. **Fig. 14** shows services distance for each of the policies defined in **Table 9**. Based on the distance, services selection order are as follows: for P_1 and P_2 services selection order is $[S_4, S_2, S_1, S_3]$, for P_3 is $[S_2, S_1, S_3, S_4]$, and for P_4 $[S_4, S_3, S_2, S_1]$.

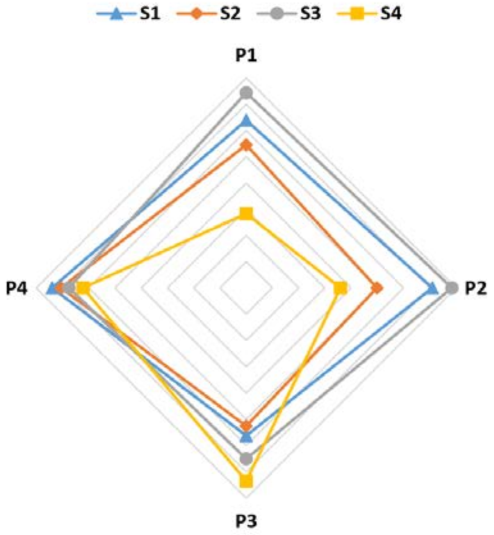


Fig. 14. Services Distance Based on Policies

Now suppose we have some data that match more than one policy. For example, policies P_3 and P_4 are very similar. Let us say that our data match P_3 and P_4. Accordingly equation 4, the total metric's weight is defined as the sum of each policy metric's weight. **Table 10** shows the new metrics weights for this special case. Based on the new weights, services selection order result is $[S_2, S_1, S_3, S_4]$. Services distance are shown in **Fig. 15**.

Table 10. Multi-Policy Metrics Weights

	Policies
	$\mathbf{P_{3\,+}P_4}$
Availability	0.437
Read Performance	0.404
Write Performance	0.437
Cost	1.120
Used Space	1.120

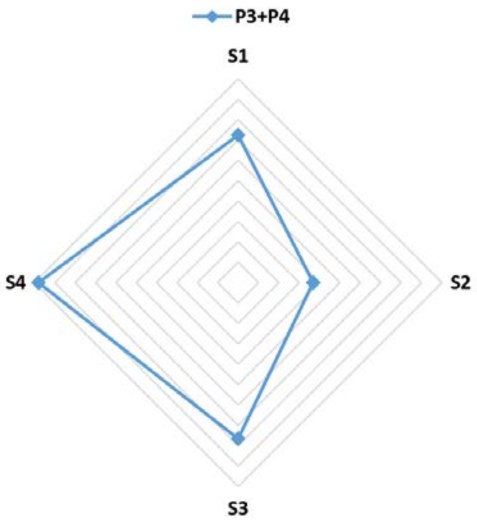


Fig. 15. Services Distance Based on P_3 and P_4

7.2.5 Services Selection Time

The time complexity of our algorithm is defined by equation 12. k represent the number of metrics, m number of policies and n number of services. Therefore, km represent the time to compute total metrics weight, kn time to calculate the Euclidean distance for each service and $n\log n$ time to order services by the distance.

$$O(km + kn + n\log n)\#(12)$$

Fig. 16, **Fig. 17** and **Fig. 18** show the selection time for different cases. Moreover, after the number of services n, the results show that k is the parameter that impacts the most the selection time, because it also affects two parts of the time equation (equation 12). Therefore, selecting the correct amount of metrics to define our services profiles becomes an important task. However, **Fig. 18** shows a more realistic scenery, where we have a fixed number of services defined by a fixed number of metrics and only the amount of policies changes accordingly to the data. Fixing k and n two parts of the time equation (equation 12) become constants and we can achieve a liner selection time, shown in **Fig. 18**.

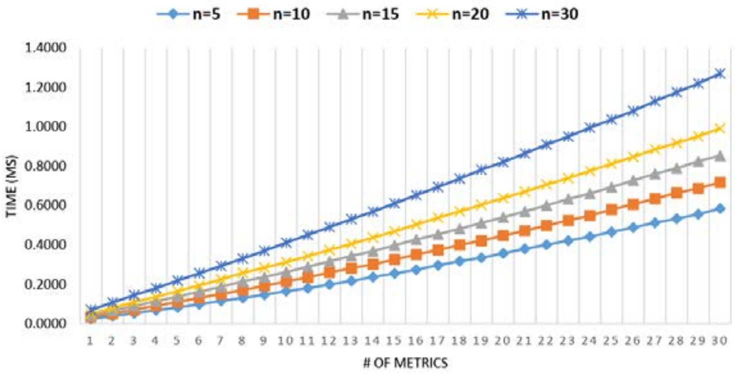


Fig. 16. Selection time with a fixed number of policies (10) and services ∈ $\{5, 10, 15, 20, 30\}$, varying the number of metrics *k*.

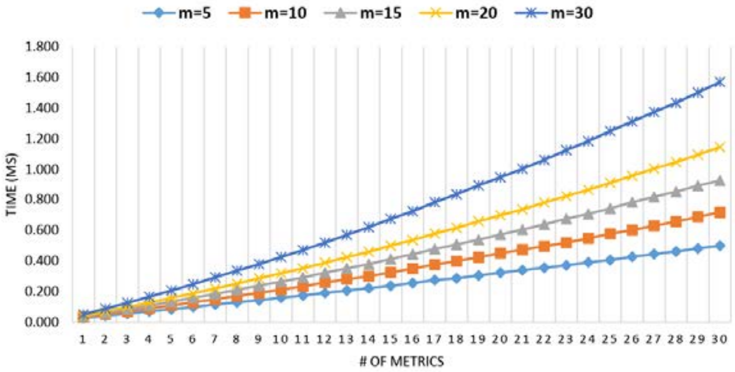


Fig. 17. Selection time with a fixed number of services (10) and policies ∈ $\{5, 10, 15, 20, 30\}$, varying the number of metrics *k*.

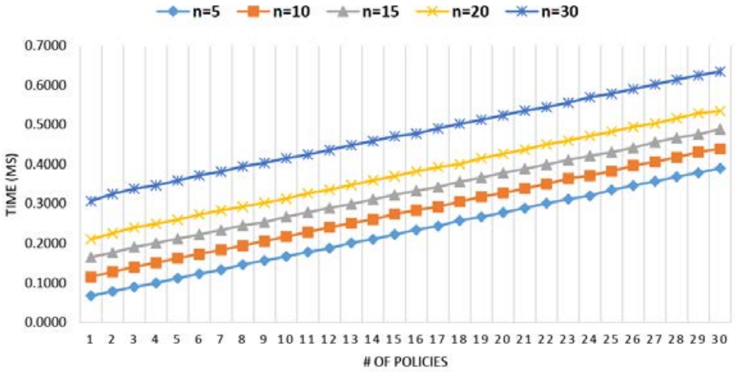


Fig. 18. Selection time with a fixed number of metrics (10) and services ∈ $\{5, 10, 15, 20, 30\}$, varying the number of policies *m*.

In comparison to [14, 21, 22, 23, 24, 25] approaches, where the selection is based on fixed metrics, our proposal can be used with any number of metrics defined by the clients. In addition, unlike these approaches our algorithm allows clients to change the importance of each metric depending on the data. [16] proposes a rule-based service selection algorithm, but clients must categorize each file in order to match the policies, while our approach automatically matches the files based on their attributes. Furthermore, we create a straightforward policy model based on metrics order, in contrast to [16], where the weight of each metric must be specified.

Table 11 and **Fig. 19** compare our selection algorithm with the solutions presented by Yang and Ren [14] (Virtual Framework for Cloud Storage Service - VCSS) and Chang et al. [22] (Probability-based Service Selection - PB). Since these algorithms do not support policies, we compare them with our algorithm using only one policy. Moreover, they only use two metrics for the selection process. Yang and Ren [14] selection is based on performance and available space. Chang et al. [22] selection is based on availability and cost. For the first comparison test, we set the number of metrics to two. **Table 11** shows that (PB) has the worst performance, for that reason (PB) was removed in **Fig. 19**, that allow us to see better the comparison between (VCSS) and our approach. **Fig. 19** shows that BoxBroker selection algorithm performs better than (VCSS) with a basic setting which is a fair comparison. However, with a more complex configuration of 10 metrics and 1000 policies, our algorithm can make a decision over 500 services in less than a millisecond.

Table 11. Algorithms Execution Time (ms)

Algorithms	**n=10**	**n=50**	**n=100**	**n=500**
BoxBroker(m=1,k=2)	0.0094	0.0165	0.0321	0.1563
BoxBroker(m=1000,k=10)	0.5176	0.5182	0.5607	0.9845
VCSS [14]	0.0092	0.0286	0.0871	0.2708
PB [22]	0.0680	5.3368	16.2033	750.4673

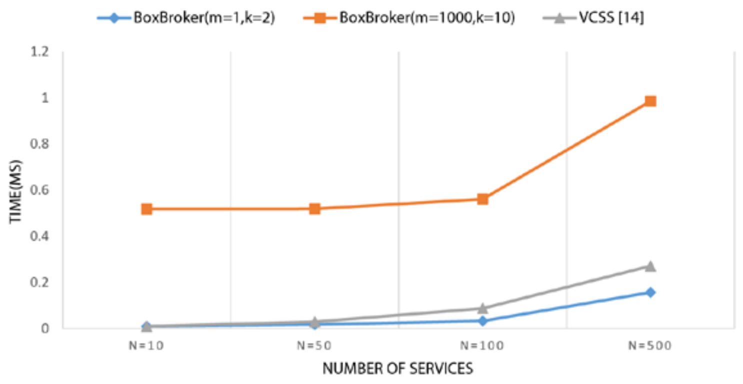


Fig. 19. Comparison of Algorithms Execution Time

7.3 File Striping Performance

In order to evaluate the performance of our framework parallel processing with striped files vs no-striped files, we used multiple file size striped in different block size. Moreover, we evaluated read, write, and delete performance for each file and block size. **Fig. 20** and **Fig. 21** present data write and read throughput in MB/s respectively. **Fig. 22** shows data deletion performance represented as the required time in seconds to delete the entire file. The results from the conducted experiments demonstrate that the combination of file striping and our framework parallel processing can improve performance and data throughput. Note that this results can vary depending on BoxBroker server processing capacity and network speed.

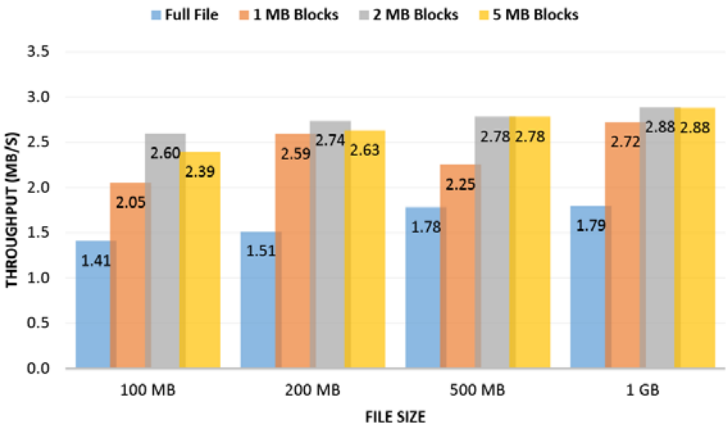


Fig. 20. File Striping Write Performance

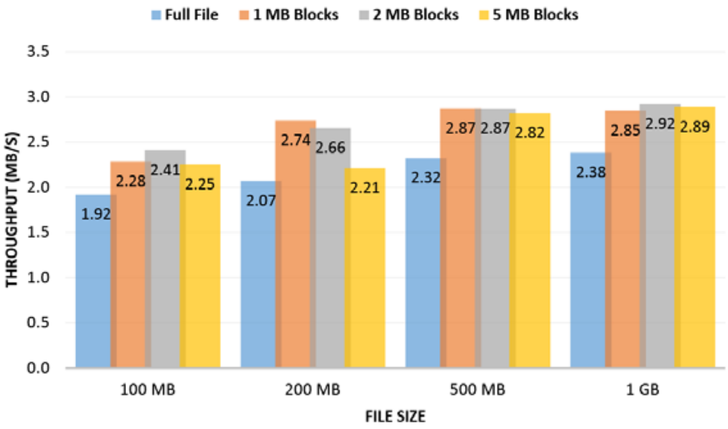


Fig. 21. File Striping Read Performance

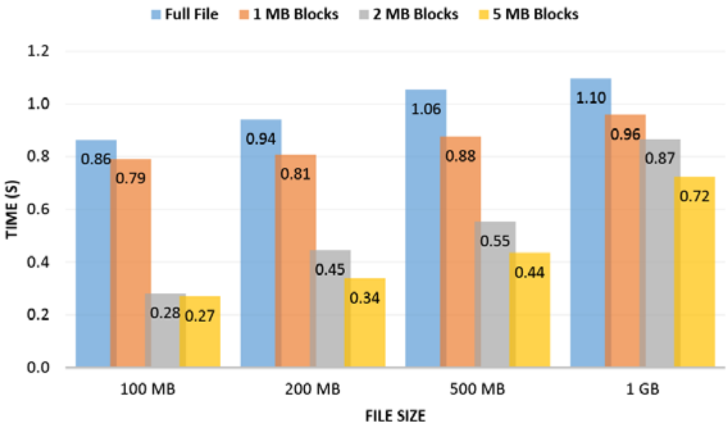


Fig. 22. File Striping Delete Performance

All the experiments in section 7 were done using a 64-bit Windows 10 Pro PC with an Intel Core i7-6700HQ CPU @ 2.6GHz, and 16 GB of RAM.

8. Conclusion & Future Work

We have presented BoxBroker, a policy-driven storage service federation framework that allows the integration of multiple storage services into a seamless storage pool. BoxBroker also implements a policy-based service selection algorithm in order to automate the data distribution and retrieval. A straightforward policy model was proposed to empower clients with the possibility of defining their own data dispersion rules. A service evaluation method for allowing dynamic decisions was implemented. Furthermore, a practical demonstration of our approach was shown and a comparison with other approaches.

Future works are led to solving the limitation of storage metric definition in our algorithm, allowing the definition of positive and negative metrics; some metrics must be minimized while others must be maximized. At the same time, we are looking forward to the implementation of a data migration component that takes advantage of our service analysis, achieving services load balancing, and I/O intensive data reallocation in high-performance services.

References

[1] L. C. Miller, S. Fadden, Software Defined Storage for Dummies, ibm platfo Edition, John Wiley & Sons, Inc., New Jersey, 2014.

[2] V. Turner, J. F. Gantz, D. Reinsel, S. Minton, The Digital Universe of Op-portunities: Rich Data and the Increasing Value of the Internet of Things, IDC Analyze the Future.

[3] Google, Google Cloud Storage (2016). Article (CrossRef Link),

[4] Amazon, Amazon S3 Cloud Storge (2016). Article (CrossRef Link)

[5] Dropbox, Dropbox Cloud Storage (2016). Article (CrossRef Link)

[6] Red Hat, Ceph Storage (2016). Article (CrossRef Link)

[7] G. community, GlusterFS Storage (2016). Article (CrossRef Link)

[8] Quobyte Inc, XtreemFS Storage (2016). Article (CrossRef Link)

[9] IBM, IBM Storage (2016). Article (CrossRef Link)

[10] HPE, HPE Storage (2016). Article (CrossRef Link)

[11] Microsoft, Microsoft Storage (2016). Article (CrossRef Link) URL

[12] D. Vellante, What is Federated Storage? (2010). Article (CrossRef Link)

[13] Hewlett Packard Enterprise, HPE 3PAR Peer Motion Software (2015). Article (CrossRef Link)

[14] D. Yang, and C. Ren, "VCSS: An Integration Framework for Open Cloud Stor-age Services," in *Proc. of IEEE 10th World Congress on Services (2014),* pp.155– 160, 2014.
Article (CrossRef Link)

[15] G. Vernik, A. Shulman-Peleg, S. Dippl, C. Formisano, M. C. Jaeger, E. K. Kolodner, M. Villari, "Data On-boarding in federated storage clouds," in *Proc. of IEEE International Conference on Cloud Computing CLOUD (2013)*, pp.244–,2013. Article (CrossRef Link)

[16] R. M. d. O. Libardi, S. Reiff-Marganiec, L. H. Nunes, L. J. Adami, C. H. Ferreira, J. C. Estrella, and *et al*, "MSSF: User-Friendly Multi-Cloud Data Dispersal," in *Proc. of 2015 IEEE 8th International Conference on Cloud Computing*, pp.341–348, 2015. Article (CrossRef Link)

[17] P. Janviriya, T. Ongarjithichai, P. Numruktrakul, C. Ragkhitwetsagul, "CloudyDays: Cloud storage integration system," in *Proc. of Proceedings of the 2014 3rd ICT International Senior Project Conference* ICT-ISPC 2014, pp.125–128, 2014. Article (CrossRef Link)

[18] R. Zhao, C. Yue, B. Tak, C. Tang, "SafeSky: A Secure Cloud Storage Middleware for End-User Applications," in *Proc. of 2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pp. 21– 30, 2015. Article (CrossRef Link)

[19] S. Shah, A. Nair, U. Thakar, "A framework to integrate multiple cloud storage services and provide consistent feature set," in *Proc. of 2014 Annual IEEE India Conference (INDICON), IEEE*, pp.1–5, 2014. Article (CrossRef Link)

[20] M. Malensek, S. Pallickara, S. Pallickara, "Autonomous Cloud Federation for High- Throughput Queries over Voluminous Datasets," *IEEE CLOUD COMPUTING, IEEE*, vol.3, No. 3, pp.40-49, 2016. Article (CrossRef Link)

[21] C. Esposito, M. Ficco, F. Palmieri, A. Castiglione, "Smart Cloud Storage Service Selection Based on Fuzzy Logic, Theory of Evidence and Game Theory," *IEEE Transactions on Computers X (c) (2015)*, pp.2348–2362, 2016. Article (CrossRef Link)

[22] C. W. Chang, P. Liu, J. J. Wu, "Probability-based cloud storage providers selection algorithms with maximum availability," in *Proc. of Proceedings of the International Conference on Parallel Processing (2012)*, pp.199– 208, 2012. Article (CrossRef Link)

[23] H. Hong Tao, W. Yan Ke, C. Cao Bingyao, Yu Fei, Y. Wu Yating, "A dynamic data allocation method with improved load-balancing for cloud storage system," in *Proc. of IET International Conference on Smart and Sustain-able City 2013 (ICSSC 2013)*, pp. 183–188, 2013.

Article (CrossRef Link)

[24] H. Nakazato, M. Nishio, M. Fujiwara, "Data allocation method considering server performance and data access frequency with consistent hashing," in *Proc. of 2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS) IEEE 2012*, pp. 1–8, 2012.

Article (CrossRef Link)

[25] W. Xie, J. Zhou, M. Reyes, J. Noble, Y. Chen, "Two-mode data distribution scheme for heterogeneous storage in data centers," in *Proc. of 2015 IEEE International Conference on Big Data (Big Data) IEEE 2015*, pp.327–332, 2015. Article (CrossRef Link)

[26] D. Jayathilake, "Towards structured log analysis," in *Proc. of JCSSE 2012 - 9th Inter-national Joint Conference on Computer Science and Software Engineering (2012)*, 259–264, 2012.

Article (CrossRef Link)

[27] M. Moh, S. Pininti, S. Doddapaneni, T.-S. Moh, "Detecting Web Attacks Using Multi-Stage Log Analysis," in *Proc. of 2016 IEEE 6th International Conference on Advanced Computing (IACC) (2016)*, pp.733– 738, 2016. Article (CrossRef Link)

[28] D. Grech, P. Clough, "Investigating Cluster Stability when Analyzing Transaction Logs," in *Proc. of 2016 IEEE/ACM Joint Conference on Digital Libraries (JCDL) (2016)*, pp.115–118, 2016.

Article (CrossRef Link)

[29] H. Chen, S. Tu, C. Zhao, H. Yongfeng, "Provenance Cloud Security Auditing System Based on Log Analysis," in *Proc. of 2016 IEEE International Conference of Online Analysis and Computing Science (ICOACS) (2016)*, pp.155–159, 2016. Article (CrossRef Link)

[30] A. Hamzehei, R. K. Wong, E. Mohammad, F. Chen, E. ShafieiBavani, "Scalable Sentiment Analysis for Microblogs based on Semantic Scoring," in *Proc. of 2015 IEEE International Conference on Services Computing Scalable (2015)*, pp.271–278, 2015. Article (CrossRef Link)

[31] OAuth 2.0 protocol (2016). Article (CrossRef Link)

[32] T. Dietterich, "Machine Learning," *McGraw-Hill Science/Engineering/Math*, vol. 28, No.4, 1996.

Article (CrossRef Link)

[33] Microsoft, Dynamic Expressions (2016). Article (CrossRef Link)

[34] Microsoft, Dynamic Expressions Operators (2016). Article (CrossRef Link)

[35] N. Iscoe, G. Williams, G. Arango, “Domain modeling for software engineering,” in *Proc. of [1991 Proceedings] 13th International Conference on Software Engineering, IEEE Comput. Soc. Press*, pp.340–343, 1991. Article (CrossRef Link)

[36] T. Dykstra, R. Anderson, Getting Started with Entity Framework 6 Code First using MVC 5, 2014. Article (CrossRef Link)

[37] Microsoft, Reflection Oriented Programing (2016). Article (CrossRef Link)

[38] Microsoft, Task Parallelism (Task Parallel Library) (2016). Article (CrossRef Link)

[39] E. BV, The Open Source Elastic Stack (2016). Article (CrossRef Link), URL

[40] M. Greenacre, R. Primicerio, Measures of distance between samples: Euclidean, *in: Multivariate Analysis of Ecological Data*, pp. 47–59, 2013.

[41] R. Kohavi, P. Langley, Y. Yun, “The Utility of Feature Weighting in Nearest-Neighbor Algorithms,” in *Proc. of Proceedings of the Ninth European Conference on Machine Learning*, pp.85-92, 1997. Article (CrossRef Link)

[42] E. Marcus, S. Hal, Blueprints for High Availability, Second Edition, Robert Ipsen, Indiana, 2003.

[43] A. Smola, S. Vishwanathan, Introduction to Machine Learning, *Cambridge university press*, 2008.

Rene Heinsen, received the bachelor's degree in Telematics Engineering from Pontificia Universidad Catolica Madre y Maestra, Dominican Republic in 2013. He is currently working toward the Master degree in Computer Engineering Department at Kyung Hee University, South Korea. His research interests includes: Grid and Cloud Computing, Big Data Storage, Parallel computing and Software Engineering.

Cindy López, received bachelor's degree in Information Systems and Computer Science from National Polytechnic School, Ecuador in 2007. She is currently working toward the Master degree in Computer Engineering Department at Kyung Hee University, South Korea. Her research interests includes: Cloud Computing, Storage Federation, Big-Data and Internet of Things.

Eui-Nam Huh received his bachelor degree in Computer Science and Engineering from Busan University, master degree of Computer Science from University of Texas, USA, and Ph.D. in Computer Science from the Ohio University, USA. His current research interests includes: Cloud Computing, Distributed Real-Time System, Grid Computing, High Performance Computing, Mobile Cloud, BigData, Internet of Things, Networking and Security. He is a professor in department of Computer Engineering and Science, director of Real-Time Mobile Cloud Research Center and Dean of Information Services and Strategy in Kyung Hee University.