Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

KATEDRA INFORMATYKI

PRACA MAGISTERSKA

KRZYSZTOF STYRC

ZARZĄDZANIE WIARYGODNOŚCIĄ I INTEGRALNOŚCIĄ DANYCH W SFEDEROWANYCH ZASOBACH CLOUD STORAGE

PROMOTOR:
dr inż. Marian Bubak

KONSULTACJA:
Piotr Nowakowski

Kraków 2012

OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

PODPIS

AGH
University of Science and Technology in Krakow

Faculty of Electrical Engineering, Automatics, Computer Science and Electronics

DEPARTMENT OF COMPUTER SCIENCE

MASTER OF SCIENCE THESIS

KRZYSZTOF STYRC

MANAGING DATA RELIABILITY AND INTEGRITY IN FEDERATED CLOUD STORAGE

SUPERVISOR:
Marian Bubak Ph.D

CONSULTANCY:
Piotr Nowakowski

Krakow 2012

Tu będą podziękowania

Contents

1. Introduction

1.1. Data reliability and integrity overview

1.2. Data integrity in VPH-Share Cloud Platform context

1.3. Objectives of this work

2. Data integrity

High-quality data availability and integrity property is a must-have requirement in many IT systems. A lot of enterprise and scientific effort has been put into development of tools and methods that support this capability. From cryptographic hash-based mechanisms that enable corruption discovery, to replication and error-correcting codes for data recovery, to security mechanisms preventing malicious data corruption. However, emerging trends in IT solutions, as cloud computing, put new challenges in this area. The following chapter presents the state of the art.

2.1. General methods and tools for ensuring data integrity

Providing a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications. The following section presents security building blocks that enable data integrity assurance. The cryptographic hash functions are core components of message authentication code algorithm to provide message integrity and authenticate the message creator. Error correcting codes are commonly deployed to be able to retrieve the original data after partial corruption.

2.1.1. Cryptographic hash functions

A cryptographic hash function is a hash algorithm that maps a message of arbitrary length to a fixed-length message digest (hash value). These algorithms enable determination of a message's integrity: any change to the message will, with high probability, result in a different message digest. This property appears very useful as a building block in various security constructions from generation and verification of digital signatures, to message authentication codes, to generation of random numbers.

A cryptographic hash function is expected to have the following properties [13]:

- **Collision resistance**: that it is computationally infeasible to find two different hash function inputs that have the same hash value. In other words, it is computationally infeasible to find x and x' for which $hash(x) = hash(x')$.
- **Preimage resistance**: that given a randomly chosen hash value, $hash\_value$, it is computationally infeasible to find an x so that $hash(x) = hash\_value$. This property is also called one-way property.
- **Second preimage resistance**: that it is computationally infeasible to find a second input that has the same hash value as any other specified input. That is, given an input x, it is computationally infeasible to find a second input x' that is different from x, such that $hash(x) = hash(x')$.

Currently, the Secure Hash Standard (SHS) [14] specifies five approved hash algorithms: SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512. Their strengths of the security properties discussed above, vary significantly. While one cryptographic hash function is suitable for one application, it might not be suitable for other. The general trend is that the longer the message digest (its hash), the stronger security guarantees, but also higher computational complexity.

Additionally, the algorithms differ in terms of the size of the blocks and words of data that are used during hashing or message digest sizes. They are presented in table 2.1.

Algorithm	Message Size (bits)	Block Size (bits)	Word Size (bits)	Message Digest Size (bits)
SHA-1	$< 2^{64}$	512	32	160
SHA-224	$< 2^{64}$	512	32	224
SHA-256	$< 2^{64}$	512	32	256
SHA-384	$< 2^{128}$	1024	64	384
SHA-512	$< 2^{128}$	1024	64	512

Table 2.1: Secure hash algorithm properties [14]

2.1.2. Error correcting codes

An error-correcting code (ECC) is an algorithm for expressing a sequence of numbers such that any errors which are introduced can be detected and corrected (up to certain level) based on the remaining numbers. All error correcting codes are based on the same basic principle: redundancy is added to information in order to correct any errors that may occur in the process of storage or transmission. In practice, the redundant symbols are appended to the information symbols to obtain a coded sequence (codeword).

ECC can be divided into two classes:

– **block codes**: that work on fixed-size blocks of predetermined size,

– **convolutional codes**: that work on bit streams of arbitrary length.

Among classical block codes the most popular are Reed-Solomon codes which are in widespread use on the CDs, DVDs and hard disk drives. Hamming codes are commonly used to prevent NAND flash memories errors. On the other hand, convolutional codes are widely used in reliable data transfer such as digital video, radio, mobile and satellite communication. Both block and convolutional codes are often implemented in concatenation.

Apart from embedding ECC in the hardware solutions, they are also being applied in software constructions to recover from eventual data corruption.

2.1.3. Message authentication codes

A message authentication code (MAC) is an authentication tag (also called a checksum) derived by applying an authentication scheme, together with a secret key, to a message [12]. The purpose of a MAC is to authenticate both the source of a message and its integrity without the use of any additional mechanisms.

MACs based on cryptographic hash functions are known as HMACs. They have two functionally distinct parameters: a message input and a secret key known only to the message originator and intended receivers.

An HMAC function is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input. The MAC is typically sent to the message receiver along with the message. The receiver computes the MAC on the received message using the same key and HMAC function as were used by the sender, and compares the result computed with the received MAC. If the two values match, the message has been correctly received, and the receiver is assured that the sender is a member of the community of users that share the key [12].

To compute a MAC over the data $text$ using the HMAC function with key K, the following operation is performed [12]:

$$MAC(text) = HMAC(K, text) = H(((K_0 \oplus opad)||H((K_0 \oplus ipad)||text))) \tag{2.1}$$

where:

– K_0 – the key K after any necessary pre-processing to form a B byte key,

– $ipad$ – inner pad, the byte $0x36$ repeated B times,

– $opad$ – outer pad, the byte $0x5c$ repeated B times,

– B – block size (in bytes) of the input to the H hash function,

– H – an approved hash function.

The Internet Engineering Task Force (IETF) published a RFC document to describe HMAC [28].

Apart from HMAC, a couple of other MACs have been proposed. Stinson [37] presented an unconditionally secure MAC based on encryption with a one-time pad. The cipher text of the message authenticates itself as nobody else has access to the one-time pad. Lai et al. [31] proposed a MAC based on stream ciphers. In their algorithm, a provably secure stream cipher is used to split a message into two substreams and each substream is fed into a linear feedback shift register (LFSR); the checksum is the final state of the two LFSRs.

2.2. Cloud storage model

Cloud computing is an emerging IT trend toward loosely coupled networking of computing resources. Its core feature is to move computing and data away from desktop and portable PCs to large data centers and provide it as a service. The popularity of this paradigm develops as it reduces IT expenses and provide agile IT services to both, organisations and individuals. Additionally, users are released from the burden of frequent hardware updates and costly maintenance, while paying for cloud services on consumption basis.

While cloud computing represents the full spectrum of computing resources, this work focuses on cloud storage services for archival and backup data. As it will be shown, this technology, apart from its advantages, introduces many problems, especially for ensuring data availability and integrity which may appear as untrustworthy.

2.2.1. General features

Cloud storage is a model of broadband network access to virtualized pool of storage resources on demand. In the spirit of cloud computing paradigm, it is mostly provided via REST/SOAP web service interface, however, other standard protocols are used. Despite incompatibilities among various cloud storage providers, as cloud computing gets more mature technology, their interfaces begin to standardize. Storage Networking Industry Association (SNIA) works toward developing a reference Cloud Data Management Interface (CMDI).

While different cloud storage solutions vary significantly, the following common properties can be derived:

– storage space is made up of many distributed resources, but still acts as one, virtualized layer,

– high fault-tolerance through redundancy and distribution of data,

– high data durability via object versioning,

– predominantly eventual consistency with regard to data replicas.

Typically, public cloud providers expose storage space as object data store, where data is organized into containers (or buckets). Each container consists of data objects (files) on which standard create, read, update, delete (CRUD) operations may be performed. Additional metadata is appended to containers and data objects such as name, size, creation/modification date or hash checksum.

Amazon Simple Storage Service (S3) [1], Rackspace Cloud Files [11] and Google Cloud Storage [5] are the most popular representatives of the illustrated cloud storage model. Despite the increasing popularity of public cloud storage providers, hybrid and private cloud solutions do exist, Openstack Swift [9] and Eucalyptus Cloud [4] to name just a few.

2.2.2. Interface and API

Current cloud storage systems mostly provide REST/SOAP web service interface to access the resources, in the spirit of Service Oriented Architecture (SOA) paradigm. While this thesis focuses on this method of access and its consequences, other providers expose different types of interface. Some of them are presented in figure 2.1 with concrete examples of solutions [26].

Despite the fact that web service interfaces enable loose coupling and technology interoperability, they require integration code with an application. Many multi-cloud libraries were created to enable interoperability across similar cloud services on a higher level of abstraction [30]. Their goal is to establish basic and uniform cloud storage access layer at the API level [7, 2].

Typically, cloud storage interfaces provide API to query, access and manage stored data, which can be divided into the following groups [1, 11, 5, 9, 4]:

– **Operations for authentication**: to secure the access to cloud storage data (mostly via token-based authentication),

– **Operations on the account**: to operate on account metadata such as managing existing containers and additional provider-specific data services,

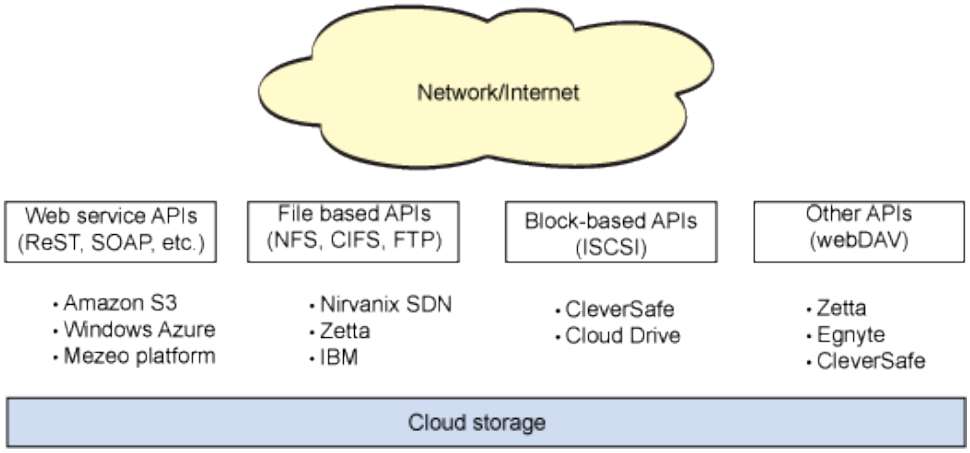


Figure 2.1: Cloud storage access methods [26]

– **Operations on the container**: to manage container policy, versioning, ACLs, lifecycle and location,

– **Operations on the data objects**: to enable CRUD operations.

There exists a growing trend to adjust provider-specific interfaces with the SNIA reference model [1, 11, 9].

2.2.3. Service Level Agreement

To provide high quality of service, cloud storage providers widely guarantee Service Level Agreement (SLA) contracts. These are mostly related to service availability during the billing cycle. The service downtime is considered as cloud network error or response errors to a valid user requests. Currently, most of the providers guarantee the availability level of 99.9% of the time [1, 11, 5].

However, if the provider will fail to provide a guaranteed level of service, the appriopriate percentage of the credit is returned to the client. In this sense, cloud storage should be still treated as best-effort. IT systems that demand uninterrupted operation simply cannot entirely rely on it.

Moreover, eventual consistency model is inherently embedded into the overwhelming majority of cloud storage architectures, which places new problems to the solutions, where strict data consistency is a crucial requirement [18, 30]. Besides eventual consistency, SLA contracts still only address the service availability, while omitting data integrity or retrievability speed issues. Even though, cloud storage service with described limitations still fit to the vast number of market applications.

Customers who require a higher data availability and integrity guarantees, still need to seek for hybrid solutions and develop sophisticated layers on top of existing infrastructure to meet their demands.

2.2.4. Constraints and limitations

Cloud storage architecture presented in previous section exhibit many advantages to potential users. Nevertheless, it also introduces a couple of drawbacks for demanding solutions.

The most striking consequence of cloud storage, is that data is stored remotely on provider's resources and user has very limited possibilities to monitor or check its data through abstract access layer. Even small security vulnerability may compromise the data of all users in public cloud model.

As it was shown in previous subsection, cloud SLA contracts still lack strong availability and integrity guarantees, rather than cost-return policy. Even though cloud storage is perceived as superb technology, a couple of serious downtimes have been reported in the last years. Amazon S3 users experienced several unavailability and data corruption periods [**?**], while Google Gmail lost data of thousands of accounts [**?**] and Google Docs enabled unauthorized access to the stored documents [**?**].

Cloud storage REST/SOAP interfaces are flexible and rich in capabilities, but when accessed remotely outside of cloud compute resources, they suffer from network latency for each HTTP request. Downloading a fragment of a file pose another challenge. It is mostly achieved by setting HTTP Range parameter to the desired value. However, only single range value is permitted. It is particularly problematic for data integrity monitoring protocols (presented in the next section) as they request a lot of small file's blocks, and for each block a separate HTTP request has to be sent, which means increased network overhead.

Moreover, cloud storage solutions lack user's code execution capability over stored data. The data has to be downloaded in order to perform computation. It makes present data integrity monitoring protocols impractical and inefficient, because they assume computation capability on the prover's side.

2.3. Approaches to data integrity in cloud storage

One of the fundamental goals of cryptography is data integrity protection. Primitives such as digital signatures and message-authentication codes (MACs), described in section 2.1, were created to allow an entity in possession of a file F to verify that it has not been tampered with. The simplest way is to use keyed hash function $h_k(F)$ to compute and store a hash value along with secret, random key k prior to archiving a file. To verify that the prover (remote server, cloud provider) possess F, the verifier releases key k and asks the prover to compute and return $h_k(F)$. By using multiple keys with their corresponding hash values, the verifier can perform multiple, independent checks. However, this approach introduces high resource overhead. It requires the verifier to store large number of hash values and the prover to read the entire file for every proof.

A more challenging problem is to enable verification of the integrity of F without knowledge of the entire file's contents. It was firstly described in general view by Blum et al. [20], who presented efficient methods for checking the correctness of program's memory. Following works concerned dynamic memory-checking in a range of settings. For instance, Clarke et al. [23] consider the case of checking the integrity of operations performed on an arbitrarily-large amount of untrusted data, when using only a small fixed-sized trusted state. Their construction employ an adaptive Merkle hash-tree over the contents of this memory. However, Naor and Rothblum showed that online memory checking may be prohibitively expensive for many applications [33]. This implies that applications requiring memory checking should make cryptographic assumptions, or use an offline version of the problem.

Unauthorized modifications to portions of files can be detected by cryptographic integrity assurance upon their retrieval. But in its basic form it does not enable such detection capability prior to the retrieval, what many other schemes aim to provide.

One of the mostly developed model of ensuring integrity of remotely stored data is the proofs of retrievability (POR). The first formal description of POR protocol was proposed by Juels and Kaliski [27]. In their scheme, the client applies error-correcting code and spot-checking to ensure both possession and retrievability of files. Shaham et al. [34] achieve POR scheme with full proofs of security and lower communication overhead. Bowers et al. [21] simplify and improve the framework and achieve lower storage overhead as well as higher error tolerance. Later on, they extend it to distributed systems [22]. However, all these schemes are focusing on static data. Before outsourcing the data file F a preprocessing steps are applied. Every change to the contents of F require re-processing, which introduces significant computation and communication complexity. Stefanov et al. [36] propose an authenticated file-system for outsourcing enterprise data to the untrusted cloud service providers with the first efficient dynamic POR.

Atenise et al. [16] presented the provable data possession (PDP) model in order to verify if an untrusted server stores a client's data without file retrieval. Key components of their scheme are public key based homomorphic verifiable tags. In the subsequent work, Atenise et al. [17] described a PDP scheme that uses only symmetric key cryptography. As a result, they achieved lower performance overhead.

A couple of practical implementations for remote integrity assurance have been developed. Bowers et al. [22] designed HAIL (High Availability and Integrity Layer) which takes advantage of data distribution over a set of servers to achieve efficient POR-like scheme. Shraer et al. [35] created Venus, a scheme that guarantees integrity and consistency for a group of clients accessing a remote storage provider. Venus ensures that each data object read by any client has previously been written by some client. Additionally, it protects against retrieving older version of the object. Bessani et al. [19] implemented DEPSKY, a system that improves the availability, integrity and confidentiality of information stored in the cloud through encryption, encoding, and replication of data on diverse clouds that form cloud-of-clouds.

In the following subsections we examine exhaustively a couple of schemes mentioned above. We present their architecture, advantages and limitations.

2.3.1. Proofs of retrievability

In a POR [27, 21] protocol, a file is encoded by a client before deploying it on cloud storage for archiving. Then, it employs bandwidth-efficient challenge-response scheme to probabilistically guarantee that a file is available at remote storage provider. Most of POR protocols proposed to date, use the technique of spot-checking in the challenge-response protocol to detect data corruption. In each challenge, a subset of file blocks is verified, and the results of a computation over these blocks is returned to the client. The returned results are checked using the original checksums embedded into the file at encoding time.

The primary POR-like protocol we consider in detail, was proposed by Juels and Kaliski [27] – a MAC-based POR scheme. In this approach, they firstly preprocess the file F by applying error-correcting codes and MAC checksums in the following steps:

1. **Error correction**: the file is divided into b blocks of the same length and apply an (n, k, d)-error correcting code, which expands each chunk of size k into size n and is able to recover from up to $d - 1$ errors. The resulting file is denoted as F'.

2. **Encryption**: the file with appended ECCs is encrypted.

3. **MAC computation**: a m number of blocks are selected in F'', their MACs computed and appended to the file.

4. **Permutation**: of file blocks to secure appended MACs against corruption.

The resulting modified file F^* is presented schematically in figure 2.2.

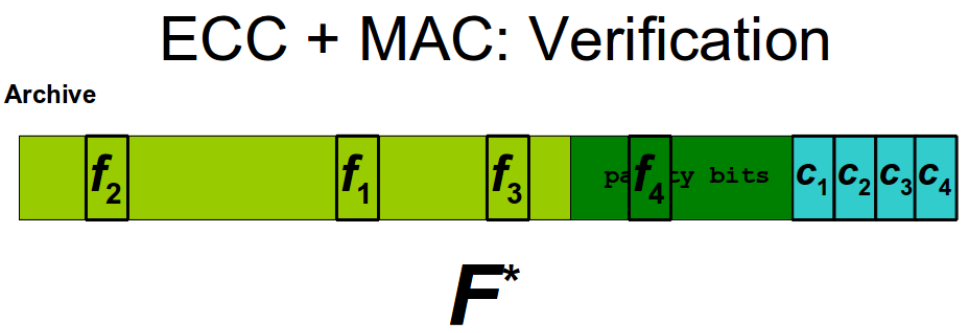


Figure 2.2: File F after MAC-based POR modifications [27]

In the same paper [27], Juels and Kaliski proposed a sentinel-based POR scheme. Similarily to the MAC-based approach it utilizes ECCs, but rather than chosing MAC blocks it embeds sentinels in random positions in F, sentinels being randomly constructed values. It is important that sentinels shall be indistinguishable from the encrypted file contents. The scheme consists of the following steps:

1. **Error correction**: the file is divided into b blocks of the same length and apply an (n, k, d)-error correcting code, which expands each chunk of size k into size n and is able to recover from up to $d - 1$ errors. The resulting file is denoted as F'.

2. **Encryption**: the file with appended ECCs is encrypted.

3. **Sentinel creation**: the randomly constructed sentinels are embedded in random positions in F'

4. **Permutation**: which randomizes sentinel positions.

In both approaches, if the prover has modified or deleted a substantial e-portion of F, then with high probability, also change roughly an e-fraction of MAC-blocks or sentinels, respectively. It is therefore unlikely to respond correctly to the verifier. Upon file retrieval, the user verifies file's checksum. If it is not valid, then it starts file recovery based on stored ECCs.

Of course, application of an error-correcting (or erasure) code and insertion of sentinels enlarges F^* beyond the original size of the file F. The expansion induced by both POR protocols, however, can be restricted to a modest percentage of the size of F. Importantly, the communication and computational costs of the protocols are low.

The obvious advantage of the presented schemes is that they can be parameterized.

Subsequent POR works [34, 21, 22, 35, 36] introduced further optimizations to the solution described. The most important are: moving MAC computation to the prover side and precomputing partial checksum values.

However, POR-like schemes are not free from drawbacks. The primary limitation is that preprocessing phase introduces non-negligible computational overhead. Moreover, it requires storage of file F in modified form. What is even more problematic, it assumes that storage service provides user's code execution capability, which is not true for current cloud storages (see section 2.2). For this reason, practical POR-like implementation would require moving prover logic for computing challenge-response queries to the verifier. As a consequence, each access to the portion of a file (MAC block or sentinel) would require separate HTTP request. As many such accesses are performed per each file, it would be impractical (except for large files, for which hundreds of short HTTP requests would be faster than downloading the entire file).

2.3.2. Data integrity proofs

Data integrity proof (DIP) [29] is a protocol, which just like POR, aims to assure that the remote archive poses the data. Unlike POR schemes, it does not involve any modifications to the stored file. The client before storing data file F, preprocesses it to create suitable metadata, which is used in the later stage of data integrity verification. The preprocessing stage consists of the following steps:

– **Generation of metadata**: the file F in divided into n blocks that each are m bits length. Then, for each data block, a set of k out of m bits are selected. The value of k is in the choice of the verifier and is a secret known only to him. Therefore, we get $n * k$ bits in total (see figure 2.3).

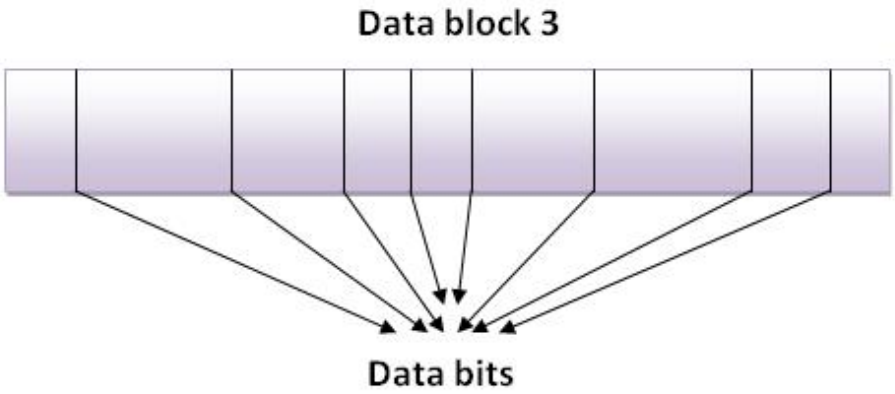


Figure 2.3: Bits generation in DIP scheme [29]

– **Encrypting the metadata**: each of the metadata from the data blocks, is encrypted by using a suitable algorithm and concatenated.

– **Appending the metadata**: all the metadata are appended to the file F, however, they can be also stored in the verifier (see figure 2.4).

To verify F integrity, the verifier utilizes challenge-response mechanism. In each challenge, it verifies a single block i specifing the positions of the k selected bits and retrieves encrypted metadata for this block to compare the values. Any mismatch between the two would mean a loss of the integrity of the clients data at the cloud storage.

While DIP scheme seems trivial, it eliminates a couple of disadvantages of the POR approach. Firstly, data integrity assurance does not require any modifications to the stored file, but also prevents the data recovery capability by ECC. It also exhibits negligible computational overhead. However, it still either assumes user's code

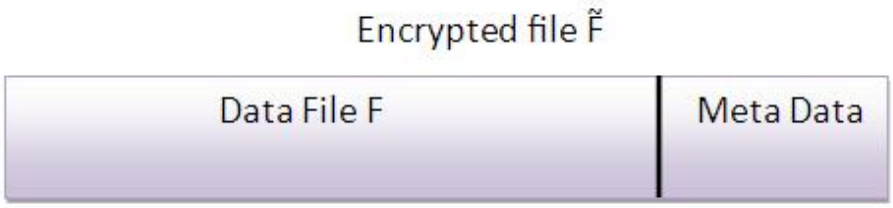


Figure 2.4: Resulting file in DIP scheme [29]

execution capability by cloud provider or requires large number of accesses to non-continuous data fragments. Such data acceses are performed in separate HTTP requests in the current cloud storages (see section 2.2), which is practically infeasible.

2.4. Conclusions

3. Data reliability and integrity service

This chapter presents the architecture of Data Reliability and Integrity (DRI) service. It starts by describing the environment of VPH-Share Cloud Platform which specifies requirements under which DRI operates. Then it defines its design and interfaces with other parts of the system. At the end, the core validation heuristic algorithm is presented.

3.1. Overview

3.2. Data and Compute Cloud Platform context

VPH-Share Data and Compute Cloud Platform project aims to design, implement, deploy and maintain cloud storage and compute platform for application deployment and execution. The tools and end-user services within the project will enable researchers and medical practitioners to create and use their domain-specific workflows on top of the Cloud and high-performance computing infrastructure. In order to fulfill this goal, Cloud Platform will be delivered as consistent service-based system that enables end users to deploy the basic components of the VPH-Share application workflows (known as Atomic Services) on the available computing resources and then enact workflows using these services. The general overview of Cloud Platform architecture with interactions to other parts of the VPH-Share is illustrated in figure 3.1.

The end-user interface for the Cloud Platform will be implemented as Master UI (web portal) enabling coarse-grained invocations of the underlying core services (more in subsection 3.2.3). The Cloud Platform itself will be deployed on available cloud and physical resources.

Data storage is an essential functionality of the Platform. It is achieved by federated cloud storage which makes use of both, cloud and other storage resources with redundancy and is accessible through common data layer – LOBCDER service. Further description in subsection 3.2.2. Atmosphere Internal Registry (AIR) serves as centralised metadata storage component which enables integration between loose-coupled services and is presented in subsection 3.2.1.

In VPH-Share project a strong emphasis is placed on providing data integrity, availability and retrievability (that it can be retrieved at minimal specified speed). To fulfill this requirement, a Data Reliability and Integrity (DRI) service was designed, implemented and deployed as one of the core Cloud Platform's services – which is a primary topic of this thesis.

MORE ON DATA VALIDATION, ITS IMPORTANCE, DIFFICULTIES AND SHORTCOMINGS HERE.

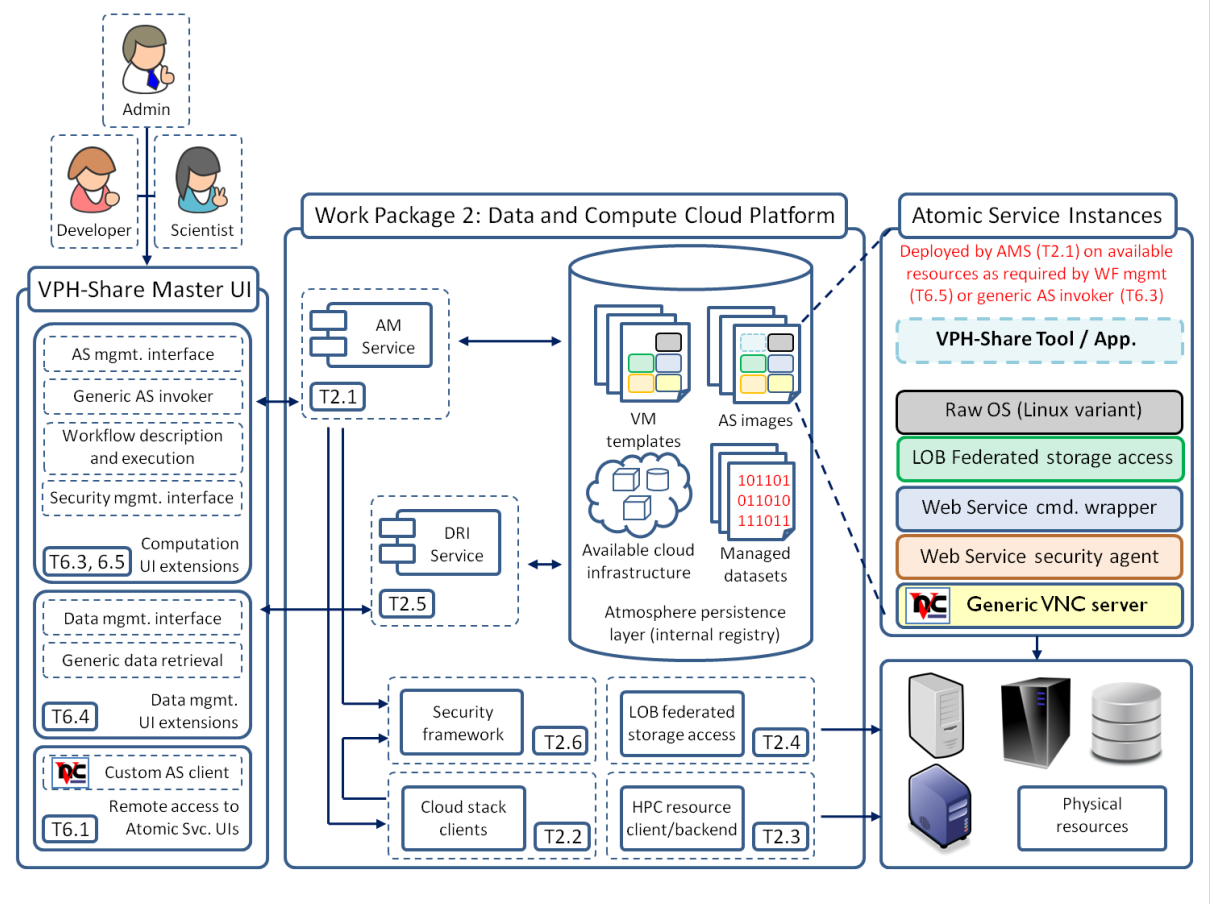


Figure 3.1: Cloud Platform architecture and its relation to other project components

3.2.1. Atmosphere Internal Registry

The Atmosphere Internal Registry (hereafter also referred to as the Atmosphere Registry, the AIR component or simply the Registry) is a core element of the Cloud Platform, delivering persistence capabilities. Its components and interactions are depicted in figure 3.2. The main function of AIR is to provide a technical means and an API layer for other components of the Cloud Platform to store and retrieve their crucial metadata. Having a logically centralised (though physically dispersed, if needed to meet high availability requirements) metadata storage component is beneficial for the platform, as multiple elements may use it not only to preserve their "memory", but also to persistently exchange data. This is facilitated through the well-known database sharing model where the data storage layer serves as a means of communication between autonomous components, making the Atmosphere Internal Registry an important element of the platform.

From DRI perspective, AIR will store necessary metadata:

– **datasets metadata** and files they contain,

– **integrity checksums** for data validation,

– **service configuration**.

Such design enables us to implement DRI as stateless service.

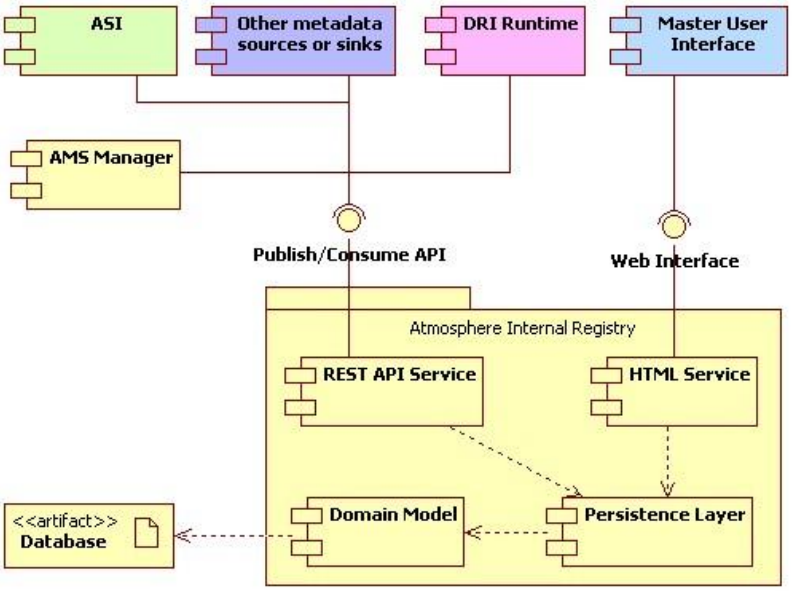


Figure 3.2: Components and interactions of AIR component

3.2.2. Federated cloud storage

Data storage is an essential part of VPH-Share Cloud Platform. The increasing popularity of cloud storage services due to high quality-cost ratio is leading more organisations to migrate and/or adapt their IT infrastructure to operate completely or partially in the cloud. However, as mentioned in section 2.3, such a solution has its limitations and implications. To overcome some of them one can leverage the benefits of cloud computing by using a combination of diverse private and public clouds. This approach is developed in Cloud Platform as federated cloud storage, where data is stored redundantly on various cloud storage services. The benefits are the following:

– **High availability** – data may be temporarily unavailable and/or corrupted for various reasons when system relies on a single cloud storage provider, as shown in recent cases (see section 2.3). In cloud federation we are able to store data redundantly and switch between providers when one becomes unavailable.

– **No vendor lock-in** – there is currently some concern that a few cloud computing providers become dominant, the so called vendor lock-in issue. Migrating from one provider to another one can also be expensive. In cloud federation we are able to easily switch between providers considering their charging or policy practices.

Federated cloud storage is not sufficient to provide data unavailability and corruption tolerance. For this purpose, an additional service has to be designed to actively monitor data integrity – DRI.

Access to the federated cloud storage is via common access layer – LOBCDER service – served by WebDAV protocol. However, DRI service will access cloud storage services directly to take advantage of cloud federation and to omit redundant LOBCDER overhead.

HERE WILL BE A PICTURE ILLUSTRATING HOW LOBCDER AND DRI WORKS ON CLOUD FEDERATION !!!

3.2.3. Master UI

3.3. DRI data model

The Cloud Platform concerns itself primarily with access to binary data, especially via file-based interface. Managed dataset represents a single entity that can be managed. At its core, it consists of a selection of files, to which a portion of metadata is appended and stored in AIR reigstry. As data integrity is a crucial requirement of the platform, the datasets can be tagged for automatic data integrity monitoring (DRI).

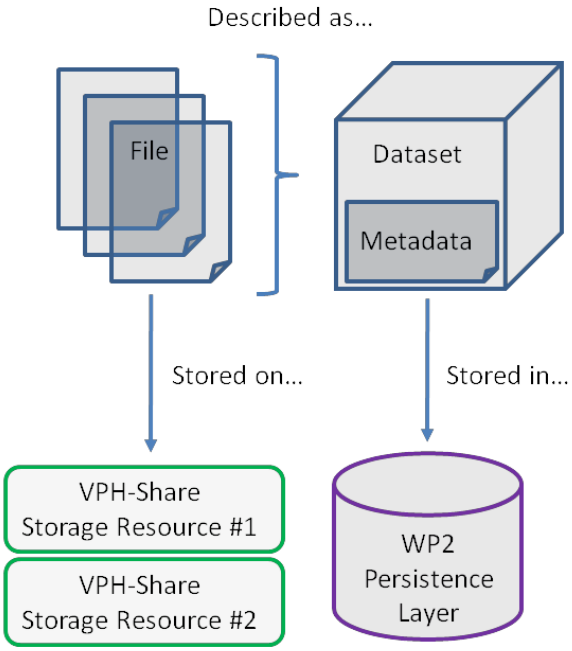


Figure 3.3: Schematic representation of a VPH-Share Managed Dataset

3.3.1. Metadata schema

Each managed dataset may consist of an arbitrary number of files (logical data) and can be stored on one or more storage resources (data source). Specific security constraints can be attached to data items, i.e. it cannot be used in public clouds. In DRI component, validation checks are of configurable policy (management policy). The schema is depicted in the figure 3.4.

The managed dataset metadata consists of the following elements:

- **owner** – reference to user ID of dataset creator,
- **list of logical data** – list of logical data ids it consists of,
- **is managed** – marker determining whether dataset's integrity is monitored,
- **DRI status** – dataset's reliability and integrity status,
- **date of registration**.

Additionally, each logical data will consist of the following attributes:

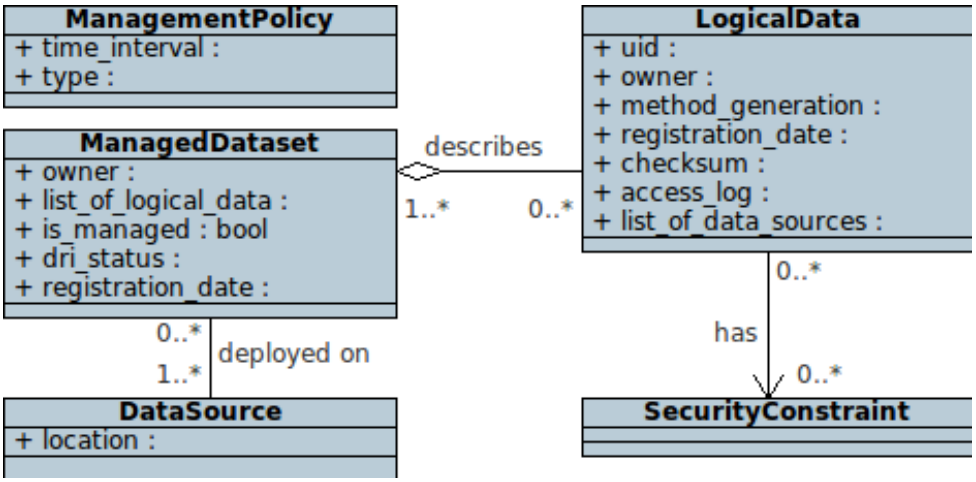


Figure 3.4: DRI metadata schema

– **owner** – reference to user ID,

– **method of generation** – whether it was uploaded manually, generated by an application or registered externally,

– **date of registration**

– **checksum** – file's value of a cryptographic hash function calculated upon registration and used to validate integrity and availability of file,

– **list of data sources** – to which the file is currently deployed,

– **access log**.

While this schema is expected to cover all the requirements addressed in the DRI service, we foresee that additional metadata can be added later without affecting already stored.

3.3.2. Tagging datasets

Before automatic verification of managed datasets can take place, it is first necessary to tag specific data as subject for management. It is foreseen that the DRI component will involve a user interface extension (portlet-based) to enable authorised users to tag specific datasets for automatic management. This interaface will display the existing data storage resources and allow creation of new managed datasets consisting of selected files.

In addition to UI based tagging, the DRI component provides API-level access for the same purpose, whenever a VPH-Share application (or workflow) needs to tag specific data as a managed dataset.

3.4. Requirements

VPH Cloud Platform puts strong empasis on data storage and availability and integrity assurance, as it is mostly static medical data of great importance. To fulfill this goal, it takes advantage of outsourcing data storage. To achieve high data availability in cases of cloud failure (see section 2.2) and avoid vendor lock-in issues, it utilizes mixture of public and private clouds. However, it is still desired to monitor periodically the availability and

integrity of data deployed at cloud storage services. DRI service was designed under several crucial requirements that dictate its architecture and potential. This section describes these requirements.

Cloud Platform requirements can be divided into two groups: functional and non-functional.

3.4.1. Functional

Functional requirements are related to the core system capabilities that it is desired to ensure. These are the following:

- **Periodic and on-request validation**: DRI has to periodically fetch datasets' metadata from the AIR registry and check the availability and integrity of managed datasets. It also has to enable API interface for this operation to be invoked by the user on demand.
- **Data replication**: DRI has to enable API interface for data replication from one data source to the other.
- **User notification about integrity errors**: validation and replication operations has to be performed asynchronously and the results have to presented to the user via suitable notification service.

3.4.2. Nonfunctional

Nonfunctional requirements are related to the quality of the core system capabilities, as it is desired that validation and replication mechanisms are performed efficiently. These are the following:

- **Bandwidth-efficient validation mechanism**: naive download and check technique is infeasible. The main DRI challenge is to design and implement efficient data validation protocol without accessing the entire file.
- **Scalablility**: as it is foreseen that the amount of data stored in Cloud Platform resources will be significant, the DRI has to present the ability to scale with the size of data. It is suggested to be achieved by deploying many independent DRI replicas.
- **Configurability**: DRI has to provide API interface or UI portlet to configure its most important parameter: validation period. It is also desired for the designed validation protocol, to be configurable, to be able to adjust error-detection probability.

3.5. Architecture

The DRI Runtime is responsible for enforcing data management policies. It keeps track of managed components and periodically verifies the accessibility and integrity of the managed data. It operates autonomously as well as on request. It also interacts and cooperates with the other important Cloud Platform's components (see section 3.2) to fulfill its goal. At the core of the service is an application that periodically polls the AIR registry for a list of managed datasets and then proceeds to verify the following:

- the availability of each dataset at locations read from the AIR registry,
- the binary integrity of each dataset (checksum-based validation).

The DRI Runtime contacts individual data sources and validates the integrity and availability of the data stored on these resources. Should errors occur, the DRI Runtime invokes a notification service to issue a warning message to subscribed system administrators (typically, the user defined as the dataset's owner).

3.5.1. Overview

The architecture of the DRI service was mostly influenced by the Cloud Platform environment (section 3.2) and the requirements and challenges it introduces (section 3.4). Figure 3.5 presents its overview.

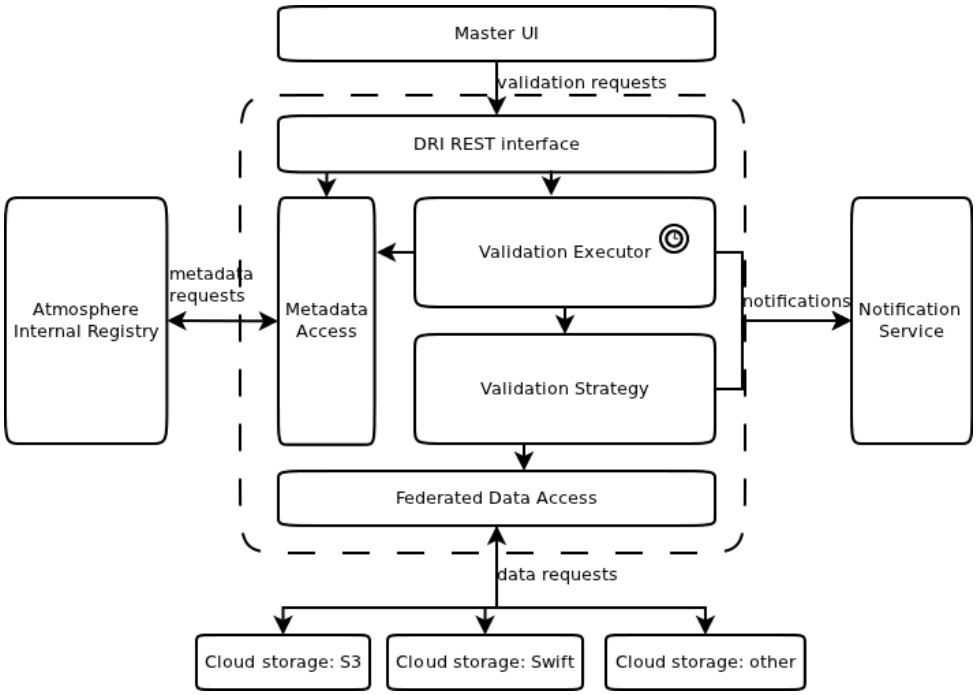


Figure 3.5: DRI architecture overview

The DRI Runtime consists of a couple of subcomponents which interact with each other as well as with other services described. It exposes REST service interface to be invoked by Master UI or the Atomic Services (see subsection 3.5.2 for details). MetadataAccess component is responsible for retrieving necessary metadata from the AIR registry. Access to the federated cloud storage is performed via FederatedDataAccess layer in order to hide the underlying complexity and differences between various cloud storage access. ValidationExecutor represents the core of the DRI. Its objective is to manage all the validation tasks, both periodical and requested by user. For each logical data it invokes ValidationStrategy component to perform its data validation algorithm.

MORE HERE HERE HERE

3.5.2. Interface and API

As hinted upon in the preceding sections, DRI provides end-user interfaces in the form of a Master UI portlet, as well as an API implemented by the Runtime service, where DRI features may be invoked directly by other VPH-Share intrastructure components.

Here we intend to focus on the API, which provides access to the low-level functionality of DRI and enables it to be configured.

As DRI exposes a stateless Web Service, all configuration parameters are stored in the Atmosphere Internal Registry. Whenever a configuration change request is invoked the DRI automatically updates policies stored in AIR. In light of this, the DRI API supports the following operations:

- ***getDataSources(): DataSourceID[]*** – returns a list of currently registered data storage resource identifiers;
- ***getDataSource(dataSourceID): DataSourceDescription*** – returns the information on a specific storage resource, as stored in the Atmosphere Internal Registry in a form of XML document describing the structure of storage resource;
- ***registerManagedDataset(DatasetDescription) : datasetID*** – tags a new dataset for management given in the form XML document describing the structure of the dataset;
- ***alterManagedDataset(DatasetID, DatasetDescription) : void*** – changes the dataset specification stored in the AIR. This action should be used to add or remove files from a managed dataset;
- ***removeManagedDataset(DatasetID) : void*** – excludes the specified dataset from automatic management. This does not delete the data, it merely stops DRI Runtime from monitoring them;
- ***getManagedDataset(DatasetID) : ManagedDatasetDescription*** – requests information on a specific managed dataset stored in the AIR, returning XML document specifing the structure of the managed dataset;
- ***getOwnerManagedDataset(User) : DatasetID[]*** – returns a list of user's managed dataset ids;
- ***assignDatasetToResource(DatasetID, DataSourceeID) : void*** – requests DRI to monitor the availability of a specific managed dataset in a specific storage resource. If this dataset is not yet present on the requested storage resource, it will be automatically replicated there;
- ***unassignDatasetFromResource(DatasetID, StorageResourceID) : void*** – requests DRI to stop monitoring the availability of a specific managed dataset on a specific storage resource. If the dataset is not present on the selected storage resource, this action has no effect;
- ***validateManagedDataset(DatasetID) : output*** – performs asynchronous validation of the specific dataset and produces a document which lists any problems encountered with the dataset's availability on the storage resources to which it had been assigned;
- ***setManagementPolicy(ManagementPolicy) : void*** – changes monitoring parameters. ManagementPolicy is an XML document specifying the frequency and type of availability checks performed on managed datasets;
- ***getManagementPolicy() : ManagementPolicy*** – retrieves an XML description of the management policy.

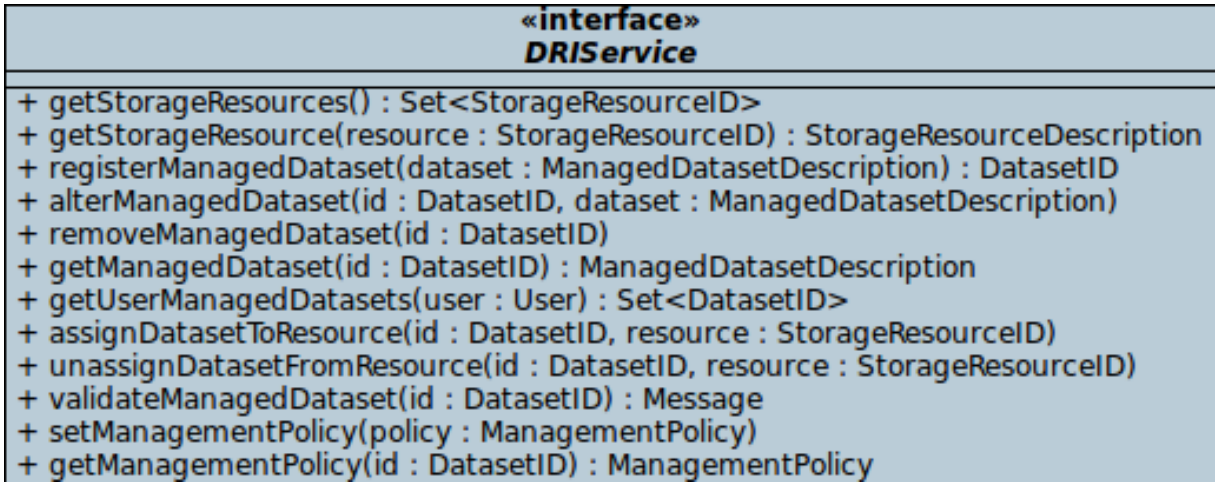


Figure 3.6: DRI Service interface

Each invocation requires to be augmented by the security token which can be intercepted and parsed by the security component residing on the virtual machine on which the DRI Runtime operates.

3.5.3. Typical use cases execution flow

The two main tasks performed by the DRI is monitoring of data integrity and replication of managed datasets among various data sources. Now, we will present a typical execution flow for this tasks through DRI subcomponents presented in figure 3.5 using sequence diagrams.

We start with data validation illustrated in figure 3.7. The $validateManagedDataset()$ method is the one designed to be called by the user, however, the incorporated logic for periodic integrity checks is the same, as ValidationExecutor fetches all the managed datasets from AIR and invokes this operation for each of them.

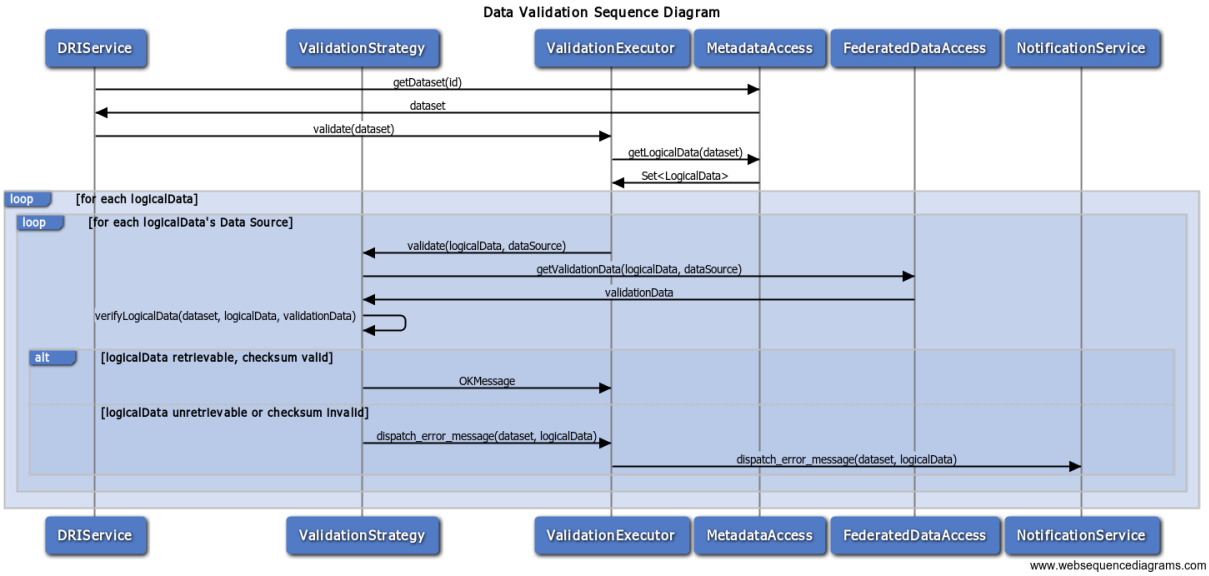


Figure 3.7: DRI $validateManagedDataset()$ call sequence diagram

Upon $validateManagedDataset()$ call, the DRI service retrieves dataset's metadata from AIR registry invoking $getDataset(id)$ on MetadataAccess object and passes it to the ValidationExecutor to apply data availability and integrity check. Subsequently, ValidationExecutor retrieves the metadata of all the logical data items which are part of the specified dataset invoking $getLogicalData(dataset)$ on MetadataAccess. Validation occurs separately for each logical data and against every data source on which it is stored by ValidationStrategy object which implements efficient validation protocol. To perform this operation, it has to get some necessary portion of data from data source by calling $getValidationData(logicalData, dataSource)$ on FederatedDataAccess object. With this necessary data, ValidationStrategy can verify whether the specified logical data is available and valid. If not, the error message is dispatched via NotificationService. It incorporates all or nothing strategy, which means that the corruption of a single logical data results in marking whole dataset as invalid. However, detailed error message informs which items' corruption has been detected on which data sources. The $validateManagedDataset()$ call performs asynchronously (no return value), but the result of the operation can be checked via NotificationService.

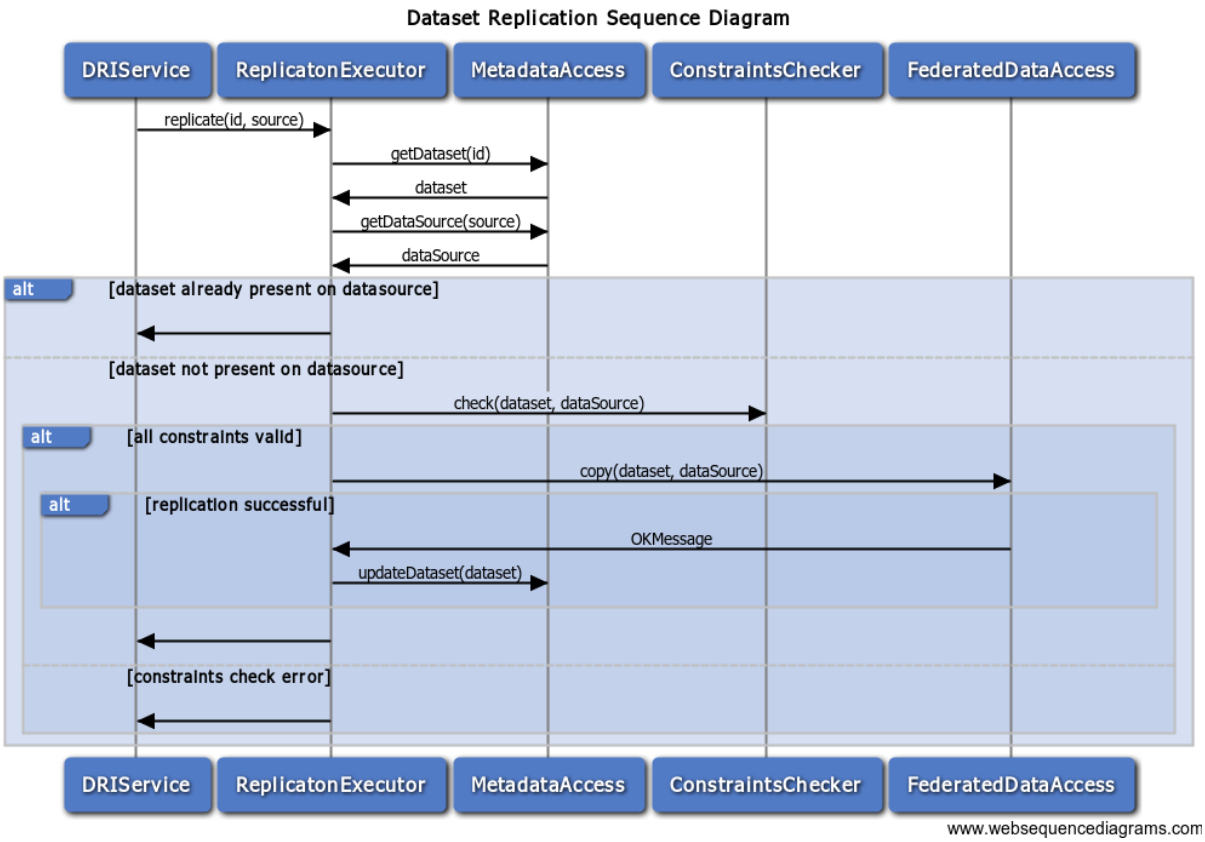


Figure 3.8: DRI $assignDatasetToResource()$ call sequence diagram

Upon *assignDatasetToResource()* call (figure 3.8), the DRIService retrieves dataset's and data source's metadata from AIR registry invoking $getDataset(id)$ and $getDataSource(source)$ on MetadataAccess object and passes it to the ReplicationExecutor. If dataset is already present on the specified location, this operation has no effect. Subsequently, ReplicationExecutor checks all the constraints, via $check(dataset, dataSource)$ call, that may be associated with the dataset (such as it cannot be stored in public clouds) and if they are valid, it performs the replication. This operation simply copies data from one data source on which the dataset is already present to the specified data source. In case of any failures, the operation aborts with no side effects. Upon successful execution, the necessary dataset metadata is updated in AIR registry ($updateDataset(dataset)$ call).

3.6. Data validation mechanism

At the heart of DRI service lies its validation heuristic algorithm which is going to be described now in detail. As it was discussed in chapter 2, a lot of effort was put into ensuring data integrity and availability on storage resources. However, cloud storage model sets new challenges in this area due to its constraints and limitations presented in section 2.3. The problem was addressed in the papers described in section 2.4, one of which - Proofs of Retrievability - became the basis for many enhancements, modifications and improvements. Each of the solution approaches difficulties with vast amount of data stored on cloud storages by creating sophisticated protocols which download only a fraction of data $(1 - 10\%)$ and try to guarantee possibly the highest error-detection rate. Unfortunately, introduced solutions do not address performance issues of these protocols with regard to typical cloud storage interfaces and VPH-Share platform requirements:

– requesting many small fragments of data greatly affects network overhead as each fragment requires separate HTTP request,

– cloud storages do not allow executing users' code,

– VPH-Share platform requires storing data in unmodified form.

These limitations make these solutions impractical. For the needs of DRI service, a new approach was designed with practical feasibility and low network overhead as main objectives in mind. Our heuristic utilizes spot-checking technique [27, 16, 29, 24] to verify data integrity with high probability. Unlike cited mechanisms [29, 27] which generally implement fine-grained spot-checking, we are aware of cloud storage limitations and employ coarse-grained spot-checking, realizing that it will result in reduced error-detection rate.

3.6.1. Algorithm description

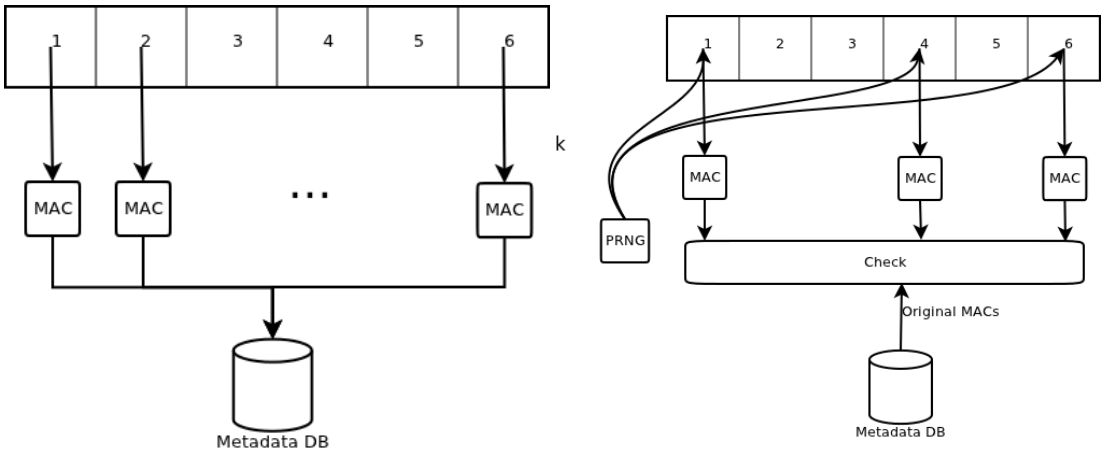
According to the data model described in section 3.1, dataset is a set of files stored on cloud storage resource. To be able to validate dataset's integrity, it is firstly necessary to retrieve dataset's data, then compute and store some checksum metadata for each file. During the validation process, the dataset's data is retrieved and checksums are computed again to compare them with the original ones. The following pseudocode illustrates this operation:

```
   Data: valid dataset id
   Result: true if dataset valid, error messages otherwise
1  dataset = get_dataset_metadata(id);
2  files = get_dataset_files(dataset);
3  for file in files do
4      for data_source in dataset.get_data_sources() do
5          data = get_validation_data(file, data_source);
6          if data == null then
7              dispatch_error_message(file, data_source);
8          end
9          result = validate_data(data, file, dataset);
10         if result is invalid then
11             dispatch_error_message(file, data_source);
12         end
13     end
14 end
```

Algorithm 1: Dataset validation algorithm

To validate a dataset, the metadata of it and all the files it consists of have to be retrieved (lines 1–2). Then, each file is validated against every data source it is stored on (lines 3–4). To validate a single file, the algorithm retrieves its necessary data (line 5). If errors occur, the file's unavailability message is reported (lines 6–8). Otherwise, integrity checksum is computed and checked with the original one (line 9). Any resulting integrity error is reported (lines 10–12).

The core part of the validation mechanism is the validation algorithm (validation protocol) which prepares dataset's metadata and then is able to validate it. As a typical integrity checking algorithm it comprises of two phases:



(a) Setup phase

(b) Validation phase

Figure 3.9: Single file validation heuristic consists of two phases: setup and validation. In setup phase (a), the file is divided into n chunks and MAC hash is computed for every data chunk which is then stored in metadata registry. In validation phase (b), the file is again divided into n chunks and pseudorandom number generator selects a set of k out of n chunk indexes that are downloaded, their checksums computed and compared with the original ones stored in metadata registry.

1. **setup** – takes place once (or after each file update) and generates checksum used during every validation phase. During this phase the file is divided into n chunks of size $\frac{F}{n}$ (where F denotes the size of entire file) and MAC hash is computed for every data chunk. Then, a set of n checksums is stored in metadata registry for further use.

2. **validation** – takes place on every dataset validation request. During this phase, the file is again divided into n chunks and pseudorandom number generator selects k out of n chunk indexes that are downloaded and their checksums computed. Then, computed checksums are compared with original ones stored in metadata registry.

These phases are graphically presented in figure 3.9.

Values k and n are configurable and can be set to fulfill specified requirements. The greater the value of n, the smaller the chunks are (see setup phase description) and more separate k HTTP requests need to be sent to maintain demanded error-detection rate. With fixed n, the greater the value of k, the higher error-detection rate.

Datasets consisting of large number of small files can lead to the performance bottleneck for two reasons: many separate HTTP requests for each small file and big storage overhead as chunks are small for small files, but for each a MAC checksum is stored. For this reason, one additional parameter $threshold$ was introduced to improve performance over small files. Data integrity for files of size smaller than $threshold$ is provided by classic entire-file SHA-256 checksum. The value of $threshold$ parameter will be established empirically based on performance test results presented in chapter 5.

3.6.2. Algorithm analysis

Validation algorithm described in the previous subsection is rather simple in its design, but poses properties that make it practically feasible:

– files are stored in unmodified form,

– network overhead and number of HTTP requests can be configured by n and k parameters,

– error-detection rate can be configured by n and k parameters.

Detailed algorithm description enables theoretical estimation of the most interesting properties that characterize our solution:

– **Error-detection rate** – expresses the probability to detect data corruption. For small changes, its value is equal to the probability that the change took place within the k blocks that are verified:

$$E_{det} = \frac{k}{n} \qquad for\ small\ changes. \tag{3.1}$$

However, if the prover has modified or deleted substantial e-portion of F, then with high probability, it also changed roughly an e-fraction of chunks.

– **Network overhead** – expresses the fraction of data that has to be retrieved in order to verify the integrity on desired level. As in our scheme, we download k out of n chunks (each of size $\frac{F}{n}$), the network overhead value is proportional to:

$$N_{over} \sim F \times \frac{k}{n}. \tag{3.2}$$

– **Execution time** – expresses the time needed to validate a file of size F. It depends on average network download speed, as well as on network latency to the cloud provider. Each HTTP request introduces latency, so the more requests are sent, the more network efficiency is affected. We estimate this value in the following way: each chunk of size $\frac{F}{n}$ is downloaded in $\frac{F}{n \times speed}$ time (where $speed$ is download speed in bits/s) plus additional request latency time. As we validate k chunks per validation phase, we get the execution time:

$$T_{exec} \sim k \times (\frac{F}{n \times speed} + latency) \tag{3.3}$$

However, the latency factor can be drastically reduced by performing a set of HTTP requests concurrently.

Metric	our approach	whole-file approach
E_{det}	$\frac{k}{n}$	1
N_{over}	$\sim F \times \frac{k}{n}$	$\sim F$
T_{exec}	$\sim k \times (\frac{F}{n \times speed} + latency)$	$\sim \frac{F}{speed} + latency$

Table 3.1: Performance metrics comparison between our and whole-file approaches

In table 3.1 we summarize the three metrics values for our approach in comparison with whole-file validation approach. Practical performance evaluation with varying values of the parameters n, k and $threshold$) and in the real cloud storage environment is presented in chapter 5.

3.6.3. Comparison with other solutions

4. DRI implementation

In the previous chapter, DRI Service architecture was described. In the following, we present the way how DRI Service went through from design into implementation. We discuss technology limitations and other compromises that had to be taken into account.

4.1. Overview

Implementation of a large software system typically involves some set of technological or paradigmatic assumptions that have to be taken into consideration while implementing its components. VPH-Share Cloud Platform takes advantage of SOA paradigm. All of the components are designed as separate web services and cooperate via REST interfaces. As a result, each component may theoretically be implemented using any technology stack. However, to avoid excessive diversity of software technologies, most of them are implemented in Java or Ruby programming languages, the practically proven open source solutions.

As it was mentioned in section 3.2, all of the Cloud Platform's core services will be deployed within Atomic Service instances, a VPH-Share application container. Atomic Service can be simply viewed as VM with add-on software and mechanism installed, such as security or federated data access layers.

DRI Service implementation was conducted according to the architecture described in chapter 3 with the best software development practices, such as testing and design patterns in mind. The main goal is to achieve the highest possible data validation efficiency, while providing an acceptable probability of unavailability or error detection. The design of DRI Service already solved some performance issues, mostly on the validation algorithm. However, implementation details have to be taken into account.

4.2. Diagram overview and implementation decisions

Ideally, project's implementation should accurately reflect its design. However, selected technology stack significantly affects this desire. One programming languages express object oriented paradigm differently than others, the REST/HTTP communication is handled in other ways or various programming platforms have just developed its own best practices and design patterns. All these dissimilarities can cause changes to the original design. That is the reason, why the architecture presented in chapter 3 represents only the conceptual and functional view of DRI component.

4.3. Implementation technologies

The Java programming language [15] was chosen as implementation language and technology stack, mainly due to its high-performance, wealth of available libraries and productivity as well as being commercially proven in many applications. At its core, DRI Service is a Java Servlet [32] component which accepts REST requests on specified URI paths and performs the requested task utilizing MetadataAccess, ValidationExecutor, ReplicationExecutor, ValidationStrategy and FederatedDataAccess, implemented as simple Java objects. In Java Servlet model, the programmer is free of the object's lifecycle management and REST/HTTP communication complexities, which is provided by the container into which application is deployed.

Another important implementation's aspect is dependency injection (DI). It is a software design pattern which releases the programmer from "dependency-hell" problem. It removes the need to provide dependencies (object instances) when constructing objects, which is error-prone. Dependencies are provided dynamically by the DI container at runtime according to the configuration. In DRI, we use Guice library [6] for DI capabilities. DI approach significantly simplifies component's testing in a service-based environment as dependable service can be simply swapped with a mock object in the configuration.

4.3.1. REST interfaces

To provide REST interface and cooperate with other Cloud Platform components, DRI utilizes Jersey library [8] – a reference implementation of the JAX-RS specification [25] and supports seamless integration with Java Servlet technology. It provides both, server and client REST interoperability via Java annotations. In the server case, the selected method is annotated to respond to the chosen HTTP request type (GET, POST, DELETE, HEAD, ...) on specified URI path, producing or consuming various parameters mostly in JSON format. In the client case, it only requires to specify REST resource's URI path as well as input and output parameter types. As a result, REST interoperability is simplified to the minimum.

OBRAZEK DRI JAKO SERVLET!!!!

4.3.2. Cloud storage access

Current cloud storage services mostly provide a standard REST interface. Despite interface similarities, it appears cumbersome to support the differences between providers. To get rid of this problem, DRI uses JClouds [7] library that provides a common API layer that abstracts cloud dissimilarities. Thus, access to the cloud storage federation is quite easy via programmer perspective. At the time of writing this thesis, JClouds supports up to 30 different cloud providers including Amazon, GoGrid, vCloud, Openstack, Azure and others. Storage access is provided as Blobstore API, which incorporates three concepts: service, container and blob. The Blobstore is a key-value store such as Amazon S3, where your account exists and where you can create containers. A container is a namespace for you data and many of them can exist. Blob is an unstructued data stored in a container referenced by its name. In all cloud storages, the combination of the account, container and blob relates directly to the HTTP URL. Access to data can be performed synchronously or asynchronously, depending on the selected Blobstore type. While Blobstore API provides cloud storage abstraction it cannot overcome specific cloud provider's limitations, for example size limits or timeouts between sensitive operations.

OBRAZEK FederatedDataAccess z wykorzystaniem JClouds!!!!

4.3.3. Request and periodic task scheduling

DRI Service periodically monitors data integrity. Periodic tasks invocation is a recurring issue in many IT systems. DRI uses Quartz library [10] for task scheduling. Quartz is a full-featured open source job scheduling service that can be integrated with, or used along side virtually any Java application – from the smallest to the largest e-commerce system. Its design is scalable, as it can be used to create simple and complex schedules for executing tens, hundreds or even ten-of-thousands of jobs. DRI Service uses Quartz in the following way: it firstly schedules one periodic job within specified period, which upon execution retrieves all managed datasets' metadata from AIR registry and schedules that many single-execution jobs – one for each dataset. Apart from periodic validation, the dataset's integrity check can be performed on request. In such case, DRI Service tries to add validation job for the specified dataset to the schedule. If a job with the same dataset id already exists in the queue, nothing happens. Otherwise, the job with specified dataset id is added to the schedule. Worth mentioning is the fact, that apart from validation jobs, there are jobs that update dataset checksums whenever its contents changed and both of them cannot collide with each other.

OBRAZEK OGÓLNY, PREZENTUJĄCY DZIAŁANIE SCHEDULERA!!!!

4.4. Validation heuristic implementation

DRI Service utilizes efficient data validation algorithm to achieve acceptable performance over large amount of data. However, apart from algorithm efficiency, its optimized implementation is greatly desirable. Due to the fact that the validation algorithm is highly oriented on network communication (see section 3.6), network bandwidth and latency are the most significant factors affecting its performance. The point of the biggest interest is access to large number of the selected chunks of data. As it was noted in section 2.2, current cloud storage interfaces do not enable efficient way to perform this operation. However, even though individual chunks of data have to be requested in separate HTTP calls, they can be invoked asynchronously in parallel to reduce round-trip time (RTT) latency. DRI employs this scheme via asynchronous Blobstore API provided by JClouds library. When DRI validates single logical data within dataset, it invokes a configurable number of asychronous data chunks requests and then waits for their completion. The scheme repeats until all the needed chunks for logical data are collected.

OBRAZEK ASYNCHRONOUS CALLS TO CLOUD PROVIDER!!!!

4.5. Deployment environment

Currently, at proof of concept stage, the DRI is deployed on Apache Tomcat [3] instance which runs on virtual machine (VM). However, in full-operational Cloud Platform it will be deployed within Atomic Service instance (simply a VM with some add-ons) as one of its core services. Apache Tomcat is a web application container which implements Java Servlet specification and provides its application environment. Nevertheless, any other application server compliant with Java Servlet specification can be used.

4.6. The use outside of Cloud Platform

4.7. Conclusions

5. Verification and testing

5.1. Functional requirements verification

A key aspect of the DRI service is to meet its requirements within Cloud Platform, which were listed in section 3.4. In the prototype phase, we are focusing mostly on data validation. As project continues, data replication mechanisms will be developed within DRI service. As previous chapters shown, we designed and implemented a service which enables periodic and on-request data validation and notifies about any integrity or availability errors. It can be used according to the REST interface described in chapter 3.

In future Cloud Platform releases Notification Service depicted in figure 3.5 will provide notification functionalities for the platform. In our prototype, we developed simple mock for this important component in the form of webpage. Its sample view during normal operation is presented in figure 5.1.

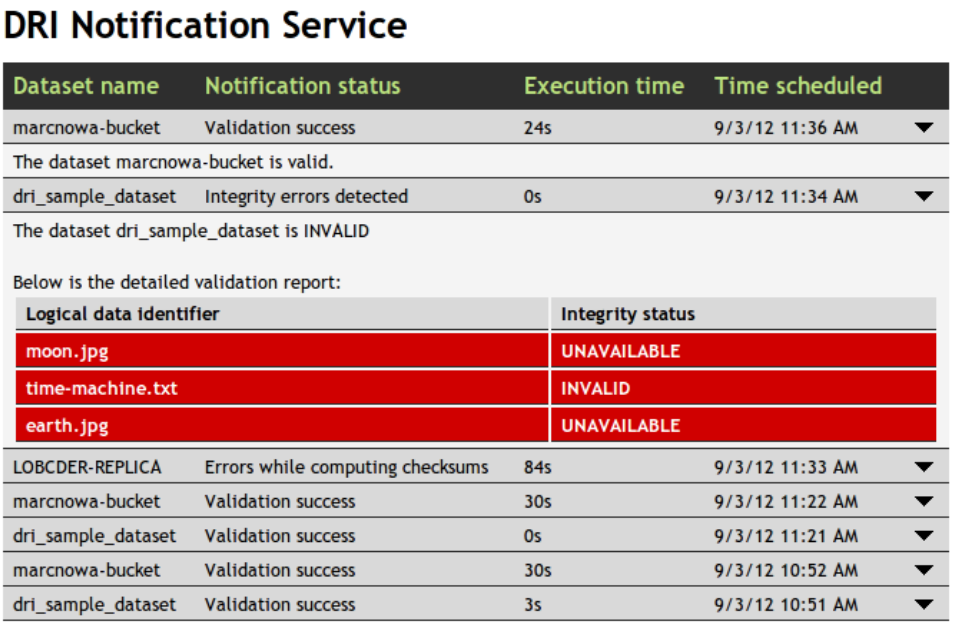


Figure 5.1: Notification service mock – sample view

Here we can see a tabular view of the notifications. Each notification initially is presented as one row. Each row shows basic information: dataset which

5.2. Efficiency metrics

5.2.1. Network bandwidth

5.2.2. Error detection rate

5.2.3. Scalability

5.3. Efficiency tests

5.3.1. Environment description

5.3.2. Experiments

5.3.3. Results

5.4. Other nonfunctional requirements verification

5.5. Conclusions and results

6. Conclusions and future work

6.1. DRI tool evaluation

6.2. Summary

6.3. Future work

Bibliography

[1] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[2] Apache Libcloud. http://libcloud.apache.org/.

[3] Apache Tomcat. http://tomcat.apache.org/.

[4] Eucalyptus Cloud. http://www.eucalyptus.com/eucalyptus-cloud.

[5] Google Cloud Storage. https://developers.google.com/storage/.

[6] Google Guice. http://code.google.com/p/google-guice/.

[7] JClouds. http://www.jclouds.org/.

[8] Jersey. http://jersey.java.net/.

[9] Openstack Swift. http://www.openstack.org/software/openstack-storage/.

[10] Quartz. http://quartz-scheduler.org/.

[11] Rackspace Cloud Files. http://www.rackspace.com/cloud/cloud_hosting_products/files/.

[12] The keyed-hash message authentication code (HMAC). Technical Report 198–1, NIST, 2008.

[13] Recommendation for applications using approved hash algorithms. Technical Report 800–107, NIST, 2009.

[14] Secure Hash Standard (SHS). Technical Report 180–4, NIST, 2012.

[15] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Prentice Hall, 2005.

[16] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of CCS '07*, pages 598–609, 2007.

[17] G. Ateniese, R. Di Pietro, L. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of SecureComm '08*, pages 1–10, 2008.

[18] D. Bermbach, M. Klems, S. Tai, and M. Menzel. MetaStorage: a federated cloud storage system to manage consistency-latency tradeoffs. In *IEEE CLOUD*, pages 452–459, 2011.

[19] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *Architecture*, pages 31–45, 2009.

[20] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Algorithmica*, pages 90–99, 1995.

[21] K. Bowers and A. Oprea A. Juels. Proofs of Retrievability: Theory and Implementation. *ACM CCSW*, 2009.

[22] K. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. *Work*, 489:187–198, 2009.

[23] D. Clarke, G. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE Symposium on Security and Privacy*, pages 139–153, 2005.

[24] Y. Deswarte, Q. Jean-Jacques, and A. Saïdane. Remote integrity checking. In *Integrity and Internal Control in Information Systems VI*, volume 140, pages 1–11. 2004.

[25] M. Hadley and P. Sandoz. JAX-RS: Java API for RESTful Web Services. Technical report, Sun Microsystems, 2008.

[26] M. Tim Jones. Anatomy of a cloud storage infrastructure. `http://www.ibm.com/developerworks/cloud/library/cl-cloudstorage/`.

[27] A. Juels and B. Kaliski. PORs: Proofs of Retrievability for Large Files. *ACM CCS*, pages 584–597, 2007.

[28] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: keyed-hashing for message authentication. http://www.ietf.org/rfc/rfc2104.txt, 1997.

[29] S. Kumar and A. Saxena. Data integrity proofs in cloud storage. *COMNETS*, 2011.

[30] T. Kurze, M. Klems, D. Bermbach, A. Lenk, S. Tai, and M. Kunze. Cloud federation. *Computing*, (c):32–38, 2011.

[31] X. Lai, R. Rueppel, and J. Woollven. A fast cryptographic checksum algorithm based on stream ciphers. In *Advances in Cryptology - ASIACRYPT '92*, pages 339–348, 1992.

[32] R. Mordani. Java Servlet specification. Technical report, Sun Microsystems, 2009.

[33] M. Naor and G. Rothblum. The complexity of online memory checking. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 573–584, 2005.

[34] H. Shacham and B. Waters. Compact proofs of retrievability. In *Proceedings of Asiacrypt 2008*, pages 90–107, 2008.

[35] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. *Electrical Engineering*, pages 19–29, 2010.

[36] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: a scalable cloud file system with efficient integrity checks. *IACR Cryptology ePrint Archive*, page 585, 2011.

[37] D. Stinson. *Cryptography: Theory and Practice*. Chapman and Hall/CRC, 2005.