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:	D2.4
Title of Deliverable:	Agile and secure prototype of the key-value store
Contractual Date of Delivery:	M24 – 9/30/2014
Actual Date of Delivery:	9/30/2014

Abstract

This deliverable presents the state of the distributed storage layer supporting the LEADS platform at M24. This layer consists in a key-value store with extended capabilities to be deployed in a federation of micro-clouds. We first describe the requirements of the key-value store in relation to the data-as-a-service (DaaS) model being developed in LEADS. Then, we recall the list of features defined in D2.2 to satisfy these requirements, and detail the state of our prototype implementation at M24. For the features currently in-progress, we review the research challenges that we have to address during the M25-M36 period in order to fulfil their developments. The last part of this document presents several experimental results that assess in practice the capabilities of the distributed storage layer.

List of Contributors

Name	Organization	E-mail
Xiao Bai	BM-Y!	xbai@yahoo-inc.com
Diego Marron	BM-Y!	diegom@yahoo-inc.com
Tim Potter	BM-Y!	tep@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
Pedro Ruivo	Red Hat	pedro@infinispan.org
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Nikolaos Giatrakos	TSI	ngiatrakos@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Nikolaos Pavlakis	TSI	npavlakis@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
Frank Busse	TUD	frank.busse@tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Etienne Rivière	Etienne.Riviere@unine.ch	2014-09-30
Approved by GA Member 1	Mark Little	mlittle@redhat.com	2014-09-20
Approved by GA Member 2	Minos Garofalakis	minos@softnet.tuc.gr	2014-09-22

Contents

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
CAP	Consistency Availability Partition tolerance
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 this 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) and for companies which are not primarily focusing on IT services.

With more details, the approach proposed in LEADS consists in aggregating into a federation of micro-clouds the infrastructures required by these multiple SMEs and client companies. 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. This approach raises several challenges. In particular, it relies on an efficient distributed storage system which runs on a collection of micro-clouds, provided by the participating SMEs and companies or offered by an infrastructure provider such as LEADS partner Cloud&Heat. WP2 is responsible for the design and implementation of this storage layer. The core challenges 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 Web resources and contributed by companies themselves.

In this deliverable, we present the state 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 recall the architecture targeted in LEADS, as well as the usage of the storage layer by higher tiers of the project (developed in WP1, WP3 and WP4). Then, we list the key features that are currently available in the storage layer and present the additions we made in comparison to D2.2 (M12). In particular, we detail the key/value API at the micro-clouds federation level, the implementation of the explicit data placement and retrieval, the multi-version support and the state of our toolkit for automated deployment and configuration of the storage layer. Then, we present several experiments that demonstrate the capabilities of our current prototype and evaluate its overall performance. The appendix of this document contains the scientific contributions which were produced by WP2 during the M13-M24 period.

2. Introduction

This document describes the state of the storage layer supporting the LEADS service at M24. Following the DoW of the project, we applied an iterative development to build the necessary functionalities supporting the storage layer requirements in LEADS. We started from a functional storage on a single micro-cloud having a modular design (described in detail in D2.2). In the last period, we designed and partially implemented the features required to support a micro-cloud federation.

The remainder of this document is organized as follows. Section 3 recalls the architecture targeted in LEADS, i.e., a federation of micro-clouds, and lists the requirements of the storage layer in regard to its usage by higher tiers. Section 4 details the features already implemented in our initial prototype, as well as the features whose integration is currently in-progress. With respect to the latter, we also introduce a list of related research opportunities. In Section 5, we present a detailed evaluation of our current prototype. We conclude this document in Section 7. The appendix contains (peer-reviewed) publications produced during the period.

3. Overview of the LEADS storage layer

In this section, we recall the architecture targeted in the context of LEADS. Then, we present an overview of the functional and non-functional requirements for the storage layer. The interested reader may consult D2.2 for further details.

3.1 Targeted architecture and associated failure model

The LEADS platform is realized by 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 fibres physical links makes the interaction over these links very efficient. Nevertheless, there is at least an order of magnitude of performance-degradation (in term of bandwidth and message delay) between the link outside and inside a micro-cloud.

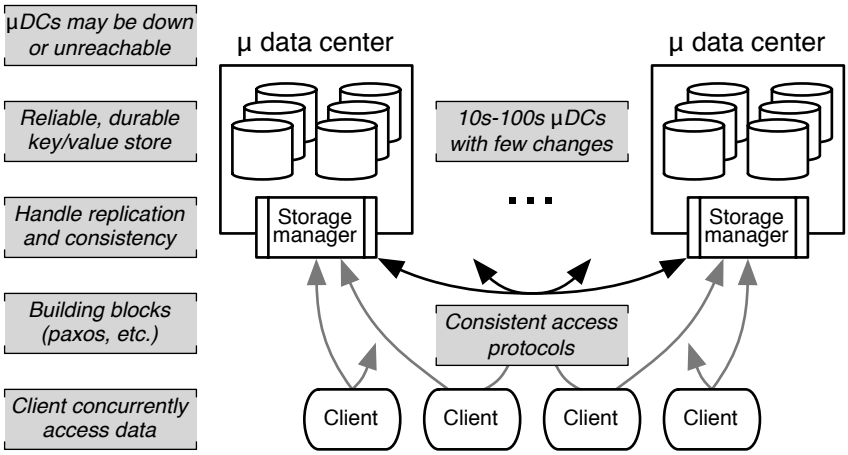


Figure 1: High-level architecture of the LEADS storage layer.

The 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 abstracted by a 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, horizontal elasticity is considered and should take two forms: adding or removing a machine to a micro-cloud, and adding or removing a micro-cloud from the system. We note here that (i) when the configuration of the storage layer changes, the load must be automatically balanced without interfering with on-going computations and with existing data collections, and (ii) the heterogeneity of the micro-clouds should also be taken into account. Item (i) falls under the scope of WP2, whereas WP4 is in charge of item (ii).

Due to the performance gap between local and wide-area network connections, *access locality* is of paramount importance to the efficiency of the system. At local scale, since a micro-cloud is small, and internal connections are fast, we employ traditional techniques to partition the load. At the federation scale, the system must support appropriate data placement and scheduling policies to leverage locality. These two aspects are under the responsibility of WP4; the storage layer should offer appropriate *information reporting* and *mechanisms* to implement WP4 policies.

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 forms of data 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 item 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: 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 levels are possible. A strong consistency level typically requires expensive synchronization and communication between nodes. The nature of the consistency level (strong, weak, etc.) 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.

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 latter case.

The LEADS platform offers querying and extraction mechanisms, defined in WP3. Queries can be persistent and operate on the data items obtained through the extraction service (WP1). Queries may be injected several times into 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 match the given query, or for which the access is not done primarily by the name of the item but according to another query.

4. Implementation

This section describes the current state of the LEADS storage layer at M24. Following the DoW, we applied an iterative development of the functionalities. We started from a functional storage deployed in a single micro-cloud (M12 prototype), on top of which we developed a micro-cloud federation storage.

Table 1: Implemented and in-progress features in the M24 prototypelists the features implemented (in green) and the features whose implementation is in-progress (in yellow). Features with a star (*) were already implemented and described in deliverable D2.2. We briefly recall their roles and functionalities in this deliverable.

Features are classified according to the fact that they target either a single micro-cloud (C) or the federation (F). Please note that in order to help the reader the numbering of the features has changed in comparison to D2.2.

Features	Single micro-cloud	Federation
Key/value store API	C11*	F11
Listener API	C12	F12
Data collection API	C13	F13
Fault-tolerance mechanisms	C21*	F21
Elasticity	C22*	F22
Consistent hashing based data placement	C31*	
Explicit and constrained data placement		F31
Data location retrieval		F32
Total order based replication	C41*	
Primary and quorum based replication		F41
Eventual consistency based replication	C42	F42
Heat maps support		F51
Automatic caching of public data		F52
Versioning	C61	F61
Redirection support		F62
Factory of atomic objects	C71	F71
FUSE-based file system interface	C81*	
Security and authentication	C91	
Support for deployment and configuration	C101	F101

Table 1: Implemented and in-progress features in the M24 prototype

In what follows, we describe in detail each of these features. Where appropriate, we also discuss ongoing research developments, as well as, future opportunities that may be investigated in the context of the LEADS storage layer.

4.1 Overview

At the scale of a micro-cloud, the LEADS storage layer consists of *Infinispan*, an open source industrial-grade key-value store [Ispn, Ispn-doc]. At the federation level, it consists of Infinispan-ensemble, or simply *Ensemble*, an aggregation of geographically-distributed Infinispan instances.

Red Hat 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. These features are first added to the development branch of Infinispan, then to eventually merged into the stable branch (currently Infinispan 7).

4.2 List of Features

In what follows, we detail the implemented features and the in-progress features of the LEADS storage layer, both at the level of a single micro-cloud and at the level of the federation of micro-clouds.

Single micro-cloud level

The core of Infinispan consists in 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. As pointed out previously, the main developer of Infinispan is Red Hat. Other partners contribute to Infinispan in a more research-oriented manner by branching and quickly developing cutting-edge ideas which will be later merged into the code base. Infinispan is open-source: all LEADS-specific developments are made public either in the main branch of Infinispan or on public code repositories.

C11* Key-value Store API (single micro-cloud)

The key-value store API of Infinispan consists in a set of Cache interfaces. Each cache extends java.util.Map [Ispn-API]. A client accesses the various caches stored in Infinispan through a management entity named *CacheManager*. The cache data structure is highly concurrent, designed from the 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 within a data center. In addition to the peer-to-peer architecture of Infinispan, to which clients can connect directly from any node provided they are running on the same cluster (known as *embedded* mode), a remote access mode (based on a classical client/server architecture) for remote clients is also supported via the HotRod protocol [HotRod]. HotRod provides the ability to run multiple instances of Infinispan 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.

C12 Listener API (single micro-cloud)

Core contribution: remote listener, converter and filter.

Streams are a common distributed programming pattern in nowadays applications (e.g., Twitter). One of the usages envisioned in LEADS consists of 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) and to

support domain-specific operators in the form of plugins (D3.4 and D5.4). To support stream-oriented architectures and in particular LEADS DaaS requirements, Infinispan offers a listener API: once a client is registered in a cache, it gets notified of every operations occurring on the data stored there. Listeners allow implementing communication between LEADS components as well as supporting the instantaneous/persistent query model described in our use cases.

At the beginning of the LEADS project, the eventing mechanism was solely available in embedded mode (e.g., the client is in the same memory space as an Infinispan node generating events). In Infinispan 7, developed during period M13-M24, Red Hat extended this API to operate in clustered mode (the client is in the same micro-cloud as an Infinispan deployment) and in remote mode (using the HotRod protocol). In addition, it is also possible now to optionally customize the listener by adding filtering mechanisms (e.g. KeyFilter classes) as well as a data conversion mechanisms (e.g. Converter classes). The first mechanism filters keys that may trigger an event, while the later converts the existing value to retain only useful information. Both mechanisms are also useful to reduce the payload sent to the client. Finally, a special listener (see D3.4) supports the definition of client-provided plugins, which allow performing domain-specific operations on incoming data.

C13 Data collections API (single micro-cloud)

Core Contribution: Map/Reduce framework and distributed entry iterator.

Infinispan offers a Map/Reduce framework to query data collections stored as caches. This framework is used in WP3 to develop relational-oriented queries (e.g., JOIN). In Infinispan 7, several optimizations have been made to this framework in order to improve performance. First, the map and the reduce phases are now running in parallel at each node. Second, the resulting data can be stored in a (user-defined) cache, thus reducing the overload at the node that started the Map/Reduce task.

In Figure 3, we illustrate the core benefits brought by these improvements of the Map/Reduce framework. More precisely, this figure compares the performances of the old and the new framework using a Map/Reduce word count task, when varying the amount of entries to count. We can observe that our modifications are more than promising. In this use case, the Map/Reduce task execution speed and throughput improvement are between fourfold and sixfold when entries are of small size.

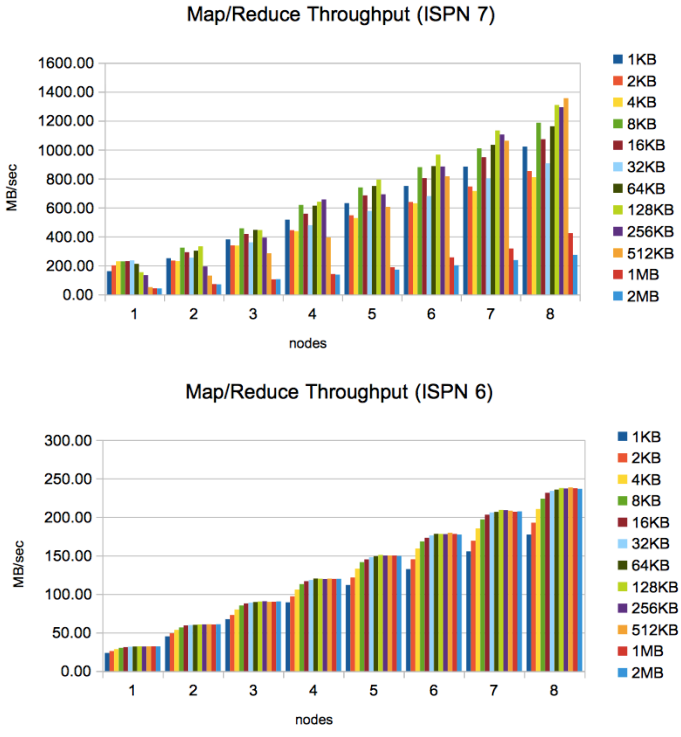


Figure 2: M/R Word Count Performance (top: Infinispan 7, bottom: Infinispan 6)

The interested reader may consult additional details on the Map/Reduce framework online [WordCount, Ispn-doc].

In addition to the Map/Reduce support, Infinispan offers a distributed Entry Iterator to retrieve collections of stored data. This mechanism allows the iteration over all entries in the cache. Iterating over all entries in the cache has always been a highly important feature. Existing methods (entrySet(), keyset() and size()) in the previous prototype of the key-value store were not a good fit because of potential out-of-memory errors. As in the cluster listener, it allows to set an optional KeyFilter and Converter. The general idea is to fetch the keys remotely in chunks to avoid overloading the requested node.

C21* Fault-tolerance mechanisms (single micro-cloud)

Infinispan offers high availability via making replicas of the shared cache state across a network as well as optionally persisting state to configurable cache stores. To that end, 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.

C22* Elasticity (one micro-cloud)

C31* Consistent hashing-based data placement (single micro-cloud)

Data placement and retrieval in Infinispan employs consistent hashing [ConsHash]. The hash function in use 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. It provides both elasticity, i.e., the ability to add and remove nodes at low-cost, and load balancing.

C41* Total order-based replication (single micro-cloud)

Consistency guarantees on the *put*/*get* operations in Infinispan are tunable. This interesting property comes from the fact that all the communications between servers are based on the JGroups library. In particular, when a cache is set at the *transactional* consistency level, operations are atomic.[1] This means that every read or write operation at the cache level behaves as in a shared-memory context: writes are totally ordered from the perspective of the reads, and if an operation *op* precedes in real-time an operation *op'* then op' sees the effect of *op.* This consistency level is achieved via the use of a total-order group communication primitive in JGroups. Finally, it is possible to obtain atomicity when using primary-copy replication inside Infinispan.

C61 Versioning (single micro-cloud)

Core contribution: convenient support to store/retrieve multiple versions of a datum.

A multi-version key-value store ensures that every *put(k,v)* operation creates a new version *v* of the datum *k,* while keeping the previous ones. Storing most of (or even all) the versions created in the history of *k*, permits the clients to retrieve the state of the data as it was at some point in time. This facility allows computing consistent cut of the stored data. A cut is similar to a snapshot of the entire state of the storage at some moment back in time, and it allows to perform queries on the state of

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

the stored Web graph a few days, weeks, etc. before the query is issued; potential applications include comparing the evolution the result of a query originally not implemented as a stream oriented query). Moreover, it allows to transparently support the storage of time series. The typical use case we target by the support of multi-version is to trace the evolution of a large-scale graph, e.g., part of the Web or a social network, over time.

Implementing multi-version support on top of a key-value store requires to build an *index* of all the versions for each key. Such an index should allow executing efficiently the queries to retrieve ranges of versions. Moreover, since the distribution of versions is unlikely to be uniform, the index should be sharded (i.e., split in multiple parts or shards, and each shard placed on a separate server) to evenly spread the load across the storage nodes.

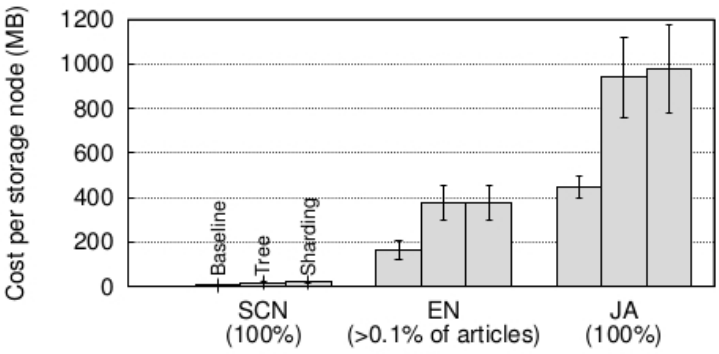


Figure 3: Storage cost with different Wikipedia datasets and for

In addition, it is worth point out that, in the case of a distributed large graph evolving over time, the storage layer needs to (i) store only the modifications occurring between one state and the following one, (ii) cache the graph, re-computing its state on-demand, and (iii) leverage the locality since most operations requires navigating the graph.

We implemented the support for multi-version on top of Infinispan that is data agnostic. Three implementations of this feature are available. We proposed and published a scientific paper that compares extensively these approaches at the IEEE SRDS 2014 conference. This paper is presented in Appendix B.

Our first implementation consists in the naïve approach of storing all the versions under the same key. The two other approaches leverages the atomic object factory (feature C17, detailed at page 17) to implement respectively a map and a sharded map per key. All these implementations have a common algorithmic structure, but they differ on some aspects having their pros and cons.

In the SRDS 2014 paper (see Appendix B), we investigated the most convenient approach to adopt in the context of the LEADS applications. To that end, we performed a detailed evaluation of the storage layer under a real workload from Wikipedia access traces. Our evaluation highlights the inherent trade-offs of each implementation, from atomic maps to tree-based and sharded tree-based indexes, and the specific adequacy to different workload patterns.

In what follows, we briefly cover one of the experimental results obtained evaluating the various approaches. The interested reader may refer to Appendix B for further details and a formal description of the algorithms.

Versioning Cost. Our evaluation of the prototype versioned key-value store consists in the re-execution of real access traces on a Wikipedia data dump stored in the key-value store. To that end, we used a cluster of 24 virtualized 4-core Xeon 2.5 Ghz machines with 4GB of memory, running Gen-

too Linux 32bits, and connected by a virtualized 1 Gbps switched network (available at UniNE). Network performance, as measured by the tools *ping* and *netperf*, is of 0.3ms for a round-trip with a bandwidth of 117MB/s. Clients runs a modified version of YCSB [YCSB] that replays Wikipedia access trace. We use three Wikipedia datasets: SCN (Sicilian), EN (English) and JA (Japanese). Properties of these datasets, such as the distribution of versions, are detailed in the publication in the annex.

Figure 2 depicts the amount of storage used per node to load each Wikipedia in the distributed storage. We compare three of the available implementations: the naive implementation *(Baseline)* that wraps all the versions of a datum k under the same key, *(Tree)* which stores independently the index and the version in different keys, the index being updated when necessary, and *(Sharding)* which relies on a pattern similar to Tree except that the index is this time spread over multiple nodes.

In Figure 2, we observe that the baseline mechanism is the less expensive for storing the versions: it costs around half the price of the two other mechanisms. This difference is mainly explained by the fact that the two other mechanisms separate data from metadata, inducing an additional cost (in term of versions and base KVS metadata). We also observe in Figure 2 that by sharding the version index one must accept a small overhead, in comparison to an approach where the index is stored at a single storage node. This comes from the fact that the threshold to create a new shard of the version index is set to a default value of 1000 versions, hence occurring in rare cases.

In our experiments, the overhead of storing the index independently from the data is around 2 times the base storage cost. In return to that cost, we show in our paper that with this approach *(i)* concurrent write-access to versioned data is possible and *(ii)* access to versioned data is one order of magnitude faster.

C71 Factory of atomic objects (single micro-cloud)

Core contribution: powerful and elegant concurrent objects abstraction.

The factory of atomic objects allows a client application to build an atomic object of her choice on top of Infinispan. An atomic object is a replicated version of a simple Java object. The factory allows creating a fault-tolerant object (since there are multiple copies whose states are synchronized, the object remains available if one or several copies are lost, depending on the replication factor). Such universal objects are 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 using the total order-based replication facility of Infinispan. Our approach is a variation of the well-known state machine replication technique [SMR] already at work in multiple distributed systems. Practically speaking, 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. Once the call gets de-serialized, it is applied to the local copy and, in case the calling process was local, the response value is returned (this mechanism is implemented via a Future object).

We used the factory of atomic objects for the implementation of the versioning support in Infinispan (feature C61). Appendix B covers the algorithmic details of the atomic object factory.

C81* FUSE-based file system interface (single micro-cloud)

Distributed file storage services (DFSS) offer a unified file system view of a distributed data store. 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. The result of this work is FlexiFS [DAIS13], a modular DFSS that implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data. FlexiFS was described in detail in our previous deliverable D2.2.

C91 Security and authentication

Core Contribution: access control over stored data and secure communication.

During the M13-M24 period, Infinispan has gained the ability to perform Access Control on CacheManagers and Caches. In terms of security and authentication, the set of new functionalities consists in:

- Coarse-grained access control to CacheManagers, Caches and data;
- Credentials for remote clients and encryption of the communication between clients and server;
- Authorization for each node to join the cluster; and
- Encryption of the communication channel between servers.

In order to maximize compatibility and integration, Infinispan uses widespread security standards where possible and appropriate, such as X.509 certificates, SSL/TLS encryption and Kerberos/GSSAPI.

Authorization in Infinispan uses role-based access control. It is configured at two levels: at the cache container and at the level of each cache. In the cache container level, the client specifies the named roles and the permissions they grant. We give an example below.

```
<role name="admin" permissions="ALL" />
<role name="reader" permissions="READ" />
<role name="writer" permissions="WRITE" />
<role name="supervisor" permissions="READ WRITE EXEC"/>
```

At the level of a cache, a client may then specify a subset of the authorized roles. For instance, continuing the example above,

```
<authorization enabled="true" roles="admin reader writer" />
```

To interact with a cache, the client needs a Java authentication and authorization service (JAAS) subject [Jaas]. A JAAS subject represents a single user, entity or system, in other words, a client, requesting authentication. This subjects need to be populated with a set of principals modelling its

properties, as well as a mapper to map the principals to the roles. The interaction is made as in the example that follows:

```
String value = Subject.doAs(mySubject, new PrivilegedAction<String>() {
        @Override
        public String run() {
                return cacheManager.getCache().get("key");
        }
});
```

For the Hot Rod clients, the authentication is done with the SASL framework [SASL]. Leveraging the use of SASL, Infinispan can support many authentication mechanisms out of the box such as, plain text, digest-MD5, or GSSAPI. In addition, SASL allows external authentication where the client-certificate identity of the underlying transport is used as the credentials.

As for Hot Rod clients, the cluster authentication is implemented using SASL and the encryption using SSL/TLS. Both protocols need to be added and configured in the JGroups configuration at the start of the Infinispan cluster.

C101* Support for deployment and configuration (single micro-cloud)

Core Contribution: JBossAS integration and OpenNebula support.

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. Moreover, such a feature allows to configure conveniently all the machines in a uniform way which is a mandatory step to provide elastic storage. Infinispan provides rich tooling in this area, with many integration opportunities. In particular, Infinispan is integrated with the Jboss Application Server [JbossAS]. JBossAS is a Platform-as-a-Service (PaaS) which implements all the features available in the Java Platform Enterprise Edition (java JEE). During the period M13-M24, Red Hat extended the JBossAS support for Infinispan, in particular aligning the embedded and JBossAS configuration files.

Additionally, we have built for the project integration purposes a standalone distribution of Infinispan that runs on top of the OpenNebula (IaaS) cloud environment [OpenNebula, Leads-dep]. Our set of scripts, disk images and OpenNebula templates are used by consortium partners to integrate their components with the storage layer. 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.

C42 Eventual consistency based replication (single micro-cloud)

Core Contribution: Weakly consistent but converging shared data.

The addition of multi-version support to Infinispan allows us to offer an eventually consistent view of the storage to the clients. This feature is currently under development |ISPN-999]. Below, we briefly cover the approach we use to implement it.

Eventual consistency states that once clients stop submitting new updates, all the replicas need to converge to the same state (of the cache) potentially after reconciliation between diverging values. Reads performed before this reconciliation may report multiple values and the application has to arbitrate between these values. Eventual consistency is a more challenging consistency model for programmers but allows for better performance in the cases where it is applicable. In the context of LEADS, because public data are immutable they are subject to few anomalies under eventual consistency. Hence, this consistency level is a good fit in such a case.

We plan to support eventual consistency by layering a reconciliation engine on top of the versioned key-value store. This engine is semantics-aware, in the sense that the clients may specify which reconciliation policy to apply when several concurrent versions of the same datum exists (e.g., last writer wins or a more specific application-dependent reconciliation strategy). The reconciliation engine works as follows: When a new versioned datum *(k,u)* is inserted, we create a versioned data *(k,v,u)* where v is a new version assigned to *(k,u)*. This tuple is then inserted in Infinispan. Upon retrieving a datum stored under the key *k*, the storage first retrieve all the latest versions stored under key *k*, then it calls the reconciliation engine which outputs arbitrarily a datum *(k,v)* which is returned to the client. A garbage collection mechanism ensures that a bounded number of versions may coexist at some point in time.

Micro-Cloud Federation Level

At the federation level, the key-value store developed in WP2 consists of Infinispan-ensemble, or simply *Ensemble*. Ensemble exports the core interface of Infinispan but is a separate software entity. This interface is implemented atop multiple Infinispan deployments, one on each micro-cloud. Each such deployment abstracts the storage interface of a micro-cloud.

Ensemble accesses a micro-cloud deployment via the HotRod protocol and the RemoteCache API. An EnsembleCache (respectively, an EnsembleCacheManager) implements the BasicCache (resp. BasicCacheContainer) API of Infinispan. To coordinate geo-distributed accesses to the storage, e.g., the concurrent creation of EnsembleCaches, Ensemble stores both the list of micro-clouds, the definition of each EnsembleCache and their allocation in a consistent and dependable index stored in the coordination service ZooFence. ZooFence is a locality-aware variation of ZooKeeper using a principled approach to service partitioning. ZooFence is another contribution of WP2, and we describe it in this section.

F11 Key/Value Store API (federation)

Core contribution: base KVS operations for the micro-cloud federation.

Ensemble exposes a Java interface that consists of two key components: an EnsembleCache and an EnsembleCacheManager. An EnsembleCache is a named and typed instance of the key-value store that spans across the micro-clouds. An EnsembleCacheManager is a container of EnsembleCaches. Both abstractions are directly inherited from the Infinispan API (respectively a Cache and a CacheContainer).

An EnsembleCache contains multiple RemoteCaches; each RemoteCache represents an Infinispan deployment at the scale of a *single* micro-cloud. Calls to a RemoteCache, and thus to the backing Infinispan instances, are implemented via the HotRod protocol [Ispn-doc]. Once an EnsembleCache is created, the user can store/retrieve data using the regular get() and put() operations. These operations are executed on the appropriate Infinispan instances according to the replication degree and the consistency criteria that characterize the EnsembleCache. We detail these parameters in features F21 and F22.

F12 Listener API (federation)

Core contribution: eventing support over multiple micro-clouds.

The novel eventing mechanism developed during M13-M24 (see feature C12) allows a full listener API at the federation level. Thanks to this mechanism, when a client wishes to listen cache events, she simply registers to the appropriate remote cache. In the case where the cache is distributed across multiple micro-clouds, a registration per micro-cloud is necessary and duplicates are removed automatically in the client-side library (this mechanism is transparent to the application).

F22 Primary and quorum based replication
F21 Eventual consistency based replication

Core contribution: support for strongly-consistent data duplication across micro-clouds.

From an abstract point of view, an EnsembleCache can be seen as the construction of dependable registers atop fault-prone ones. Indeed, in the targeted LEADS architecture, each Infinispan deployment operating in a micro-cloud is subject to be unresponsive from the outside (e.g., the micro-cloud is subject to a power outage or the internet link fails). Based on this observation, we developed Ensemble using existing algorithms and principles taken from dependable shared-memory computing literature.

We distinguish two types of EnsembleCache: a ReplicatedEnsembleCache fully replicates the data it holds to the set of micro-clouds it has been assigned to, whereas a DistributedEnsembleCache partially replicates its content across its set of allocated micro-clouds. (The usage of the terms "replicated" and "distributed" come from the terminology in use in Infinispan).

ReplicatedEnsembleCache. With a ReplicatedEnsembleCache, data is fully replicated across multiple micro-clouds. A ReplicatedEnsembleCache can be instantiated in various flavours, each flavour representing a distinct consistency level. Specifically:

- *WeakEnsembleCache* implements a ReplicatedEnsembleCache with weakly consistent operations. The implementation is straightforward: to execute a *put()* operation, the client applies *put()* to a quorum of RemoteCaches, and to execute a *get()*, the client accesses (at random) one of these RemoteCaches.
- *SWMREnsembleCache* implements an atomic single-writer multiple-reader atomic API of a Cache on top of several RemoteCaches. In this case, the system makes use of a primary-based replication schema: every read access the primary, and a write first writes to the primary then asynchronously and in parallel to the other caches.
- MWMREnsembleCache implements the complete Cache API at the atomic level, offering an atomic multiple-readers multiple-writers cache. To implement this abstraction, Ensemble relies on the following classical quorum algorithm: when the client executes a *put(k,v)*, a MWMREnsembleCache first retrieves the greatest version (timestamp) t stored at a quorum of RemoteCaches; then it executes *put(k,(t+1,v))* at a quorum of RemoteCaches. This last write is conditional. A *get(k)* operation executes a similar operation, i.e., first it retrieves the value v associated with the greatest timestamp, and then it writes back this value before returning.

DistributedEnsembleCache. With a ReplicatedEnsembleCache, data is stored in a dependable manner and it can also be replicated at the local micro-cloud in order to improve the response time of Ensemble to clients' requests. On the other hand, such a construction does not fully leverage the available storage in the micro-cloud federation. The notion of a DistributedEnsembleCache retains the dependability property of a ReplicatedEnsembleCache, while allowing data to be replicated across several, but not all, of the micro-clouds. A DistributedEnsembleCache also allows using explicit partitioning and placement strategies for data, and to improve locality of accesses in some scenarios. Similarly to a ReplicatedEnsembleCache, a DistributedEnsembleCache implements a BasicCache API on top of multiple caches. However, in this case the underlying caches are (i) either ReplicatedEnsembleCaches or RemoteCaches, and (ii) at construction time a Partitioner object is given to map keys to the appropriate caches.

This gives us the following constructor:

```
public DistributedEnsembleCache(String name,
                                 List<? extends EnsembleCache<K, V>> caches,
                                 Partitioner<K, V> partitioner)
```

where a Partitioner is an interface exporting the following method:

```
public abstract EnsembleCache<K,V> locate(K k)
```

As an example, the ModuloPartitioner implements a simple modulo operation on the hash value of *k* to retrieve the EnsembleCache that should store the content indexed by key *k*.

Frontier Mode. In a DistributedEnsembleCache, *put* and *get* operations behave as usual. This means that a *put(k,v)* inserts tuple *(k,v)* in the EnsembleCache *E* returned by *locate(k),* and *get(k)* operation returns the tuple *(k,v)* stored at *E*. When a DistributedEnsembleCache operates in frontier mode, put operations work as previously, but *get()* operations solely returns data located in the local site. As we shall illustrate in Section 7, this mode allows us to transparently port existing applications to the geo-distributed scale.

F31 Explicit and constrained data placement (federation)

Core contribution: fine-tuned data placement for the client-side.

For performance reasons, WP3 queries must access data items that, to the highest extend, should be located 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 that we define as *data-to-data affinity.* The LEADS storage layer is aware of these locality constraints and the two-level hierarchical nature of the platform as,

(1) services developed in WP4 leverage both data-to-data and data-to-query affinity to improve the performance of the LEADS architecture, and
(2) Ensemble implements an *explicit* and *constrained* data placement strategy.

In **Error! Reference source not found.**, we illustrate this strategy implemented in Ensemble. At the creation of an EnsembleCache, the user gives an explicit set of Infinispan instances deployed on the micro-cloud federation that are backing the EnsembleCache. For instance, the green cache in **Error! Reference source not found.** is replicated on the micro-clouds 1, 3 and 4. Alternatively, the user can provide a replication degree and a list of prohibited micro-clouds (e.g., as the stored private data may not be in a region under a particular legislation). Based on this information, EnsembleCacheManager compute an appropriate set of backing instances. For instance, to create the yellow cache in **Error! Reference source not found.**, the user can specify a replication degree of 3 and bans the micro-clouds 1 and 2.

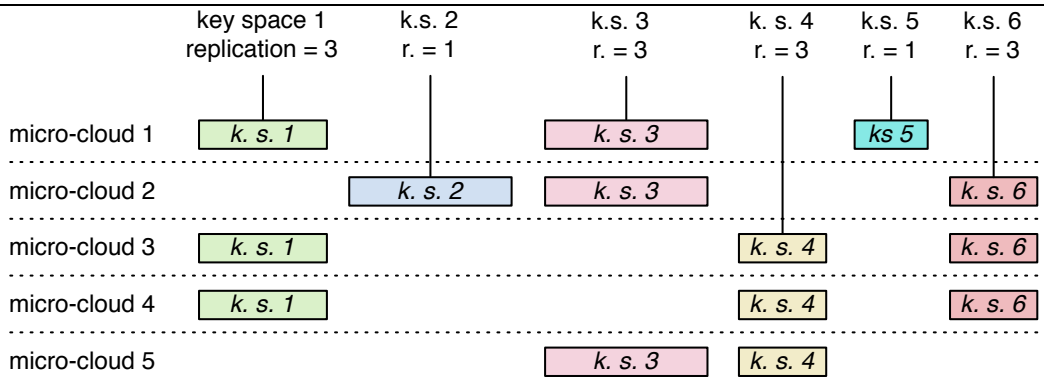


Figure 4: Illustration of the data placement based on an explicit shared index.

In the current state of our prototype, Ensemble is already able to implement a constrained and indexed data placement. The two functionalities are available through either a Java interface or a REST API.

REST API for an EnsembleCache. The proposed interface maps all EnsembleCacheManager into REST calls, and adds for convenience two simple get() and put() operations. The data format in use is JSON. We give examples of the REST API in Appendix A.

F32 Data location and retrieval (federation)

Core contribution: principled distributed service partitioning with application to indexing

Accesses across micro-clouds should be rare. This imposes that the location and retrieval of an EnsembleCache is executed in a single step. Moreover, the explicit index is at core of the Ensemble architecture, and thus, it should be shared between all the micro-clouds. To satisfy both requirements, Ensemble stores the index inside a globally, strongly-consistent shared tree.

To implement this index, our initial choice was to use an Apache ZooKeeper instance that spans the micro-clouds federation. ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination [Zk10]. It is based on the seminal state machine replication (SMR) approach. By replicating the shared tree on multiple servers, ZooKeeper operations are wait-free despite failures, and by executing them in the same order at all replicas, they are atomic (a.k.a, linearizable).

However, it is well-known that this last strategy has a performance cost: because ZooKeeper serializes all commands, it does not leverage the intrinsic parallelism of the workload. This situation is even more problematic in a geo-distributed setting where remote accesses are necessary even for shared data which are only accessed locally.

Recently, we worked on a scalable solution to address the above problem. Our solution, to be presented at the IEEE SRDS 2014 conference (see Appendix C), consists in partitioning the shared tree exposed by ZooKeeper [SRDSb].

With more details, we proposed and formulated specific conditions under which *any* distributed service is *partitionable*. We also present a general algorithm to build a dependable and consistent partitioned service, and apply it to build ZooFence. ZooFence is a coordination service which mimics ZooKeeper API. Under the hood, it orchestrates several instances of ZooKeeper and presents the exact same API and semantics to its clients. It automatically splits the shared tree among ZooKeeper instances while being transparent to the application. By reducing the convoy effect on operations and leveraging the workload locality, this approach allows proposing a coordination service with a greater scalability than with a single ZooKeeper instance. The interested reader may consult Appendix C for further details.

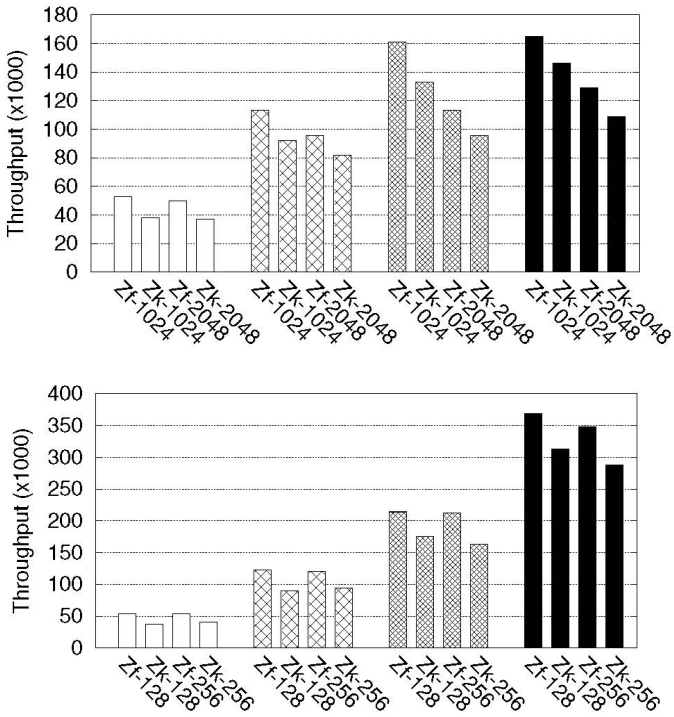


Figure 5: BookKeeper performance *(from left to right, the amount of written entries is 100, 250, 500 and 1000)*

Next, we briefly cover an example empirical evaluation that illustrates the obtained results. The complete paper describing this work is available in the Appendix C.

BookKeeper. Figure 5 presents a detailed performance comparison of BookKeeper [BookKeeper], a dependable concurrent logging service, when using ZooKeeper and ZooFence. We vary the length of a log entry, from 128 to 2,048 bits, and the number of entries, from 100 to 1,000, written by a client before it switches to a novel log. In both figures, Zk and Zf stands, respectively, for ZooKeeper and ZooFence. The throughput is measured as the total amount of operations per second. When clients write 1,000 entries (or more, not shown on these plots), the two systems achieve close performance. In such a case, the throughput is limited by either the storage nodes replicating the log, or the network. During our experiments, the MTU (maximal packet size) is set to 1,500 bytes. This explains the performance gap between large and small entries. On the other hand, when clients create concurrently more logs, fast operations on the metadata storage matter. In such a case, because ZooFence provides parallel accesses to the shared tree, it outperforms ZooKeeper. The difference increases as clients access new logs more frequently. In our experiments, ZooFence improves the throughput of BookKeeper by up to 45%.

F22 Fault-tolerance mechanisms (federation)

Core contribution: mechanisms to sustain micro-cloud failures

Avoiding the interruption of the storage service is a paramount property of the LEADS storage layer. As pointed out in D2.2, Infinispan already satisfies this guarantee in a single micro-cloud thanks, in particular, to the JGroup communication library [JGroup]. At the federation level, implementing this

guarantee requires two components: a failure detector that monitors micro-clouds, and a set of distributed mechanisms to implement a (configurable) policy when such an event occurs. These two components are currently under development, and both of them offer challenging research problems. We briefly cover them in what follows.

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. Moreover, we note that responding to a micro-cloud failure incurs into a lot of processing to reach the nominal state of execution. This trade-off between the response time to such an event and the cost of the recovery process is a challenging direction to investigate.

Recall that the state of Ensemble consists in two globally shared objects backed by ZooFence. Based on this design, we plan to implement mechanisms to migrate data to/from a micro-cloud as event handlers registered on the global shared indexes. For instance, when a micro-cloud *uc1* is removed, an event is triggered at every other micro-cloud. Each EnsembleCacheManager then searches for all the caches that it owns which relies on *uc1*, and starts migrating the data. Such an approach is reminiscent of the live object paradigm of Birman et al. [LiveObj]. When implementing these mechanisms, we plan contributing to the Menagerie library [menagerie], an open-source project that supports collections on top of ZooKeeper (and thus ZooFence).

F23 Elasticity (federation)

Core contribution: ability to add/remove a micro-cloud in the federation

In our current prototype, Ensemble clients have the ability to add and remove micro-clouds. This mechanism is available using a Java API or using the REST API (see Appendix A). Recall that at the scale of a single micro-cloud, Infinispan is elastic, that is it supports the on-the-fly addition or removal of a node to the system without any service disruption (see feature C22). As a consequence, the additional ability of Ensemble to add/remove micro-cloud on-the-fly guarantees full elasticity at the federation level. Both levels of elasticity are used by the scheduler implemented in WP4. In particular, such elastic features of Ensemble allow the scheduler to adapt the amount of machines *and* of micro-clouds in use to the workload changes, by dynamically provisioning and de-provisioning resources in an autonomous manner, such that at each point in time the available resources match the current demand as closely as possible.

We note here that in its current state, our prototype still lacks a mechanism to modify on-the-fly the replication factor and the composition of an EnsembleCache. These mechanisms are the same as the ones required to sustain the failure of a single micro-cloud as discussed above. They will be provided for milestone M30.

F61 Versioning (federation)

Core contribution: support for data versioning across micro-clouds.

Initially, we will consider that support for versioned data at the federation level inherits directly from its counterpart at the single micro-cloud level. From an pragmatic point of view, this means that when a versioned datum *(k,u,v)* with value u and version v is put in either a replicated or a distributed EnsembleCache, it is added to all the caches that supports it.

The simple aforementioned implementation requires the versioning mechanism to be aware of the geo-distributed infrastructure. For instance, if this mechanism is based on vector clocks, the dimension of the vector must be equals the total number of Infinispan nodes in the federation. As a consequence, such a naïve approach might be expensive. Consequently in a second step, we will consider a versioning mechanism that takes into account the geo-distributed aspect of the LEADS infrastructure.

F71 Factory of atomic objects (federation)

Core contribution: powerful abstraction to coordinate geo-distributed processes

Compare-and-swap[2] is a core 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 this abstraction is not part of the standard high-level API commonly used via the message-passing paradigm. More typically, distributed systems based on the message-passing paradigm 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 central sequencer, or the Paxos consensus algorithm [Paxos].

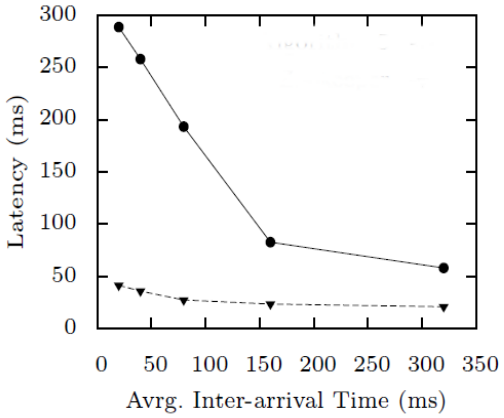


Figure 6: Comparison between Zookeeper (bottom) and a key-value store based implementation (top) of a critical section

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 propose a novel design and implementation of the compare-and-swap primitive on top of the Cassandra distributed key-value store. The key idea is to leverage the fact 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. In what follows, we present two experimental results that assess the benefits of our approach. Appendix D contains the detail of this work.

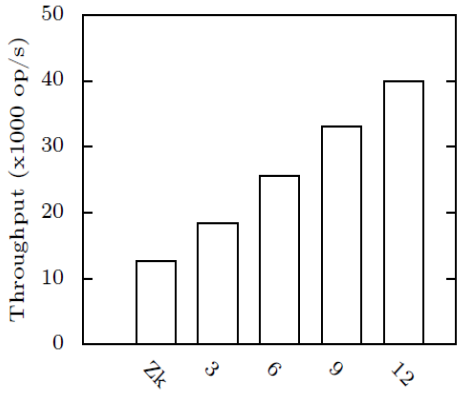


Figure 7: Scalability of a key-value store based implementation of compare-and-swap in comparison to Zookeeper

Error! Reference source not found. and **Error! Reference source not found.** present a comparison of our approach against Zookeeper. These

[2] 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.

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 **Error! Reference source not found.**, we evaluate the time to access a critical section when multiple clients compete for it. This bottom line shows the performance results of Zookeeper when the critical section is implemented via the creation of a znode (an elemental node and coordination object in the ZooKeeper distributed tree abstraction). The upper line 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 **Error! Reference source not found.**, 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 concurrently accessing the critical section increases, Zookeeper can be up to three times faster. This performance difference is the price to pay to have *no centralization point in the system*.

On the other hand, having no execution bottleneck pays off when concurrent clients access disjoint objects. In **Error! Reference source not found.**, 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. Using 3 servers is necessary to tolerate the failure of a single one, and using more servers can only decrease the throughput for write operations. With 3 servers, our system delivers 18.4K op/s , whereas ZooKeeper only delivers 12.6K op/s. This gab is explained by the bottleneck nature of the ZooKeeper leader which serializes all updates. Our prototype achieves 33K op/s when using 9 servers, and 40K op/s with 12. In this last case, our system is 3.2 times faster than Zookeeper on 3 machines.

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. During the M25-M30 period, we plan to further assess them in a geo-distributed context.

F101 Support for deployment and configuration (federation)

Core contribution: tools to deploy LEADS storage across multiple micro-clouds.

Configuring and deploying a distributed service on top of an IaaS (Infrastructure-as-a-service) is a complex administrative task. Not only does it require a fine knowledge of the underlying IaaS which executes the VMs, but it also necessitates a partly automated procedure to avoid, at most, repetitive operations.

Feature C101 (partly described in D2.2) a toolkit to easily deploy and configure a distributed service in a micro-cloud running OpenNebula. This toolkit has been extended to support OpenStack which is the IaaS running on Cloud&Heat micro-clouds. Furthermore, the toolkit gives now access to two additional services: ZooKeeper and BookKeeper, besides the existing Infinispan and Cassandra support. We also provide a set of scripts to emulate a micro-cloud federation. This emulation makes use of the Linux traffic shaping tools [TC] and it can constrain both the latency and the bandwidth between several configurable sets of VMs.

In the section that follows, we present several experiments that demonstrate the capabilities of Ensemble and evaluate its performance. We run these experiments in a controlled environment by emulating a federation of micro-clouds as described above.

5. Evaluation

In this section, we present several experimental results conducted in a cluster of virtualized dual-core machines with 2GB of memory and communicating through a Gigabit network. In these experiments, we emulate a micro-cloud federation that consists of 3 micro-clouds. Each micro-cloud runs 3 Infinispan machines in distributed mode and it uses of a Murmur3-based consistent hashing to locate data. Inside a micro-cloud, the replication factor is set to 1, i.e., a single Infinispan machine holds each data item. At the federation scale, depending on the experiments it varies between 1 and 3.

During our experiments, we use the Yahoo Cloud Serving Benchmark (YCSB) for which we implemented an Ensemble binding. YCSB is a framework with the goal of facilitating performance comparisons of the new generation of cloud data serving systems. In this benchmark, a client machine emulates multiple clients that access a cloud data storage service. The access pattern varies according to (i) the amount of object stored, (ii) the amount of operations executed in total by the clients, and (iii) the nature of this workload, i.e., the read/write ratio and the object popularity distribution.

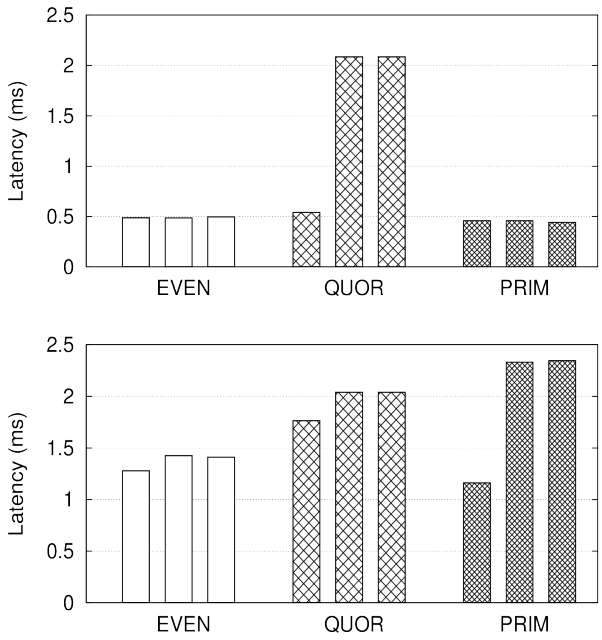


Figure 8: Impact of consistency in Ensemble *(top read operations, bot write operations, rep. factor varies left to right from 1 to 3)*

We evaluate the performance of Ensemble when executing various workloads. We vary the number of clients, the replication factor, the latency between micro-clouds and the consistency degree of the exposed cache. Note here that when Ensemble exposes a weakly consistent cache and the replication factor across micro-clouds equals 1, the system demonstrates performance of an Infinispan deployment in a single micro-cloud.

During an experiment, each client executes around 10^5 operations over 10^5 data items. By default, the YCSB benchmark truncates results in order to obtain a 95% confidence interval. In all our experiments, the client machine was not a bottleneck.

Flat evaluation. Our first set of experiments compares the different consistency degrees available in Ensemble in a basic configuration, that is, when (i) all the communication is flat, i.e., inside and between micro-clouds it relies on the underlying Gigabit network, and (ii) a single client accesses Ensemble. Such an experimental setting offers minimal noise and aims at comparing the algorithmic structures of the 3 designs.

Figure 8 reports the latency of a client operation when executing a single type of operation: in Figure 3(top), only read operations, and conversely in Figure 3(bottom), only write operations. The replication factor of the cache varies, and is set to respectively 1, 2 and 3 from left to right.

In Figure 8(top), we observe that the performance of eventual consistency and primary replication are close. This comes from the fact that in both cases, the client accesses a single remote cache to execute an operation. On the contrary, in a quorum-based implementation, a read first retrieves the latest version of the key from a quorum of replicas then it writes this value back to ensure the consistency of posterior reads. When the cache is replicated, this incurs a large performance penalty over the two other implementations.

Figure 8(bottom) indicates that for every consistency criteria, replicating a cache across several micro-clouds induces a performance cost. As all the put operations are asynchronous, this cost is small when the EnsembleCache is eventually consistent. On the contrary, in a primary-replica based implementation, the cost is higher since the first write has to be synchronous before accessing the other replicas. When the client uses a quorum of remote caches to maintain consistency, the critical path of a write operation is the same as in the case of a read operation, i.e., two round trips. Nevertheless, because these operations are executed in parallel, they can be faster than a primary-based implementation.

Performance of Ensemble. In the following set of experiments, we consider that micro-clouds are linked by a network offering a latency of 15ms and a bandwidth of 10Mb/s. The client machine is in one of the three geographical locations, thus it is local to one of the Infinispan instances. This setting corresponds to a deployment of the LEADS infrastructure spanning a small geographical region. We sum-up it below where the underlined IP address corresponds to the client.

UCloud 1 = (192.168.79.107 192.168.79.108 192.168.79.109 192.168.79.117)
UCloud 2 = (192.168.79.111 192.168.79.112 192.168.79.113)
UCloud 3 = (192.168.79.114 192.168.79.115 192.168.79.116)

During these experiments, we evaluate two types of YCSB workloads:

- Workload A: an update-heavy workload, where half of the operations are writes and the distribution is uniform; and
- Workload B: a read-heavy workload, characterized by a set of 95% of read operations and where the accesses follow a zipfian (skewed) distribution.

The first workload corresponds to a use case where clients attach private information to public data and later retrieve them. The second workload models query intensive operations to compute various metrics on stored data (e.g., PageRank).

	EVEN	QUOR	PRIM
Workload A	**7618**	**363**	**5439**
Workload B	**9641**	**497**	**8361**

Figure 9 – YCSB Performance of Ensemble *(in ops/sec)*

In Figure 9, we report the maximal performance of Ensemble for the two workloads, varying the consistency of the EnsembleCache that supports the operations. The replication degree is set to 2 in the first workload; this corresponds to a setting where private data is replicated for durability. In the second workload, the replication degree is set to 3. Such a setting models that public data is locally available in the micro-cloud before the computation starts.

We observe in Figure 9 that there exists one order of magnitude between the maximal throughput of the quorum-based implementation and the two other implementations. Such a large difference is explained by the fact that none of the operation in this case is local to the micro-cloud where the client executes. On the contrary, for the two other modes, the operations are largely executed locally before they reach the other micro-clouds. When the consistency level of Ensemble is set to eventual consistency, these operations are all executed asynchronously in the background. This explains that the system is in that case faster than when it relies on a primary replication schema. When the load is composed mostly of read operations (Workload B), the price to pay in order to obtain consistent access is nevertheless reasonable (around15%).

6. Conclusion

The present document is the deliverable D2.4 of the LEADS project. This deliverable presents an overview of the current state of the storage layer at M24. This layer consists in a key-value store with extended capabilities to support a federation of micro-clouds. In this document, we first recall the architecture targeted in LEADS, as well as the use cases of the storage layer by higher tiers of the project (WP3 and WP4). Then, we list the key features that are currently available in the storage layer and present the additions we made in comparison to D2.2. In particular, we detail the key/value API at the federation level, the implementation of the explicit data placement and retrieval, the multi-version support and the state of our toolkit for deployment and configuration of the storage layer. Further, we present several experiments in an emulated environment that evaluate the performance of our prototype.

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

[Zk10] 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),

[Hibernate] Hibernate, *hibernate.org*

[Multivers] WP2, multi-version support in Infinispan, Pierre Sutra, (Feb. 2014).

[B-Tree] A Practical Scalable Distributed B-Tree, M.K. Aguilera et al., VLDB'08.

[B-Tree+] Minuet: A Scalable Distributed Multiversion B-Tree, B. Sowell et al., VLDB'12.

[TC] Linux Traffic Shaping Tools, http://tldp.org/HOWTO/Traffic-Control-HOWTO.

[LiveObj] Programming with Live Distributed Objects, Ostrow K. Ostrowski et al., ECOOP 2008.

[SRDS14a] On the Support of Versioning in Distributed Key-Value Stores, P. Felber et al., SRDS 2014.

[SRDS14b] ZooFence: Principled Service Partitioning and
Application to the ZooKeeper Coordination Service, R. Halalai et al., SRDS 2014.

[COMP14] Construction universelle d'objets partagés
au-dessus d'un entrepôt clé-valeur, Sutra, COMPAS'AR.

|Protobuf] code.google.com/p/protobuf

[Avro] avro.apache.org

[Wcount] http://blog.infinispan.org/2014/06/mapreduce-performance-improvements.html

[Jaas] http://docs.oracle.com/javase/7/docs/api/javax/security/auth/Subject.html

[SASL] http://en.wikipedia.org/wiki/Simple_Authentication_and_Security_Layer

[ISPN-999] https://issues.jboss.org/browse/ISPN-999

Appendix A

Below, we provide several examples of the REST API to access Ensemble.

- Register a site
  - Call format
    ```
    POST /sites HTTP/1.1
    Accept: application/json
    Content-Type: application/json
    Content-Length: <size>
    {
      "name" : <site name>,
      "endpoints" : [ <URL1>, <URL2>, ..., <URLn> ]
    }
    ```
  - Arguments

    Parameter *name* specifies the name of the site to be registered. If not specified, a string based on the first endpoint's URL is used. Parameter e*ndpoints* specifies the Hotrod communication endpoints to reach the site. At least one endpoint must be specified.
  - Return format
    ```
    HTTP/1.1 200 OK
    Content-Type: application/json
    Content-Length: <size>
    {
      "name" : <site name>,
      "endpoints" : [ <URL1>, <URL2>, ..., <URLn> ]
    }
    ```

- Inspect site registration / fields
  - Call format
    ```
    GET /sites/<site name> HTTP/1.1
    Accept: application/json
    ```
  - Comment

    The return format is the same as for registering the site.

- Create an ensemble cache
  - Call format (for ReplicatedEnsembleCache)
    ```
    POST /caches HTTP/1.1
    Accept: application/json
    Content-Type: application/json
    Content-Length: <size>
    {
      "name" : <new cache name>,
      "replication" : <replication factor>,
      "consistency" : <consistency level>,
      "sites" : [ <site1>, <site2>, ..., <siteN> ]
    ```

```
}
```

Arguments

All arguments are optional. Parameter *name* specifies the name of the cache to be created. By default, a unique random name is generated. Parameter *replication* specifies the number of sites (micro-clouds) to use to fully replicate. By default, one site is used. Parameter *consistency* specifies the consistency level of the created cache. Possible values are: SWMR, linearizability with a single writer and multiple readers, MWMR, linearizability with a multiple writers and multiple readers, WEAK, no consistency, being this the default value. Finally, parameter *sites* explicitly specifies the names of the sites to use for replication. By default, a random selection is used.

Return format

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: <size>
{
   "name" : <new cache name>,
   "replication" : <replication factor>,
   "consistency" : <consistency level>,
   "sites" : [ <site1>, <site2>, ..., <siteN> ]
}
```

Comment

All the actual values used in the cache description fields are returned to the caller.

Call format (for DistributedEnsembleCache)

```
POST /caches HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: <size>
{
   "name" : <new cache name>,
   "caches" : [ <cache1>, <cache2>, ..., <cacheN> ]
   "partitioner" : {
      "url" : <partitioner URL>,
      "parameters" : {
         <parameter1>,
         <parameter2>,
         ...,
         <parameterN>
      }}}
```

Arguments

Parameter *name* specifies the name of the cache to be created. It is optional, and by default, a unique random name is generated. Parameter *caches* specifies the names of the caches that compose the collection. Parameter *partitioner* defines an object that splits the key space into partitions (in other words, it maps every possible key to one of the given caches in the collec-

tion). It is defined by an URL (usually a class in the server class path) and a number of parameter passed to the object constructor.

Comment

Several standard partitioners are provided. For instance, the uniform partitionner takes as input a list of micro-clouds and a regular expression. It ensures that each micro-cloud holds an equal share of the keys satisfying the expression.

Appendix B to D : three scientific papers (attached as PDFs).

REFERENCES?

On the Support of Versioning in Distributed Key-Value Stores

Pascal Felber, Marcelo Pasin, Étienne Rivière,
Valerio Schiavoni, Pierre Sutra
University of Neuchâtel, Switzerland
first.last@unine.ch

Fábio Coelho, Rui Oliveira,
Miguel Matos, Ricardo Vilaça
HASLab, INESC TEC & U. Minho, Portugal
fabio.a.coelho@inesctec.pt
{rco,miguelmatos,rmvilaca}@di.uminho.pt

***Abstract*—The ability to access and query data stored in multiple versions is an important asset for many applications, such as Web graph analysis, collaborative editing platforms, data forensics, or correlation mining. The storage and retrieval of versioned data requires a specific API and support from the storage layer. The choice of the data structures used to maintain versioned data has a fundamental impact on the performance of insertions and queries. The appropriate data structure also depends on the nature of the versioned data and the nature of the access patterns. In this paper we study the design and implementation space for providing versioning support on top of a distributed key-value store (KVS). We define an API for versioned data access supporting multiple writers and show that a plain KVS does not offer the necessary synchronization power for implementing this API. We leverage the support for listeners at the KVS level and propose a general construction for implementing arbitrary types of data structures for storing and querying versioned data. We explore the design space of versioned data storage ranging from a flat data structure to a distributed sharded index. The resulting system, ALEPH, is implemented on top of an industrial-grade open-source KVS, Infinispan. Our evaluation, based on real-world Wikipedia access logs, studies the performance of each versioning mechanisms in terms of load balancing, latency and storage overhead in the context of different access scenarios.**

***Keywords*-versioning, key-value store, listeners.**

I. INTRODUCTION

Applications processing massive amounts of data favor storage on key-value stores (KVSs) over traditional relational databases for their much better scalability. Some of these applications are based on a computational model that considers the evolution of data over time, in the form of *versioned* data. We consider the following motivating examples.

Business intelligence extraction can be performed on periodic crawls of the Web graph. Such analysis may consider the evolution of mentions of products and other assets on Web pages, analyze trends, track the origin of data and rumors, or try to determine Web influencers. These are made possible by storing for each page, the different versions obtained with successive crawls.

Collective and collaborative editing platforms such as Wikipedia naturally deal with, and give access to, versioned data. Most reads are for the latest version of a given page. The ability to access previous versions is nonetheless required by the access model, in order to be able to compare versions and restore content that has been deleted by mistake. The ability to access previous versions also allows mining complex information from past states of a wiki, e.g., to detect trends in vocabulary usage.

Other examples of applications that directly rely on versioned data include log mining, forensics and generally data mining for large sets of unstructured data where versions of the data for time windows in the past are considered. Web content based on a timeline-dependent set of information, such as blogs and Twitter streams archives, also form naturally versioned data.

For each of our motivating applications, the data store must not only store and expose the latest version of the data associated with a given key but a very large number of versions may exist for each key, which must be persistently stored by the versioned KVS. An order relation - given by the semantics of the application - between these identifiers allow operations based on versions ranges.

In this paper we are interested in the case where versioning support is *explicitly* part of the data model and exposed by the API of the KVS. We name such a KVS a *versioned KVS*. We consider more specifically the construction of *multiple-writer* versioned KVS, where several clients may write new versions to any given key concurrently. This choice is driven by our motivated examples, where updates may come from multiple concurrent and un-synchronized clients.

We note that, for concurrency control, distributed KVSs already associate version numbers to updated values of the same key. These versions are used internally. At any given time, a small number of them may exist for each key. KVSs offering weak consistency models such as eventual consistency may expose these versions to the applications upon reads, to let the application reconcile multiple unordered updates. These small set of co-existent values for a given key are temporary in nature and do not correspond to our requirement of storing large sets of versions in a persistent and long-lived manner. The notion of versions exposed in the data model is actually independent of the notion of versions used for concurrency control. The two mechanisms can actually co-exist. In an eventually-consistent versioned KVS, a particular version might temporarily be associated with several values, and exposed to the application for reconciliation upon a read. This paper only considers

versioned KVSs offering strong consistency support.

There are several design options available for implementing a versioned KVS. These options differ in the level at which versioning is implemented. The KVS can be oblivious to versioning, and the logic of maintaining multiple versions for a key can be handled by the clients. The implementation can also be integrated in the KVS itself, either by means of objects exposing the versioning semantics on a single node, or by means of explicit indexes stored in the KVS which is then aware of the access semantics. Cost and interest of these various solutions differ depending on the number of versions per key, the size of the objects, the number of keys, the access pattern (mostly read, mostly writes, read/write), and the nature of these accesses (for example, whether most accesses are to the latest version or not, whether appends happen at the tail or not, or even if accesses to ranges of versions dominate).

A. Contributions

In this paper, we explore the design options available for building a versioned KVS on top of an existing KVS. To support versioning, the implementation needs to maintain specific data structures. In some of our designs, the KVS needs to be aware of the semantics of these structures. Our first contribution is to prove that a KVS API providing solely $put()$ and $get()$ operations does not have the necessary synchronization power to implement multi-writer versioning. Our second contribution is to describe the construction of atomic objects of any type on top of a KVS enriched with support for listeners. Our third contribution is ALEPH, a generic versioning system atop such a listenable KVS. Our final contribution is to perform a thorough evaluation of ALEPH under a real workload from Wikipedia access traces. The evaluation highlights the inherent tradeoffs of each implementation, from atomic maps to tree-based and sharded tree-based indexes, and the specific adequacy to different workload patterns.

B. Outline

The remainder of this paper is organized as follows. Section II reviews related work and the support of versioning in existing KVSs. In Section III, we define an API for a versioned KVS and prove, in Section IV, that a plain KVS is not sufficient to implement a multi-writer versioned KVS. We then define the notion of a listenable KVS, upon which we build a universal construction. We describe our different alternative implementations of versioning in Section V. In Section VI, we evaluation ALEPH and discuss the results. Section VII concludes the paper.

II. RELATED WORK

This paper addresses the ability to access and query data with a potentially very large number of versions that may exist for a long time. In particular, we are interested in explicit support for versions and mechanisms to retrieve ranges of versions, including the latest one.

Temporal databases [9–11] provide specific support for storing, querying, and updating historical data. Commercial database management systems (DBMSs), such as IBM DB 10, Oracle Database 11g and Teradata, recently introduced temporal-database features in the form of SQL extensions, based on SQL:2011 [12]. According to the SQL:2011 standard, a table can be an application-time period table, a system-versioned table, or both [11]. Application-time period tables are useful to capture periods where data is valid. System-versioned tables are useful to maintain an accurate history of data changes and thus is similar to the multi-version support we target in this paper. Queries on system-versioned tables retrieve the table content at a given timestamp, (e.g: `FOR SYSTEM_TIME AS OF TIMESTAMP t`), or between a range of timestamps, (e.g: `FOR SYSTEM_TIME BETWEEN TIMESTAMP t1 AND TIMESTAMP t2`). System-versioned tables are highly coupled to relational databases, where timing information is added as metadata to tables, and requires deep modifications to an existing DBMSs.

Most distributed KVSs lack any support for long-lived versioned values, while others only offer limited multiple-versions support. Table I presents a brief comparison of versioning support in several popular KVSs.

KVSs using multi-version concurrency control (MVCC) [13] usually associate version numbers to data, typically using timestamps or version vectors [14]. However, versions are used internally, and applications have no access to the full history of an object. Loosely consistent KVSs may even expose at any given time a small number of versions to the applications, so they are able to reconcile with multiple updates.

Cassandra [1] columns have timestamps used for conflict resolution and relies on the Last-Writer-Wins (LWW) approach [15]. Dynamo and Riak are loosely consistent KVSs and may expose concurrent versions to the application. The $put()$ operation of Dynamo [7] receives a version; the $get()$ operation optionally returns a list of objects with conflicting versions. A read operation in Riak [2] by default returns the most recent version, using vector clocks: clients need to resolve conflicts when needed.

MongoDB [4] is a document database without any built-in support for multi-version. Mongo MVCC [5] offers MVCC atop MongoDB but in contrast to most MVCC implementations it keeps a complete history of old data, enabling access to older versions of all documents at any time. Mongo MVCC uses the principles from distributed version control systems, such as Git, allowing the creation of branches containing different versions of documents. Mongo MVCC by default hides old versions of documents: users can recover them using their unique identifier (the commit's ID).

Apache HBase [6] is a distributed KVS with a versioned data model and architecture inspired by BigTable [16]. In

Name	Historic versions	Multi-Versioning Support Technique
Cassandra [1]	No	Columns have timestamps that are used for conflict resolution.
Riak [2]	No	Use vector-clocks by default. Can be disabled and fall back to timestamps based on LWW.
CouchBase and CouchDB [3]	No	MVCC. Conflicts must be solved by the application. Old versions are discarded upon file-compactation operations.
MongoDB [4]	Yes, with MVCC [5]	Support for versioned branches.
HBase [6]	Yes	TTL associated with each revisions. Upon expiration, row is trashed.
Dynamo [7]	No	Timestamps and eventual consistency based on LWW.
HyberTable [8]	Yes	Configurable number of managed versions, stored in reverse-chhronological order. Query predicates can filter versions.

Table I
CLASSIFICATION OF KVSS AND THEIR SUPPORT TO LONG-TERM DATA VERSIONING. LWW=LAST-WRITER-WINS, MVCC=MULTI-VERSION CONCURRENCY CONTROL

HBase, the maximum number of versions can be defined per table and versions are stored in descending order. Read operations, $get()$ and $scan()$, can specify the quantity or the range of versions to be retrieved. HyperTable [8], another Bigtable's clone, never discards old versions.

Most KVSs lack proper long-term versioning support, including an API to access sorted ranges of versions. In the remainder of this paper we study the design space for versioning support on top of unmodified distributed KVS, and we propose a framework based on universal atomic objects to maintain the data structures needed for versioning.

III. VERSIONED KEY-VALUE STORE

This section defines the notion of versioned data and the interface of a versioned KVS. Accesses to such a data store are made through the $put()$ and $get()$ operations of a *plain* KVS extended with capabilities for a client to retrieve past versions. Below, we also list a set of desirable properties for a versioned KVS that serve as guidelines in our implementation.

A. Notion of Versioned Data

We are interested in any type of versioned data that might be stored in a KVS. To model this, we consider three abstract sets: a set of keys $\mathcal{K}$, each key identifying a *datum*, a set of values $\mathcal{U}$, and a set of versions $\mathcal{V}$. A tuple $(k, u, v) \in \mathcal{K} \times \mathcal{U} \times \mathcal{V}$ is called a *versioned datum*.

Clients of the data store create versioned data, e.g., (k_1, u_1, v_1) and (k_2, u_2, v_2) along time. To capture this, we assume the existence of a *version order* $<$ such that $(\mathcal{V}, <)$ is a bounded join-semilattice, i.e., a partially ordered set ensuring that(i) given any two elements $v, v' \in \mathcal{V}$, the least upper bound, or *join*, of v and v' is in $\mathcal{V}$, and (ii) $\mathcal{V}$ contains some smallest element v_0, named the initial version.

Numerous instances of the above abstraction $(\mathcal{V}, <)$ have been proposed in the past. These include timestamps [14], vector clocks [17], version vectors [18, 19], or more recently, version vectors with exception [20] and interval tree clocks [21]. Depending on how concurrency is tracked and for which purposes, the dimension of $\mathcal{V}$ may vary from a single dimension to the size of the data set ($\mathcal{K}$ in our case), the number of storage nodes, or even the number of clients.

Algorithm 1 Versioned KVS Interface

1: $put(k : \mathcal{K}, u : \mathcal{U})$
2: **post:** **let** $v = \sqcup\{(k, \_, v') \in S\}$
3: $S \leftarrow S \cup (k, u, v)$
4:
5: $put(k : \mathcal{K}, u : \mathcal{U}, v : \mathcal{V})$
6: **pre:** $\forall (k, \_, v') \in S : v' < v$
7: **post:** $S \leftarrow S \cup (k, u, v)$
8:
9: $get(k : \mathcal{K}) \rightarrow u : \mathcal{U}$
10: **pre:** $(k, u, v) \in S \wedge \forall (k, \_, v') \in S : \neg\ (v' > v)$
11:
12: $get(k : \mathcal{K}, v : \mathcal{V}) \rightarrow u : \mathcal{U}$
13: **post:** $(\_, u, v) \in S$
14:
15: $getRange(k : \mathcal{K}, v_1 : \mathcal{V}, v_2 : \mathcal{V}) \rightarrow R : 2^{\mathcal{U} \times \mathcal{V}}$
16: **pre:** $v_1 < v_2$
17: **post:** $R = \{(u, v) \mid (k, u, v) \in S \wedge v_1 \leq v \leq v_2\}$
18:
19: $getSuccessor(k : \mathcal{K}, v_1 : \mathcal{V}) \rightarrow (u, v) : \mathcal{U} \times \mathcal{V}$
20: **pre:** $(u, v) \in S \wedge v < v_1 \wedge \forall (k, \_, v') \in S : \neg\ (v' < v)$
21:
22: $getPredecessor(k : \mathcal{K}, v_1 : \mathcal{V}) \rightarrow (u, v) : \mathcal{U} \times \mathcal{V}$
23: **pre:** $(u, v) \in S \wedge v > v_1 \wedge \forall (k, \_, v') \in S : \neg\ (v' > v)$
24:

B. Versioned KVS: API Definition

A *versioned* KVS consists in a set of storage nodes that offer an API to its clients to access versioned data. Clients can add a new datum, add a new version of it, retrieve a specific version, or retrieve a range of values spanning a range of versions. We consider that a versioned KVS is an automata whose initial state S consists in an empty set of versioned data. Algorithm 1 details the semantics of the interface that clients employ to access the store. This interface defines the following set of operations:

- $put(k, u)$ adds (k, u, v) to the store, where v is the least upper bound (denoted $\sqcup$) of all existing versions of k;
- $put(k, u, v)$ adds the versioned datum (k, u, v);
- $get(k)$ returns (any of) the latest value stored at key k;

- $get(k, v)$ returns the value of key k stored at version v;
- $getRange(k, v_1, v_2)$ returns all data versions at key k whose versions falls into the range $[v_1, v_2]$;
- $getPredecessor(k, v_1)$ returns (any of) the latest versioned datum whose version is lower than v_1; and
- $getSuccessor(k, v_1)$ returns (any of) the earliest versioned datum whose version is greater than v_1.

As pointed out in Table I, existing KVSs offer some features to support versioning. They implement all or part of the interface described in Algorithm 1. Consistency of this interface varies from one store to another. Cassandra [1] exposes $put(k, u, v)$ and $get(k, v)$ operations using timestamps provided by clients; this interface is either sequential or eventual consistency. The clients of Riak [2] can use version vectors and dotted version vectors to track changes. In both cases, versions are only exposed to reconcile storage nodes that replicate the same datum at the application level. INFINISPAN [22] does not offer built-in support for data versioning. However, like in many others KVSs, its data model supports secondary indexes and clients may execute range queries on them.

C. Design objectives

Clients of different KVSs have different capabilities and guarantees when querying and retrieving versioned data. Nevertheless, a careful examination of existing versioned KVSs reveals a set of common properties that any implementation should offer. Below, we list these essential properties.

(Progress) Clients of different applications may concurrently access the same KVS. As a consequence, we require that calls to the interface are wait-free, i.e., they return after some bounded amount of time regardless of what the other clients do.

(Multi-writer) The versioned KVS should allow multiple writers to insert different versions of a datum concurrently.

(Scalability) A versioned KVS should support a large number of versions.

(Performance) The time complexity of any operation of the interface should be sublinear in the number of stored versions.

(Load-balancing) The amount of versions of some datum on the storage nodes should be as balanced as possible even when the distribution of versions per key is highly skewed.

These properties serve as design objectives for the different versioned KVS implementations detailed in Section V. We shall also use them in our empirical comparison in Section VI.

Consider a naive versioning mechanism where all the versions of a datum indexed by k are stored as a blob under key k, retrieved as such, updated locally and re-submitted to the KVS. This mechanism is not appropriate as(i) it does not offer any load balancing. (ii) the more versions it stores, the more it is expensive, and (iii) it does not support concurrent writters - if two versions are written concurrently, one of them might be lost. These osbervations examplify the importance of the properties we defined above. In the next section, we further refines them by characterizing the synchronization power of a versioned KVS.

IV. UNIVERSAL CONSTRUCTION ON A LISTENABLE KVS

After defining the versioned store API in the previous section, we now explore whether a plain KVS can implement this interface or if additional mechanisms are required. This question is of practical importance as it allows determining the nature and complexity of the mechanisms required to support data versioning. We contribute an impossibility result: the construction of a versioned store on top of a plain KVS is impossible as it requires a synchronization power strictly greater than what a plain KVS allows. Then, we present an augmentation of a plain KVS that overcomes this limitation, in the form of a listenable data store with the ability for clients to follow the modifications occurring on the store through remotely registered listeners. Finally, we describe how the properties of a listenable store can be used to propose a universal construction that allows building any strongly-consistent shared object on top of it. We use this universal construction to build and maintain versioning information with various data structures in Section V.

A. The Separation Result

We start by showing that it is impossible to build a versioned store on top of a plain KVS. To that end, we first prove that a strongly-consistent wait-free versioned KVS can solve consensus for any number of participants, i.e., that the consensus power of the interface described in Algorithm 1 is infinite. Then, we show that the consensus power of a plain KVS is one, leading to a separation result.

Let us first recall that in consensus, processes propose values and must reach agreement on one of them. More precisely, consensus is defined by the $propose()$ operation which takes as input a *proposed* value and returns some *decision*. Every run of consensus satisfies the following properties. *(Termination)* Every correct process eventually decides some value. *(Integrity)* Every process decides at most once. *(Validity)* If a process decides v, then v was proposed by some process. *(Agreement)* Two processes can't decide differently. The consensus power of a shared object o is the maximum amount of processes that may solve consensus with atomic and wait-free shared objects of the same type than o and registers. Herlihy [23] shows that this hierarchy is strict for shared objects, in the sense that if object o has a consensus power of n, it cannot implement consensus for $n + 1$ processes.

Theorem 1: If $(\mathcal{V}, <)$ is a totally ordered set then the consensus power of the versioned store interface is infinite.

Proof: We consider an asynchronous system of n processes $\{p_i, \ldots, p_n\}$, and for each process $p_{i \in [\![1,n]\!]}$, we

note u_i the value proposed by p_i. Let k be some key. Every process p_i executes the following code to solve consensus: Upon a call to $propose(u_i)$, process p_i executes $put(k, u_i)$. Then, it fetches the content of $getSuccessor(k, v_0)$ in the pair (u, v) and decides the value stored in u.

Consider some history h of the above algorithm, and note l the linearization of the calls to $put(k, u_i)$ and $getSuccessor(k, v_0)$ that appear in h. First, we observe that the value returned by $getSuccessor(k, v_0)$ is necessarily proposed by one of the participating processes, and that every process decides at most once. This proves that our algorithm ensures the Validity and Integrity clauses of consensus. Then,, for any process p_i, a call to $put(k, u_i)$ appears before $getSuccessor(k, v_0)$. As a consequence, the precondition of $getSuccessor(k, v_0)$ holds and every call $getSuccessor(k, v_0)$ by some correct process returns in h. This shows Termination. Finally, observe that any complete call to $getSuccessor(k, v_0)$ returns the value u_j for which the corresponding operation $put(k, u_j)$ appears first in the linearization l. As a consequence, Agreement holds. ■

A shared memory can implement the operations $put(k, v)$ and $get(k)$ of a plain KVS. As a consequence, the FLP impossibility result [24] tells us that the consensus power of such an interface is one. From the strictness of Herlihy's hierarchy [23], we deduce the separation result that a plain KVS cannot implement a versioned KVS.

B. Listenable Key-Value Store

At the light of Theorem 1, we have to augment the synchronization power of the plain KVS interface to support versioned data. One solution would be to add some strong synchronization primitive at the interface, such as a compare-and-swap operation. However, this choice is not appealing since (i) it requires a complex helping mechanism to ensure progress of operations under contention, and (ii) a synchronization primitive would not leverage the client-server nature of the interface. In this paper, we consider another possibility, which is clients being able to listen to modifications made to the store.

We define a *listenable* KVS as a plain KVS augmented with the following operations:

- $regListener(f, k)$: it registers the function f as a listener of the modifications occurring on key k. Every time the KVS executes a put operation on k, the callback $f(k, u)$ is executed, where u is the new value of k.
- $unregListener(f, k)$: to unregister the callback f.

In the remainder of this section we assume that operations of such a listenable KVS are linearizable. This means that the $put()$ and $get()$ operations behave like in an atomic register, and that once a callback is registered, it gets notified of all the modifications according to the linearization order in which they occur.

A universal construction [23] is an algorithm to share atomically any sequential code. The next section explains how to implement this construction on top of a listenable KVS. In Section V, we use this universal construction to build a versioned data store.

Algorithm 2 Universal Construction – code at process p

```
 1: Shared Variables:
 2:     K                                              // Listenable KVS
 3:
 4: Local Variables:
 5:     s ∈ States                                     // initially ⊥
 6:     r ∈ Values                                     // initially ⊥
 7:     Q                                  // a FIFO queue; initialy ⊥
 8:
 9: open(k)
10:     K.regListener(callBack)
11:     (x, _, f) ← get(k)
12:     if (x, _, f) = ⊥ then
13:         s ← s_0
14:     else if f = PER then
15:         s ← x
16:     else
17:         K.put(k, (⊥, p, RET))
18:         wait until s ≠ ⊥
19:
20: close(k)
21:     K.put(k, (s, p, PER))
22:     K.unregListener(callBack)
23:     s ← ⊥, r ← ⊥, Q ← ⊥
24:
25: invoke(k, op)
26:     r ← ⊥
27:     K.put(k, (op, p, INV))
28:     wait until r ≠ ⊥
29:     return r
30:
31: When callBack(k, (x, p', f))
32:     if f = INV then
33:         if s ≠ ⊥ then
34:             (s, v) ← τ(s, x)
35:             if p = p' then
36:                 r ← v
37:         else if Q ≠ ⊥ then
38:             Q ← Q ∘ ⟨x⟩
39:     else if f = RET then
40:         if s ≠ ⊥ then
41:             K.put(k, (s, p, PER))
42:         else if p = p' then
43:             Q ← ⟨⟩
44:     else if f = PER ∧ s = ⊥ then
45:         s ← x
46:         for x ∈ Q do              // In the order defined by Q.
47:             (s, v) ← τ(s, x)
48:
```

C. Universal Construction

Our construction is a variation of the seminal state machine replication approach [25]. It allows sharing any sequential data type between multiple processes with linearizability semantics [26]. In what follows, we first recall the formal definition of a (sequential) data type and then we detail our

construction on top of a listenable KVS.

A sequential data type is an automaton defined by: a set of states $States$, an initial state s_0 in $States$, a set of operations Ops, a set of response values $Values$, and a transition function $\tau : States \times Ops \rightarrow States \times Values$. Hereafter, and without lack of generality, we shall assume that every operation op is *total*, meaning that $States \times \{op\}$ is in the domain of τ.

We present our universal construction in Algorithm 2. At any process p, our algorithm maintains the following four variables: K represents the listenable KVS, s is the logical state of the shared object at process p, r is a reference to the response value of the last local call issued by p, and Q is a FIFO queue. Initially, process p assigns a null value ($\perp$) to all local variables.

As mentioned previously, the core of our construction inherits from the state machine approach. When a process p invokes an operation op on a shared object o, op is transmitted via a $put(k, (op, p))$ to the KVS, where k is the unique key identifying o. Upon an execution of the callback function $callBack(k, (op, p'))$, operation op is applied to the local copy of object o. Then, in case op is registered as a local call to o, i.e., $p = p'$, the response value is returned to the calling process.

This approach offers consistency, durability and it allows processes to create and destroy shared objects. To this end, the variable K stores tuples of the form (x, p, f), where (i) x is either an operation, or an object state, (ii) p identifies the process that executed this insertion on the KVS, and (iii) f is a flag that indicates the type of the insertion. An insertion flagged with `INV` indicates that process p called object o, and in such a case, x is an operation. If f equals `RET`, process p aims at retrieving the persistent state of the shared object. Such a state s is forwarded by another process that opened previously object o via an insertion of the form $(s, p, \texttt{PER})$ in the store.

Process p opens an object when it executes the operation $open()$. This call registers the callback function $callBack()$, then retrieves the tuples stored in the KVS at key k. Three different cases may occur:

1) The KVS does not contain any value at key k (line 12). In such a case, p assigns to s the initial state s_0 of the object.
2) If now the tuple retrieved in the KVS is of the form $(x, \_, \texttt{PER})$ then x is an object state and p assigns x to variable s (line 15). Notice here that s is precisely the object state after applying all the operations linearized before process p opens object o. The registration of $callBack()$ in the KVS before the operation $get()$ ensures that p keeps track of the state for all the operations that occur after $open()$ in the linearization order.
3) Finally, when the tuple stored at key k does not contain an object state, the process waits until another process transmits such an information (lines 17 and 18). This is achieved by (i) storing a request flagged with `RET` in the KVS, (ii) initializing Q to the value of an empty list (line 43), (iii) storing all the calls to o that occur after the opening request of p (line 38), and (iv) once the state is retrieved, applying these operations in the order defined by Q to variable s (lines 44 to 47). In case no process is available, the opening fails and process p is notified by an exception (not described in the code of Algorithm 2.)

When the process p stops accessing object o, it executes the operation $close()$. This operation inserts a tuple $(s, p, \texttt{PER})$ inside the KVS. Then, it unregisters the callback function. Finally, local variables are erased (lines 21 to 23).

The listenable KVS ensures durability of objects in case processes properly close them. Nevertheless, if the last process that opens the object crashes, this property is lost. To avoid this situation, we require that at least $F + 1$ processes have the object opened at any point in time, where F is the maximal amount of crashes that may occur during an execution.

Notice that we may improve performance of our construction by considering sequentially consistent objects. To achieve this, we proceed as follows. We annotate every operation with a flag indicating if it modifies, or not, the object. When an operation op is called, in case op is read-only, we apply it locally and return the result to the calling process. A less intrusive approach consists in cloning the state of the object, execute tentatively the call on the copy, and return immediately the result in case the state does not change.

V. Implementation of Versioning Support

In this section we decribe ALEPH, a generic versioning support for any listenable KVS, as well as three representative versioning implemented within. Each mechanism offers different guarantees in terms of load balancing, latency and storage overhead. We present an extensive evaluation of these mechanisms and their inherent tradeoffs in Section VI.

A. Overview

The architecture of ALEPH consists in two tiers (see Figure 1). The *storage nodes* of the KVS form the bottom tier. They expose a listenable KVS interface, and ALEPH uses them to store both versioned data and version indexes. The upper tier executes operations on indexes using the universal construction described in Section IV. This indexation tier is generic and we present several variations in the following. Clients communicate with the indexation tier using remote procedure calls to store and query versioned data (Figure 1-❶). Upon receiving the remote procedure call, the contacted indexing node issues an operation on the appropriate index, by means of the universal construction presented in Section IV. This event triggers a chain of operations at the storage nodes level (Figure 1, steps ❷ and ❸). Finally, the indexing node

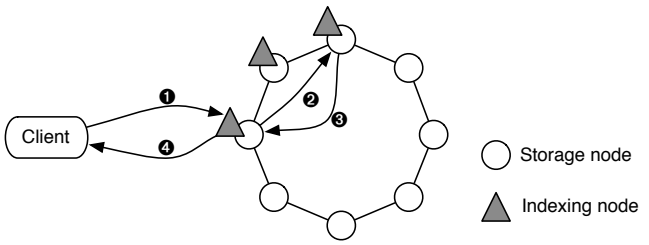


Figure 1. General architecture of the versioned KVS.

returns the response to the calling client (Figure 1-❹). ALEPH collocates each indexing node with a storage node. This design choice improves performance since data can transit in a shared memory space. Notice that, nevertheless, this is not mandatory in our architecture.

B. Storage

ALEPH can potentially use any listenable KVS as the storage layer. The evaluation presented in Section VI uses INFINISPAN [22]. INFINISPAN is a simple yet efficient one-hop DHT that relies on consistent hashing to store and locate data. In more details, it supports the following features:

(Routing) INFINISPAN uses a one-hop routing design, i.e., every node knows all storage nodes in the ring.

(Elasticity) Upon joining, a node chooses a random identifier along the ring and fetches the ring structure from some other DHT node. It then informs its neighbors that it is joining.

(Storage) The storage layer uses consistent hashing [27] to assign blocks to nodes with a replication factor r: a data block with a key k is stored at the r nodes whose identifiers follow k on the ring.

(Reliability) INFINISPAN builds on the JGroups communication library [28]. This library relies on failure detectors to maintain a consistent view of the system. The repair mechanisms of consistent hashing are triggered upon a lack of response of a storage node within a timeout.

(*Consistency*) INFINISPAN implements the listenable KVS interface with sequential consistency gurantees. INFINISPAN achieves this by using primary-backup replication and the ability for clients to execute a $get()$ operation at any of the replicas. Events are forwarded by the primary replica to the registered listeners. Upon a primary change, an idempotency mechanism guards the application against duplicated events.

ALEPH is implemented in 3,146 SLOC of Java, i.e. an increase of 1% over the INFINISPAN base-code.

C. Indexation

Aside from the storage nodes, ALEPH employs a set of *indexing nodes*. Each indexing node exports the versioned KVS interface presented in Algorithm 1. Clients initially retrieve the list of indexing nodes, and randomly choose one of them to connect. Clients then access the interface through remote method invocations to their indexing node. At each indexing node, a *versioning mechanism* maps operations on Algorithm 1 to appropriate accesses on versioned data indexes. The choice of the versioning mechanism implemented by the indexing nodes of ALEPH is configurable at start time. In the remainder of this section, we detail three representative mechanisms.

Algorithm 3 Tree-based Versioning – code at process p

```
1:  Shared Variables:
2:      K                                          // Listenable data store
3:
4:  put(k, u, v)
5:      choose some unique key l
6:      K.put(l, u)                                // store value u at key l
7:      T ← K.open(k)                              // open the tree stored at key k
8:      T.add(v, l)                                // update the tree
9:      K.close(k)                                 // close the tree
10:
11: get(k)
12:     T ← K.open(k)                              // retrieve the tree
13:     (_, l) ← T.last()                          // compute the latest entry
14:     K.close(k)                                 // close the tree
15:     return K.get(l)                            // return the corresponding value
16:
```

Baseline: The first versioning mechanism we consider is the naive algorithm presented at the bottom of Section III-C. All the versions are stored in a sorted map, under the key identifying the corresponding datum. Everytime a versioned operation is executed, the map is entirely fetched from the KVS then updated accordingly. Since every versioned operation requires at most two accesses to the listenable KVS API, this versioning mechanism is optimal when the amount of versions per datum is small. On the other hand, when the number of versions is large, this versioning mechanism is expensive as it requires to retrieve all existing versions. Furthemore, it does not support concurrent writers, nor does it offer load balancing.

Tree-based Consistent Indexes: ALEPH supports a second versioning mechanisms that builds sorted trees to index the versions with the universal construction depicted in Section IV. Algorithm 3 details this approach for the most relevant operations of the versioning interface. This algorithm indexes the versions of a datum inside a dedicated tree (when versions are non-comparable a canonical order is chosen). To execute a versioned operation on behalf of a client, an indexing node opens the tree of versions, invokes the corresponding operation on the tree, then closes it. With more details, to implement a call to $put(k, u, v)$, the indexing node first picks some unique key l at which it stores the value u. Then, the node opens the tree T stored at key k and adds the pair (v, l) to T before closing it (lines 5 to 9). When retrieving the latest version of some datum k, the indexing node first computes the greatest entry $(\_, l)$ in the tree T (line 13), and

returns the value stored at key l in the KVS (line 15). For the performance reasons we detailed in Section IV-C, ALEPH implements this versioning mechanism with sequentially consistent trees. Moreover, to save the cost of registering a listener for read-only operations, indexing nodes postpone the installation of listeners (Algorithm 2, line 10) until a modification occurs.

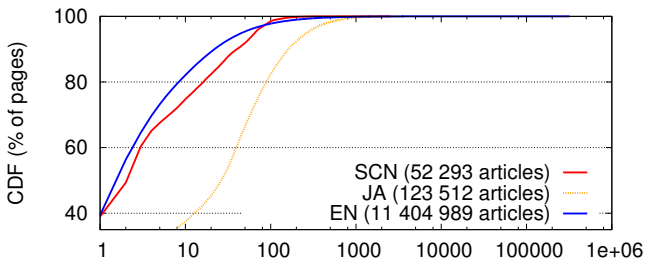
Sharding the Trees: As we shall see in practice in Section VI, the tree-based versioning mechanism works fine in most cases, but fails for data having a large number of versions. We describe a versioning mechanism that overcomes this limitation by scattering the different versions into multiple trees. In detail, for each datum k ALEPH makes use of one *sharded tree* stored at key k. A sharded tree consists in a sorted map $M = \{(v_i, T_1), (v_2, T_2) \ldots\}$ of trees distributed and replicated in the storage layer. The tree T_i stores the version of k that are greater are equal to v_i, but smaller than the version v_{i+} indexing tree T_{i+1}. The sorted map M, as well as the trees referenced by M are implemented using the universal construction introduced in Section IV-C. Upon the insertion of a pair (u, v) in the sharded tree, the indexing node retrieves the map of trees and finds the last tree T whose version v' is smaller than v and adds (u, v) to T. Then, if v is smaller than the version v' referencing T in M, v' is updated with v in M. In case T contains more than κ elements, the greatest tuple (u_m, v_m) in T is removed, and added to the successor of T in M. If such a successor S does not exists, the indexing node creates it in M. Upon the retrieval of one or more versioned data in T, e.g., when executing $getRange(k, v_1, v_2)$, the indexing node exploits the fact that, at any point in time, the trees in M are both disjoint and sorted.

VI. EVALUATION

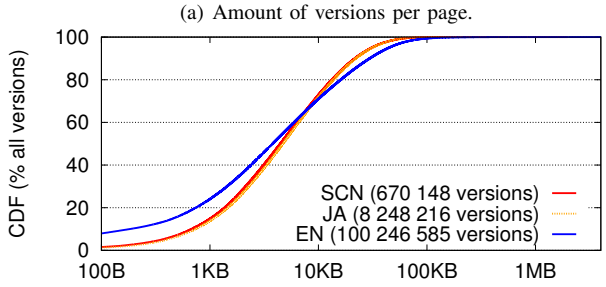
Our evaluation consists in re-executing real access traces on a Wikipedia dump stored by ALEPH. We ran our experiments on a cluster of 24 virtualized 4-core Xeon 2.5 Ghz machines with 4GB of memory, running Gentoo Linux 32bits, and connected by a virtualized 1 Gbps switched network. Network performance, as measured by *ping* and *netperf*, is of 0.3ms for a round-trip with a bandwidth of 117MB/s. Clients runs a modified version of YCSB [29] that replays Wikipedia access traces on the interface defined in Algorithm 1. In the remainder of this section we study the workload properties, discuss the modifications implemented in YCSB, and finally present our evaluation results along several dimensions.

A. Workload Characteristics

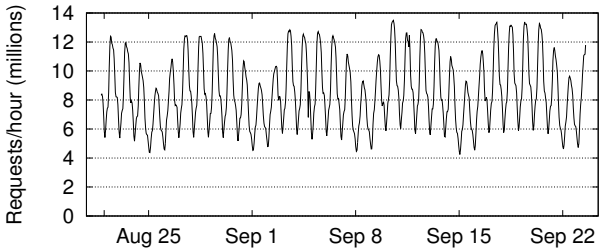
We use the dump of Wikipedia as of January 3rd, 2008, published by the Wikibench benchmark [30]. Among other information, it contains the page identifier and the list of versions. We use this log to recreate all the versions of the Wikipedia pages in ALEPH.



(a) Amount of versions per page.



(b) Size of each version.



(c) Access log to *en.wikipedia.org* (2007-08-22 to 2007-09-24).

Figure 2. Workload characterization.

Clients use an access log from the English Wikipedia web servers [30] spanning August to September 2007. This log contains (i) read accesses to the last versions, (ii) read accesses to older versions, and (iii) range queries to retrieve all versions (but not the content) of a page. As expected, most requests (98.9%) in this workload consists in read access to the last version. The remaining 0.64% and 0.46% consist in reads of older versions and range queries, respectively.

We start by analyzing the distribution of versions per page for three different Wikipedia languages: English (EN), Japanese (JA) and Sicilian (SCN). These languages were chosen because their size span different ranges: 11,404,989 (EN), 123,512 (JA) and 52,039 (SCN) articles. Results are presented in Figure 2(a). Even though the vast majority of pages have less than 10 versions, a small fraction of the pages have hundreds to thousands of versions. These are precisely the ones that might pose scalability issues, for instance, when storing all the versions on a single node.

Figure 2(b) shows the page size distribution for each language on a logarithmic scale. The three languages follow the same distribution with an average page size of 3.86KB, and a small fraction (0.0002%) is bigger than 1MB.

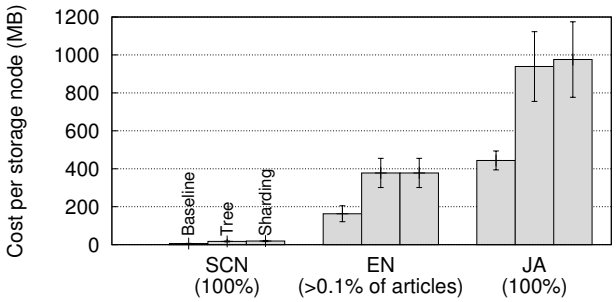


Figure 3. Storage cost for each Wikipedia.

Figure 2(c) shows the distribution of requests over time for the EN workload. The workload exhibits sudden spikes that need to be accommodated properly. In the most busy period, the system should sustain around 4,000 ops/second.

B. Client

The YCSB benchmark [29] executes create, read, update and delete (CRUD) operations, following a chosen ratio of operations and a key distribution. To replicate the access pattern shown in Figure 2(c), YCSB had to be heavily modified. First, the benchmark respects the order in which keys are requested in the trace log, Second, it issues the three types of (versioned) read operations occurring in the log. To further stress ALEPH, multiple clients can execute the log. In such a case, the benchmark orchestrates clients to replay the trace log as fast as possible.

C. Experimental Results

This section reports several experimental results on the use of ALEPH. We configure ALEPH to employ the three versioning mechanisms covered in Section V-C; namely (Baseline) a baseline implementation storing all the versions of a page under a blob in the KVS, (Tree) a tree-based indexation of the versions, and (Sharding) a mechanism sharding the index with the κ threshold fixed to 1000.

We perform our tests in-memory to reduce noise due to persistence storage. When populating ALEPH, we scaled down the size of each Wikipedia page by a factor 10 and use only 0.138% of the English Wikipedia (EN). This is necessary to satisfy the hardware constraints of our cluster.[1]

Storage cost: Figure 3 depicts the amount of memory used per storage node to load each Wikipedia in ALEPH. As expected, the baseline mechanism is the less expensive. It costs around half the price of the two other mechanisms. This difference is because such mechanisms separate data from metadata (indexes of versions). We also observe in Figure 3 that sharding the version index brings a small overhead in comparison to an approach where the index is stored at a single storage node. This comes from the fact that the threshold κ to create a new shard of the version index is fixed to 1000, hence occurring in rare cases.

[1] A 32bits Java virtual machine addresses at most 2.5GB of memory. Thus, the cluster offers at most $24 \times 2.5 = 60$GB of effective storage.

Technique	SCN	EN	JA
Baseline	14s	189s	392s
Tree	105s	419s	1450s
Slow-down	7.5	2.2	3.6
Sharding	158s	559s	1662s
Slow-down	11.3	2.9	4.2

Table II
TOTAL INSERTION TIME PER LANGUAGE (SECONDS).

Insertion performance: Table II shows the total time taken for a versioning mechanism to install each of the Wikipedia dumps into ALEPH. Depending on the Wikipedia, the baseline technique is 2.2 to 11.3 times faster. Such a gap comes from the fact that INFINISPAN does not offer a fast call to store multiple key-value pairs at once. Hence, in the case of Tree and Sharding, the implementation simply iterates over all the versions of a page to install them. Still, as one can see by the *Slow-down* factor, this cost is amortized for larger workloads.

Latencies Tradeoffs: Figure 4 compares the three versioning mechanisms executing an hour of the trace log on the EN dataset. This figure shows the last decile of the latency distribution (as a CDF) for read (bottom) and read range (top) queries. We grow the number of clients executing the log (from left to right). The x-axis indicates the latency in milliseconds on a logarithmic scale. Figure 4 only reports the results of Baseline for 20 clients. As expected, the Baseline mechanism is expensive and does not scale: read and read range queries require on average 106ms and 152ms, respectively. On the other hand, Tree and Sharding versioning mechanisms perform similarly. We observe that 95% of the read queries take less than 25ms even under high load (100 clients). For range queries, Tree is more efficient than Sharding, even in the tail of the distribution. We believe that the benefits of sharding the index of versions might require a more version-intensive dataset.

VII. CONCLUSION

This paper studies the requirements and tradeoffs to add versioning support to a key-value store (KVS). First, we prove that a simple $put()$ and $get()$ KVS interface doesn't provide sufficient synchronization power to support versioned data. To sidestep this result, we then consider a KVS enriched with support for listeners, and we explain how to build atomic objects of arbitrary type on top of its interface. Using this construction, we implement and evaluate various versioning mechanisms on top of industrial-grade KVS, INFINISPAN. Our empirical results, based on access traces and datasets from Wikipedia, suggest that the integration of versioning support into an existing KVS is practical, although trade-offs, in terms of operation latencies and storage costs, must be taken into account.

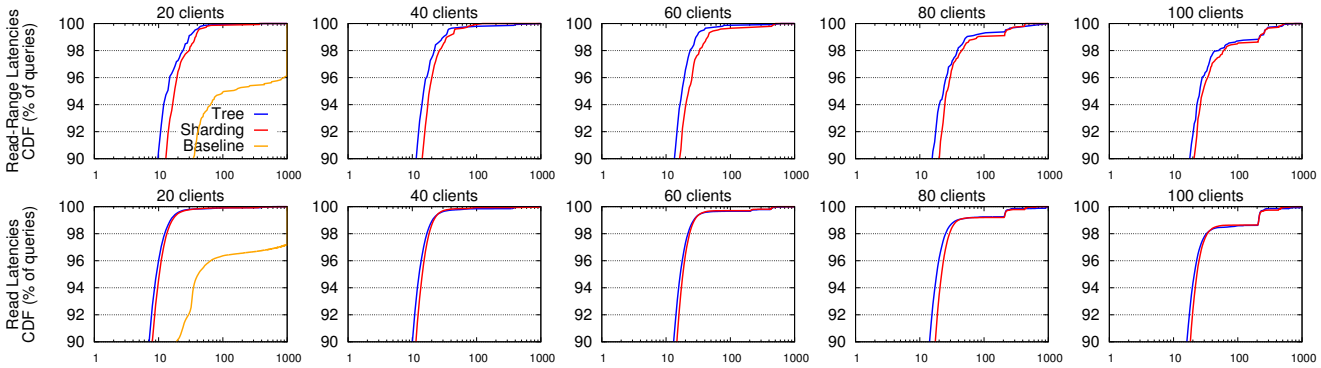


Figure 4. Read (bottom) and Read-Range (top) latencies for increasing number of clients.

VIII. Acknowledgements

We are thankful to the authors of Wikibench [30] to have publicly released their datasets, as well as the Infinispan developer community. The research leading to this publication was partly funded by the European Commission's FP7 under grant agreement number 318809, LEADS project and 611068, CoherentPaaS project, as well as the ERDF-European Regional Development Fund through the COMPETE Programme and by national funds through the FCT - Portuguese Foundation for Science and Technology - within project FCOMP-01-0124-FEDER-037281.

References

[1] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structured Storage System," in *Large Scale Distributed Systems and Middleware (LADIS)*, October 2009.

[2] "Riak," http://basho.com/riak.

[3] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[4] "Mongodb," https://www.mongodb.org.

[5] "Mongo MVCC," https://github.com/igd-geo/mongomvcc.

[6] L. George, *HBase: The Definitive Guide*. O'Reilly Media, 2011.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP 2007*, pp. 205–220.

[8] "HyperTable," http://hypertable.org.

[9] R. T. Snodgrass, *Developing Time-oriented Database Applications in SQL*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[10] C. Date and H. Darwen, *Temporal Data and the Relational Model*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[11] K. Kulkarni and J.-E. Michels, "Temporal features in SQL:2011," *ACM SIGMOD Record*, vol. 41, no. 3, pp. 34–43, Oct. 2012.

[12] F. Zemke, "What's new in SQL:2011," *ACM SIGMOD Record*, vol. 41, no. 1, pp. 67–73, 2012.

[13] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] P. R. Johnson and R. H. Thomas, "The maintenance of duplicate databases." *Internet RFC 677*, 1976.

[16] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *TOCS*, vol. 26, no. 2, 2008.

[17] T. A. Marsland and Z. Yang, *Global States and Time in Distributed Systems*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994.

[18] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 240–247, May 1983.

[19] J. Almeida, P. Almeida, and C. Baquero, "Bounded version vectors," *Distributed Computing*, vol. 3274, pp. 102–116, 2004.

[20] D. Malkhi and D. B. Terry, "Concise version vectors in WinFS," *Distributed Computing*, vol. 20, no. 3, pp. 209–219, 2007.

[21] P. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," *Principles of Distributed Systems*, vol. 5401, pp. 259–274, 2008.

[22] F. Marchioni and M. Surtani, *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[23] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[24] M. J. Fischer, N. A. Lynch, and M. S. Patterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[25] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[26] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, 1990.

[27] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *ACM STOC '97*, pp. 654–663.

[28] B. Ban, "JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org," 2007.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM SoCC*, 2010, pp. 143–154.

[30] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009, http://www.globule.org/publi/WWADH_comnet2009.html.

ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service

Raluca Halalai, Pierre Sutra, Étienne Rivière, Pascal Felber
University of Neuchâtel, Switzerland
first.last@unine.ch

***Abstract*—Cloud computing infrastructures leverage fault-tolerant and geographically distributed services in order to meet the requirements of modern applications. Each service deals with a large number of clients that compete for the resources it offers. When the load increases, the service needs to scale. In this paper, we investigate a scalability solution which consists in partitioning the service state. We formulate specific conditions under which a service is partitionable. Then, we present a general algorithm to build a dependable and consistent partitioned service. To assess the practicability of our approach, we implement and evaluate the ZooFence coordination service. ZooFence orchestrates several instances of ZooKeeper and presents the exact same API and semantics to its clients. It automatically splits the coordination service state among ZooKeeper instances while being transparent to the application. By reducing the convoy effect on operations and leveraging the workload locality, our approach allows proposing a coordination service with a greater scalability than with a single ZooKeeper instance. The evaluation of ZooFence assesses this claim for two benchmarks, a synthetic service of concurrent queues and the BookKeeper distributed logging engine.**

I. Introduction

Distributed services form the basic building blocks of modern computer architectures. A large number of clients access these services, and when a client performs a command on a service, it usually expects the service to be responsive and consistent. The seminal state machine replication (SMR) approach offers both guarantees. By replicating the service on multiple servers, the commands are wait-free despite failures, and by executing them in the same order at all replicas, they are linearizable. However, it is well-known that this last strategy has a performance cost: because SMR serializes all commands, it does not leverage the intrinsic parallelism of the workload.

To overcome the above problems, several directions have been investigated. First, operations that do not change the service state can be executed at a single replica. This approach implies dropping linearizability for sequential consistency, but such a limitation is unavoidable in a partially-asynchronous system [1]. Second, SMR can leverage the commutativity of updates to improve response time. This strategy exhibits a performance improvement of at most 33% in comparison to the baseline [2]. Third, one can partition the state of the service and distribute the partitions between replicas [3]. When the workload is fully parallel, the scale-out of the partitioning approach is optimal. Hence, we consider this approach as the most promising direction.

To illustrate in practice the benefits of partitioning, let us consider Figure 1. In this figure, we compare a partitioned queue service versus a non-partitioned one. In more details, we execute a hypothetical consumer/producer workload where (i) each consumer pulls resources from some defined queue, and (ii) each producer pushes with 80% chance a resource to a random queue, and with 20% to all the queues. In the top-curves, we partition the queues, each being implemented with an instance of the Apache ZooKeeper coordination service [4]. In the non-partitioned case (bottom-curve), all queues employ the same ZooKeeper. The total computational power remains the same for the partitioned and non-partitioned implementations. For 32 clients, the partitioned approach outperforms the non-partitioned one by a factor of 1.6.

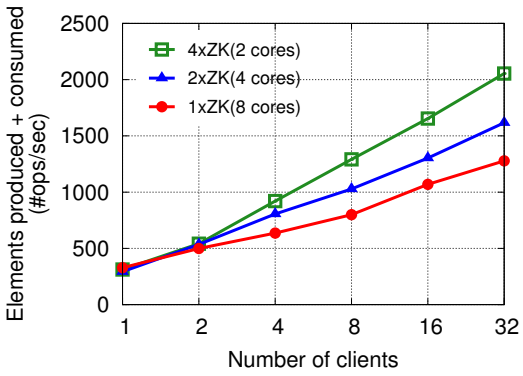


Fig. 1. Partitioned versus non-partitioned queue service.

Despite its obvious interest, to the best of our knowledge, few research efforts have been devoted to a principled study of service partitioning. In this paper, we try to bridge this gap. Our first contribution is to show that the partitioning theorem of Marandi et al. [3] omits some cases. We extend it and state a more general result under which it is possible to partition a shared service. Our second contribution consists in a general algorithm to partition a service. Our third contribution is ZooFence, a system that partitions the ZooKeeper coordination service following the previously introduced algorithm. ZooFence orchestrates multiple vanilla ZooKeepers, delegating portions of its state to each of them, and forwarding the operations that act on their respective partitions. In our last contribution, we evaluate ZooFence with two benchmarks, a synthetic concurrent queues service in a geo-distributed setting and the BookKeeper logging service. This evaluation shows that ZooFence improves the performance of the coordination

service compared to a single ZooKeeper, while offering at core the same guarantees.

The remainder of this paper is organized as follows. We formulate our results on service partitioning in Section II. Section III presents our general partitioning algorithm. We give an overview of ZooFence and the internals of its implementation in Section IV. We present a detailed evaluation of ZooFence in Section V. Section VI surveys related work. We conclude in Section VII. For readability purposes, we defer all our proofs to the appendix.

II. Consistent Service Partitioning

In what follows, we define the elements of our system model and the notion of partition. Further, we present two results that characterize if the partitioning of a service is consistent. These results form the guidelines of our approach to split a shared service into multiple parts in order to leverage the parallelism of its operations.

A. System Model

A service is specified by some serial data type. The serial data type defines the possible states of the service, the operations (or *commands*) to access it, as well as the response values from these commands. Formally, a serial data type is an automaton $S = (States, s^0, Cmd, Values, \tau)$ where $States$ is the set of states of S, $s^0 \in States$ its initial state, Cmd the commands of S, $Values$ the response values and $\tau : States \times Cmd \to States \times Values$ defines the transition relation. A command c is *total* if $States \times \{c\}$ is in the domain of τ. Command c is *deterministic* if the restriction of τ to $States \times \{c\}$ is a function. Hereafter, we assume that all commands are total and deterministic. We use $.st$ and $.val$ selectors to respectively extract the state and the response value components of a command, i.e., given a state s and a command c, $\tau(s,c) = (\tau(s,c).st, \tau(s,c).val)$. Function τ^+ is defined by repeated application of τ, i.e., given a sequence of commands $\sigma = \langle c_1, \ldots, c_{n\geq 1}\rangle$ and a state s:

$$\tau^+(s,\sigma) \triangleq \begin{cases} \tau(s,c_1) & \text{if } n = 1, \\ \tau^+(\tau(s,c_1).st, \langle c_2, \ldots, c_n\rangle) & \text{otherwise.} \end{cases}$$

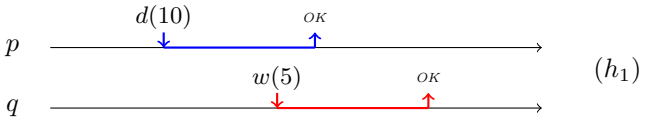
Two commands c and d *commute*, written $c \asymp d$, if in every state s we have:

$$c \asymp d \triangleq \begin{cases} \tau^+(s,\langle c,d\rangle).st = \tau^+(s,\langle d,c\rangle).st \\ \tau^+(s,\langle d,c\rangle).val = \tau^+(s,c).val \\ \tau^+(s,\langle c,d\rangle).val = \tau^+(s,d).val \end{cases}$$

For any two commands c and d, we write $c = d$ when in every state s, $\tau(s,c) = \tau(s,d)$. By extension, for some command c and some sequence $\sigma = \langle c_1, \ldots, c_{n\geq 2}\rangle$, we write $c = \sigma$ when $\tau^+(s,\langle c_1, \ldots, c_n\rangle) = \tau(s,c)$.

To illustrate the above notations, let us consider a bank account equipped with the usual withdraw and deposit operations. We define $States$ as $\mathbb{N}$, with $s_0 = 0$. A deposit operation $d(10)$ brings s to $s + 10$ and returns OK. In case the bank prohibits overdrafts, a withdraw operation $w(x)$ returns NOK if $s < x$; otherwise it brings s to $s - x$.

a) History: We consider a global time model and some bounded set of client processes that may fail-stop by crashing. A history is a sequence of invocations and responses of commands by the clients on one or more services. When command c precedes d in history h, we write $c <_h d$. We use timelines to illustrate histories. For instance the timeline below depicts the interleaving of commands $d(10)$ and $w(5)$, executed by respectively clients p and q in some history h_1.



Following Herlihy and Wing [5], histories have various properties according to the way invocations and responses interleave. For the sake of completeness, we recall these properties in what follows. A history h is *complete* if every invocation has a matching response. A *sequential* history h is a non-interleaved sequence of invocations and matching responses, possibly terminated by a non-returning invocation. When a history h is not sequential, we say that it is *concurrent*. A history h is *well-formed* if (i) $h|p$ is sequential for every client process p, (ii) for every command c, c is invoked at most once in h, and (iii) for every response $res_i(c)\,v$ there exists an invocation $inv_i(c)$ that precedes it in h.[1] A well-formed history h is *legal* if for every service S, $h|S$ is both complete and sequential, and denoting $\langle c_1, \ldots, c_{n\geq 1}\rangle$ the sequence of commands appearing in $h|S$, if for some command c_k a response value appears in $h|S$, it equals $\tau^+(s^0, \langle c_1, \ldots c_k\rangle).val$.

b) Linearizability: Two histories h and h' are said *equivalent* if they contain the same set of events. Given a service S and a history h of S, h is *linearizable* [5] if it can be extended (by appending zero or more responses) to some complete history h' equivalent to a legal and sequential history l of S with $<_{h'} \subseteq <_l$. In such a case, history l is named a *linearization* of h. For instance, the history h_1 above is linearizable since it is equivalent to a sequential one in which $d(10)$ occurs before $w(5)$. The histories of a service S are all the linearizable histories that are constructible with the commands of S. A service S *implements* a service T when for every linearizable history h of S, there exists a linearizable history h' of T such that h' is a high-level view of h [6].[2]

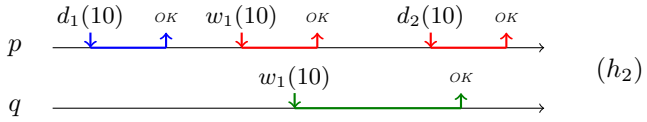
c) Partition: Given a finite family of services $(S_k)_{1\leq k\leq n}$, the synchronized product of $(S_k)_k$ is the service defined by $(\prod_k States_k, (s_1^0, \ldots, s_n^0), \bigcup_k Cmd_k, \bigcup_k Values_k, \tau)$ where for every state $s = (s_1, \ldots, s_n)$ and every command c in some Cmd_k, the transition function τ is given by $\tau(s,c) = ((s_1, \ldots, \tau_k(s_k,c).st, \ldots, s_n), \tau_k(s_k,c).val)$. Given a service S, the family $(S_k)_{1\leq k\leq n}$ is a *partition* of S when its synchronized product satisfies(i) $States \subset \prod_k States_k$, (ii) $s^0 = (s_1^0, \ldots, s_n^0)$, and (iii) for every command c, there

[1] For some service or client x, $h|x$ is the projection of history h over x.

[2] A high-level view is generally constructed via a refinement mapping from the states of S to the states of T [7].

exists a unique sequence σ in $\bigcup_k Cmd_k$, named the *sub-commands* of c, satisfying $\sigma = c$. The partition $(S_k)_k$ is said *consistent* when it implements S.

To illustrate the notion of partition, let us go back to our banking example. A simplistic bank service allows its clients to withdraw and deposit money on an account, and to transfer money between two accounts. We can partition this service into a set of branches, each holding one or more accounts. A transfer of an amount x between accounts i and j is modeled as the sequence of sub-commands $\langle w_i(x).d_j(x)\rangle$, where $w_i(x)$ and $d_j(x)$ are respectively a withdrawal and a deposit of the amount x on the appropriate branch. However, precautions must be taken when concurrent commands occur on the partitioned service. For instance, the following history should be forbidden by the concurrency control mechanism to avoid money creation (concurrent withdrawals cannot both succeed as the balance is not sufficient).



In the section that follows, we characterize precisely when the partition of a service is correct.

B. Partitioning Theorems

When there is no invariant across the partition and every command is a valid sub-command for one of its parts, the partition is *strict*. We first establish that a strict partition is always consistent.

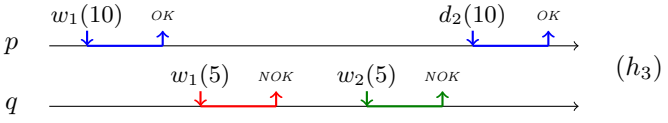
Theorem 1. *Consider a service S and a partition $(S_k)_k$ of S. If both $\prod_k States_k = States$ and $Cmd = \bigcup_k Cmd_k$ hold then $(S_k)_k$ is a consistent partition of S.*

The above theorem is named the locality property of linearizability [5]. It states that the product of linearizable implementations of a set of services is linearizable. From the perspective of service partitioning, this suggests to implement each part as a replicated state machine. Such an idea forms the basic building block of our protocols.

When commands contain several sub-commands, Theorem 1 does not hold anymore. Nevertheless, it is possible to state a similar result when constraining the order in which sub-commands interleave. This is the observation made by Marandi et al. [3], despite a small omission in the original paper. Below, we show where the error occurs and propose a corrected and extended formulation. To state our results, we first need to introduce the notion of conflict graph.

Definition 1. *(Conflict Graph) Consider a history h of a partition $(S_k)_k$ of some service S. The* conflict graph *of S induced by h is the graph $G_h = (V, E)$ such that V contains the set of commands executed in h, and E is the set of pairs of commands (c, d) with $c \neq d$, for which there exist two sub-commands c_i and d_j executed on some S_k such that $c_i <_{h|S_k} d_j$.*

In [3], the authors claim that the partition $(S_k)_k$ is consistent, provided that h is linearizable for each part S_k and G_h is acyclic. Unfortunately, this characterization is incorrect because G_h does not take into account the causality with which commands are executed in h. We argue this point with a counter-example. Let us consider again that our banking service is partitioned in a set of branches. Clients p and q execute the three commands $\langle w_1(10), d_2(10)\rangle$, $w_1(5)$ and $w_2(5)$ as in history h_3 where account 1 is initially provisioned with the amount 10.



Since both withdrawals of client q fail, history h_3 is not linearizable. However, G_{h_3} remains acyclic since it does not capture that process q creates the order $w_1(5) <_{h_3} w_2(5)$.

In what follows, we prove an extended and corrected formulation of the partitioning result of Marandi et al. [3]. Our characterization is based on the notion of semantic graph that we define next.

Definition 2. *(Semantic Graph) Consider a history h of a partition $(S_k)_k$ of some service S. The* semantic graph *of S induced by h is the graph $G_h = (V, E)$ such that V contains the set of commands that appear in h and E is the set of pairs (c, d), with $c \neq d$, for which either(i) there exist two non-commuting sub-commands c_i and d_j in some part S_k such that $c_i <_{h|S_k} d_j$, or (ii) $c <_h d$, where we note $c <_h d$ when all the sub-commands of c precede all the sub-commands of d in history h.*

In contrast to the notion of conflict graph, a semantic graph takes into account the commutativity of sub-commands. To understand why, assume that the banking service allows unlimited overdraft. In such a case, any interleaving of the sub-commands would produce a linearizable history. The partitioning theorem that follows generalizes this observation. It states that the acyclicity of non-commuting sub-commands in the semantic graph is a sufficient condition to attain consistency. A proof appears in Appendix A.

Theorem 2. *A partition $(S_k)_k$ of a service S is consistent if for every history h of $(S_k)_k$, there exists some linearization l of h such that the semantic graph of S induced by l is acyclic.*

III. Protocols

Building upon our previous theorems, this section describes several constructions to partition a shared service. Our presentation follows a refinement process. We start with an initial construction requiring that strictly disjoint services form the partition, then we introduce a more general technique that can accommodate with any type of partitioning. Our last construction improves parallelism at the cost of constraining how the partition is structured. To ease the presentation of our algorithms, we shall be assuming hereafter that sub-commands are idempotent and that no two sub-commands

Algorithm 1 Base construction – code at client p

1: $invoke(c)$:=
2: **let** S_k such that $c \in Cmd_k$
3: **return** $\mathcal{M}(S_k).invoke(c)$

in the same command access the same part. Nevertheless, all of our algorithms can be easily adapted to handle the cases where such properties do not hold.

A. Initial Construction

We depict in Algorithm 1 a first construction when the partition $(S_k)_k$ of service S is strict. This algorithm makes use of a mapping $\mathcal{M}$ satisfying that for every S_k, $\mathcal{M}(S_k)$ is a replicated state machine implementing S_k. When a client p executes a command c on S, it uses $\mathcal{M}$ to retrieve the state machine implementing S_k, where S_k is the service on which c executes (line 2). Then, client p invokes the command on $\mathcal{M}(S_k)$ and returns the result of this invocation (line 3).

Since $(S_k)_k$ is strict and $\mathcal{M}(S_k)$ is a linearizable implementation of S_k, Algorithm 1 implements a consistent partition of S by Theorem 1. Besides, the implementation of $(S_k)_k$ obtained through Algorithm 1 is wait-free [8]. This property is inherited from the underlying replicated state machines that support Algorithm 1. In addition, this base construction is optimal in terms of scalability since, when clients access uniformly the parts, the throughput of the partitioned service is $|(S_k)_k|$ times the throughput of S.

B. A Queue-based Construction

In what follows, we refine Algorithm 1 to handle the case where multiple sub-commands compose a command. A naive solution would consist in modifying Algorithm 1 so that when the client process p executes a command $c = \langle c_1, \ldots, c_n \rangle$, it applies in order all the sub-commands $c_1, \ldots, c_n$ to the appropriate part. Such an approach however fails since(i) an invariant may link different parts of the partition, and (ii) if client p crashes in the middle of its execution, not all the parts will reflect the effects of command c.

Algorithm 2 depicts a solution to deal with these two issues. This algorithm ensures that either all the sub-commands of a command execute, or none of them, and that the state of the partitioned service remains consistent. It is based on a shared FIFO queue abstraction (variable Q) and an eventual leader election service (variable Ω). Clients use Q to submit the commands they wish to execute. Submitted commands are then executed in the queue order by the leader elected by Ω.

With more details, our algorithm works as follows. Upon invoking a command $c = \langle c_1, \ldots, c_n \rangle$, a client p appends c to the queue Q, then it starts participating in the leader election (line 8). In case p is elected, it processes the commands in Q (line 15). For each such command d, p executes all the sub-commands of d once every non-commuting command before d has been executed (line 15). The result of the last sub-command of d is stored as the response of d in the queue Q (lines 17 to 20). This pattern is repeated until the leader, which might not be p, executes command c.

Algorithm 2 Queue-based construction – code at client p

1: **Shared Variables:**
2: Ω // a leader election
3: Q // an atomic queue
4:
5: $invoke(c)$:=
6: $r \leftarrow \bot$
7: $Q \leftarrow Q \circ (c, r)$
8: $\Omega.register()$
9: **wait until** $r \neq \bot$
10: $\Omega.unregister()$
11: $Q \leftarrow Q \setminus (c, r)$
12: **return** r
13:
14: **when** $p = \Omega.leader()$
15: **let** $(d, r') \in Q : \forall (e, \hat{r}) <_Q (d, r') : \hat{r} \neq \bot \vee d \asymp e$
16: **let** $d_1, \ldots, d_m : d = \langle d_1, \ldots, d_m \rangle$
17: **for all** $j \in [\![1, m]\!]$ **do**
18: **let** $S_k : d_j \in Cmd_k$
19: $r'' \leftarrow \mathcal{M}(S_k).invoke(d_j)$
20: $r' \leftarrow r''$

The leader election service Ω allows a process to register (line 8) and to unregister (line 10). This service satisfies that eventually(i) only registered processes are elected, and (ii) at least one correct process considers itself as the leader. We note here that property (ii) was previously mentioned in [9], and that Ω is a form of restricted leader election [10]. This makes Ω strictly weaker than the leader oracle used in consensus [11].

C. Ensuring Disjoint Access Parallelism

Both the protocol of Marandi et al. [3] and Algorithm 2 order submitted commands through some global shared object: an instance of the Ring Paxos protocol in the case of [3], and a shared queue for Algorithm 2. As a consequence, the synchronization cost of executing a command is related to the number of concurrent commands. This defeats the primary goal of partitioning which is to scale-up the service by leveraging parallelism for commands that access different parts of the service. Such a property is named *disjoint-access parallelism* (DAP) in the literature on shared memory computing [12]. In what follows, we depict a refinement of our previous algorithm that ensures the following DAP property:

Definition 3 (Disjoint-Access Parallelism)**.** *Consider an algorithm $\mathcal{A}$ implementing a partition $(S_k)_k$ of some service S. We say that $\mathcal{A}$ is* disjoint-access parallel *when in each of its histories h there exists a linearization l of h, such that if p and q concurrently executing commands c and d contend on some shared object in $\mathcal{A}$, there exists a non-directed path linking c to d in the conflict graph of l.*

Algorithm 3 depicts our construction of a DAP consistent partitioning. For each part S_k, we assign respectively a queue $Q[k]$ and a leader election $\Omega[k]$. When a client p invokes a command $c = \langle c_1, \ldots, c_m \rangle$, it iteratively executes each sub-command c_i on the appropriate replicated state machine. To that end, client p adds c_i to the queue $Q[k]$ and then joins leader election $\Omega[k]$ to execute all the sub-commands in $Q[k]$. The helping mechanism in Algorithm 3 is similar to the one we employed in Algorithm 2: when p is the leader and a sub-

Algorithm 3 DAP construction – code at client p

```
1: Shared Variables:
2:     Ω                                   // an array of leader election objects
3:     Q                                   // an array of atomic queues
4:
5: invoke(c) :=
6:     return invoke_sub(first(c))
7:
8: invoke_sub(c_i) :=
9:     r ← ⊥
10:    let S_k : c_i ∈ Cmd_k
11:    Q[k] ← Q[k] ∘ (c_i, r)
12:    Ω[k].register()
13:    wait until r ≠ ⊥
14:    Ω[k].unregister()
15:    Q[k] ← Q[k] \ (c_i, r)
16:    return r
17:
18: when p = Ω[l].leader()                 // for some l
19:    let (d_j, r') ∈ Q[l] : ∀(e_j', r̂) <_Q[l] (d_j, r') : r̂ ≠ ⊥ ∨ d_j ≍ e_j'
20:    r'' ← M(S_l).invoke(d_j)
21:    if d_j ≠ last(d) then
22:        r'' ← invoke_sub(d_j+1)
23:    r' ← r''
```

command d_j occurs before c_i in the queue $Q[k]$, p must first execute d_j as well as the sub-commands following it, before it can execute c_i (lines 18 to 23). This pattern ensures the correctness of our construction in the case where the following property holds:

(P1) There exists an ordering $\ll$ of $(S_k)_k$ such that for any two sub-commands c_i and c_j accessing respectively parts S_k and $S_{k'}$, if c_i precedes c_j in c then $S_k \ll S_{k'}$ holds.

Unfortunately, property P1 does not hold for every partition $(S_k)_k$ of a service S. For instance in our previous banking example, P1 only holds if money transfers between accounts in different branches occur in some canonical order: e.g., $\langle w_i(x), d_j(x)\rangle$ is allowed if and only if $i < j$ holds.

Our key observation is that we can nevertheless enforce the acyclicity of the semantic graph by implementing the partition in a hierarchical manner. We achieve this via two modifications to Algorithm 3. First, we replace P1 by the fact that:

(P2) Function $\mathcal{M}$ returns a set of replicated state machines for each S_k such that for any two sub-commands c_i and c_j accessing respectively parts S_k and $S_{k'}$, if c_i precedes c_j in c then either $\mathcal{M}(S_k) \subseteq \mathcal{M}(S_{k'})$ or the converse holds.

Second, upon executing a sub-command c_i (at line 20 in Algorithm 3), we apply c_i in some canonical order to all the replicated state machines in $\mathcal{M}(S_k)$ before returning the value r''. These two modifications ensure that the premises of Theorem 2 hold for any partition of some shared service S. We sketch a proof of correctness for Algorithm 3 in Appendix B.

Going back to the design of a partitioned banking service, applying P2 requires the addition of a special account t replicated at all branches, such that when a money transfer occurs between two accounts in different branches, money goes through account t, i.e., $\langle w_i(x), d_t(x).w_t(x).d_j(x)\rangle$.

Algorithm 3 with property P2 is the general method we employ to partition a service. We implemented it in ZooFence where we partition the shared tree interface exposed by the Apache Zookeeper coordination service. By partitioning the tree, ZooFence reduces contention and leverages the locality of operations. Both effects contribute to improve latency of operations and increase overall throughput. We describe ZooFence in detail in the next section.

IV. The ZooFence Service

In this section, we present an application of our principled partitioning approach to the popular Apache ZooKeeper [4] coordination service. The resulting system, named ZooFence, orchestrates several independent instances of ZooKeeper. The use of ZooFence is transparent to applications: it offers the exact same semantics and API as a single instance of ZooKeeper. However, the design of ZooFence allows avoiding synchronization between parts when it is not necessary. This reduces the impact of convoy effects that synchronization causes [13].

The partitioning of the ZooKeeper service follows the approach introduced in Algorithm 3. ZooFence splits the tree structure between multiple ZooKeeper instances. Commands that access distinct parts of the tree run in parallel on distinct instances, while guaranteeing both strong consistency and wait-freedom. Commands that access a single part of the tree run on a single instance. This section presents the main components of ZooFence, discusses our design choices with regard to Algorithm 3, then details some specific aspects of its implementation. We give a high-level specification of ZooFence in Appendix C.

A. Overview

Figure 2 depicts the general architecture of ZooFence. The system has four components:(i) A set of independent ZooKeeper instances that ZooFence orchestrates; (ii) A client-side library; (iii) A set of queues storing commands that need to be executed on multiple instances; and (iv) A set of executors that fetch commands from the queues, delegate them to the appropriate instances, and return the result to the calling clients.

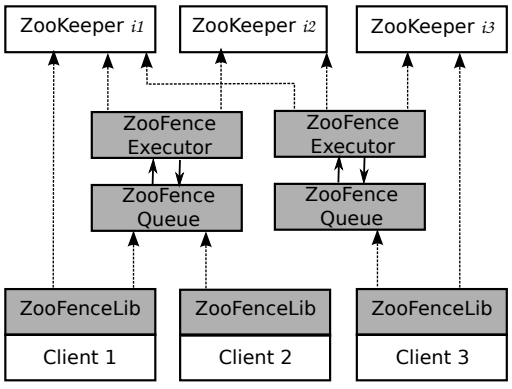


Fig. 2. ZooFence design.

B. Client-Side Library

ZooFence clients execute commands through a client-side library that implements the ZooKeeper API. This interface

consists of a set of commands accessing a concurrent tree data structure composed of *znodes*. A znode in the tree stores some data, and is accessible via a *path* as in UNIX filesystems. Znodes can be *persistent*, meaning they belong to the tree until a client explicitly deletes them, or *ephemeral*, in which case they are automatically removed once the client that created them disconnects or crashes. In addition, znodes can be *sequential*. For such znodes, the system automatically appends a monotonically increasing counter to their names at creation time. A client can manipulate a znode through read or write commands. Read commands, e.g., `exists`, `getChildren` or `getData` return a sub-state of the shared tree without modifying it. Write commands such as `create`, `delete` or `setData`, modify the state of the tree and return metadata information to the client.

Clients execute a command on one or more of the orchestrated ZooKeeper instances according to a *flattening function*. This function plays the role of function $\mathcal{M}$ in Algorithm 3 and maps paths to ZooKeeper instances. Multiple flattening functions can be used. The choice of a flattening function depends on how the application accesses the concurrent tree structure. When a client executes a command c on a znode n, ZooFence determines, using the flattening function, the set of ZooKeeper instances $I = f(n)$ on which c is executed. Command c is *trivial* when $f(n)$ returns a single ZooKeeper instance. In such a case, the command is directly forwarded to that instance. If c is *non-trivial*, it is inserted into the appropriate queue. The executor associated to the queue forwards the command to the corresponding ZooKeeper instances, then returns the result to the client.

The flattening function f satisfies property P2. This means that if a znode n with parent p is mapped to a set of instances $I = f(n)$ then I is also a subset of $f(p)$. To understand why, consider that znode `/a` is mapped to instances $\{i_1, i_2, i_3\}$, and `/a/b` to $\{i_1, i_2\}$. In case a client executes `create(/a/b)`, because both instances i_1 and i_2 hold a copy of `/a`, the creation of `/a/b` succeeds if and only if `/a` was created previously. This ensures that a znode n with parent p is in the tree if and only if p also exists in the tree. The next section provides additional details on the internals of ZooFence, explaining how we maintain this key invariant.

C. Executor

The core notion of ZooFence is the executor. Each executor implements the logic of the sub-commands execution mechanism we depicted at lines 18 to 23 in Algorithm 3. There is one executor for each set of ZooKeeper instances replicating a common path. For instance, in the above example, there is one executor for `/a`, replicated at $\{i_1, i_2, i_3\}$ and one for `/a/b`, replicated at $\{i_1, i_2\}$. As explained previously, each executor is associated with a FIFO queue. When a client executes a non-trivial command, it adds that command to the corresponding queue according to the flattening function. The executor scans the queue in order to retrieve the next command c it has to execute. Then, it forwards c to all the associated ZooKeeper instances, merges their results, and sends the final result back to the client before deleting c from the queue.

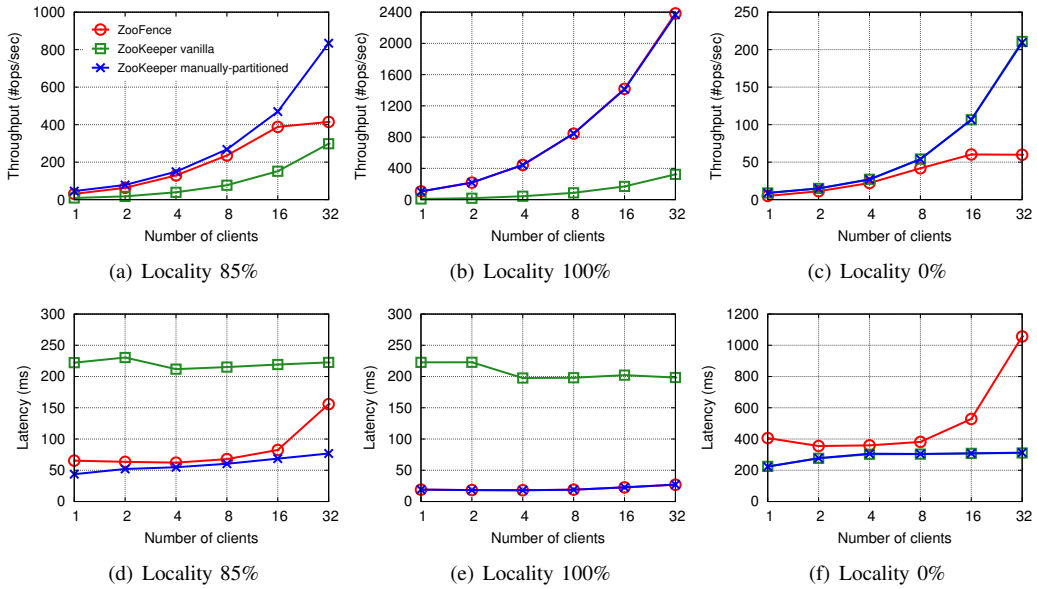
d) Queue synchronization: Using multiple executors improves the performance of ZooFence, but requires additional synchronization. To illustrate this point, let us consider again that `/a` is replicated at $\{i_1, i_2, i_3\}$ and `/a/b` at $\{i_1, i_2\}$. The set of instances associated with znode `/a/b` has a smaller cardinality than the set of instances associated with its parent, `/a`; we call `/a/b` a *fringe* znode. Assume that a client attempts to delete `/a`, while another client concurrently attempts to delete `/a/b`. Due to the tree invariant, the deletion of `/a` succeeds only if it does not have any children. In the scenario above, if the deletion of znode `/a/b` finishes on i_1, but not on i_2, before the deletion of znode `/a` is executed, then `/a` would be deleted from i_1, but not from i_2, leaving replicas in an inconsistent state. We solve the problems related to fringe znodes by synchronizing queues following the approach depicted at lines 21 to 22 in Algorithm 3: Upon creation of such a znode, the executor adds the command to the parent queue. Upon deletion, the executor first executes the command then, in case of success, it adds the command to the parent queue. When adding a command to the parent queue, the executor waits for a result before returning.

e) Failure Recovery Mechanism: The executor is a dependable component of ZooFence. To ensure this guarantee, we replicate each executor and employ the same leader election mechanism as in Algorithm 3. When the previously elected executor is unresponsive or has crashed, ZooFence nominates a new one and resumes the execution of commands. ZooFence prevents inconsistencies that might result, as follows: (i) We ensure idempotency at the client side; and (ii) We use the command semantics to resume incomplete commands on the associated ZooKeeper instances.

f) Queue monitoring: Executors retrieve commands from their respective queues and keep them in a local cache. Instead of actively polling the queue for new commands, executors use the ZooKeeper event notification mechanism, watches, which are triggered when the queues are modified. Executors check their local caches every time the watch set on their queue is triggered; if the cache is empty, the executor performs a `getChildren` command to repopulate its cache.

g) Asynchronous commands: Each executor retrieves commands from its local cache and forwards them to its set of ZooKeeper instances. We use asynchronous commands between an executor and its ZooKeeper instances, regardless of the type of command issued by the ZooFence client. The only difference between synchronous and asynchronous client commands is local, in the ZooFence client library; for synchronous commands, the library blocks and waits for the reply to arrive. Since ZooKeeper guarantees FIFO order even for asynchronous commands, this optimization which is transparent to the clients, increases the throughput of the executor without changing the semantics of synchronous

h) Colocation: An executor is a stand-alone process of ZooFence that runs on a different machine than the clients. To reduce network latency for queue accesses, an executor usually



(a) Locality 85% (b) Locality 100% (c) Locality 0%

(d) Locality 85% (e) Locality 100% (f) Locality 0%

Fig. 3. Comparison of ZooFence against a vanilla and a manually-partitioned ZooKeeper.

executes at the same site as the ZooKeeper instance that hosts its corresponding queue.

D. Implementation Details

Our prototype implementation of ZooFence is written in Java and contains around 2,500 SLOC. It is built upon the out-of-the-box ZooKeeper 3.4.5 distribution. Clients transparently instantiate a ZooFence deployment when the connection string contains multiple ZooKeeper instances separated by a "|" character, e.g., "127.0.0.1:2181 | 127.0.0.1:2182". As discussed in Section IV-B, ZooFence clients forward non-trivial commands to executors via queues. Client commands are serialized and stored in *persistent sequential znodes*; we use the *sequential* attribute offered by ZooKeeper to assign a sequence number to each command in the queue. Queues are stored in dedicated administrative ZooKeepers. Executors are identified by *ephemeral znodes*, also stored in administrative ZooKeepers. The choice of the administrative ZooKeepers among the existing instances is currently not automated and left as a future work. Clients open TCP connections to executors before putting commands in queues. When commands complete, executors send the results over the respective connections.

V. Evaluation

We evaluate our ZooFence prototype in two scenarios: a synthetic concurrent queue service deployed at multiple geographically-distributed sites, and the BookKeeper distributed logging service [14]. We compare ZooFence against vanilla ZooKeeper deployments in terms of throughput, latency and scalability.

A. Concurrent Queues Service

In this experiment, we show that ZooFence can leverage access patterns locality to improve both throughput and responsiveness of distributed applications. To that end, we emulate a geographically-distributed deployment. Each site consists of dual-core machines with 2 GB RAM, and hosts one ZooKeeper instance and several clients. Inside a site, machines communicate using a (native) gigabit network. Between sites, we set up the round-trip delay to approximately 50 ms. Our experiment uses a single executor, collocated with the administrative ZooKeeper on an eight-core machine at a different site than all clients.

In this environment, we deploy a service consisting of five concurrent queues. Four of the queues exhibit strong locality, and are used exclusively by clients from the same site (i.e., *queue1* is used only by clients from *site1*, *queue2* from *site2*, etc.). We refer to these queues as *site queues*. The fifth queue is used by clients from all sites. We refer to it as the *geo-distributed queue*. We vary the number of clients within each site from one to eight. Each client runs two producer processes. Producers mostly create znodes in their site queue, but occasionally write to the geo-distributed queue as well. We note that this is a write-heavy workload, which represents a worst-case scenario for both ZooFence and ZooKeeper.

Our experiments compare in terms of performance three different deployments:(1) ZooFence with a flattening function that assigns site queues to local ZooKeeper instances, and the geo-distributed queue to all instances. (2) A vanilla ZooKeeper, which involves all the available ZooKeeper instances: a leader and three followers, with synchronization bound set to 175 ms. All queues are stored by this ZooKeeper instance. This is the baseline for our experiments – how ZooKeeper would be used in the present. (3) A manually-partitioned ZooKeeper. Each machine runs two ZooKeeper instances: one instance stores the site queue, and the other one stores the geo-distributed queue; the latter is a ZooKeeper instance covering all sites. This deployment is the optimum an experienced ZooKeeper administrator can achieve: writes accessing the geo-distributed queue are broadcast to all sites, otherwise they are served locally.

Figure 3 presents the latency and the throughput of the

system for each of the three deployments when we vary the number of clients and the percentage of operations on the geo-distributed queue.

As shown by figures 3(a) and 3(d), with a locality of 85%, ZooFence is close to the manually-partitioned ZooKeeper. This happens because in both deployments most operations occur inside a site, avoiding the cross-site communication in most cases. The vanilla ZooKeeper deployment exhibits lower throughput and higher latency because all queues are replicated across all sites. Since the leader ZooKeeper propagates updates to its followers, the inter-site communication penalty cannot be avoided.

When locality is maximum, ZooFence is identical to the manually-partitioned ZooKeeper (figures 3(b) and 3(e)). The vanilla ZooKeeper deployment performs significantly worse because it fully replicates all znodes.

Finally, when all operations are performed on the geo-distributed queue (figures 3(c) and 3(f)), the performance of ZooFence becomes worse than that of the other two deployments, both in terms of throughput and latency. This is due to the overhead incurred by our execution mechanism for operations on replicated znodes: the executor fetches commands from the execution queue, delegates them to the responsible ZooKeeper instances based on the flattening function and forwards the result to the client.

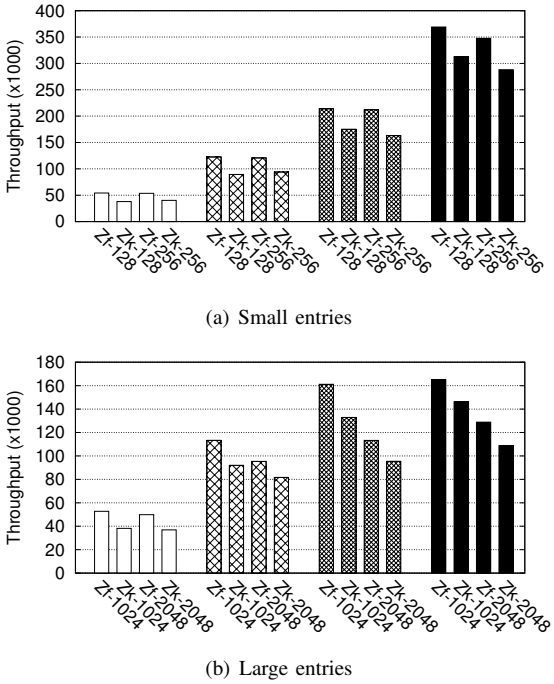
The executor itself can become saturated and degrade performance, in terms of both throughput and latency. In this experiment, we used a single executor, deployed on the same VM as the one hosting the administrative ZooKeeper. The executor becomes saturated at around 60 operations per second, as shown in Figure 3(c) and Figure 3(a) (only 15% of operations go through the executor, and 15% * 400 = 60). This is mainly due to our design choice of making ZooFence modular, on top of ZooKeeper. This choice implies that our system does not benefit from optimizations such as batching, which require tighter integration. Since the executor is in a separate site, it pays the latency penalty two times for global operations by communicating with the ZooKeeper instances and the clients, which limits performance.

We have performed a similar experiment for read workloads, where conclusions are similar. ZooFence allows local reads to exhibit low latency, and not interfere with the performance of remote partitions.

Overall, our experiments show that ZooFence performs close to an optimal deployment when access patterns exhibit strong locality. ZooFence can enable even inexperienced administrators to obtain good performance without the burden of partitioning the state manually.

B. BookKeeper

This section presents experimental results that assess the performance gains of ZooFence over ZooKeeper. To that end, we compare the two systems in a cluster deployment when supporting the BookKeeper logging service [14]. Below, we first give a brief description of BookKeeper, then detail our experimental protocol and comment on our results.



(a) Small entries

(b) Large entries

Fig. 4. BookKeeper performance *(from left to right, the amount of written entries is 100, 250, 500 and 1000).*

BookKeeper is a distributed system that reliably stores streams of records. Each stream is stored in a write ahead log, or *ledger*, that contains multiple *entries*. A ledger is replicated at one or more dedicated storage servers, named *bookies*. BookKeeper uses the ZooKeeper tree structure to stores the metadata of the ledgers, as well as the set of available bookies. The tree also ensures that a single client writes to a ledger at a time. BookKeeper supports close-to-open semantics: when writing to a ledger, a client appends new entries at a quorum of its replicas. Changes to an open ledger are visible to a concurrent client only when the ledger is closed.

Our experiments consist in evaluating the maximal throughput of BookKeeper when clients concurrently open ledgers and start appending entries to them. In all our experiments, we employ 18 bookies with a replication factor of 3. Such a setting ensures that the system can sustain the failure of one bookie without interruption. We vary both the length of an entry and the frequency at which clients create a new ledger. This last parameter is controlled by the number of entries each client writes to a ledger before closing it.

Figures 4(a) and 4(b) present a detailed performance comparison of BookKeeper when using ZooKeeper and ZooFence. We vary the length of an entry from 128 to 2048 bits, and the number of written entries from 100 to 1000. In both figures, Zk stands for ZooKeeper, while Zf means ZooFence; the throughput is measured as the total amount of operations per second. When clients write 1000 entries (or more, not shown on the plots), the two systems achieve close performance. In such a case, the throughput is limited by either the bookies or the network. During our experiments, the MTU is set to 1500 bytes. This explains the performance gap between large and small entries. On the other hand, when clients open concurrently more

ledgers, fast operations on the metadata storage matter. In such a case, because ZooFence provides parallel accesses to the shared tree, it outperforms ZooKeeper. The difference increases as clients access new ledgers more frequently. In our experiments, ZooFence improves the throughput of BookKeeper by up to 45%.

VI. Related Work

Coordination services find their roots in pioneering works on locks and synchronization primitives [15]. The need for coordination spreads over multiple areas, from distributed databases [16], file stores [17], multicore systems [18] and Cloud computing [19].

Several paradigms exist for coordinating distributed processes in the Cloud. Microsoft Azure [20] exports a common lock interface. Clients of Google Chubby [21] make use of a lease mechanism to gain exclusive access to a shared resource. The API of Apache ZooKeeper [4] consists in a concurrent tree data structure, close to UNIX filesystems. All these three systems back-up committed operations in a replay log. The Corfu system [22] implements an atomic log on top of dedicated hardware. With Tango [23], developers have access to any type of strongly consistent in-memory object, e.g., queue, map. Each Tango object is backed by the Corfu log.

Coordination is closely related to dependability. Indeed, redundancy is the usual mean to mask failures, but replicas of a service need to coordinate in order to implement a consistent service execution. State machine replication (SMR) is the seminal approach [24] to build dependable distributed services. It allows a set of replicas to agree on the order in which they execute service operations. SMR relies on a consensus algorithm, e.g., Paxos [25], and this approach is at heart of the Cloud systems we reviewed above. Several recent works (e.g., [26, 27]) observe that there is no need to order commuting service operations. This observation was used recently to build a distributed database system [28].

The seminal CAP result [29] states that a system cannot be at the same time responsive, consistent and robust to network outages. In the same vein as the SMR approach, ZooFence favors consistency over availability. This means that availability can be forfeited in the presence of network faults. For instance, if one of the ZooKeeper instances implementing ZooFence cannot progress, commands accessing the data it replicates are frozen until the instance come-back.

The virtual synchrony paradigm [30] is close, but slightly different from SMR. Under this paradigm, distributed processes execute a sequence of views, agreeing in each view upon the participants and the set of received messages (but not on their order). Virtual synchrony is used to build the ISIS middleware and its successors [31].

Some approaches [32, 33] leverage application semantics to compute dependencies among commands in order to parallelize SMR. Commands are assigned to groups such that commands within a group are unlikely to interfere with each other. Eve [33] allows replicas to execute commands from different groups in parallel, verifies if replicas can reach agreement on state and output, and rolls back replicas if necessary. Parallel State-Machine Replication (P-SMR) [32] proposes to parallelize both the execution and the delivery of commands by using several multicast groups that partially order commands across replicas. These two techniques aim at speeding-up the execution of commands at each replica by enabling parallel execution of commands on multi-core systems. Our approach is orthogonal to this body of work. It improves performance by enabling different replicated state machines to execute commands in parallel without agreement.

Marandi et al. [3] employ Multi-Ring Paxos to implement consistent accesses to disjoint parts of a shared data structure. However, by construction, if an invariant is maintained between two or more partitions, the approach requires that a process receives all the messages addressed to the groups, defeating the purpose of DAP. Oster et al. [34] and Preguiça et al. [35] construct a shared tree in a purely asynchronous system under strong eventual consistency. In both cases however, the tree structure is replicated at all replicas.

Concurrently to our work, Bezerra et al. [36] have described recently an approach to partition a shared service. To execute a command, the client multicasts it to the partitions in charge of the state variables read or updated by that command. Each partition executes its part of the command, waiting (if necessary) for the results of other partitions. In comparison to our approach, this solution(i) does not take into account the application semantics implying in some cases an unnecessary convoy effect, and (ii) it requires to approximate *in advance* the range of partitions touched by the command. In the ZooKeeper use case (Volery), P-SMR stores the tree at all replicas and commands modifying the structure of the tree (`create` and `delete`) are sent to all replicas. In contrast, ZooFence exploits application semantics to split the tree into overlapping sub-trees, one stored at each partition.

The transactional paradigm is a natural candidate to partition a concurrent tree (e.g., [37]). In ZooFence, there is no need for transactional semantics because the implementation of the tree is hierarchical. A transactional history is serializable when its serialization graph is acyclic [16]. Theorem 2 can be viewed as the characterization of strictly serializable histories over abstract operations.

Ellen et al. [38] prove that no universal construction can be both DAP and wait-free in the case where the implemented shared object can grow arbitrarily. We pragmatically sidestep this impossibility result in our algorithms by bounding the size of the partition.

VII. Conclusion

This paper presents ZooFence, a system that automatically partitions ZooKeeper. ZooFence reduces contention and increases the overall throughput of applications using ZooKeeper, especially in geo-distributed scenarios. This system is based on a principled approach to partition a distributed service that we present in detail. We assess the practicability of a prototype implementation of ZooFence on two benchmarks: a concurrent queue service and the BookKeeper distributed logging engine.

Our experiments show that when locality is optimum, ZooFence improves ZooKeeper performance by almost one order of magnitude.

VIII. Acknowledgments

We are thankful to the anonymous reviewers and our shepherd for their many useful comments. The research leading to this publication was partly funded by the European Commission's FP7 under grant agreement number 318809 (LEADS), as well as by the Swiss National Science Foundation under Sinergia Project No. CRSII2_136318/1.

References

[1] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, May 1994.

[2] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," in *Proceedings of the 30th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, Madrid, Spain, Oct. 2011, pp. 255–264.

[3] P. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, ser. DSN, 2011.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX technical conference*, ser. ATC, 2010.

[5] M. Herlihy and J. Wing, "Linearizability: a correcteness condition for concurrent objects," *ACM Trans. on Prog. Lang.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[6] L. Lamport, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, 1986.

[7] M. Abadi and L. Lamport, "The existence of refinement mappings," in *3rd Annual IEEE Symposium on Logic in Computer Science*, ser. LICS, July 1988.

[8] M. Herlihy, "Wait-free synchronization," *ACM Trans. on Prog. Lang. and Systems*, vol. 11, no. 1, Jan. 1991.

[9] P. Sutra and M. Shapiro, "Fault-tolerant partial replication in large-scale database systems," in *European Conf. on Parallel Computing*, ser. Euro-Par, 2008.

[10] M. Raynal and J. Stainer, "From a store-collect object and Ω to efficient asynchronous consensus," in *European Conf. on Parallel Computing*, ser. Euro-Par, 2012.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[12] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *13th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC, 1994.

[13] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing research agenda," *ACM SIGACT News*, vol. 40, no. 2, pp. 68–80, Jun. 2009.

[14] Apache Software Foundation, "Apache Bookkeeper," 2013. [Online]. Available: zookeeper.apache.org/bookkeeper

[15] E. W. Dijkstra, "The origin of concurrent programming," P. B. Hansen, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 2002, ch. Cooperating Sequential Processes, pp. 65–138.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[17] Sun Microsystems, Inc., "NFS: Network file system protocol specification," Network Information Center, SRI International, RFC 1094, Mar. 1989.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *19th ACM Symposium on Operating Systems Principles*, ser. SOSP, 2003.

[20] B. Calder and *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *23rd ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011.

[21] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *7th symposium on Operating Systems Design and Implementation*, ser. OSDI, 2006.

[22] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "Corfu: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 10:1–10:24, Dec. 2013.

[23] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *24th ACM Symposium on Operating Systems Principles*, ser. SOSP, 2013.

[24] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[25] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[26] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, 2002.

[27] L. Lamport, "Generalized consensus and paxos," Microsoft, Tech. Rep. MSR-TR-2005-33, March 2005.

[28] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *8th ACM European Conference on Computer Systems*, ser. EuroSys, 2013.

[29] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[30] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *11th ACM Symposium on Operating Systems Principles*, ser. SOSP, 1987.

[31] K. P. Birman, "Replication and fault-tolerance in the ISIS system," in *10th ACM Symposium on Operating Systems Principles*, ser. SOSP, Dec. 1985.

[32] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *34th International Conference on Distributed Computing Systems*, ser. ICDCS, July 2014.

[33] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: Execute-verify replication for multi-core servers," in *10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI, 2012.

[34] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *ACM Conference on Computer-Supported Cooperative Work*, ser. CSCW, Nov. 2006.

[35] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *29th International Conference on Distributed Computing Systems*, ser. ICDCS, Montréal, Canada, Jun. 2009, pp. 395–403.

[36] C. E. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *45th International Conference on Dependable Systems and Networks*, ser. DSN, June 2014.

[37] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed b-tree," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 598–609, Aug. 2008.

[38] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers, "Universal constructions that ensure disjoint-access parallelism and wait-freedom," in *ACM Symposium on Principles of Distributed Computing*, ser. PODC, 2012.

APPENDIX

A. Proof of Theorems 1 and 2

Theorem. *Consider a service S and a partition $(S_k)_k$ of S. If both $\prod_k States_k = States$, and $Cmd = \bigcup_k Cmd_k$ hold then $(S_k)_k$ is a consistent partition of S.*

Proof: Let h be a linearizable history of $(S_k)_k$. For each part S_k, history $h|S_k$ is linearizable. Thus, there exists per S_k a history h'_k completing $h|S_k$ and a sequential history l_k equivalent to h'_k such that l_k is legal for S_k and $<_{h'_k} \subseteq <_{l_k}$ holds.

Name $<$ the transitive closure of the union of $(\bigcup_k <_{l_k})$ and $<_h$. For the sake of contradiction, assume that $<$ is not a partial order over the commands in h. It follows that for some commands $c_1, \ldots, c_{n>2}$, we have a cycle $c_1 < \ldots < c_n < c_1$. Clearly, we cannot have in this cycle neither only orders $<_h$, nor at most one order $<_h$. Thus, consider some i for which we have $c_{i-1[n+1]} <_h c_i <_{l_k} c_{i+1[n+1]} <_h c_{i+2[n+1]}$. Necessarily, c_{i+1} does not precede c_i in h. From which it follows (in a global time model) that $c_{i-1} <_h c_{i+2}$ holds. By applying this reduction, we conclude that there exists a cycle with only one order $<_h$; a contradiction.

We append to h all the responses that are in the histories h'_k and not in h to form H. Since $(S_k)_k$ is a partition of S, H is a history of S. Moreover, every command c is complete in H as it was complete in some h'_k.

We concatenate the commands in $(l_k)_k$ in the order $<$ to form L. By definition of $<$, we have $<_H \subseteq <_L$. History L is by construction sequential. Then, consider that a transition from $s = (s_1, \ldots, s_n)$ to $s' = (s'_1, \ldots, s'_n)$ occurs in L for some command $c \in Cmd_k$ with a response value val. Since $(S_k)_k$ is strict, we have $\tau(s, c) = ((s_1, \ldots, \tau_k(s_k, c).st, \ldots, s_n), \tau_k(s_k, c).val) = (s', val)$. This leads to $(s, c, val, s') \in \tau$. As a consequence, the history L is legal. It follows that L is a linearization of H. ∎

Theorem. *A partition $(S_k)_k$ of a service S is consistent if for every history h of $(S_k)_k$, there exists some linearization l of h such that the semantic graph of S induced by l is acyclic.*

Proof: Let us assume the existence of some history l such that(i) for every S_k, history $l|S_k$ is a linearization of $h|S_k$, and (ii) the semantic graph $G_l = (V, E)$ of S induced by l is acyclic. We have to show that there exists some linearizable history H of S such that H is a high-level view of h.

First, let us define the following mapping $\mathfrak{F}$ from h to a set of invocation and response events: Given a command $c = \langle c_1, \ldots, c_m \rangle$ that appears in h, we map every event e regarding the sub-commands of c, except $res(c_m)$, via $\mathfrak{F}$ to $inv(c)$. Provided that the event $res(c_m)$ exists in h, we map it via $\mathfrak{F}$ to $res(c)$. Then, we consider the following relation $\ll$ on the image of h via $\mathfrak{F}$;

$$e \ll e' = \begin{array}{l}(\exists (c, d) \in E : e = inv(c) \ \wedge \ e' = inv(d)) \\ \vee \ (\mathfrak{F}^{-1}(e) <_h \mathfrak{F}^{-1}(e'))\end{array}$$

Clearly, $\ll^*$ is a partial order on $\mathfrak{F}(h)$. Hence, we can define H as the concatenation of the invocations and response events in $\mathfrak{F}(h)$ following some topological order compatible with $\ll^*$. By construction, H is a higher-level view of h. Moreover, since $(S_k)_k$ is a partition of S, H is a history of S.

Second, we append to h all the responses that are in l and not in h to form h'. Since for every S_k, history $l|S_k$ is a linearization of $h|S_k$, every sub-command in l is complete. As a consequence, history h' completes history h. Then, following some order compatible with E, for every command c incomplete in h', we append to h' some sequential execution of the sub-commands in c missing in h'. Following the algorithm we employed to construct H, we then build H' based on h'. By construction, H' completes H.

Now, let us choose L as some sequential history equivalent to H' and satisfying $<_{H'} \subseteq <_L$. We argue by induction that L is legal for the service S. Consider some command $c = \langle c_1, \ldots c_m \rangle$ returning in L a value val, and assume that $L_{|<c}$ is legal. Note $s = (s_1, \ldots, s_n)$ the final state of S in $L_{|<c}$. Without lack of generality, we assume that $m = |(S_k)_k|$ and that every sub-command c_k executes on a distinct S_k. We have to show that(i) for every k, denoting $\hat{s}_k$ the state of S_k in $l|S_k$ before the execution of c_k, we have $\hat{s}_k = s_k$, as well as, (ii) val is the response returned by c_m in l. We observe that (ii) holds by definition of $\mathfrak{F}$. Then for (i), let d_k be the sub-command of some command d preceding c in L that produces s_k. Observe that for every command e_k such that $d_k <_{l|S_k} e_k <_{l|S_k} c_k$, because $<_L$ is compatible with the order E, d_k and e_k commute. From which we deduce that $\hat{s}_k = s_k$. ∎

B. Correctness of Algorithm 2 and Algorithm 3

In this section, we sketch a correctness proof of Algorithms 2 and 3. To that end, we consider some partition $(S_k)_k$ of a service S. First of all, we observe that since for every part S_k, $\mathcal{M}(S_k)$ is a linearizable implementation of S_k, all our constructions are linearizable implementation of $(S_k)_k$. In the case where $|\mathcal{M}(S_k)| > 1$, this follows from the idempotency of sub-commands and the fact that we apply each sub-command c_i in some canonical order at all the replicated state machines in $\mathcal{M}(S_k)$ before returning the response of c_i.

Algorithm 2. First of all, consider that some client p executes a command c. Since p registers at line 8, by the properties of the leader election service Ω, eventually some correct client process q executes lines 15 to 20 for command c. Then, since every replicated state machine $\mathcal{M}(S_k)$ is wait-free, all the sub-commands of c returns. It follows that eventually p returns from its invocation of c. Besides, we observe that the preconditions at line 15 implies that if two commands are concurrently executed in some execution of Algorithm 2, they must be commuting. As a consequence, the semantics graph produced by any execution of Algorithm 2 is acyclic. Applying Theorem 2, we conclude that the partition implemented by Algorithm 2 is consistent.

Algorithm 3. Consider some linearization l of an history h produced by Algorithm 3. At first glance, assume that any two commands of S executed in l are commuting. It follows that the semantic graph induced by l is acyclic, and we may apply

Theorem 2. Then, assume for the sake of contradiction that G_l contains a cycle $\mathcal{C}$ of non-commuting commands. We note $(S_{k'})_{k'}$ the restriction of $(S_k)_k$ to the parts appearing in cycle $\mathcal{C}$. Then, we consider the two following cases depending on whether property P1 or P2 holds:

(P1.) Consider a sub-command c_i applying on part S_i such that S_i is the smallest part for the order $\ll$ over the parts $(S_{k'})_{k'}$, and c_i creates an order (d, c) or (c, d) in G_l.

(Case (d, c).) The sub-command c_i is preceded by some d_i with $d \in \mathcal{C}$ in l. By definition of c_i, every sub-command preceding c_i in c commutes with all the sub-commands of the commands in $\mathcal{C}$. Hence, by the helping mechanism at lines 18 to 23, every sub-command d_j commutes with the sub-commands of c before c_i and is executed before c_i in l. Then, since some d_j is preceded by a sub-command $e_{j'}$ with $e \in \mathcal{C}$, the very same reasoning tells us that the sub-commands of e commute with the sub-commands before c_i and that they are executed before c_i in l. Hence, by induction, no order $(c, \_)$ occurs in $\mathcal{C}$; a contradiction.

Case (c, d) This case is symmetrical to the previous one, and thus omitted.

(P2.) Assume that both relations (c, d) and (d, e) are in G_l. Note respectively (c_i, d_i) and (d_j, e_j) the pair of sub-commands that created them. In addition, let S_i and S_j be the parts on which respectively (c_i, d_i) and (d_j, e_j) do not commute. By property P2, we have either $\mathcal{M}(S_i) \subseteq \mathcal{M}(S_j)$ or the converse that holds. From which it follows that some replicated state machine linearizes c_i and e_j. By applying this reasoning to the cycle G_l, we obtain the desired contradiction.

Let us now consider that some correct client process p executes a command c. Since no two sub-commands in c access the same part, command c never blocks waiting for itself at line 19 Then, consider that when executing line 19, a client process always returns the first command that satisfies the predicate. In such a case, a short induction on the eventual properties of the leader election services in S together with the fact that G_l is acyclic, tell us that every sub-command eventually returns from its invocation. From which we deduce that Algorithm 3 is wait-free.

Finally, we observe that if two commands c and d access the same replicated state machine in some $\mathcal{M}(S_k)$, there must exist a non-directed path in conflict graph of l. Hence, Algorithm 3 is disjoint-access parallel.

C. High-Level Specification of ZooFence

We consider a tree as an undirected graph $T = (N, E)$ in which any two vertices are connected by exactly one simple path. We note n_0 the root of the tree. Each node n in the tree stores some (initially null) data $n.d$. As usual, we make no distinction between a node and the unique path starting from n_0 that reaches it. For some path p, we note $\mathit{parent}(p)$ the parent of p, and given two paths p and p', we write $p \sqsubseteq p'$ when p prefixes p'. Clients manipulate the tree through the following interface:

- `exist`(x) return true iff x is in T.
- `getChildren`(x) returns the children of x in T.
- `create`(x): if $\mathit{parent}(x)$ exists in T, add x as a child of $\mathit{parent}(x)$ and returns true; otherwise false is returned.
- `delete`(x): if x has no children then deletes it from T and returns true; otherwise returns false.
- `update`(x, d): if node x exists in T, updates its content with data d and returns true; otherwise false is returned.

To partition the above interface, we need to satisfy property P2. We translate this as the fact that if $p \sqsubseteq p'$ then $\mathcal{M}(p') \subseteq \mathcal{M}(p)$. Multiple mappings are possible, and as pointed out in Section IV, such a choice is application-dependent. For instance, let $\kappa, \lambda \in \mathbb{N}$ be some *flattening parameters*. We define $\mathcal{M}(n_0)$ as the set of all the replicated state machines. Then, for every path p of length $|p|$, if $|p| \neq 0\,[\kappa]$, we remove deterministically λ replicated state machines from $\mathcal{M}\mathit{parent}(p)$; otherwise we let $\mathcal{M}(p)$ equals $\mathcal{M}(\mathit{parent}(p))$.

Based on the above assignment of nodes to replicated state machines, we then partition the tree as follows: Each replicated state machine implements a shared tree that exposes the very same interface as the one we described above. Commands at the client level are implemented by the following sequences of commands at the replicated state machine level (underlined):

```
exist(x)       = return M(x).exist(x)
update(x, d)   = return M(x).update(x, d)
getChildren(x) = return M(x).getChildren(x)
create(x)      = return M(parent(x)).create(x)
delete(x)      = if M(x).getChildren(x) ≠ ∅
                   return false
                 M(parent(x)).delete(x)
                 return true
```

A Practical Distributed Universal Construction with Unknown Participants

Pierre Sutra, Étienne Rivière, and Pascal Felber

University of Neuchâtel, Switzerland

Abstract. Modern distributed systems employ atomic read-modify-write primitives to coordinate concurrent operations. Such primitives are typically built on top of a central server, or rely on an agreement protocol. Both approaches provide a *universal construction*, that is, a general mechanism to construct atomic and responsive objects. These two techniques are however known to be inherently costly. As a consequence, they may result in bottlenecks in applications using them for coordination. In this paper, we investigate another direction to implement a universal construction. Our idea is to delegate the implementation of the universal construction to the clients, and solely implement a distributed shared atomic memory on the servers side. The construction we propose is obstruction-free. It can be implemented in a purely asynchronous manner, and it does not assume the knowledge of the participants. It is built on top of *grafarius* and *racing* objects, two novel shared abstractions that we introduce in detail. To assess the benefits of our approach, we present a prototype implementation on top of the Cassandra data store, and compare it empirically to the Zookeeper coordination service.

1 Introduction

The management of conflicting accesses to shared data plays a key role in executing correctly and efficiently distributed applications. In general, strongly consistent operations on shared data are serialized either through a central server, or using the replicated state machine approach (e.g., with the Paxos consensus protocol [1]). These two techniques implement a wait-free universal construction, that is, a general mechanism to obtain responsive atomic objects [2]. It is however well-established that these two mechanisms are costly. This comes from the fact that in both cases a server serializes all updates emitted by the clients, creating a potential bottleneck in the system. Furthermore, central servers require human intervention to be constantly operational, and replicated state machines are known to be difficult to deploy and maintain.

In this paper, we propose to delegate the logic of strongly consistent operations to the client side, and to replace the central server/replicated state machine by a distributed shared memory. The resulting universal construction is dependable, while being conceptually simpler than state-machine-replication. Similar in spirit to [3, 4], or more recently [5], we aim at bridging the gap that exists in practice

between shared memory literature on universal constructions and their counterparts in distributed systems. Our approach is nonetheless different as we do not rely on a shared log to order all accesses, but instead make use of a distinct set of registers to implement each object used by the application. This leverages the intrinsic parallelism of the workload.

To achieve this, our first contribution is an obstruction-free universal construction on top of an asynchronous distributed shared memory that works even if the participants are unknown. We base our construction on two novel abstractions: a *grafarius* and a *racing*. A grafarius is close to the more common notion of ratifier, or adopt-commit object [6, 7]. A racing object encapsulates the behavior of algorithms that repeatedly access new objects to progress. By combining these two abstractions, we devise an obstruction-free universal construction whose time complexity is optimal during contention-free executions.

Our previous solution makes use of an unbounded amount of memory to store the state of the object it implements. We solve this problem with a second contribution, in the form of a novel memory management mechanism. We formalize the notion of recycled objects then propose a mechanism to recycle all the base objects of our previous implementation. In a distributed system, the time complexity of the resulting universal construction during uncontended executions is constant, and it uses $O(k^2)$ shared registers, where k is the amount of processes that actually access the construction.

Our third contribution is a practical assessment of this approach. We present a prototype implementation on top of the Cassandra distributed data store [8] which we compare to Zookeeper, a state-of-the-art coordination service [9]. Several empirical results show that our system achieves results comparable to Zookeeper when clients rarely contend on shared objects, and that in addition, it exhibits a good scalability factor. For instance, with 12 servers and when the workload is completely parallel, our system is as dependable as a 3 servers deployment of Zookeeper, while being 3.2 times faster. This last property comes from the fact that our approach exhibits no bottleneck. Thus, the more it scales-out, the more likely operations that access distinct objects execute in parallel on the servers, improving performance.

Paper Outline. Section 2 surveys related work. In Section 3, we introduce the notions of grafarius and racing objects, and we present our first universal construction. We refine this construction to bound its memory footprint in Section 4. Section 5 describes a prototype implementation of our algorithm on top of Cassandra, and we evaluate it against Zookeeper. We close in Section 6. For the sake of clarity, all the proofs are deferred to the appendix.

2 Related Work

Our work deals with the problem of transforming a sequential object into a concurrent strongly-consistent one. Such a mechanism is named in literature a *universal construction*. At core of this construction is consensus, an abstraction with which processes agree on the next state of the concurrent object. In a distributed system, the classical approach to implement consensus is the Paxos

algorithm [1]. Due to the impossibility result of Fischer et al. [10], Paxos is indulgent [11]. This means that Paxos guarantees safety at all times but provides progress only under favorable circumstances. The alpha of consensus [12] captures the indulgent part of Paxos. This notion is close to the ranked-register object [13] which models the Disk Paxos algorithm of Gafni and Lamport [4].

Processes executing Paxos iteratively calls the alpha abstraction with a tuple (k, v), where k is a round number and v some (appropriately chosen) proposal value. Each such call translates the execution of a round in the original algorithm of Lamport [1]. A ratifier, or *adopt-commit*, object [6] is a one-shot object encapsulating the safety property of a round. Hence, from a high-level perspective, the alpha of consensus can be seen as successive (consistent) calls to adopt-commit objects (see [7] or [14, Fig.5]). In Section 3.3, we present the *racing* object that allows abstracting such iterative calls.

When there is no assumption on the proposed values, the result of Aspnes and Ellen [15] tells us that the solo time complexity of an adopt-commit belongs to $\Omega\left(\sqrt{\log n}/\log\log n\right)$. Surprisingly, some consensus algorithms exhibit constant solo decision time (e.g., [16]). This difference is explained by the convergence property of adopt-commit objects which requires processes to commit a value in case they all propose it. In Section 3.2, we introduce the notion of *grafarius* object. A grafarius can be seen as an adopt-commit object with a weak convergence property, namely a process has to commit its value only if it executes solo. As shown in Section 3.4, we can build an obstruction-free consensus with constant solo decision time on top of the grafarius and racing objects.

Some algorithms, e.g. [14, 16] in shared-memory, or [3–5, 13] in distributed systems, use strong synchronization primitives to implement consensus. On the contrary, our approach relies solely on a set of registers emulated by the underlying distributed system. As a consequence of this choice, our universal construction is obstruction-free. The work of Fich et al. [17] describes a practical transformation to convert an obstruction-free algorithm into a wait-free one. Jayanti et al. [18] proves an $\Omega(n)$ lower bound on the solo decision time and the space complexity of obstruction-free implementations.

During a step-contention free execution, processes do not contend on the base objects that implement the desired abstraction. The work of Attiya et al. [14] studies obstruction-free implementations that only make use of history-less primitives during step-contention free executions, but might rely on stronger ones under contention. The authors show that such implementation have $\Omega(n)$ space complexity, and that they exhibit $\Omega(\log n)$ time complexity in step-contention free executions.

The time complexity of the wait-free universal construction of Herlihy [2] is $O(n)$. Jayanti and Toueg [19] propose a variation of this construction which does not use unbounded integers. The space complexity of this last algorithm is $O(n^2)$. Attiya et al. [14] present an obstruction-free universal construction that employs an unbounded amount of memory. In Section 4.3, we describe a space-bounded universal construction that works in the case where processes participating to the construction are unknown. In a distributed system, it makes use of $O(k^2)$

registers, where k is the amount of processes that actually access the construction. To achieve this, we present a novel recycling mechanism in Section 4.2. At core of our mechanism is the observation that properly recycled grafarius objects can be concurrently accessed in different rounds.

3 The Construction

This section first introduces our system model. Then, it details the grafarius and the racing objects. Based on these two abstractions, we further depict a consensus algorithm that exhibits a constant time complexity in the contention-free case. This algorithm is our core building block to obtain an efficient universal construction. All the objects we present hereafter are depicted in the asynchronous shared-memory model, and they all support a bounded yet unknown amount of processes. These two assumptions reflect the message-passing system we target.

3.1 System Model & Notations

We consider an asynchronous message-passing system characterized by a complete communication graph where both communication and computation are asynchronous. Processes take their identities from some bounded set Π, with $n = |\Pi|$. The set Π is not accessible to processes for computation, but they may execute operations on the identities (e.g., equality tests).

During an execution, a process can fail-stop by crashing, but we assume that at most $\lceil \frac{n}{2} \rceil - 1$ such failures occur. There exists an implementation of an asynchronous shared-memory ($\mathcal{ASM}$) under such an assumption [20, 21]. Consequently, we shall write all our algorithms in the $\mathcal{ASM}$ model where processes communicate by reading and writing to atomic multi-writer multi-reader (MWMR) registers.

In what follows, we detail how to implement higher level abstractions using the shared registers. Most of the objects we describe in this paper are linearizable [22]. An object is *one-shot* when a process may call one of its operations at most once. When there is no limit to the number of times a process may invoke the object's operations, the object is *long lived*. Besides, we shall be considering the following two progress conditions on the invocations and responses of operations [23]:(*Obstruction-freedom*) if at some point in time a process runs solo then eventually it returns from the invocation; and (*Wait-freedom*) a process returns from the invocation after a bounded number of steps.

In this paper, we are most interested in executions where processes rarely contend on shared objects. The canonical case of such an execution is the *solo* execution in which a single process executes computational steps. This class of execution is appropriate for one-shot objects but we need extending it for long-lived ones. To that end, we define the notion of *contention-free* execution that is an execution during which calls to the implemented shared object do not interleave. The *contention-free time complexity* of an algorithm is the worst case number of steps made by a process during such executions.

Algorithm 1 Grafarius – code at process p

```
1: Shared Variables:
2:     s                                              // A splitter object
3:     c                                              // Initially, false
4:     d                                              // Initially, ⊥
5:
6: adoptCommit(u) :=
7:     if ¬ s.split() then
8:         c ← true
9:         if d ≠ ⊥ then
10:            return (adopt, d)
11:        d ← u
12:        return (adopt, u)
13:    d ← u
14:    if c then
15:        return (adopt, u)
16:    return (commit, u)
```

3.2 Grafarius

The first abstraction we employ in our construction is a shared object named *grafarius*. A grafarius is a one-shot object defined on a domain of values $\mathcal{V}$. It exports a single operation $adoptCommit(u \in \mathcal{V})$ that returns a pair $(flag, v)$, with $flag \in \{adopt, commit\}$ and $v \in \mathcal{V}$. During every history of a grafarius, and for every process p that invokes $adoptCommit(u)$, the following properties are satisfied:(*Validity*) If p adopts v, some process invoked the operation $adoptCommit(v)$ before. (*Coherence*) If p commits v, every process either adopts or commits v. (*Solo Convergence*) If p returns from its invocation before any other process invokes *adoptCommit* then p commits u. (*Continuation*) If some process returns before p invokes *adoptCommit*, p adopts or commits a value proposed before it invokes *adoptCommit*.

The grafarius is closely related to the notion of adopt-commit object introduced by Gafni [6]. Nevertheless, the two abstractions are not comparable. On the one hand, the solo convergence property of a grafarius is weaker than the convergence property of an adopt-commit object. This avoids the lower bound $\Omega\left(\sqrt{\log n}/\log\log n\right)$ on the time complexity to execute *adoptCommit* in $\mathcal{ASM}$ [15], while being sufficient to implement obstruction-free consensus. On the other hand, an adopt-commit object does not satisfy the continuation property, meaning that a process can return a value $(u, adopt)$ despite that such invocation follows a call which returned $(v, adopt)$. The continuation property improves convergence speed under contention. This also makes the grafarius a decidable object, which is needed by our memory management schema. We give further details regarding this last point in Section 4.

Algorithm 1 depicts a wait-free implementation of a grafarius. This algorithm makes use of a splitter object that detects a collision when two processes concurrently access the shared object. We first remind below how a splitter works, then we detail the internals of Algorithm 1.

The splitter object was first introduced by Lamport [24] then later formalized by Moir and Anderson [25]. A splitter is a one-shot shared object that exposes a single operation: *split*(). This operation takes no parameter and returns a value in

Algorithm 2 Racing on $\mathcal{L}$ – code at process p

```
1: Shared Variables:
2:     L                                          // A map from Π to ℕ, initially ∅
3:
4: Local Variables:
5:     F                                          // A function from ℕ to ℒ
6:     last                                       // Initially, 0
7:
8: enter() :=
9:     L[p] ← last
10:    S ← codomain(L)
11:    let m = max(S)
12:    if last = m then
13:        last ← m + 1
14:    else
15:        last ← m
16:    return F(last)
```

$\{true, false\}$.[1] When a process returns *true*, we shall say that it *wins* the splitter; otherwise it *loses* the splitter. When multiple processes call *split*(), at most one receives the value *true*, and if a single process calls *split*(), this call returns *true*. Furthermore, when a process calls *split*() after some other process returned, it necessarily loses the splitter. A splitter is implementable in a wait-free manner with atomic MWMR registers (see [25, Fig. 2] for further details).

Algorithm 1 works as follows. Upon calling *adoptCommit*(u), a process p tries to win the splitter (line 7). If p fails, it raises the flag c to record that a collision occurred, i.e., the fact that two processes concurrently attempted to commit a value. Then, in case a decision was recorded, p adopts it; otherwise p adopts it own value (lines 9 to 12). On the other hand, if p wins the splitter, it writes its proposal u to the register d. Then, process p commits u if it detects no conflict, otherwise p adopts it (lines 14 to 16).

3.3 Notion of Racing

Many algorithms (e.g., [3, 26]) repeatedly access new objects to progress. A *racing* is a long-lived object that captures such an iterative pattern. Its interface consists of a single operation *enter*(p, l), defined on a countably infinite domain $\mathcal{L}$ of *laps*. During a history h, a process p *enters* lap l when *enter*(p, l) occurs in h. Process p *leaves* lap l when l is the last lap entered by p and p enters a new one. The following invariant holds during every history of a racing:*(Ordering)* There exists a strict total order $\ll_h$ on the set of entered laps in h such that for every process p that enters some lap l, either(i) some process left l before p enters it, or (ii) the last lap left by p is the greatest lap smaller than l for the order $\ll_h$.

Let us consider an unbounded counter c at each process, and an indexing function F from $\mathbb{N}$ to $\mathcal{L}$. Whenever a process p enters a new lap, suppose that p increments c and then returns the object $F(c)$. This simple local algorithm implements a linearizable racing. However, because this construction does not

[1] A splitter is generally defined with the returned values $\{L, S, R\}$. Here, we make no distinction between L and R.

Algorithm 3 Consensus – code at process p

```
1: Shared Variables:
2:     R                                          // A racing on grafarius objects
3:     d                                          // Initially, ⊥
4:
5: propose(u) :=
6:     while true do
7:         if d ≠ ⊥ then
8:             return d
9:         o ← R.enter()
10:        (f, u) ← o.adoptCommit(u)
11:        if f = commit then
12:            d ← u
```

bound the amount of laps a process has to retrieve before knowing the most recent one, it might be expensive when contention occurs.

Algorithm 2 presents a more efficient approach that allows a process to skip the laps it missed. This algorithm makes use of an initially empty shared map L from Π to $\mathbb{N}$. We map x to the value y via L when writing $L[x] \leftarrow y$; operation $\mathit{codomain}(L)$ returns the codomain of L. For some process p, the local variable *last* stores the index of the last lap entered by p. When it calls $\mathit{enter}()$, process p stores the *last* index in L (line 9). Then, p retrieves the content of L and computes the maximum element m in its codomain. Process p assigns $m + 1$ to *last*, if $\mathit{last} = m$ holds, and m otherwise (lines 12 to 15). The value of $F(\mathit{last})$ is then returned as the result of the call.

Time Complexity. The adaptive collect object of Attiya et al. [27] can implement the shared map L used in Algorithm 2, without having the knowledge of Π. In such a case, the time complexity of Algorithm 2 is $O(k)$, where $k \leq n$ denotes the amount of processes that actually access the racing object.

3.4 Racing-based Consensus

Using the racing abstraction introduced in the previous section, we now depict an obstruction-free implementation of consensus. Recall that consensus is a shared object whose interface consists of a single method *propose*. This method takes as input a value from some set $\mathcal{V}$ and returns a value in $\mathcal{V}$ ensuring both (*Validity*) if v is returned then some process invoked $\mathit{propose}(v)$ previously, and (*Agreement*) two processes always return the same value.

Algorithm 3 describes our implementation of consensus. In this algorithm, processes compete on two shared abstractions: a racing R on grafarius objects, and a decision register d. When a process p suggests a value u for consensus, it attempts to commit u by entering the next grafarius object in R (line 9). Every time p executes *adoptCommit* on a grafarius object o, p updates its proposed value with the response returned by o (line 10). In case the grafarius returns a committed value, this value is stored in d as the result of the call to *propose* (lines 11 and 12).

Time complexity. The call to the splitter object in Algorithm 1 requires four computational steps [25]. Besides, the solo time complexity of the adaptive collect object of Attiya et al. [27] belongs to $O(1)$. It follows that Algorithm 3 solves

Algorithm 4 Universal Construction – code at process p

```
1: Shared Variables:
2:     R                                          // A racing on consensus objects
3:
4: Local Variables:
5:     C                                          // Initially, R.enter()
6:     s                                          // Initially, s_0
7:
8: invoke(op) :=
9:     while true do
10:        d ← C.d
11:        if d ≠ ⊥ then
12:            s ← d[1]
13:            C ← R.enter()
14:        else
15:            (s', v) ← τ(s, op)
16:            if s = s' then
17:                return v
18:            d ← C.propose((p, s'))
19:            if d[0] = p then
20:                return v
```

consensus in $O(1)$ steps during solo executions. This fast resolution of consensus allows us to implement a universal construction with a linear time complexity when no contention occurs. We detail our approach in the next section.

3.5 A Fast Obstruction-free Universal Construction

A universal construction is a general mechanism to obtain linearizable shared objects from sequential ones. A sequential object is specified by some serial data type that defines its possible states as well as its access operations. Formally, a serial data type is an automaton defined bya set of states ($States$), an initial state ($s_0 \in States$), a set of operations (Op), a set of response values ($Values$), and a transition relation ($\tau : States \times Op \rightarrow States \times Values$). Hereafter, and without lack of generality, we shall assume that every operation op is *total*, i.e., $States \times \{op\}$ is in the domain of τ.

Algorithm 4 depicts our obstruction-free linearizable universal construction. The algorithm uses a single shared variable: a racing on obstruction-free consensus objects named R. When a process p invokes an operation via $invoke(op)$, p first checks the decision of the latest consensus object it entered (line 11). If a decision was taken, p updates its local variable s with the new state of the object. Then, p enters the next consensus (lines 12 to 13). Once p reaches the last consensus that was decided, variable s stores a state of the object that is older than the time at which p invoked op. At this point, process p executes tentatively the operation on s and stores the result in the pair (s', v). When s equals s', the invocation does not change the result of the object and p can immediately returns v. Otherwise, p proposes the pair (p, s') to change the state of the object to s'. If it successes, process p returns the response v (lines 19 and 20).

Time complexity. As pointed out previously, the case we consider to be the most frequent one is the contention-free case, that is when multiple processes access the object but interleavings do not occur. In the worst case, a process freshly calling $invoke()$ in a contention-free execution first retrieves the largest

decided consensus, then it enters the next consensus and decides. From our previous time complexity analysis of Algorithms 2 and 3 and the lower bound result of Jayanti et al. [18], the contention-free time complexity of Algorithm 4 is optimal and belongs to $\Theta(k)$.

4 Managing Memory Usage

Every time the state of the object implemented by the universal construction changes, Algorithm 4 accesses a new consensus instance. This implies that the number of consensus instances is not bounded and may rapidly exhaust available memory. In this section, we present a novel recycling technique that addresses this problem. To that end, we first introduce several definitions that capture the notion of recycled objects. Then, we depict a mechanism to recycle the objects used in Algorithm 4.

4.1 Preliminary Notions

Intuitively, every time an object is reused, it should behave according to its specification. We formalize this idea in the definitions that follow.

Definition 1 (*Round & Decomposition*). *Given some history h, a round r of h is a sub-history of h such that every invocation complete in h is complete in r. A* decomposition *of h is an ordered set of rounds $\{r_1 \ldots . r_{m \geq 1}\}$ satisfying $h = r_1 \ldots . r_m$.*

Definition 2 (*Recycled Object*). *Consider a history h of some object o. We say that o is a* recycled *object of type $\mathfrak{T}$ during h, when there exists a decomposition of h such that every round r in this decomposition is a correct history for an object of type $\mathfrak{T}$.*

In order to illustrate these definitions, let us consider two processes p and q, and a shared object o exporting an operation op. We can decompose history $h_1 = inv_{p,1}(op).inv_{q,1}(op).res_{q,1}(op)u.res_{p,1}(op)v.inv_{p,2}(op)$ in rounds $r_1 = inv_{p,1}(op).inv_{q,1}(op).res_{q,1}(op)u.res_{p,1}(op)v$ and $r_2 = inv_{p,2}(op)$. However, if we consider that $op = propose$ and $u \neq v$, there is no decomposition of h_1 for which o is a recycled consensus object.

The usual approach to recycle an object is to reset all its fields once the processes have stopped accessing it, that is once all the operations pending in a round have completed. The universal construction of Herlihy [2] implements this idea by provisioning for each process $O(n^2)$ cells, each cell storing the state of the implemented object. An array of $O(n)$ bits associated to each cell indicates when it can be reset by its owner.

Since the participants to the universal construction are unknown in our context, we cannot employ the previous approach. Instead, we propose to recycle the objects used in Algorithm 4 by signing each modification with the round at which it occurs. An operation that updates such an object will be oblivious to modifications made in prior rounds. If now the operation is in late, that is when a new round has started before it returns, the operation will observe the object in a state consistent with one of the rounds to which it is concurrent. We develop this idea in the next section, then apply it to Algorithm 4.

4.2 Recycling Objects

As a starter, let us remind the definition of a decidable object. This category of objects contains consensus, but also the splitter and the grafarius objects we described in Section 3.

Definition 3 (Decidable Object). *A decidable object o is a shared object whose state contains a* decision *register d taking its value in some set $\mathcal{V}$, the domain of o, union $\perp \notin \mathcal{V}$, and which initially equals $\perp$. The object is said* decided *when $d \in \mathcal{V}$ holds. For every operation op of o, once o is decided, there exists some deterministic function f of d such that $f(d)$ is a valid response value for op.*

As an example of the previous definition, let us consider a grafarius object. We observe that when the decision register d does not equal $\perp$, the pair $(adopt, d)$ is a sound response for the call *adoptCommit*.

The first step of our recycling mechanism consists in recycling the MWMR registers that form the basic building blocks of our algorithms. We detail it below.

(**Construction 1**) Let $(\mathcal{T}, <)$ be a set of timestamps totally ordered by some relation $<$ and containing a smallest element $0 \in \mathcal{T}$. For every register x having some initial state s_0, we initialize x to $(0, s_0)$. Then, consider some timestamp t. When a value v is written to x, we write (t, v) to x. Now, upon reading from x, the value returned is the value u in case x contains (t', u) with $t \leq t'$, and s_0 otherwise.

In a second step, we extend this technique to decidable objects as follows.

(**Construction 2**) For some decidable object o, a call to $recycle(o, t)$ returns a copy of o such that upon a call to an operation op of o,(i) if the object is decided then we return $f(d)$, and otherwise (ii) op is executed but read and write operations on the shared registers that implement o are replaced by the steps described in Construction 1 using timestamp t.

For some decidable object o, we shall write $recycle(o)$ the object obtained by proxying every call to the operations of o by corresponding calls to $recycle(o, t)$ for some timestamp t. Proposition 1 establishes that, provided the timestamps are appropriate, $recycle(o)$ implements a recycled object of the same type as o.

Proposition 1. *Consider a decidable object o of type $\mathfrak{T}$ and some history h of $recycle(o)$ during which the following invariant holds:*

(P1) For any pair of operations op and op', executed respectively on $recycle(o, t)$ and $recycle(o, t')$ in h, if op' does not precede op in h and $t' < t$ holds, there exists an operation on $recycle(o, t')$ that precedes op' in h and writes to the decision register d of o.

Then, $recycle(o)$ implements a recycled object of type $\mathfrak{T}$ during history h.

Algorithm 5 Universal Construction – code at process p

```
1:  Shared Variables:
2:      L                                          // A map from Π to ℕ, initially ∅
3:
4:  Local Variables:
5:      F                                          // A function from ℕ to consensus objects
6:      s                                          // Initially, s_0
7:      last                                       // Initially, 0
8:      ts                                         // Initially, 0
9:      C                                          // Initially, enter()
10:
11: invoke(op) :=
12:     while true do
13:         d ← C.d
14:         if d ≠ ⊥ then
15:             (s, last, ts) ← (d[1], d[2], d[3])
16:             C ← enter()
17:         else
18:             (s', v) ← τ(s, op)
19:             if s = s' then
20:                 return v
21:             d ← C.propose((p, s', free(), ts + 1))
22:             if d[0] = p then
23:                 return v

24: function free() :=
25:     S ← codomain(L)
26:     let (γ, Γ) = (min(S), max(S))
27:     if γ > 0 then
28:         return γ − 1
29:     return Γ + 1

30: function enter() :=
31:     L[p] ← last
32:     return recycle(F(last), ts)
```

4.3 Application

Algorithm 5 depicts our second obstruction-free universal construction. In comparison to Algorithm 4, we introduce two modifications:(i) processes now compete to decide which consensus will store the next state of the object, and (ii) consensus objects are recycled using the mechanism we presented in Construction 2.

With more details, Algorithm 5 works as follows. We implement a racing on consensus objects with variables L and F. When an operation changes the state of the implemented object, the calling process proposes to consensus the new state s' together with the index of the consensus object that will be used next and its associated timestamp (line 21). The index is determined by a call to the function *free*(). This function retrieves the codomain of L, and computes the smallest consensus index that is not currently accessed by a process (lines 25 to 29). In case all objects between 0 and Γ are busy, where Γ is the greatest index accessed so far, the index $\Gamma + 1$ is returned.

Algorithm 5 recycles the consensus objects in the codomain of F using the timestamping schema we introduced in Section 4.2. During an execution, for every object $recycle(o)$ with $o \in codomain(F)$, the algorithm maintains the invariant P1 of Proposition 1. This ensures that accesses to variables L and F implement a racing on consensus objects, reducing Algorithm 5 to Algorithm 4.

Time & Space Complexity. The contention-free time complexity of Algorithm 5 is the same as for Algorithm 4, i.e., it belongs to $\Theta(k)$ in $\mathcal{ASM}$. From the code of function *free*(), Algorithm 5 employs at most $k+1$ consensus objects. In a distributed system, a quorum system can implement a collect object by emulating $O(k)$ shared registers. It results that in such a model the contention-free time complexity of Algorithm 5 measured in message delay is $O(1)$, and that its space complexity belongs to $O(k^2)$.

5 Empirical Assessment

To assess the practicability of our approach, we evaluate in this section a prototype implementation of Algorithm 5. This implementation is built on top of the Apache Cassandra distributed data store [8] which provides a distributed shared memory using consistent hashing and quorums of configurable sizes. In what follows, we describe the internals of our implementation then detail its performance in comparison to the Apache Zookeeper coordination service [9]. For the sake of reproducibility, the source code of our implementation as well as the scripts we run during the experiments are publicly available [28].

5.1 Implementation Details

Cassandra offers a data model close to the classical relational model at core of the database systems. The smallest data unit in Cassandra is a column, a tuple that contains a name, a value and a timestamp. Columns are grouped by rows, and a column family contains a set of rows. Each row is indexed by a key, and stored at a quorum of replicas (following a consistent hashing strategy). A client can read a whole row and write a column. The consistency of such operations is tunable in Cassandra. When the cluster running Cassandra is synchronized and both read and write operations operate on quorums, Cassandra provides an atomic snapshot model. This storage system also supports eventually consistent operations. When this consistency level is employed, a write operation accesses a quorum of replicas, while a read occurs at a single replica. Cassandra reconciles replicas via a timestamp-based mechanism in the background.

Prototype Implementation Our implementation uses the Python programming language and it consists of the different shared objects we detailed in the previous sections (splitter, grafarius, consensus, and universal construction). The conciseness of Python allows the whole implementation to contain around 1,000 lines of code. Our implementation closely follows the pseudo-code of the algorithms. Each object corresponds to a row in a column family, and is named after the type of the object. When an object relies on lower-level abstractions, e.g., consensus employs multiple grafarius objects, the objects' keys at the low-level are named after the key at the higher one, e.g., *consensus:12:grafarius:3*. By changing the consistency of the decision register d in Algorithm 3, we can tune the consistency of our universal construction. When d is eventually consistent, the universal abstraction is sequentially consistent for read operations; otherwise it is linearizable. In Zookeeper, updates are linearizable while read operations are sequentially consistent. For that reason, when we compare the performance of our implementation to Zookeeper during the experiments, we use the sequentially consistent variation of our algorithm.

5.2 Evaluation

We conducted our experiments on a cluster of virtualized 8-core Xeon 2.5 Ghz machines running Gentoo Linux, and connected with a virtualized 1 Gbps switched network. Network performance, as measured by ping and netperf, are 0.3 ms for

a round-trip and a bandwidth of 117MB/s. Each machine is equipped with a virtual hard-drive whose read/write (uncached) performance, as measured with `hdparm` and `dd`, are 246/200 MB/s. A *server machine* runs either Cassandra or Zookeeper. A *client machine* emulates multiple clients accessing concurrently the shared objects. During an experiment, a client executes 10^4 accesses to one or more objects. We used 1 to 20 clients machines, emulating 1 to 100 clients each, and 3 to 12 server machines. In all our experiments, the client machines were not a bottleneck.

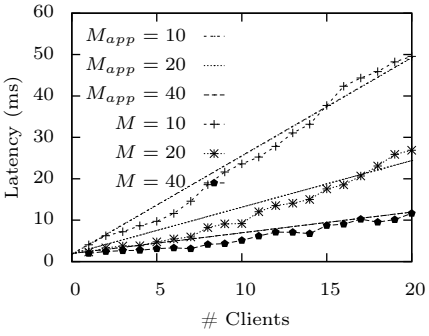


Fig. 1. Compare-And-Swap

Compare-and-swap We first evaluate in Figure 1 the performance of our implementation when clients execute compare-and-swap operations, and the system is composed of 3 server machines. Recall that a compare-and-swap object exposes a single operation: $C\&S(u, v)$. 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*. In Figure 1, we plot the latency to execute a compare-and-swap operation as a function of the number of clients and the arguments of the operations. The initial state of the compare-and-swap object is 0. Each client executes in closed-loop an operation $C\&S(k, l)$, where k and l are taken uniformly at random from the interval $[\![0, M[\![$ with M some maximum value.

When the size of the interval $[\![0, M[\![$ shrinks, each $C\&S()$ operation is more likely to success in transforming the state of the object; hence contention increases. Consequently, as observed in Figure 1, performance degrades. Contention between clients occur mainly on the splitter objects that form the building blocks of Algorithm 1. We briefly analyze how contention is related to performance next.

An operation $C\&S(u, v)$ is successful when the state is changed from u to v. Let us note ρ the ratio of successful operations, that is $1/M$, and λ_s (respectively λ_f) the latency to execute solo a successful (resp. failed) operation. In Figure 1, the light lines (M_{app}) plot for each value of M the curve $\lambda_f(1-\rho)+\lambda_s\rho n$. This is a reasonable approximation where the term $\lambda_s\rho n$ follows Little's law [29] and translates the convoy effect [30] on successful operations.

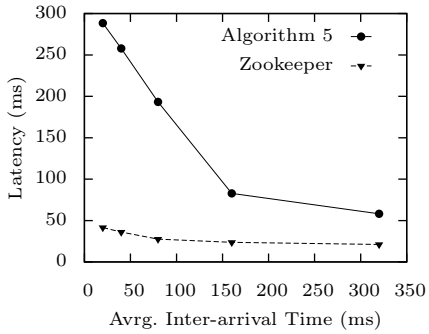


Fig. 2. Critical Section

Critical section In Figure 2, we compare the performance of our implementation to Zookeeper, when clients access a critical section (CS). Such an object is not in line with the non-blocking approach, but it is commonly used in distributed applications. We implemented the CS on top of our universal construction using

a back-off mechanism. For Zookeeper, we employed the recipe described in [9]. Figure 2 presents the average time a client takes to enter then leave the CS, and we vary the inter-arrival time of clients in the critical section according to a Poisson distribution.

We observe in Figure 2 that when the inter-arrival time is high, and thus little contention occurs, a client accesses the CS with Zookeeper in 20 ms. For Algorithm 5, it takes 60 ms, but the performance degrades quickly when clients access more frequently the CS. This comes from the fact that(i) we implemented a spinlock and thus clients are constantly accessing the system, and (ii) as pointed out previously, when clients are competing on splitter objects, the performance of our algorithm degrades.

Scalability Our last set of experiment assess the scalability of our approach. To that end, we compute the maximal throughput of our prototype implementation when clients access different objects, precisely $C\&S()$ for $M = 10$. The amount of available server machines varies from 3 to 12 servers. In all cases, we implement a register with the help of a quorum of 3 servers. We compare our results to an instance of 3 Zookeeper machines. Zookeeper does not implement natively a compare-and-swap operation. We devised the following implementation relying on the versioning mechanism exposed to the clients by Zookeeper. When a client executes $C\&S(u, v)$ it first retrieves the value w and the attached version k of the znode identifying the object. In case $w = u$, the client attempts writing v with version $k + 1$. If this write fails, the client re-execute $C\&S(u, v)$. In our experiments, a single client accesses each object. Thus it never retries and this implementation is optimal.

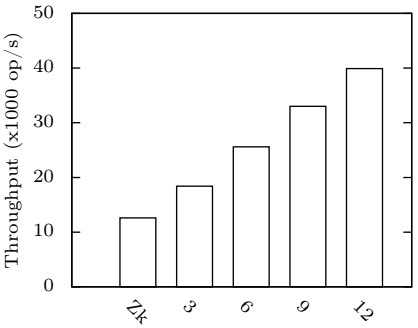


Fig. 3. Scalability

Figure 3 depicts the maximal throughput with 3 to 12 servers. With 3 servers, our system deliver 18.4K op/s and ZooKeeper 12.6K op/s. The bottleneck nature of the ZooKeeper leader which serializes all updates explains this gap. Our prototype achieves 33K op/s when using 9 servers, and 40K op/s with 12. In this last case, our system is 3.2 times faster than Zookeeper on 3 machines.

6 Conclusion

This paper presents a novel algorithmic solution to implement a distributed universal construction when participants are unknown. Contrary to previous works, which mostly focus on state machine replication, our approach employs solely a distributed asynchronous shared memory, the logic of consistent operations being delegated to the client side. Hence, and as exemplified by our prototype, we can implement it in a client library that runs on top of an off-the-shelf distributed shared memory. To obtain this result, we introduce two novel shared abstractions: a grafarius and a racing, which we believe are of interest on their own. We also present a new mechanism to recycle the base objects at core of our construction.

Bibliography

[1] L. Lamport, "The part-time parliament," in *ACM Trans. Comput. Syst.*, 1998.
[2] M. Herlihy, "Wait-free synchronization," in *ACM Trans. Program. Lang.*, 1991.
[3] M. Balakrishnan *et al.*, "Corfu: a shared log design for flash clusters," in *NSDI'12*.
[4] E. Gafni *et al.*, "Disk paxos," in *Dist. Comput.*, 2000.
[5] M. Balakrishnan *et al.*, "Tango: Distributed data structures over a shared log," in *SOSP'13*.
[6] E. Gafni, "Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony," in *PODC'98*.
[7] J. Aspnes, "A modular approach to shared-memory consensus, with applications to the probabilistic-write model," in *PODC'10*.
[8] A. Lakshman *et al.*, "Cassandra: a decentralized structured storage system," in *SIGOPS Oper. Syst. Rev.*, 2010.
[9] F. P. Junqueira *et al.*, "The life and times of a ZooKeeper," in *PODC'09*.
[10] M. J. Fischer *et al.*, "Impossibility of distributed consensus with one faulty process," in *J. ACM*, 1985.
[11] R. Guerraoui, "Indulgent algorithms (preliminary version)," in *PODC'00*.
[12] R. Guerraoui *et al.*, "The information structure of indulgent consensus," in *IEEE Trans. Comput.*, 2004.
[13] G. Chockler *et al.*, "Active disk paxos with infinitely many processes," in *PODC'02*.
[14] H. Attiya *et al.*, "The complexity of obstruction-free implementations," in *J. ACM*, 2009.
[15] J. Aspnes *et al.*, "Tight bounds for anonymous adopt-commit objects," in *SPAA'11*.
[16] V. Luchangco *et al.*, "On the uncontended complexity of consensus," in *Dist. Comput.*, 2003.
[17] F. Fich *et al.*, "Obstruction-free algorithms can be practically wait-free," in *Dist. Comput.*, 2005.
[18] P. Jayanti *et al.*, "Time and space lower bounds for nonblocking implementations," in *SIAM J. Comput.*, 2000.
[19] P. Jayanti *et al.*, "Some results on the impossibility, universality, and decidability of consensus," in *Dist. Alg.*, 1992.
[20] H. Attiya *et al.*, "Sharing memory robustly in message-passing systems," in *J. ACM*, 1995.
[21] N. A. Lynch *et al.*, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *FTCS'97*.
[22] M. Herlihy *et al.*, "Linearizability: a correcteness condition for concurrent objects," in *ACM Trans. on Prog. Lang.*, 1990.
[23] M. Herlihy *et al.*, "On the nature of progress," in *OPODIS'11*.
[24] L. Lamport, "A fast mutual exclusion algorithm," in *ACM Trans. Comput. Syst.*, 1987.
[25] M. Moir *et al.*, "Fast, long-lived renaming," in *Dist. Alg.*, 1994.
[26] R. Guerraoui *et al.*, "Anonymous and fault-tolerant shared-memory computing," in *Dist. Comput.*, 2007.
[27] H. Attiya *et al.*, "An adaptive collect algorithm with applications,", in *Dist. Comput.*, 2002.
[28] P. Sutra, "http://github.com/otrack/pssolib," 2013.
[29] A. O. Allen, *Probability, Statistics, and Queueing Theory with Computer Science Applications*. 1990.
[30] M. Blasgen *et al.*, "The convoy phenomenon," in *SIGOPS Oper. Syst. Rev.*, 1979.

A Correctness of Algorithms 1 to 4

The two theorems that follow prove the correctness of our implementations of a grafarius and a racing objects.

Theorem 1. *Algorithm 1 implements a wait-free grafarius.*

Proof. Algorithm 1 employs only wait-free shared objects. Hence, to the light of its pseudo-code, it conserves this liveness property. Validity and solo convergence are both immediate. To prove the continuation property, we observe that when a process p returns from a call to *adoptCommit*, (i) $d \neq \perp$ holds, and (ii) if a process q invokes *adoptCommit* after p returned, it loses the splitter. It remains to show that Algorithm 1 also ensures the coherence property. To that end, consider that in some execution ρ of Algorithm 1, a process p commits a value u. Note t_0 the time in ρ at which p executes line 14. Since p commits u, c always equals *false* before t_0. This implies that p wins the splitter s in ρ. From the properties of a splitter object, we conclude that every process, except p, that completes its invocation in ρ must execute line 8. This observation and the fact that $c = \mathit{false}$ at t_0 tell us that no process executes line 8 before t_0. Then, observe that p executes line 13 at some time $t_1 < t_0$. As a consequence, $d = u$ after t_0, and every process other than p that completes its invocation in ρ executes line 10. The coherence property holds in ρ.

Theorem 2. *Algorithm 2 implements a wait-free linearizable racing.*

Proof. From the pseudo-code of Algorithm 2, we conclude that this algorithm is wait-free. Then, consider a complete history h produced by the following mapping:[2] when p invokes *enter*() in Algorithm 2, it calls $\mathit{enter}(p, l)$ in h with $l = F(\mathit{last})$ the response value of the call, and when the call of p returns, we append the corresponding response to h.

Let $(\mathit{lin}, <_{\mathit{lin}})$ be the linearization of h induced by the order in which the operations on variable L occurs in h. In more detail, if $\mathit{enter}(p, l)$ and $\mathit{enter}(q, l')$ are concurrent then we choose the order $\mathit{enter}(p, l) <_{\mathit{lin}} \mathit{enter}(q, l')$ if p does not read the value of $L[q]$ written by q (line 9); otherwise, we choose the converse order.

Fix $\ll_{\mathit{lin}}$ as the order between the entered laps in *lin* induced by the indexing function F. Assume a process p enters some lap $l = F(\mathit{last})$ in *lin*. We observe that p executes either line 13 or line 15. In the first case, this implies that p has just left $F(\mathit{last} - 1)$ which is the greatest lap smaller than l in which a process has entered during *lin*. In the second case, we observe that necessarily a process left l. and moreover that this event occurs in *lin* before p enters l. In both cases, the ordering property holds.

Below, we prove a key proposition characterizing algorithms that employ a racing on decidable objects. (The notion of decidable object is recalled in Section 4.1.)

[2] All our implementations are at least obstruction-free. Thus, when proving linearizability, we can simply consider a complete history.

Proposition 2. *Consider an algorithm $\mathcal{A}$ that accesses a racing on decidable objects. Suppose that in $\mathcal{A}$, when some process p enters a new object, the last object left by p is decided. Then, during every execution ρ of $\mathcal{A}$, at the time p enters a new object o, all the objects $o' \ll_\rho o$ are decided.*

Proof. By induction. The property holds clearly for the first object in the order $\ll_\rho$. Then, consider that some process p enters an object o and assume by induction that all the objects prior to o for the order $\ll_\rho$ are decided. By property (i) of the ordering property of a racing object, we can consider without lack of generality that p is the first process to enter o in ρ. Then, by property (ii), process p left previously the greatest object smaller than l for the order $\ll_\rho$; name this object o'. Applying the induction hypothesis, we know that for every object $o'' \ll_\rho o'$, o'' is decided at that time. Moreover, before p enters o, $\mathcal{A}$ ensures that o' is decided. Hence, the property holds for o.

Based on the above proposition, we prove next that Algorithm 3 and Algorithm 4 implement respectively a consensus and a universal abstraction.

Theorem 3. *Algorithm 3 implements an obstruction-free linearizable consensus.*

Proof. First of all, we observe that a process which starts executing solo necessarily commits the value it proposes into the next grafarius it enters alone. This is ensured by the solo convergence property. As a consequence, Algorithm 3 is obstruction-free. Besides, the validity of consensus follows from the validity property of a grafarius.

It remains to prove that the agreement property is also satisfied. To that end, consider a complete history h produced by an execution ρ of Algorithm 3 in which some process p returns a value v. We observe that a grafarius o returned $(commit, v)$ before p executes line 7. Suppose then that some process q adopts, or commits, a value u by accessing the smallest object o' higher than o for the order $\ll_\rho$. By the ordering property of a racing, when q enters o' either(i) process q just left o, and by the coherence property of a grafarius, it must propose v to o', or (ii) a process q' left o' before q enters it, and by a short induction on the continuation property of a grafarius, q must return the value v when it accesses o'. Hence, we have $u = v$.

Next, let us consider that two processes p and q return respectively u and v during h. Name o and o' the two grafarius accessed to decide these values. (If the call of either p or q returns at line 7, the situation boils down to this case.) If $o = o'$, the coherence property of a grafarius tells us that $u = v$. If now $o \neq o'$, we can assume without lack of generality that $o \ll_\rho o'$ holds and by induction we also obtain that $u = v$. Hence, Algorithm 3 maintains the agreement property of consensus during the history h.

Theorem 4. *Algorithm 4 implements an obstruction-free linearizable universal construction.*

Proof. Variable R is a (wait-free) racing on obstruction-free consensus objects. Thus, at the light of its pseudo-code, Algorithm 4 is obstruction-free.

We prove now that the implementation is linearizable. Consider a complete history h produced by some execution ρ of Algorithm 4, and let op be some operation in h executed by a process p. Operation op is *trivial* (respectively *regular*) when it returns at line 17 (resp. line 20). The value of variable C on process p at the time the operation returns is the consensus *associated* to op.

Note $\ll_\rho$ the order induced by the racing object R on the consensus objects associated to at least one operation. Let *lin* be the history produced by ordering the operations in h according to their associated consensus such that(i) trivial operations are ordered after the regular ones having the same associated consensus, and (ii) we keep the order between trivial operations having the same associated consensus. We prove below that *lin* is a linearization of h.

First, we show that $<_h \subseteq <_{lin}$. Let op and op' be two operations of h executed by respectively p and q. Consider that $op <_h op'$ holds, and name c and c' their respective associated consensus. (Case $c = c'$.) When the two operations have the same associated consensus, we observe that by the agreement property of consensus they cannot be both regular (line 18). Since the consensus c must be decided for op to execute, the case op trivial and op' regular is impossible Hence, we have op regular or trivial, and op' trivial. By the properties (i) and (ii) of our construction of the history *lin*, we conclude that $op <_{lin} op'$ holds. (Case $c \ll_\rho c'$.) By construction, we have $op <_{lin} op'$. (Case $c' \ll_\rho c$.) Operation op is associated to c. As a consequence, there exists a time t at which $c.d$ equals $\perp$ when the test at line 11 is executed in op. Similarly, we note t' the time at which $c'.d$ equals $\perp$ in op'. Since op precedes op' in h, it must be the case that $t < t'$. But $c' \ll_\rho c$ implies by Proposition 2 that at time t' consensus c is decided; a contradiction.

It remains to prove that *lin* is a legal sequence. To that end, consider an operation op by a process p in *lin*. Name v its response value, c its associated consensus, and op' the first regular operation that precedes op in *lin*. In case op is trivial, by property (i) of our construction, op' is the regular operation associated to the same consensus as op. Hence, this operation sets the object to a state s such that $\tau(s, op) = (s, v)$ (see lines 12, 15 and 18). Assume now that op is regular. When p enters c at line 13, by the ordering property of a racing object, we can consider two cases:(i) Process p has left the greatest lap smaller than l for the order $\ll_\rho$. Then, variable s contains the state of the object after op' and all the operations between op' and op are trivial. Thus, the response v is correct. (ii) The consensus c was left by some process before p enters it. In that case, c is decided at that time; a contradiction to the fact that op is regular.

B Proof of Proposition 1

In this section, we show that when invariant P1 holds in some complete history h, *recycle*(o) is a linearizable implementation of a recycled object of type $\mathfrak{T}$. To prove this statement, we construct from h a history *lin* decomposable in rounds, each round of *lin* being a correct history for some object of type $\mathfrak{T}$.

Our construction of *lin* relies on the following distinction between the operations appearing in h: operations that write to the decision register of o are called

modifiers, whereas operations that read a value in the decision register d of o and immediately return $f(d)$ are named *observers*.

Claim. For every observer op, there exists a unique modifier op' such that(i) op observes the decision written by op', (ii) op' is either concurrent or prior to op, and (iii) there is no modifier op'' such that $op' <_h op'' <_h op$. We shall say in the following that op' is the modifier *associated* to op.

Proof. Consider an operation op that executes on $recycle(o, t)$. If op is an observer, then according to Construction 2, the decision register d contains a value different than $\perp$. From Construction 1, we deduce that op reads a value $(t' \geq t, \_)$ from d. Since the register d is atomic, a unique operation op' writes this value to d. This operation is necessarily concurrent, or prior to op, and moreover there must be no modifier op'' such that $op' <_h op'' <_h op$.

Claim. For every modifier op on $recycle(o, t)$ in h, if op' is a modifier on $recycle(o, t')$ and $t' < t$ holds then op' precedes op in h.

Proof. We proceed by induction. Consider that the claim holds for all the modifiers prior to some operation op on $recycle(o, t)$. Then, assume for the sake of contradiction that an operation concurrent, or following, op writes to the decision register d of o with a timestamp $t' < t$. Name op' the first such operation. By P1, there exists an operation op'' on $recycle(o, t')$ that precedes op' in h and that writes to d. From our induction hypothesis and the choice of op', every modifier between the response of op'' and the invocation of op', writes to d with a timestamp $t'' \geq t'$. Hence from the code of Construction 2, op' observes that o is decided when it is executed; a contradiction to the fact that this operation is a modifier.

We detail how to build history *lin* in Construction 3. Further, we show that *lin* is a higher-level view of h. Then, we prove the existence of a decomposition of *lin* for which each round is correct for the type $\mathfrak{T}$.

(**Construction 3**) Consider the invocation and response events e that occur in the history h. In case $e = inv_p(recycle(o, t).op)$, we add $inv_p(recycle(o).op)$ to *lin*; otherwise $e = res_p(recycle(o, t).op)\, v$ for some response value v, and we add $res_p(recycle(o).op)\, v$ to *lin*. The order in which we add those events in *lin* is(i) for modifiers, the order in which they write last to the decision register d of o, and (ii) for observers, the order in which they read d.

By the fact that registers are atomic, Construction 3 implies that *lin* is sequential. Moreover, since every operation reads or writes to d and we order them according to their accesses to d, the following claim is immediate.

Claim. For any two operations op and op' in h, if op precedes op' in h then op precedes op' in *lin*

In what follows, we prove the correctness of *lin*.

Claim. There exits a decomposition of *lin* such that in every round r of this decomposition, r is a correct history for an object of type $\mathfrak{T}$.

Proof. For every timestamp $t \in \mathcal{T}$ that appears in h, we define the round r_t as the sub-sequence of modifiers in *lin* that occur on $recycle(o, t)$ in h, together with the observers (if any) for which one of these operations is a modifier. We order rounds according to their associated timestamps. Denoting $\{t_1, \ldots, t_m\}$ the ordered set of timestamps that appears in h, we prove below that $lin = r_{t_1} \ldots . r_{t_n}$ holds.

Consider an operation op accessing $recycle(o, t)$, and that this operation is complete in h. According to Construction 3, either op belongs to r_t, or by Claim B, there exists a modifier associated to op in some round $r_{t'}$, and op belongs to that round. We observe that in both cases, op is complete in the round it belongs to.

Then, let us consider two operations op and op' that belong to rounds r_t and $r_{t'>t}$. In each of the three cases that follow, we prove that op and op' do not interleave. (Case op and op' modifiers.) Since $t < t'$, by Claim B, op precedes op' in h. By Construction 3, this also holds in *lin*. (Case op modifier and op' observer.) Note op'' the modifier in round $r_{t'}$ associated to op'. This modifier is executed on $recycle(o, t')$, whereas op is executed on $recycle(o, t < t')$. Applying Claim B, we know that op'' precedes op in h, and thus also in *lin*. By Construction 3, we deduce that $op <_{lin} op'' <_{lin} op'$ holds. (Case op observer and op' modifier.) This case is symmetric to the previous one and thus omitted. (Case op and op' observers.) Let op_1 and op_2 be the modifiers respectively associated to op and op' in rounds r_t and $r_{t'}$. Since $t < t'$, by Claim B, op_1 precedes op_2 in h. By Construction 3, we deduce that $op_1 <_{lin} op <_{lin} op_2 <_{lin} op'$ holds.

It remains to show that every round in the above decomposition is a correct history for an object of type $\mathfrak{T}$. To achieve this, let us now consider a round r_t and some operation op in r_t. If op is an observer, then it should return the decision set by its associated modifier in r_t, and this modifier precedes op. Otherwise, by construction all the operations in r_t are executed on $recycle(o, t)$. and by definition of *lin*, all the modifiers preceding the round r_t were executed on $recycle(o, t')$, for some timestamp $t' < t$. Thus, from Construction 1, all the modifiers in r_t are oblivious of the modifications that occur in previous rounds. As a consequence, we conclude that r_t is a correct history for an object of type $\mathfrak{T}$.

C Correctness of Algorithm 5

As pointed out in Section 4.3, the correctness of Algorithm 5 relies on Proposition 1. Below, we state that the recycled consensus objects used in Algorithm 5 satisfy this proposition, then we reduce Algorithm 5 to Algorithm 4.

Proposition 3. *During every execution ρ of Algorithm 5, for every natural k, $recycle(F(k))$ implements a recycled consensus object.*

Proof. Choose an execution ρ of Algorithm 5 and suppose that P1 holds in ρ for all objects $recycle(F(k))$, up to some operation op on $recycle(o, t)$. The case $t = 0$ is obvious, hence from now we assume $t > 0$. Consider an operation op' on

recycle(o, t') such that op' does not precede op in ρ and $t' < t$ holds. Let p be the process that executes op in ρ.

Operation op on *recycle*(o, t) occurs either at line 13 or at line 21. Since $t > 0$, process p executed a call to function *enter*() at line 32 before, and this call returned *recycle*$(F(l), t)$, with $o = F(l)$. From the code at line 15, naturals l and t are both contained in the decision register of some object *recycle*$(\_, t-1)$. Then because P1 applies up to op, some process p'' invoked $op'' =$ *propose*$((\_, \_, l, t))$ on *recycle*$(\_, t-1)$ previously. At this point, we may consider the two following cases. (Case $t' = t-1$) The index l computed by p'' is the result of a call to function *free*() at line 21. But from the code at lines 25 to 29, at that time $L[p'']$ equals l. A contradiction. (Case $t' < t-1$) We apply our induction hypothesis to op'' and op'.

Proposition 4. *During every execution ρ of Algorithm 5, $h|$enter() is a correct history for a racing on consensus objects.*

Proof. The domain of *enter*() consists of all the objects *recycle*(o, t) with $o = F(l)$ and $t \in \mathcal{T}$. For some execution ρ of Algorithm 5, we define $\ll_\rho$ as the order induced by $<$ on $\mathcal{T}$, that is *recycle*$(o, t) \ll_\rho$ *recycle*(o', t') holds iff $t < t'$.

By Proposition 3, *recycle*(o) is a recycled consensus object. Thus, we may consider a decomposition of $\rho|$*recycle*(o) in rounds $\{r_1, \ldots, r_{m \geq 1}\}$, where each round r_i is a correct history for consensus. Choose some object *recycle*(o, t). When some process p executes an operation op on *recycle*(o, t), we may consider the round r_i to which operation op belongs. Because r_i is a correct history for a consensus object, this is also the case for the history $\rho|$*recycle*(o, t). This proves that the domain of *enter*() is a set of consensus objects.

Then, we observe that when p accesses $c_i =$ *recycle*$(F(l), t)$ with some operation op, p executed a call to *enter*() previously. From the code at line 15 and the code of function *enter*(), the pair (l, t) is the result of a decision stored in *recycle*$(o', t-1)$. Hence, some process entered *recycle*$(o', t-1)$ previously and this object is the greatest object smaller than *recycle*(o, t) for $\ll_\rho$.

We can now state the main result of this section:

Theorem 5. *Algorithm 5 implements an obstruction-free linearizable universal construction.*

Proof. From Proposition 4 and the code of Algorithm 5, Algorithm 5 implements Algorithm 4. Applying Theorem 4, we deduce that Algorithm 5 implements an obstruction-free linearizable universal construction.