LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D2.2
Title of Deliverable:	Initial prototype of the key-value store
Contractual Date of Delivery:	M12 – 9/30/2013
Actual Date of Delivery:	9/30/2013

Abstract

This deliverable presents the specification and the initial implementation of the distributed storage layer at core of the LEADS platform. This layer consists in a key-value store with extended capabilities to be deployed in a federation of micro-clouds. We first list the requirements of the key-value store with relation to the big data services we aim at implementing in LEADS. Further, we detail a list of features that implement these requirements, and describe the state of our prototype implementation. For the features currently in-progress, we review the research challenges that we have to address in order to fulfil their implementations. As required in the DoW of the LEADS project, our current prototype is fully functional in the case of a single micro-cloud at M12.

List of Contributors

Name	Organization	E-mail
Xiao Bai	BM-Y!	xbai@yahoo-inc.com
Matthieu Morel	BM-Y!	matthieu@yahoo-inc.com
Ata Turk	BM-Y!	ata@yahoo-inc.com
Emmanuel Bernard	Red Hat	ebernard@redhat.com
Jonathan Halliday	Red Hat	jonathan.halliday@redhat.com
Mark Little	Red Hat	mlittle@redhat.com
Mircea Markus	Red Hat	mmarkus@redhat.com
Manik Surtani	Red Hat	msurtani@redhat.com
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Ioakim Perros	TSI	imperros@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	Christof.Fetzer@tu-dresden.de
André Martin	TUD	Andre.Martin@tu-dresden.de
Do Le Quoc	TUD	Do@se.inf.tu-dresden.de
Frezewd Lemma Tena	TUD	Frezewd_Lemma.Tena@mailbox.tu-dresden.de
Lenar Yazdanov	TUD	Lenar.Yazdanov@tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Etienne Rivière	Etienne.Riviere@unine.ch	2013-09-15
Approved by GA Member 1	Mark Little	mlittle@redhat.com	2013-09-30
Approved by GA Member 2	Minos Garofalakis	minos@softnet.tuc.gr	2013-09-25

Contents

List of Figures

GLOSSARY

EU	European Union
FP7	Seventh Framework Programme
Micro-cloud	A cluster of machines with virtualization capabilities
Node	A machine inside a micro-cloud
VM	Virtual Machine
IaaS	Infrastructure-as-a-service
DaaS	Data-as-a-service
KVS	Key-Value Store
FUSE	Filesystem in User Space
DoW	Declaration of Work

1. Executive Summary

Most of the knowledge that can be derived from public data sets is only available to a handful of Internet-scale corporations. Such knowledge is however at the heart of promising business models for companies that have the capacity to store and analyze large amount of data. The objective of LEADS is to investigate a novel approach that facilitates data-as-a-service (DaaS) such that the real-time processing of large amount of public data becomes economically and technically feasible even for small and medium enterprises (SMEs). More precisely, the approach followed in LEADS consists in aggregating into a federation of micro-clouds the infrastructures of multiple SMEs. This aggregation increases the amount of data storage and computational power available to a level where an SME has enough sufficient resources to process public data.

The approach followed in the LEADS project raises several challenges. In particular, the LEADS platform relies on an efficient distributed storage system. Each SME brings into the federation a few servers with virtualization capabilities – named a *micro-cloud*. The LEADS distributed storage layer sits on top of this set of micro-clouds. Designing the storage layer is addressed in WP2. The core difficulties in designing such a layer are (i) the dependability of the whole infrastructure in face of faults at both scales - inside and between the micro-clouds, (ii) the necessity of leveraging the two-level infrastructure of LEADS for performance, and (iii) the ability to deal with big data extracted from public sources – typically the Internet.

In this deliverable, we detail the specification and the initial implementation of the storage layer at core of the LEADS platform. This layer consists in a key-value store with extended capabilities to be deployed in a federation of micro-clouds. We first list the requirements of the key-value store with relation to the big data services we aim at implementing in LEADS. Further, we detail a list of features that implement these requirements, and describe the state of our prototype implementation. Then, this deliverable reviews the research challenges that we have to address in order to fulfil the implementation of features still in -progress. As required in the DoW of the LEADS project, our current prototype is fully functional in the case of a single micro-cloud at M12.

2. Introduction

This document describes a specification of the storage layer that supports the LEADS service model, and its initial implementation at M12. Following the DoW of the project, we apply an iterative development of the functionalities of the storage layer. We start from a functional storage on a single micro-cloud having a modular design and we are currently adding micro-cloud federation capabilities to it. This document defines precisely the features that are expected to be supported by the final implementation, and lists the research efforts related to them.

The remainder of this document is organized as follows: Section 2 introduces the architecture targeted in LEADS, i.e., a federation of micro-clouds, and lists the requirements of the storage layer. In Section 3, we describe a high-level implementation of these requirements. Our implementation is defined in terms of *features;* a group of features implements a *requirement*. In Section 4, we detail the features currently implemented in our initial prototype, as well as, the features which integration is currently in-progress. For each in-progress feature, we introduce a list of related research opportunities. Section 5 closes this document with a brief summary of the state of our prototype as well as the directions we are taking to enhance it.

3. High-level requirements of the storage layer

In this section, we present an overview of the functional and non-functional requirements for the LEADS storage layer. These requirements were previously detailed for internal use in the project in working document WD2.1, and form the input of the storage layer specification that we describe in Section 4.

3.1 Targeted architecture and associated failure model

The LEADS platform is a collection of micro data-centers or *micro-clouds* (Figure 1). Each micro-cloud consists of a dozen of servers with virtualization capabilities connected to a local-area network. Connections to clients of the platform, to other micro-clouds and to clouds in the public space take place through a wide-area network. The availability of optical fibers makes such links efficient. Nevertheless, there is at least one order of magnitude between the link performance (in term of bandwidth and message delay) outside and inside a micro-cloud.

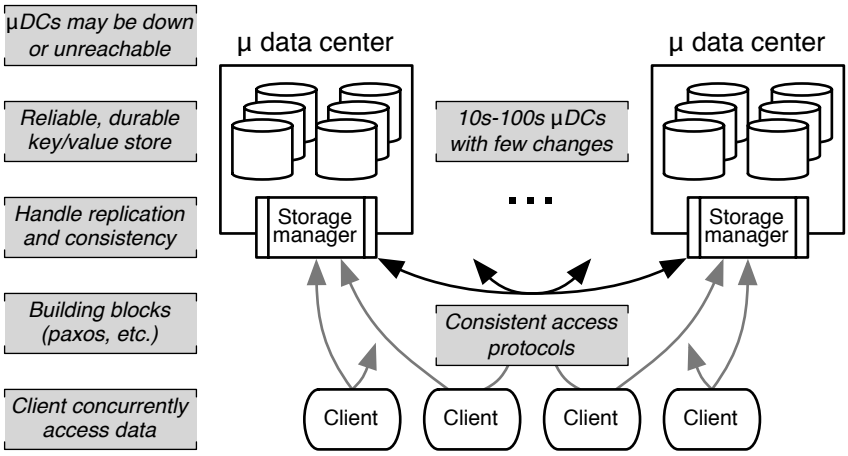

Figure 1: High-level architecture of the LEADS storage layer.

The above architecture targeted in LEADS requires considering a *two-level failure model* where both a server inside a micro-cloud and a micro-cloud as a whole can fail. Indeed, while the simultaneous failure of all servers inside a micro-cloud is unlikely, each micro-cloud is typically connected through a single link to other micro-clouds, to the clients, and to public data sources. The failure of this unique link results in the apparent failure of the whole micro-cloud for externally connected components. Other external events such as natural disasters or power outages can also disrupt the whole micro-cloud.

As a consequence of the above failure model, we must consider a *decentralized architecture*. In other words, the use of a centralized, omniscient, and/or permanently connected component that would run in one of the micro-cloud is prohibited. If we need a component that orchestrates the storage layer and the placement of data, this component must be virtualized using fully decentralized and dependable mechanisms. Inside each micro-cloud, dependability mechanisms will ensure that the services provided to the clients and to other micro-clouds are dependable and highly available. At the higher level, additional mechanisms will treat each micro-cloud as a single, fallible entity of the system.

Elasticity refers to the ability to add or remove machines to the system. In the LEADS storage layer, elasticity should take two forms: adding or removing a machine to a micro-cloud, and adding or removing a micro-cloud from the system. It should be noted that when the configuration of the storage layer change, the load must be automatically balanced without interfering with on-going computations.

Due to the performance gap between local and wide-area network connections, *access locality* is paramount to the efficiency (in particular, energy-efficiency) of the system. At local scale, since a micro-cloud is small, and internal connections are fast, we employ traditional techniques. At the federation scale, the system must support appropriate data placement and scheduling policies to leverage locality. These two aspects are mainly targeted in WP4; the storage layer should simply offer appropriate *information reporting* to allow implementing them.

3.2 Data-as-a-Service model and impact on the storage layer

The service model of the LEADS DaaS (data-as-a-service) platform combines two forms of data: public data extracted from various sources, e.g., from the Web, and private data created by the users of the service. Accesses to both data types must be simple, and the API should correspond to well-established abstractions for manipulating data. Below, we further detail the two types of data, and how they impact the storage layer.

Public data is immutable. It is stored and made available in multiple versions. Each version of a data items reflects the time at which it was created in the storage layer. The storage of public data is not encrypted. If applicable, it can be compressed and uncompressed on the fly. Public data is typically structured - for instance it can form a graph. Navigation through this data may benefit from indexing mechanisms and access methods that expose this structure. However, the primary interface proposed to access data is flat, that is the storage system remains oblivious of the nature and the links between data elements. Since public data is immutable, it should be replicated as needed inside a micro-cloud and across multiple micro-clouds (a minimum number of replicas can be necessary to ensure durability of the data in the presence of micro-clouds faults).

Private data is mutable and its privacy must be preserved through appropriate *encryption mechanisms*. Users and applications may also require that a private data element is stored in a particular region, or outside a particular region (e.g., for legal reasons). Since private data is mutable, it must be replicated on multiple machines to guarantee dependability. Furthermore, accesses must be possible on any copy and, therefore, consistency must be enforced accordingly. Multiple consistency level are possible. A high consistency level typically requires expensive synchronization and communication between nodes. The need for strong (or weak) consistency depends on the data and on the application. The support for several, co-existing consistency models, permits a wider range of applications to operate on top of the storage layer. In addition, it reduces the cost and overall energy usage of the platform.

While the performance of the storage layer for accessing private data is important, it is even more for public data. Indeed, massive computations on public data are expected to extract meaningful information, and are thus likely to be accessed by many clients simultaneously. In contrast, private data sets are expected to be smaller and should experience less traffic. Therefore, functionalities can be considered as more important than raw performance in this last case.

The LEADS platform features querying and extraction mechanisms, defined in WP3. Queries can be persistent and operate on the data items obtained through the extraction service (WP1). The same (or similar) queries may be sent several times in the system and therefore operate on similar data sets. The storage layer supports these queries by allowing the creation of *collections*. *A collection* is defined on a subset of the data and it enables a fast and direct access to the items that correspond to a given criteria, or for which the access is not done primarily by the name of the item but according to another criteria. Collections are specific to a given query and thus to a given client application. As a consequence, they represent sensitive data: an attacker getting access to the unencrypted version of a collection may be able to determine the common aspect of the documents indexed, and thus, part of the corresponding queries. This calls for appropriate privacy-preserving encrypted collection.

3.3 Summary

Below, we summarize the high-level requirements of the LEADS storage layer.

(R1) Data access

A LEADS client accesses data through simple and well-known interfaces.

(R2) Dependability

The LEADS storage layer operates despite the failure of multiple nodes, or the failure of a whole micro-cloud.

(R3) Elasticity

The storage layer supports the ability to add and remove nodes inside a micro-cloud, and to add and remove micro-clouds.

(R4) Consistency

Public data is immutable and strongly consistent. The leads storage layer supports different tenable consistency levels for mutable private data, including strong and weak consistency.

(R5) Locality and performance

Immutable public data are replicated/cached at will to improve locality and thus performance of accesses. Private data are mutable, and consequently it is cached with care to face the trade-off between consistency and performance.

(R6) Data placement and indexing (with WP1, WP4)

The leads storage layer supports explicit data placement to satisfy administrative/legal requirements and improve locality.

(R7) Versioning

Public and private data are available in several versions. Upon relocation of data, previous versions might be accessed via redirections.

(R8) Scheduling (with WP4)

Query placement algorithms from WP4 decide which micro-cloud should host a query, and require information about data placement and affinity of queries with data coming from the storage layer.

(R9) Encrypted private data

A query typically operates on a subset of the data that is pre-computed upon data extraction, or generated by other queries. Such a subset is kept in a collection. The content of a collection may be sensitive data in itself. As a consequence, it needs to be encrypted and support for this encryption must be built into the storage layer.

4. Detailed specification of the storage layer

We now specify a high-level implementation of the requirements listed in the previous section. Our implementation is defined in terms of *features*. Features are grouped into *functionalities*, which refine the requirements we listed previously in Section 3. Table 1 – Decomposition in features of the LEADS storage layerbelow sums-up our specification.

The reminder of this section gives a detailed explanation of the functionalities and the features the LEADS storage layer should provide. For each functionality, we first recall its context and motivation. Then, we define how the storage layer implements that functionality in terms of API and expected functional behaviour. Due to the two-level structure of the LEADS infrastructure, a feature implementing functionality might be needed at the level of a micro-cloud, at the federation level, or at both levels.

Features	one micro-cloud	federation
Key/value store API	F2.11	F2.12
Listener API	F2.13	F2.14
Fault-tolerance mechanisms	F2.21	F2.22
Elasticity	F2.23	F2.24
Consistent hashing based data placement	F2.31	
Explicit and constrained data placement		F2.32
Data location retrieval		F2.33
Total order based replication	F2.41	
Primary-replica based replication		F2.42
Eventual consistency based replication	F2.43	F2.44
Heat maps support		F2.51
Automatic caching of public data		F2.52
Versioning	F2.61	F2.62
Redirection support		F2.63
Factory of atomic objects	F2.71	
FUSE-based file system interface		F2.81
Encrypted collections		F2.91
Support for deployment and configuration	F2.101	F2.102

Table 1 – Decomposition in features of the LEADS storage layer requirements

4.1 Key-value store API

Motivation. Traditional, highly normalised data has been proven not to scale in a large-scale distributed environment. Common solutions involve storing denormalized data - either as key/value pairs of objects, or as documents. This has been the cornerstone of modern, highly scalable distributed storage systems collectively known as NoSQL [NoSQL]. Following this approach, the storage layer should privilege flat data structures with simple access patterns over complex ones (e.g., the relational model). The canonical example of data denormalization is a key-value store API. We define this interface as *the main interface to the LEADS storage layer*.

Functional Description. The LEADS storage layer supports a key-value store API. This means that a client can *create* and *remove* a *key space*, and can *put* and *get* a key with an associated value into a particular key space. This API is available at the scale of a micro-cloud, and at the scale of the federation of micro-clouds. Two parameters given at the key space creation time, indicates the scale at which the key space is created, and whether it contains private or public data. In addition, the storage layer supports a *listener* pattern via a client registration mechanism that takes a key space as argument. Once registered, a client receives notifications for the events occurring on the registered keys (creation, modification and deletion).

Features.

- [F2.11] Key/value store API (one micro-cloud)
- [F2.12] Key/value store API (federation)
- [F2.13] Listener API (one micro-cloud)
- [F2.14] Listener API (federation)

4.2 Dependability and elasticity mechanisms

Motivation. Existing Cloud storages, such as *Cassandra* [LM10], can span multiple large data centers. Geo-replication mechanisms allow such systems to have a very high degree of dependability. Similarly, the LEADS infrastructure is a federation of micro-clouds, and thus at core a two-level architecture. This architecture must be resilient to failures at both scales. This implies that (i) each micro-cloud is a reliable machine, through internal dependability mechanisms, and (ii) at the highest level, the federation of micro-clouds is also a dependable system that handles transparently the failure of a micro-cloud.

Functional Description. At the level of a micro-cloud, the storage layer implements failure detection mechanisms and supports the *addition* and *removal* of nodes (elasticity). At the level of the micro-clouds federation, the addition (or removal) of a micro-cloud is not a frequent event. Still, it must be supported without service interruption. Due to the performance gap between local and remote accesses, we note here that migrating the load to a novel micro-cloud is an expensive process. This raises the observation that re-configurations must occur *only when necessary*. To satisfy this constraint, the membership information at the micro-clouds federation level is *strongly consistent,* and the failure detection mechanism is slow but *accurate*. Both features are implemented on top of a geographically distributed instance of the Zookeeper data sharing service [Zk]. We note here that Zookeeper has not been designed for large-scale data replication; in particular, all accesses are orchestrated through a (rotating) leader.

Features.

- [F2.21] Fault-tolerance mechanisms (one micro-cloud)
- [F2.22] Fault-tolerance mechanisms (federation)
- [F2.23] Elasticity (one micro-cloud)
- [F2.24] Elasticity (federation)

4.3 Data placement, indexing and locality (interactions with WP4)

Motivation. Each data item is associated with a name inside a naming space - for instance, a hierarchical naming space for a file system, or a key space for a key-value store. The ability to locate data in this space is the responsibility of an *indexing service* implemented at the storage layer level. Another important aspect of data management is locality. Data locality is the likelihood of a set of data items

to be accessed together. In the context of LEADS, affinity mechanisms from WP4 enforce data placement to favour data locality. Moreover, a LEADS user may constrain data placement, e.g., for legal reasons. These last observations motivate that the LEADS storage layer needs to support an explicit indexing service at the micro-clouds federation level.

Functional Description. The LEADS storage layer is able to explicitly place data on the micro-clouds. Once selected for storing a data item, a micro-cloud uses an arbitrary internal mechanism for balancing the storage load between its internal servers. Direct placement is implemented via a (globally replicated) *explicit data index*. This index maps key spaces to the micro-clouds holding a copy of the space. A key space might be replicated over multiple micro-clouds to ensure data availability and to improve performance. In order to avoid incorrect data migration from one cloud to another, the explicit index is strongly consistent. Each micro-cloud might also cache a local copy of the index to improve read performance. The storage layer is able to return the location(s) of a data item, and to receive data placement requests from mechanisms implemented in WP4. Requests for placement can be explicit, or can be based on constraint, e.g., a key space shall not be placed on a micro-cloud that is in a given geographical region.

Features.

- [F2.31] Consistent hashing based data placement (one micro-cloud)
- [F2.32] Explicit and constrained data placement (federation)
- [F2.33] Data location retrieval (federation)

4.4 Replication and consistency of private data

Motivation. Replication is a fundamental technique to improve data availability and performance. Nevertheless, dealing with updates on replicated data requires a considerable amount of synchronisation among the replicas, in order to keep them coherent. Relaxed consistency models (e.g., eventual consistency [EC]) require developers to work with less consistent data and are generally harder to use, but they present much better performance than strictly consistent ones. This is the inherent trade-off between replication, consistency, and performance [CAP]. Finding an appropriate level of replication and consistency depends on the access patterns envisioned and the application invariants. Classical solutions range between (1) increasing the synchronisation among the replicas in order to semantically enforce the same update order in all of them, at the expense of adversely impacting the system's performance, and (2) reducing the guarantees offered to the applications with relaxed data consistency models. These solutions are vastly developed in the literature and the best trade-offs are usually dependent on the applications. In general, strong consistency (aka. linearizability [LIN]) should be ensured for mutable data. Nevertheless, as application programmers have the ability to gauge their application's tolerance to weaker consistencies, the storage should allow them to specify the consistency levels required for each data set. Following most cloud storage systems currently available, the LEADS storage layer should offer strong and eventual consistency.

Functional Description. The consistency level of a key space (put and get operations) is defined at creation time. This consistency level can be of two types: weak and strong. At the level of a single micro-cloud, strong consistency is supported using a total order group communication primitive. At the level of a federation of micro-clouds, the technique employed is primary-backup. Eventual consistency at the level of a micro-cloud is implemented by using vector clocks and an anti-entropy mechanism [Dyn]. At the federation level, a lazy reconciliation scheme in the storage management layer is used.

Features.

- [F2.41] Total order based replication (one micro-cloud)
- [F2.42] Primary-backup based replication (federation)
- [F2.43] Eventual consistency with vector clocks (one micro-cloud)
- [F2.44] Eventual consistency with lazy replication (federation)

4.5 Caching public data

Motivation. The LEADS storage layer processes both public and private data. As explained in the previous section, each type of data has a different best replication strategy. In the case of public data, there is no need to synchronize the replicas since public data is *immutable*. Consequently, replication in that case should only be driven by the access pattern to create as many local caches as necessary. Non-discriminated growth of data replication should not pose a problem.

Functional Description. The data storage implements an automatic caching mechanism for public data. This caching mechanism is based on *heat map* of public data at the federation level, and its granularity is a key space.

Features.

- [F2.51] Heat maps support (federation)
- [F2.52] Automatic caching of public data (federation)

4.6 Versioning

Motivation. Some queries will need to compare previous and new version of data, for instance to estimate the evolution over time of public data. For private data, query results can be compared with previous results from the same query to detect trends in these results.

Functional Description. The LEADS storage layer is able to store not only the latest version of each data item, but also a certain number of versions in the past. The amount of stored versions w.r.t. time follows a Zipf law. Out-dated versions are garbage-collected via an appropriate mechanism. When a key space is migrated from one micro-cloud to another, the system chooses between completely updating the shared index, or partially by making versions prior to the migration accessible via multiple hops.

Features.

- [F2.61] Versioning (one micro-cloud)
- [F2.62] Versioning (federation)
- [F2.63]Redirection support (federation)

4.7 Rich shared data structures

Motivation. Simple shared data structures, such as a key-value store are inefficient to implement certain cloud applications. For instance, porting a legacy object-oriented application on a key-value store requires a complex re-engineering process. To offer an alternative and stronger application support, the LEADS storage layer offers a rich shared object programming layer. This programming layer allows an application to construct a shared object of its choice[1] on top of the key-value store

[1] Provided the class is marshallable.

API. In particular, these constructs can be used by WP3 mechanisms for implementing complex queries.

Functional Description. The rich data structure layer consists in a *factory of objects*. Each object is strongly-consistent. It is built in a non-intrusive manner on top of the key-value store API. At the creation time of the factory, the client defines the class of objects being created by the factory. Each shared object created by the factory is named after its corresponding storage key in the key-value store. A shared object is *elastic* and it both can scale-out and scale-up; this means that multiple clients located on the same node, or on different nodes, can transparently *share* the object. Furthermore, an object can be stored durably by the application and later retrieved.

Features.

- [F2.71] Factory of atomic objects(one micro-cloud)
- [F2.71] Factory of atomic objects(federation)

4.8 File-storage Interface

Motivation. The access to the DaaS service must be simple for clients. The API must correspond to well-established abstractions for manipulating data. A file system interface corresponds to these requirements. It allows re-using existing data management systems and simplifies the integration of client applications.

Functional Description. The LEADS storage layer includes a FUSE-based file system interface, in addition to lower-lever client interfaces and protocols. As previously with the strongly consistent object factory, this file system is implemented in a non-intrusive manner on top of the key-value store.

Features.

- [F2.81] FUSE-based file system interface (federation)

4.9 Encrypted collections of private data

Motivation. WP3 implements queries operate on a subset of public data, as well as on previously generated private data. Typically, a query makes use of one or more criteria to execute a computation. For instance, it may select all the data items for which an attribute has a particular value. Going through the entire data set on all micro-clouds for locating the data items that correspond to the criteria is impractical. Moreover, a purely Map/Reduce-type approach [MR] would have a high cost compared to the amount of data that is actually used. A sounder approach is to pre-populate, during the extraction of data, or when data is generated from other queries, a specific *collection* that contains the data items of (potential) interest to the query. This collection can be a map indexing the items according to a query-specific attribute, a set of attributes to compute statistics on top of them, or a simple array structure. Such collections of private data represent a threat to query privacy. For instance, they may list the data items that correspond to the query output, and accessing this information in clear, or computing the similarities between them, may allow an attacker to determine the whole query, or the criteria that were used in the first place.[2] As a consequence, only the initiator of the query must be able to access the information contained in these collections.

[2] Notice here that this is distinct to the fact that data are themselves encrypted – a concern orthogonal to the storage layer.

Functional Description. The LEADS data storage layer would support the generation and maintenance of encrypted collections and allow navigating through the items listed in these indexes. These collections shall be provided in cooperation with WP3 encrypted querying mechanisms.

Features.

- [F2.91] Encrypted collections of private data (federation)

4.10 Management

Motivation. Managing a large-scale distributed system such as a federation of micro-clouds is difficult. To facilitate this task, the LEADS storage layer should implement a decentralized management service that can be easily accessed from any entry point of the system. In particular, this service should expose an interface to monitor the system, configure its components and adapt its parameters such as the replication degree or the caching and elasticity policies.

Functional Description. The LEADS storage layer implements a distributed component that monitors and configures on the fly all the aspects of the system. In particular, such a component can drive scripts to deploy the storage layer inside and between micro-clouds instances. Inside a micro-cloud, an instance of the storage layer is encapsulated inside a VM image and is supported by the underlying IaaS (infrastructure-as-a-service).

Features.

- [F2.101] Support for deployment and configuration (one micro-cloud)
- [F2.102] Support for deployment and configuration (federation)

5. State of the implementation

This section depicts the current state of the LEADS storage layer at M12. Following the DoW, we apply an iterative development of the functionalities. We start from a functional storage on a single micro-cloud having a modular design and we will incrementally add micro-cloud federation capabilities on top of this initial prototype.

Table 2 – Implemented and in-progress features in the initial prototypelists the implemented features (in dark grey) and the features whose implementation is in-progress (in light grey). Implemented feature with a star (*) were existing prior to the beginning of the LEADS project. In what follows, we first give an overview of the storage layer. Then, we detail each implemented feature and each in-progress feature. When appropriate, we discuss research opportunities that may be investigated in order to fulfil the implementation of the LEADS storage layer.

Features	one micro-cloud	federation
Key/value store API	F2.11*	F2.12
Listener API	F2.13*	F2.14
Fault-tolerance mechanisms	F2.21*	F2.22
Elasticity	F2.23*	F2.24
Consistent hashing based data placement	F2.31*	
Explicit and constrained data placement		F2.32
Data location retrieval		F2.33
Total order based replication	F2.41*	
Primary-replica based replication		F2.42
Eventual consistency based replication	F2.43	F2.44
Heat maps support		F2.51
Automatic caching of public data		F2.52
Versioning	F2.61	F2.62
Redirection support		F2.63
Factory of atomic objects	F2.71	F2.72
FUSE-based file system interface		F2.81
Encrypted collections of private data		F2.91
Support for deployment and configuration	F2.101	F2.102

Table 2 – Implemented and in-progress features in the initial prototype

5.1 Overview

At the scale of a micro-cloud, the LEADS storage layer consists in Infinispan, an open source key-value store [Ispn, Ispn-doc]. At the federation level, it consists in Infinispan-ensemble, an aggregation of geographically-distributed Infinispan instances. RedHat is the main developer of Infinispan. The other LEADS partners (BM, TSI, TUD and UniNE) contribute to Infinispan in collaboration with Red Hat by implementing specific features of the storage layer. The core of Infinispan is a specialised shared data structure, tuned to and geared for a great degree of concurrency — especially on multi-CPU and multi-core architectures. Most of the internals of Infinispan are essentially lock- and synchronization-free, favouring state-of-the-art non-blocking algorithms and techniques wherever possible.

5.2 Implemented Features

Below, we detail the implemented features and the in-progress features of the LEADS storage layer both at the level of a micro-cloud (Infinispan) and at the level of the federation of micro-clouds (Infinispan-ensemble).

F2.11* Key-value Store API (one micro-cloud)

The key-value store API of Infinispan is a cache interface that extends java.util.Map [Ispn-API]. This data structure is highly concurrent, designed ground-up to take advantage of the characteristics of modern architectures while at the same time providing distributed sharing capabilities. A cache object is backed by a peer-to-peer network architecture to distribute the state efficiently in a data centre. In addition to the peer-to-peer architecture of Infinispan, a client/server mode is also supported via the HotRod protocol [HotRod]. HotRod provides the ability to run farms of Infinispan instances as servers and connect to them using a plethora of clients — both written in Java as well as other popular open source and proprietary platforms.

F2.13* Listener API (one micro-cloud)

Streams are a common distributed programming pattern. For instance, one of the usages envisioned in LEADS consists in a set of workers aggregating the output of a distributed crawling engine [D1.2] to compute the PageRank score of webpages of interest [D3.2]. To support stream-oriented architectures and in particular LEADS requirements, Infinispan employs a listener API: once a client is registered on a cache, it gets notified of every operations occurring on the data stored in the cache. Listeners allow implementing communication between LEADS components as well as supporting the instantaneous/persistent query model described in our use cases.

F2.22* Fault-tolerance mechanisms (one micro-cloud)

Infinispan offers high availability via making replicas of state across a network as well as optionally persisting state to configurable cache stores. In more details, Infinispan is built upon the JGoups communication layer [JGroups]. This layer allows a set of servers to construct a reliable membership service, by employing failure detection, appropriate group communication primitives and discovery services.

F2.23* Elasticity (one micro-cloud)

& F2.31* Consistent hashing-based data placement (one micro-cloud)

Data placement and retrieval in Infinispan employs consistent hashing [ConsHash]. The hash function it employs is the MurmurHash algorithm, a non-cryptographic hash function suitable for general hash-based lookup [Murmur]. Consistent hashing is a familiar technique in distributed data storage systems that provides both elasticity (the ability to add and remove nodes) at low-cost, and load balancing. Infinispan also support round-robin data placement.

F2.41* Total order-based replication(one micro-cloud)

Consistency guarantees on the put/get operations in Infinispan are modular. This interesting property comes from the fact that all the communications between servers are based on the JGroups li-

brary. In particular, when a cache is set at the transactional level, operations are atomic.[3] Below, we explain how we used this feature to implement a factory of atomic objects.

F2.71 Factory of atomic objects (one micro-cloud)

The factory of atomic objects allows a client application to build an atomic object of its choice on top of Infinispan. This object is be persistent and elastic (both vertically, by increasing the number of local threads, and horizontally, by increasing the number of distributed clients accessing it).

We built the factory on top of the total-order facility of Infinispan. Our approach is a variation of the well-known state machine replication technique [SMR] already at work in multiple distributed systems. In more details, when the object is created, we store both a local copy and a proxy registered as a cache listener. We serialize every call in a transaction consisting of a single put operation. When the call is de-serialized it is applied to the local copy and, in case the calling process was local, the return value is returned (this mechanism is implemented via a Future object).

F2.101 Support for deployment and configuration (one micro-cloud)

When one starts thinking about running a distributed storage system on several dozens of servers, as in our typical micro-cloud architecture, management is not an extra but a necessity. Infinispan provides rich tooling in this area, with many integration opportunities. For instance, it is integrated with JBossAS [Jboss], an application server that implements the Java Platform Enterprise Edition (java JEE).

We have also built for testing purposes a standalone distribution of Infinispan that runs on top of the (IaaS) OpenNebula cloud environment [OpenNebula, Leads-dep]. Our set of scripts, disk images and OpenNebula template are available on-line, and are already used by consortium partners for integration purposes. This standalone version runs a tailored Gentoo Linux distribution. The core of the distribution is very modular. In addition to Infinispan, it can deploy multiple LEADS services such as our distributed crawler prototype, or the Zookeeper synchronization service. A LEADS service is defined by an OpenNebula template that consists of two disk images. The first image contains the core of the Linux system; the second disk image contains the LEADS service. To deploy a service on an OpenNebula platform, the user needs to perform only very few manual steps. For instance, to run Infinispan, the user instantiates the appropriate template on the desired nodes. Once all nodes are started, Infinispan is up and running and a HotRod client can access it. The discovery of the running nodes executing Infinispan is supported by an IP broadcast facility of the JGroups library.

5.3 Features in-progress and research opportunities

In what follows, we describe the Infinispan features that are currently in-progress, as well as several research challenges associated to them.

F2.12 Key/Value Store API (federation)

& F2.14 Listener API (federation)

We are currently implementing a component of Infinispan that encapsulates the logic of the LEADS storage layer and that is suitable at the micro-clouds federation level. In more details, this compo-

[3] In the context of LEADS, we will not use the transactional features of Infinispan as this is not a requirement of the project.

nent, named *Infinispan-ensemble*, exposes the same interface as Infinispan, i.e., a cache object. Infinispan-ensemble is based on the principle of aggregating remote cache objects, which are distant cache objects transparently accessed via the HotRod client-server protocol. Metadata regarding Infinispan-ensemble are stored into an instance of the Zookeeper coordination service [Zk]. The Zookeeper instance spans multiple micro-clouds for both reliability and read performance. The metadata stored in Zookeeper contains the membership information of the Infinispan-ensemble, as well as the information related to data placement (detailed in features F2.32 and F2.33).

In Infinispan-ensemble, each Infinispan instance runs on a micro-cloud independently of the others, thus offering, from the federation point of view, the abstraction of a single entity. Such architecture ensures a separation of concerns between the micro-clouds and the federation itself, allowing distinct appropriate mechanisms to operate at each level.

The cache operations of Infinispan-ensemble are implemented in a non-intrusive manner on top of the remote cache objects. For instance, if a client executes a get operation, Infinispan-ensemble either contacts Zookeeper, or uses a local version of the data placement information to redirect the call to one of the micro-clouds replicating the appropriate data. The listener API in Infinispan-ensemble follows the same semantics as in Infinispan. It relies on an on-going evolution of the HotRod protocol that allows a client to register as a listener on a remote cache object.

Research opportunity. The non-intrusive architecture of Infinispan-ensemble raises several questions of interest in regard to data consistency and replication. More precisely, the planned features F2.42 and F2.44 provide respectively an atomic and an eventually consistent put/get interface to the client. A naive approach for implementing this interface can consist in delegating the writes to all the micro-clouds replicating the key to the client that executes the put operation. However, it might not be practical since a client can fail in the middle of such an operation. A more formal treatment of this question consists in solving the problem of constructing shared registers on top of abortable shared registers that model the key-value store interface at the level of a micro-cloud.

F2.22 Fault-tolerance mechanisms (federation)

&F2.23 Elasticity (federation)

The Infinispan-ensemble component is a deterministic automaton whose state consists solely of the information stored into the information of Zookeeper spanning the micro-clouds federation. This automaton allows the addition and removal of micro-clouds through appropriate operations, and keeps track of the association between micro-clouds and the data stored at the scale of the micro-clouds federation (described shortly in feature F2.32).

Infinispan-ensemble is de facto a dependable component. Nevertheless it needs to track the failure of a micro-cloud, in order to avoid an interruption of the provided service. To that end, we plan using in Infinispan-ensemble a failure detection mechanism provided by the JGroups library. It should be noted here that since the process of recovering from a micro-cloud failure is expensive, we should pay the cost of a recovery only when necessary. This implies that we shall use accurate failure detectors at the federation level.

At the level of the federation of micro-clouds, the addition and deletion of a micro-cloud is an infrequent event, still it must be supported without service interruption. To support the elasticity of the micro-cloud federation, Infinispan-ensemble exposes meta-information in its interface such as the load of each micro-cloud and the number of available servers at each distinct geographical location.

We can afford that the balancing of the load to the new micro-cloud is progressive and takes some time: unlike a distributed hash table mechanism where the addition of a new node implies immediately splitting the load of the surrounding nodes, we can accept that the data placement and query placement mechanisms from WP4 gradually fills the new micro-cloud with requests and data. For instance, an under loaded micro-cloud might be used preferentially to support replicas of new data items, or replicas of data items that are likely to be accessed conjunctly.

Similarly to the addition of a micro-cloud, the removal of a micro-cloud requires handing over the responsibility of the data it replicates to other micro-clouds to maintain the desired replication factor. It also requires re-launching the distributed computations that were running on the micro-cloud but that did not committed on other micro-clouds. Notice here that this latter mechanism is under the responsibility of WP4.

Research opportunity. To the best of our knowledge, there are few solutions regarding the problem of failures detection between disjoint groups of nodes as it appears in the context of the micro-clouds federation. Related to failure detection is the problem of handling the elasticity of the federation of micro-clouds. Indeed, as noted previously, responding to a micro-cloud failure incurs lots of processing to reach the nominal state of execution for the micro-cloud federation. Trade-offs between the response time to such an event and the cost of the recovery process can be a challenging direction to investigate.

F2.32 Explicit and constrained data placement (federation)

& F2.33 Data location and retrieval (federation)

As pointed out in Section 4.3, most existing data storage systems use *implicit* indexing based on *hashing* mechanisms. For instance, we described such a mechanism for Infinispan in Section 5.2. With implicit indexing, the keys of data items are *hashed* to a different key space. This indexing mechanism yields to a good repartition between storage nodes, but it loses the ordering and thus the locality of the original key space.

An implicit data placement enforces the placement on some nodes, and does not allow dynamically changing their locations. Consequently, such a mechanism is inadequate for the micro-clouds federation architecture targeted in LEADS. The LEADS storage layer must be aware of the two-level hierarchical nature of the platform, and it should take into account the difference in performance between the lower and the higher layer, both in terms of bandwidth and message delay. Queries implemented in WP3 must access data items that, to the highest extend, should be present on the micro-cloud executing the query. We call this requirement *data-to-query affinity.* Moreover, data items likely to be processed together have to be close geographically; a requirement named *data-to-data affinity.* The services developed in WP4 leverage both data-to-data affinity and data-to-query affinity to improve the performance of the LEADS architecture. To support these services, Infinispan-ensemble, the implementation of the LEADS storage layer at the federation level makes use of an *explicit* data placement and retrieval strategy. We illustrate this mechanism in Figure 2.

We can observe in Figure 2 that the explicit index consists of a mapping from key spaces to the corresponding implementation in a micro-cloud (or micro-clouds, in case of a replication factor greater than one, as will be the general case to ensure data availability in the presence of faults at the federation level). A micro-cloud stores a key space as an Infinispan cache object. The characteristics of the key space at the federation level such as the replication degree and the consistency model must be

satisfied. However there is no need that all the cache objects implement exactly the same policy, as long as the characteristics of the key space are respected. For instance, if a key space requires strong consistency, it can be implemented with an atomic primary cache object, and eventually consistent secondary copies.

We note here that mapping a key space to a micro-cloud and not each key to a micro-cloud is mandatory. Indeed, the amount of data items stored in the LEADS storage layer is expected to be in the order of billions. It is thus impractical to store the explicit correspondence between individual data items and the micro-clouds storing them.

A requirement for the performance of accesses between micro-clouds is that the resolution of a name is performed (with high probability) in a *single hop*. This means that, in a given micro-cloud, a request for a data item whose name does not fall into one of the key space replicated at the micro-cloud, leads to a request sent towards the appropriate micro-cloud. To implement this functionality, Infinispan-ensemble needs to maintain in a strictly consistent manner the associations between the key spaces and micro-clouds. For instance, in Figure 2, the system must maintain the association between the pink key space (key space 3) and the three micro-clouds replicating it. To achieve this, Infinispan-ensemble stores the explicit index inside the Zookeeper instance spanning the micro-clouds federation. Nevertheless, to improve performance of the location and retrieval mechanisms, each micro-cloud also maintains locally a loosely consistent copy of the index.

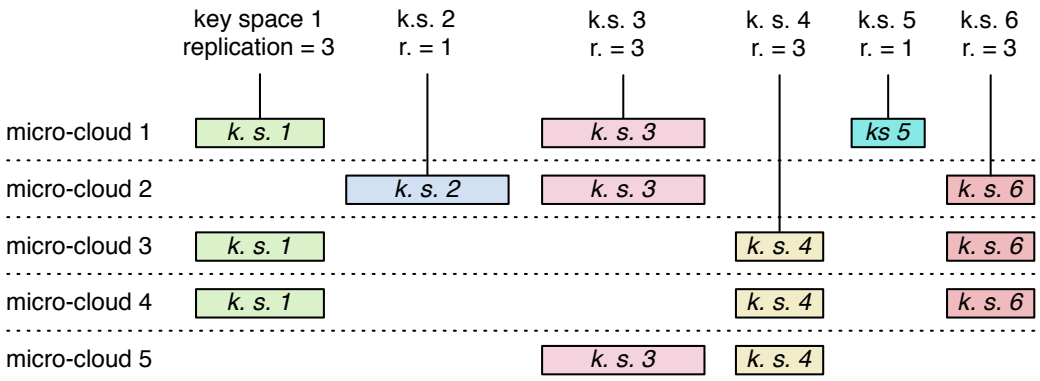

Figure 2: Illustration of the data placement based on an explicit shared index.

Research opportunity. Zookeeper is extensively used in data enters to implement various data storage systems. A typical use case of this coordination service is to maintain the metadata of a storage system. For instance, HBase uses the content of /HBase to store the location of the server hosting the root of all tables. In the context of LEADS, we aim at deploying a large-scale data store, and maintain the consistency of its metadata in Zookeeper. To the best of our knowledge, Zookeeper is seldom employed in such a context. The two key reasons are (i) that the service cannot leverage data locality since, at core, all the servers replicate the whole Zookeeper tree, and (ii) the communication pattern of Zookeeper is biased toward the leader that orders the operations. An interesting avenue of research is to investigate how to improve the performance of Zookeeper at large-scale. This requires to (i) leverage the fact that data accesses are disjoint with high probability in order to implement efficiently the shared tree in Zookeeper, as well as (ii) investigate the trade-off between bandwidth usage and message delay to cope with the long distant links between micro-clouds.

F2.72 Factory of atomic objects (federation)

Compare-and-swap[4] is an important building block to implement non-blocking data structures and operations. In particular, it is well known that any atomic object can be implemented on top of a compare-and-swap primitive [WaitFree]. However, in the message-passing world, this abstraction is not part of the standard high-level API. More typically, message-passing systems feature a synchronization service that replaces it (e.g., Zookeeper or Google Chubby [Chubby]). Such a service is implemented using the state machine replication approach [SMR], and relies either on a centralized sequencer, or the Paxos consensus algorithm [Paxos].

Synchronization services are not efficient at large-scale. We underlined previously that this inefficiency comes from the fact that all the modifications go through a leader node that orchestrates the service. To cope with this problem, we have investigated recently the implementation of a compare-and-swap primitive on top of the Cassandra distributed key-value store [SRF13]. Our key idea is to leverage that concurrent processes access distinct objects with high probability. In such a situation, we show that it is possible to implement a compare-and-swap primitive in a fully asynchronous and distributed manner, while providing atomicity and strong progress guarantees.

Figure 3 and Figure 4 report preliminary results when we compare our approach to Zookeeper. These results were obtained in a cluster of virtualized Xeon 2.5 GHz machines running Ubuntu 12.04 GNU/Linux and connected by a 1Gbps switched network. In all experiments, a set of clients compete on a shared object, simulating a concurrent workload. Our implementation is in Python, and uses the standard interfaces of Zookeeper and Cassandra.

In Figure 3, we evaluate the time to access a critical section when multiple clients compete for it. This bottom curve shows the performance results of Zookeeper when the critical section is implemented via the creation of a znode. The top curve depicts the performance of a spinlock object implemented on top of our distributed compare-and-swap primitive, and guarded by an exponential back off mechanism. In Figure 3, the inter-arrival time to the critical section follows a Poisson distribution. This experiment shows that when little contention occurs, the performance of our key-value store based implementation and Zookeeper are close. Nevertheless when the number of clients accessing concurrently the critical section increases, Zookeeper can be up to three times faster. This substantial performance difference is the price to pay to have *no centralization point in the system*.

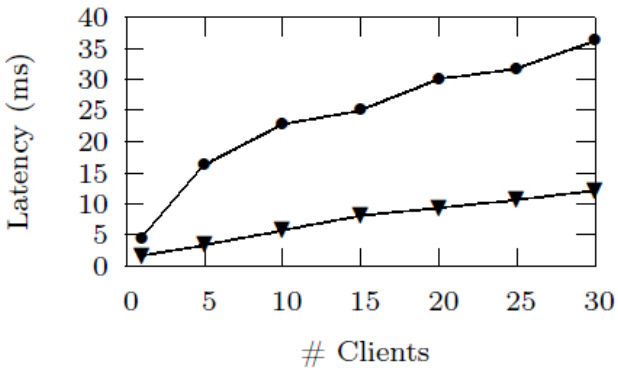

Figure 3: Comparison between Zookeeper (bottom) and a key-value store based implementation (top) of a critical section

[4] A compare-and-swap object exposes a single operation: cas{u,v}. This operation ensures that if the old value of the object equals u, it is replaced by v. In such a case the operation returns true; otherwise it returns false.

On the other hand, having no execution bottleneck pays off when concurrent clients access disjoint objects. In Figure 4, we present the scalability factor of our approach in comparison to Zookeeper. These results were obtained when clients access distinct compare-and-swap objects with high probability. The results for Zookeeper are reported for 3 servers. This is the maximal throughput we can obtain since all the operations update the shared objects.

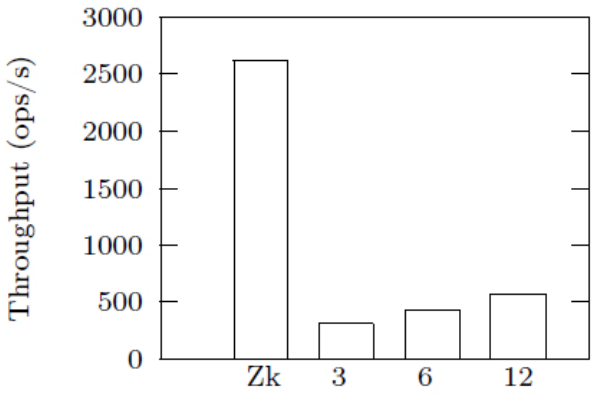

Figure 4: Scalability of a key-value store based implementation of compare-and-swap in comparison to Zookeeper

In Figure 4, we observe that our approach scales when the number of servers increases. With 6 and 12 servers we obtain respectively a multiplicative factor of 1.4 and 1.6 in comparison to a deployment of 3 servers. This scalability factor is small, nevertheless it shows that contrarily to the classical state machine replication approach, the system scales-up when more servers are added.

Research opportunity. Our preliminary results are promising. They show that one can implement a strong synchronization primitive on top of an off-the-shelf key-value store, and thus avoid a bottleneck in the system where all synchronization calls are serialized. Yet, to be more convincing, we should improve our base implementation. In particular, the implementation should be more efficient when a process accesses in isolation the compare-and-swap primitive.

F2.81 FUSE-based file system interface (federation)

Distributed file storage services (DFSS) offer a unified filesystem view of distributed data stores. As for any distributed storage service, the expected properties of a DFSS are consistency, availability, and tolerance to partitions. The CAP theorem [CAP] states that a distributed storage system can fully support at most two of these three properties simultaneously. Since a micro-cloud may be temporarily disconnected, and such an event should be supported in LEADS, partition tolerance is mandatory in our targeted architecture. Besides, an important non-functional requirement in LEADS is a clear separation of concerns between the level of the micro-clouds and the federation level. Consequently, a DFSS system at the federation level must be built using the data access primitives available at the lower level. To fulfil both requirements, we have worked on the implementation of a DFSS on top of a regular key-value store. Our preliminary work in this direction is FlexiFS [DAIS13]. FlexiFS is a modular DFSS that implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data. In what follows, we briefly describe our findings in developing FlexiFS, as well as the future directions we shall investigate so as to implement feature F2.33.

FlexiFS is a fully functional and configurable DFSS service. The key insight of FlexiFS is a novel decomposition of a DFSS in put and get operations augmented with a compare-and-swap synchronization primitive. In the context of LEADS, this primitive is implemented using features F2.71 and F2.72 that provide a universal construction of atomic objects at respectively the level of a micro-cloud and at the federation level.

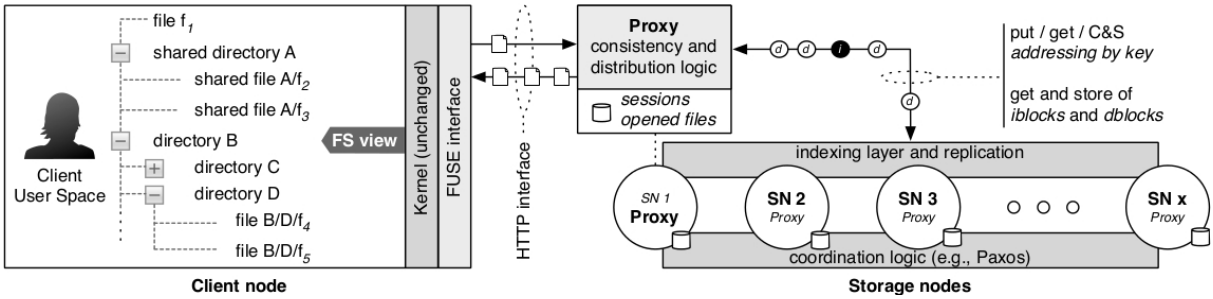

Figure 5: General Architecture of FlexiFS

We depict the general architecture of FlexiFS in Figure 5: General Architecture of FlexiFS. A FlexiFS client accesses the filesystem through a filesystem in user space (FUSE) implementation. FUSE is a loadable kernel module that provides a filesystem API to the user space. Each access to the filesystem is transformed into a Web service request and is routed toward a proxy node that acts as an entry point to the distributed storage. The proxy redirects requests to the adequate storage node(s), which store or return data blocks via the put/get interface.

FlexiFS decouples metadata from data storage. For each file, an inode block contains the metadata information about the file, e.g., size and user/group ownership. One or more data blocks hold the content of the file. Both data and inode blocks are stored in the distributed key-value store. The structure of the file system employed in FlexiFS follows a Unix-like graph structure.

When the user executes an access to the file system, the proxy splits the access in multiple get and put operations followed (if needed) by a compare-and-swap operation to update the metadata of the file. For instance, upon the creation of a file or directory, the proxy first stores a corresponding inode block in the distributed storage. Then, it executes a compare-and-swap operation on the parent directory to add the file. Performing a compare-and-swap ensures that no two clients create the same file concurrently. If the file was concurrently created, the proxy returns an error to the client. Otherwise the proxy opens the file and indicates that the operation was successful to the client.

Research opportunity. FlexiFS shows that decomposing a DFSS in flat accesses to a data storage layer is possible. We have also learned from this work that sharing the files can be done at a high level of consistency with a small performance[5]. Two directions of research follow this preliminary work. First, FlexiFS is implemented using expandable storage blocks, which requires an appropriate garbage collection mechanism to recycle them. This non-trivial mechanism is currently missing in our implementation. The second direction of research is the extension to the micro-clouds federation of the FlexiFS approach. To that regard, we are confident that our approach is applicable at this level, provided that a key-value store API exists at the level of the micro-clouds federation, as well as a mean to implement efficiently a compare-and-swap primitive. These two interfaces are provided respectively by features F2.12 and F2.72.

[5] Typical file system workloads exhibit more data accesses than metadata accesses. Under such circumstances, the performance penalty of implementing a strong sharing semantics over a weak one is small.

6. Conclusion

This deliverable presents a specification of the storage layer in the LEADS platform. This layer consists in a key-value store with extended capabilities to support a federation of micro-clouds. We present the targeted architecture, and list the features that have to be supported by the storage layer. Further, we describe our prototype implementation built on top of Infinispan. We list on-going work on Infinispan and some research opportunities related to them. As required in the DoW, our current prototype implementation is fully functional in the case of a single micro-cloud.

7. References

[BCQ+11] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In Proceedings of the sixth conference on Computer systems, EuroSys '11, pages 31–46, New York, NY, USA, 2011. ACM.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40, 2010.

[NoSQL] http://en.wikipedia.org/wiki/NoSQL

[Zk] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIXATC'10).USENIX Association, Berkeley, CA, USA, 11-11.

[EC] Werner Vogels. 2009. Eventually consistent. Commun. ACM 52, 1 (January 2009), 40-44.

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

[LIN] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 3 (July 1990), 463-492.

[SMR] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (December 1990)

[Ispn] http://www.jboss.org/infinispan

[Ispn-doc] https://docs.jboss.org/author/display/ISPN/User+Guide

[JGroups] http://www.jgroups.org

[ConsHash] Karger, D.; Sherman, A.; Berkheimer, A.; Bogstad, B.; Dhanidina, R.; Iwamoto, K.; Kim, B.; Matkins, L.; Yerushalmi, Y. Web Caching with Consistent Hashing, 1999 Computer Networks 31 (11): 1203–1213.

[Murmur] http:// en.wikipedia.org/wiki/MurmurHash

[Jboss] http://www.jboss.org/jbossas

[ONebula] http://opennebula.org

[MR] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1 (January 2008).

[Dyn] Giuseppe De Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07)

[Leads-dep] http://leads-project.eu/wiki/doku.php?id=knowledge:storage:deployment:about

[DAIS13] José Valerio, Pierre Sutra, Etienne Rivière, Pascal Felber: Evaluating the Price of Consistency in Distributed File Storage Services. DAIS 2013.

[Chubby] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06).

[WaitFree] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst, 1991.

[SRF13] Pierre Sutra, Etienne Rivière, Pascal Felber : An Efficient Distributed Obstruction-Free Compare-And-Swap Primitive, 2013 (https://github.com/otrack/PSSOLib)

[Paxos] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst. 16, 2 (May 1998),

[D1.2] Leads deliverable D1.2