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

```
Shared Variables:
    $K$                                        // Listenable KVS

Local Variables:
    s $\in$ $States$                               // initially $\perp$
    r $\in$ $Values$                               // initially $\perp$
    $Q$                              // a FIFO queue; initialy $\perp$

$open(k)$
    $K.regListener(callBack)$
    $(x, \_, f) \leftarrow get(k)$
    if $(x, \_, f) = \perp$ then
        $s \leftarrow s_0$
    else if $f = \texttt{PER}$ then
        $s \leftarrow x$
    else
        $K.put(k, (\perp, p, \texttt{RET}))$
        wait until $s \neq \perp$

$close(k)$
    $K.put(k, (s, p, \texttt{PER}))$
    $K.unregListener(callBack)$
    $s \leftarrow \perp, r \leftarrow \perp, Q \leftarrow \perp$

$invoke(k, op)$
    $r \leftarrow \perp$
    $K.put(k, (op, p, \texttt{INV}))$
    wait until $r \neq \perp$
    return $r$

When $callBack(k, (x, p', f))$
    if $f = \texttt{INV}$ then
        if $s \neq \perp$ then
            $(s, v) \leftarrow \tau(s, x)$
            if $p = p'$ then
                $r \leftarrow v$
        else if $Q \neq \perp$ then
            $Q \leftarrow Q \circ \langle x \rangle$
    else if $f = \texttt{RET}$ then
        if $s \neq \perp$ then
            $K.put(k, (s, p, \texttt{PER}))$
        else if $p = p'$ then
            $Q \leftarrow \langle\rangle$
    else if $f = \texttt{PER} \wedge s = \perp$ then
        $s \leftarrow x$
        for $x \in Q$ do              // In the order defined by $Q$.
            $(s, v) \leftarrow \tau(s, x)$
```

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

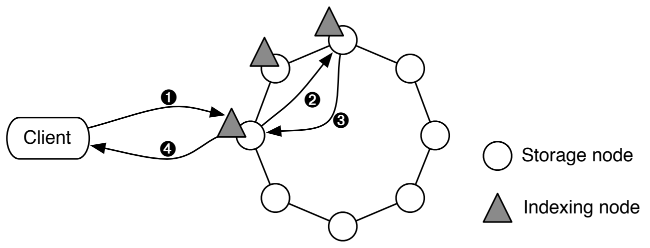


Figure 1. General architecture of the versioned KVS.

returns the response to the calling client (Figure 1-4). 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

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:      $T \leftarrow K.open(k)$                    // 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:     $T \leftarrow K.open(k)$                    // retrieve the tree
13:     $(\_, l) \leftarrow T.last()$               // compute the latest entry
14:     $K.close(k)$                                // close the tree
15:     return $K.get(l)$                           // return the corresponding value
16:
```

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.

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.

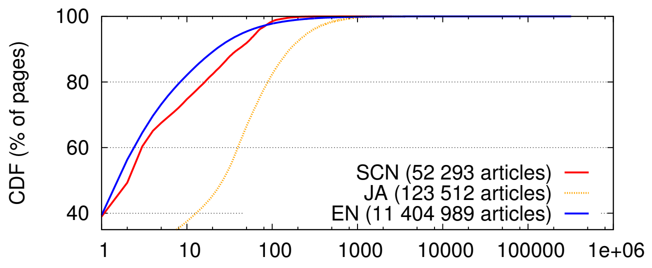
Sharding the Trees: As we shall see in practice in Section VI, the tree-based versioning mechanism works fine in most cases, but fails for data having a large number of versions. We describe a versioning mechanism that overcomes this limitation by scattering the different versions into multiple trees. 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

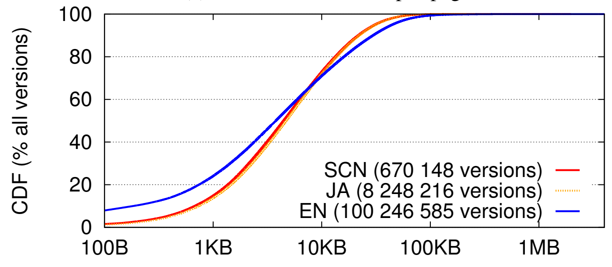
Our evaluation consists in re-executing real access traces on a Wikipedia dump stored by ALEPH. We ran our experiments on a cluster of 24 virtualized 4-core Xeon 2.5 Ghz machines with 4GB of memory, running Gentoo Linux 32bits, and connected by a virtualized 1 Gbps switched network. Network performance, 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*

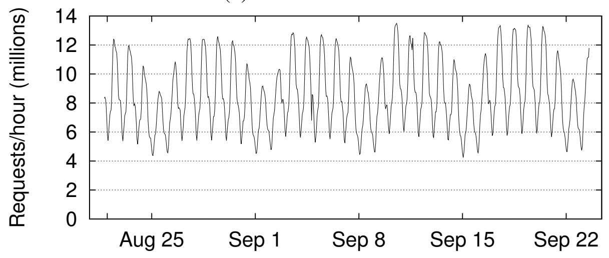
We use the dump of Wikipedia as of January 3rd, 2008, published by the Wikibench benchmark [30]. Among other information, it contains the page identifier and the list of versions. We use this log to recreate all the versions of the Wikipedia pages in ALEPH.



(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.

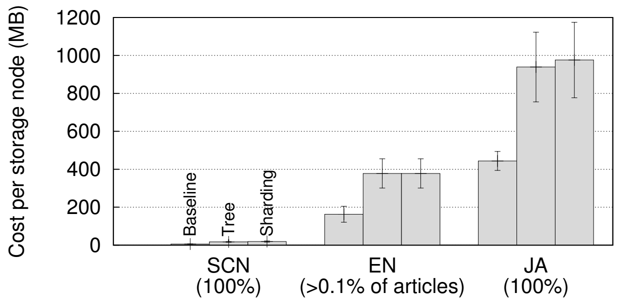


Figure 3. Storage cost for each Wikipedia.

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.

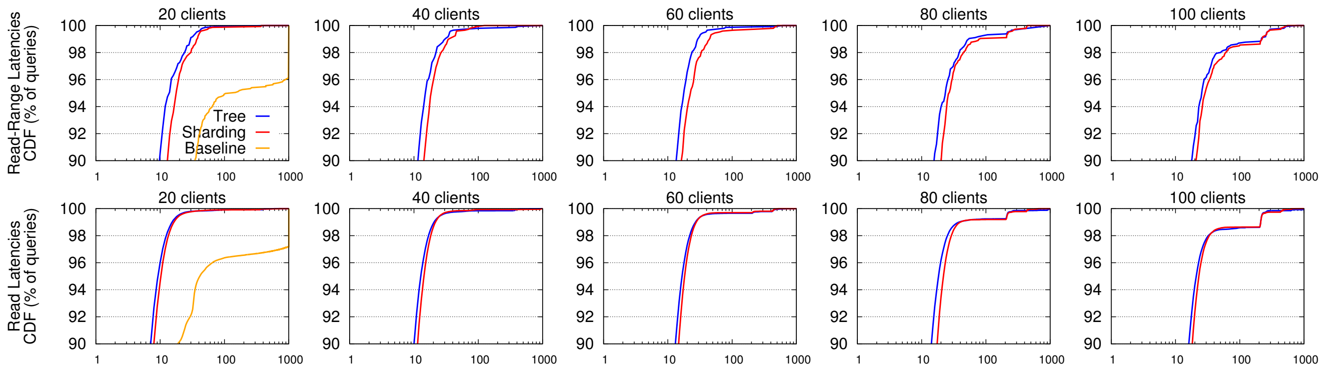


Figure 4. Read (bottom) and Read-Range (top) latencies for increasing number of clients.

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.