CRESON: Callable and Replicated Shared Objects over NoSQL

Pierre Sutra*, Etienne Rivière†, Cristian Cotes‡, Marc Sánchez Artigas‡, Pedro Garcia Lopez‡,
Emmanuel Bernard§, William Burns§ and Galder Zamarreño§
*Télécom SudParis, CNRS, Université Paris-Saclay, France. e-mail: pierre.sutra@telecom-sudparis.eu
†University of Neuchâtel, Switzerland. e-mail: etienne.riviere@unine.ch
‡Universitat Rovira i Virgili, Tarragona, Spain §Red Hat

***Abstract*—In a Cloud environment, the ability to share and persist objects simplifies the design of applications. Storing objects in a NoSQL database ensures their availability and provides scalability to applications. When Object-NoSQL Mapping is performed at the client side, objects that are accessed by several clients are repeatedly converted between their in-memory and serialized representations. This negatively impacts performance and increases replication costs. In this paper, we describe the design of CRESON, a system supporting *callable* objects over NoSQL, in which application objects are mapped and instantiated directly on the storage nodes. CRESON supports composition by reference and ensures strong consistency. Objects are replicated and maintained coherent using State Machine Replication. The implementation of CRESON leverages the support of a listenable key-value store (LKVS), a novel NoSQL storage abstraction that we introduce in this paper. We discuss the performance and complexity of CRESON with the example of the portage of a personal cloud storage service, initially developed using an object-relational mapping over a sharded PostgreSQL database. Our results show that CRESON offers a simpler programming experience both in terms of learning time and lines of code, while performing better on average and being more scalable.**

I. Introduction

Applications built for cloud environments must outsource the management of their state to a dedicated storage layer. This allows scaling the number of application instances with the number of client requests. The use of a common storage layer is also instrumental for sharing mutable state between application instances. To support application availability, faults must be masked by replicating application data on several storage nodes. As shared data is subject to concurrent accesses, the storage layer must enforce that replicas remain consistent. NoSQL databases answer these needs. They provide a simpler interface than relational databases, typically in the form of a key/value store with limited querying capabilities, but offer better scalability and support elasticity, i.e. they allow adding and removing storage nodes dynamically. NoSQL databases also enforce a large variety of consistency guarantees for concurrent accesses, ranging from eventual consistency [1], [2] to stronger guarantees such as linearizability [3]–[5].

Most applications manipulate their state in the form of *objects* associating fields and operations, or methods, under the popular object-oriented paradigm. In order to benefit from dependability and elasticity, it is natural to consider the use of a NoSQL database to store and share these objects.

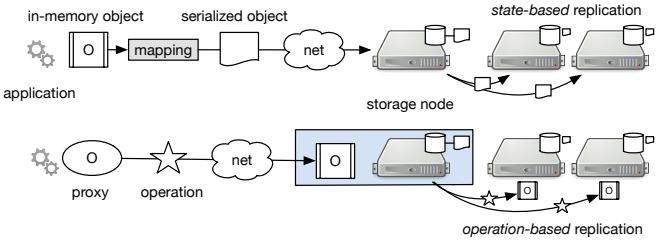


Fig. 1. Client-side Objet-NoSQL Mapper (top) and CRESON (bottom)

Storing and accessing an object in a database requires a mapping between the in-memory state and the serialized representations in the database. This mapping can be performed directly by the programmer, but most often a framework automates the process and offers language integration. This allows simply marking the types of objects that must be stored remotely and hides the actual mapping phase. Examples of Object-NoSQL Mappers (ONM) [6] are Hibernate OGM, Morphia, and Objectify. For relational databases, Object-Relational Mappers (ORM) such as Hibernate [7] provide the same functionality. In these frameworks, the mapping is performed solely at the client application side, where in-memory objects also reside. Such an approach is illustrated by the upper part of Figure 1: The database manipulates only the serialized representations of objects, and it does not have access to their methods. Any access to a remotely-stored object requires to retrieve its serialized form and instantiate the object locally. When disposing of this object, its state is serialized again and pushed back to the database. The conflict between the two types of representation is called the *impedance mismatch*.

The impedance mismatch impacts the performance and the ability to efficiently share objects [8]. The cost of repeated conversion and transmission can be high, in particular for large objects. Further, state-based replication happens at the level of the entire object, requiring to ship the full serialized form between replicas upon each modification. Most operations access multiple and/or composed objects. Navigating a complex data structure requires multiple interactions with the database, in order to retrieve each object in sequence. Caching objects at the application side and pre-fetching linked objects [9] can lower the impact of the impedance mismatch, but hinders offering strong consistency. As an example, Objectify, the

ONM for the Google App Engine store, supports caching and transactions, but it is not thread-safe, i.e. it does not support the sharing of objects by multiple application instances [10].

Contributions. We are interested in the support of *callable* objects over NoSQL, where calling an object does not happen at the application side after retrieving its serialized representation, but instead on the storage nodes, as illustrated in the lower part of Figure 1. Our approach, named CRESON, targets applications that need to share composed objects with strong consistency guarantees, while requiring the dependability, scalability and elasticity properties of NoSQL databases.

CRESON limits the impact of the impedance mismatch thanks to two important design decisions, which distinguish it from client-side ONM. First, objects are mapped only between their serialized representation and their in-memory representation once at each storage node hosting an object replica, and this is also where the method calls are performed. The mapping does no longer repeatedly happen at each application instance upon opening the object or following a concurrent modification. Second, in order to avoid repeatedly shipping large serialized representations between replicas, we adopt an operation-based replication in the form of *State Machine Replication* (SMR). Only operations on an object are replicated and applied independently in the same order at its replicas. CRESON offers the strong consistency guarantee of linearizability for concurrent operations. It allows the programmer to *compose* objects in arbitrary acyclic graphs, and maintains linearizability across composed objects. This allows easily building consistent applications through shared *facade* objects embedding complex and multi-object operations without the need for transactions.

CRESON inherits the performance, scalability and elasticity guarantees of the NoSQL storage it builds upon. In particular, it enables disjoint-access parallelism: Operations that access distinct objects execute in parallel. We introduce and motivate a novel NoSQL abstraction, the *Listenable Key-Value Store* (LKVS), that allows the efficient and wait-free implementation of shared objects. Our implementation adds the LKVS abstraction to the Infinispan [5] NoSQL data grid, and layers the components of CRESON upon it. CRESON provides a simple interface thanks to its integration with the Java language using annotations, intended to be easy to learn for developers using existing ONM and the Java Persistence API.

We evaluate the complexity and performance of using CRESON for building a highly-available application. We modify StackSync, a personal cloud storage application similar to Dropbox, to use CRESON for storing and sharing objects representing users' filesystem metadata. This application previously employed an object-relational mapping with the PostgreSQL database, and PL/Proxy for sharding and stored procedures. Such a configuration led to scaling challenges and to the lack of support for elasticity, as re-sharding at the database level requires to halt the application. We evaluate the performance of StackSync using a large-scale trace from the *Ubuntu One* cloud storage service. Our results show that CRESON consistently outperforms the ORM-based solution, being able to sustain a higher throughput and to offer shorter response times.

Outline. The remainder of this paper is organized as follows. We cover the related work in Section II. We present the design of CRESON in Section III, and detail our case study and the system interface in Section IV. We discuss selected implementation details in Section V, and present our evaluation in Section VI, before concluding in Section VII.

II. Related work

We review the related work in light of the objectives of CRESON to allow consistent sharing of persistent application objects. We summarize the key aspects of the main systems we review in Table I. **Evaluation** indicates whether a method call is evaluated on a local copy of the shared object, or at a remote server. In the latter case, **Composition** denotes the ability to transparently call a shared object within a shared object. **Replication** describes the support for state-based or operation-based replication. We finally detail the **Consistency** in presence of concurrent accesses and the support for **Elasticity**. Our work bears similarities with distributed objects, object-oriented databases, object-relational databases, transactional stores and object-database mappings. We covered the latter in the introduction.

Distributed objects. The *distributed object* paradigm takes its roots in remote procedure calls [22] and the ability to load custom procedures in a data store [23]. It allows composing applications through remote calls to objects exporting methods, defined as *guardians* in Argus [11]. Other notable implementations of this paradigm include Orca [13], Arjuna [12] and GARF [24]. Orca and Arjuna offer the ability to access and modify objects within transactions. GARF support replication with the help of a group communication primitive. Argus

System	Evaluation	Composition	Replication	Consistency	Elasticity
Argus [11]	R	×	–	T	×
Arjuna [12] and Orca [13]	R	×	S/O	T	×
JAVA RMI	R	√	–	S	×
CORBA obj. group [14]–[17]	R	√	O	S	×
DB4O (Obj. oriented DB)	L	–	–	W	×
Versant (Obj. oriented DB)	L	–	S	W	×
Thor [9], [18]	L	–	S	T	×
ObjectStore [19]	L	–	–	T	×
Postgres [20] w. PL/Proxy	R	√[1]	S	T	×
Hibernate OGM (Obj.-NoSQL)	L	–	S	T	√
Objectify (Object-NoSQL)	L	–	S	T[1]	√
F1 [21] (NewSQL)	L	×	S	T	√
CRESON	R	√	O	S	√

Criteria:
Evaluation of methods call: Local (L) or Remote (R);
Composition (none, embedding=× or by reference=√);
Replication (none, state-based=S or operation-based=O);
Consistency (weak=W, strong=S or transactional=T);
Elasticity support.
[1]: with limitations.

TABLE I
SOLUTIONS FOR REMOTE STORAGE AND EVALUATION OF OBJECTS.

leaves object replication and composition in the hands of the programmer. Globe [25] defines the notion of a *distributed shared object*, which is a group of replicas of a distributed object for which the programmer can plug in mechanisms for replication, migration, communication and concurrency control.

Java Remote Method Invocation (RMI) and CORBA [26] are two prominent implementations of the distributed-object paradigm. Java RMI supports composition by reference but does not specify how the servers hosting objects should offer replication or elasticity. In the context of CORBA, systems such as Electra [14], Eternal [15] and OGS [27] provide fault-tolerant distributed objects using the *object group* design pattern [28]. Under this pattern, the Object Request Broker (ORB) delegates each method call to a dependable communication primitive that sends it to the object replicas. The ORB supports object composition, being able to deal with issues related to call idempotence at the caller and at the replicas (these are named response collation and request filtering respectively in [27]). We explain how CRESON addresses these two problems in Section V. In general, the object group design pattern suffers from a proliferation of groups [17], [29] and does not offer a persistent, nor elastic data store *per se*.

Object-oriented databases. Object-oriented databases [30], [31] do not require an explicit mapping phase between in-memory objects and their intermediate representation at the storage layer. Similarly to CRESON, object-oriented databases are integrated with several object-oriented programming languages, and they allow to create and access persistent objects. O++ [32] extends C++ and additionally features querying capabilities. ZODB targets the Python language, and offers support for transactions and distributed evaluation. DB4O is a discontinued object-oriented database. Unlike CRESON, it does not support remote evaluation, but requires fetching the object locally, and re-uploading it after mutation. There is no guarantee for consistency when two processes fetch and update the same object. Versant is a closed-source follow-up to DB4O that supports replication. After each mutation, the system requires shipping the full state of the object to the replicas, instead of coordinating operations as with CRESON's state-machine replication approach. Thor [9], [18], Mneme [33] and ObjectStore [19] are other examples of object-oriented databases that use local evaluation and offer transactional support. These systems manipulate object representations directly as memory pages. They implement the retrieval and local copy of missing pages upon a reference to a non-local object, and benefit from locality of accesses.

Object-oriented databases are generally considered to suffer from poor scalability due to the lack of disjoint-access parallelism, which CRESON addresses by its use of a NoSQL store. Recent examples of object-oriented databases can nonetheless be found in the context of embedded systems, such as Perst and Realm. These databases support multi-threaded accesses and ACID semantics for transactions on multiple objects, but target a different deployment context than CRESON, with no necessity for replication.

Object-relational databases. Object-relational databases are similar to relational ones with object-oriented specificities. One prominent system in this category is Postgres [20]. Postgres supports user-specified types, stored procedures (e.g. with PL/Proxy) and type inheritance. Many of the ideas coming from object-relational databases have been incorporated into the SQL:1999 standard via structured types, and are now available in commercial databases. The SQL:1999 standard also introduces language constructs to overcome the inherent limitations of relational algebra, such as the absence of a transitive closure operator [34]. Our use case in Section IV describes and evaluate Postgres and PL/Proxy in detail.

Transactional stores. Mutiple distributed transactional stores were proposed recently, e.g., [35]–[41]. These stores do not necessarily offer a relational or semi-relational model. In particular, several works [37], [40], [42] propose to port the transactional memory paradigm [43] to distributed systems. The use of distributed transactions in an object-oriented application requires the use of an explicit mapping, similarly to relational databases. In most approaches, the store employs a deferred update schema: the transaction is evaluated locally and tentatively using remote data, then, if it commits, the changes are pushed back to the storage tier. This requires the use of local evaluation, followed by the push back of serialized representation to the store, as with ORM and ONM solutions. Sinfonia [35] proposes a programming paradigm for distributed applications based on a multi-word read-modify-write primitive. In Sinfonia, the mapping of data to storage nodes is explicit and left to the application. There is no automatic sharding, and elasticity must also be handled by the application. Spanner [39] uses accurate clocks to execute transactions in a geo-distributed manner. The system shards automatically data into multiple fragments and uses a consensus protocol to maintain the consistency of each fragment. The NewSQL F1 [21] database uses Spanner. Applications employ a simplified ORM that requires to make both join operations and the traversal of relationships between records explicit. The system also includes a distributed query engine with streaming capabilities.

Dedicated data structures. Over the past few years, multiple dedicated NoSQL solutions focusing on a particular data type have appeared, for example, trees [44], locks [45], tuple spaces [46], [47] and sets [48]. Applications generally use them to manipulate metadata and for coordination purposes. The authors of [49] show that these dedicated solutions have a limited versatility. The Extensible Coordination approach [50] addresses this issue, allowing clients to embed user-defined operations directly in the storage layer. These operations manipulate the ZooKeeper tree [44]. With CRESON, the developer is not bound to a particular interface. An application may use the shared objects to implement coordination and synchronization primitives.

Summary. CRESON represents a new alternative for storing and accessing objects in a NoSQL store. It targets shared

objects composed by reference and strong consistency. As shown in Table I, the design choices of CRESON positions it in particular as a sound alternative to object-NoSQL mappers.

III. Design

This section details the design of CRESON. In III-A, we give an overview of the system. We detail CRESON guarantees in III-B. We follow by presenting the NoSQL storage on which CRESON is built in III-C. Object creation and disposal as well as methods calls appear in III-D, and object composition in III-E. We provide a discussion on our design choices and our consistency guarantees in III-F. Language integration is presented together with the use-case application in the next section.

A. Overview

Figure 2 presents the general architecture of CRESON.

The server side of CRESON is a NoSQL data store on which we build the components dealing with object management. This NoSQL data store implements a novel interface that we define in this paper, the *Listenable Key-Value Store* (LKVS). A LKVS allows to process calls at the server side and notify the client application in return. In more detail, in addition to the usual $put()$ and $get()$ calls, a LKVS allows registering an external *listener* on a particular key, together with an associated *handler*. When a $put()$ call occurs for that key, the handler processes the value, and may send a callback to the listener.

At the *client* side, the application accesses a shared object through a *proxy* [51]. CRESON creates this proxy when the application instantiates the shared object locally. The first call to the proxy registers it as a listener of the LKVS associated with a *session* handler. For instance, in Figure 2, the light gray proxy is linked to the light gray session handler.

At a storage node, session handlers referring to the same object share an *object* handler (see object O1 in Figure 2). These object handlers represent collectively a set of elastic, durable and consistent shared objects. A session handler relays calls to the object handler and sends back response values to the associated proxy.

Operation-based replication. When using replication, $put()$ calls are ordered and multicast to all storage nodes holding a copy of the handlers, and henceforth an in-memory copy of the object state. These $put()$ calls carry operations (e.g., a method call). The state of the object is not propagated between replicas. The object handlers actually implement a variation of *state machine replication* (SMR) [52], that is able to deal with the lifetime of objects and with their composition. The use of operation-based replication requires that the state of the object at each of the replicas remains identical when subject to the same sequence of method calls. Shared objects must therefore be of a *deterministic* type: the behavior of an object in a given state must be unique for a given set of arguments. This requirement is not specific to CRESON, but is common to all operation-based replication systems (e.g., [53], [54]).

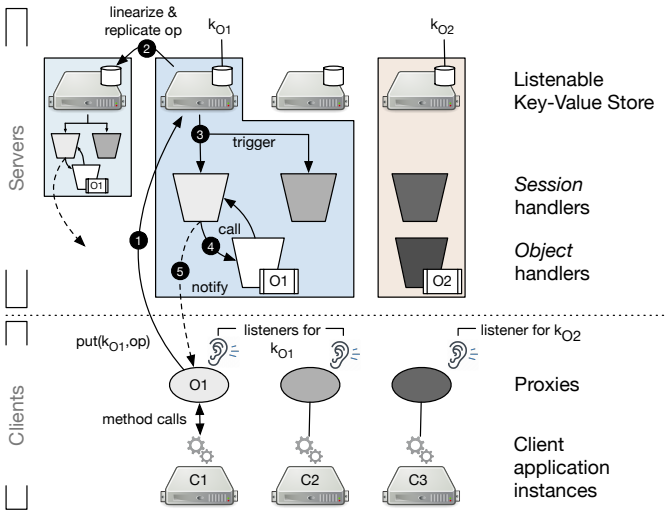


Fig. 2. CRESON architecture

B. Guarantees

CRESON offers the following key functional guarantees to the programmer of a distributed application.

(Strong Consistency) Objects shared with CRESON are linearizable [55]. This means that if two application processes concurrently execute method calls, these calls will appear as executed in some sequential order. Furthermore, this sequential order respects the real-time ordering of the calls.

(Progress) CRESON supports wait-free shared objects [56]. This means that calls to a shared object by application processes do not fail-stop but always return.

(Composition) CRESON allows the programmer to compose objects in arbitrary acyclic graphs. The composition of objects in CRESON maintains linearizability.

(Persistence) Every object created and hosted by CRESON is persistent. If the application stops then later restarts, the shared objects it was manipulating are intact. This property allows to easily program stateless application components, improving scalability and dependability.

We also provide two important non-functional guarantees:

(Disjoint-access Parallelism) Computations that access distinct objects operate on disjoint components at the storage nodes. As a result, CRESON is scalable for concurrent accesses to distinct objects, and only features scalability restrictions for concurrent accesses to the same object, as is inevitable when providing linearizability [44], [55], [57].

(Elasticity) CRESON can provision or remove storage nodes without interruption.

C. Listenable Key-Value Store

We now detail the listenable key-value store (LKVS) that is the foundation of the server side of CRESON. To build the LKVS, we consider a slight variation of the virtual synchrony paradigm [58], as detailed next.

Interface. A LKVS exposes a common map interface to $put()$ and $get()$ values. In addition, it allows a clients to watch for modifications by registering a *listener* together with a *handler*. The listener is typically a distant process, but the handler is a pluggable piece of code executed at the LKVS side. A listener can receive a notification after a newly written value has been processed by its associated handler. In detail, a LKVS presents the following interface:

- $put(k, u)$ stores value u under key k;
- $get(k)$ returns the value stored under key k, or $\perp$ if no such value exists;
- $regListener(k, l, h_l)$ registers the listener l and its associated handler h_l for key k;
- $unregListener(k, l)$ unregisters the listener l, removing both l and h_l from the LKVS.

Registering a listener l together with its handler h_l for some key k has the following semantics: Consider that a call to $put(k, u)$ occurs, and note v the old value stored under key k. A handler defines a function $handle(u, v)$. If calling $handle(u, v)$ on h_l returns $w \neq \perp$, then listener l eventually receives at least once the event $\langle k, w \rangle$. In such a case, we shall say that the event $\langle k, w \rangle$ *passes* the handler h_l.

Requirements. To build an LKVS in a message-passing distributed system where process can fail-stop, we use the virtual synchrony paradigm [58]. A *view* refers to a set of processes. Virtual synchrony allows a process to send ($send$) and receive ($receive$) messages, and to get notified of a view change ($viewChange$). When the event $viewChange(V)$ occurs at process p, we say that p installs view V; initially all processes install some view V_0. During a run, the installation of the successive views guarantees that (Primary Component) they follow some total order; (Virtual Synchrony) processes agree on the messages they receive in a view; and (Total Order) processes also agree on the order in which they receive these messages. In addition, we shall consider that the (Precise Membership) property holds, that is, *(i)* every faulty process gets eventually excluded from the view, and *(ii)* every correct process is eventually present in the view.

In the original virtual synchrony paradigm, messages are addressed to all processes in the view. Differently, we consider that a message can be addressed to any subset of the view. Such a variation is straightforwardly implementable using the original paradigm, by discarding messages appropriately. More interestingly, with genuine atomic multicast [59] and an eventually perfect failure detector [60], we construct a disjoint-access parallel implementation as follows: *(i)* to send a messsage, we multicast it, and *(ii)* whenever a process leaves/joins the system or is suspected, a corresponding message is multicast to all the processes, triggering an appropriate view change.

Based on virtual synchrony, we define a replication mapping as follows. Let us consider some replication factor ρ. For some view V and some key k, we assume a locally computable function $rep(k)_V$ that returns the ρ processes in charge of replicating the item under key k in view V. As exemplified by many NoSQL solutions (e.g., [5], [61]), we may use consistent hashing [62] to implement a replication mapping.

Algorithm 1 LKVS – code at process p

```
1:  Variables:
2:      D                                   // Initially, ∀k : D(k) = ⊥
3:      H                                   // Initially, ∀k : H(k) = ∅
4:      V                                   // Initially, V_0
5:
6:  Task Front-end
7:      get(k) :=
8:          eff: send ⟨GET, k⟩ to rep(k)_V
9:               wait until received ⟨ACK, u⟩
10:              return u
11:     put(k, u) :=
12:         eff: send ⟨PUT, k, u⟩ to rep(k)_V
13:              wait until received ⟨ACK⟩
14:     regListener(k, l, h_l) :=
15:         eff: send ⟨REG, k, l, h_l⟩ to rep(k)_V
16:              wait until received ⟨ACK⟩
17:     unregListener(k, l) :=
18:         eff: send ⟨UREG, k, l⟩ to rep(k)_V
19:              wait until received ⟨ACK⟩
20: Task Back-end
21:     doGet(k) :=
22:         pre: received ⟨GET, k⟩ from q
23:         eff: send ⟨ACK, D(k)⟩ to q
24:     doPut(k, u) :=
25:         pre: received ⟨PUT, k, u⟩ from q
26:         eff: v ← D(k)
27:              D(k) ← u
28:              forall h_l ∈ H(k) do
29:                res ← h_l.handle(u, v)
30:                if res ≠ ⊥
31:                  send ⟨k, res⟩ to l
32:              send ⟨ACK⟩ to q
33:     doRegListener(k, l, h_l) :=
34:         pre: received ⟨REG, k, l, h_l⟩ from q
35:         eff: H(k) ← H(k) ∪ {h_l}
36:              send ⟨ACK⟩ to q
37:     doUnregListener(k, l) :=
38:         pre: received ⟨UREG, k, l⟩ from q
39:         eff: H(k) ← H(k) \ {h_l}
40:              send ⟨ACK⟩ to q
41:     doViewChange :=
42:         pre: viewChange(V)
43:         eff: I ← {k : p ∈ rep(k)_V \ rep(k)_V}
44:              O ← {(k, q) : p ∈ rep(k)_V,
                          q ∈ rep(k)_V \ rep(k)_V}
45:              forall (k, q) ∈ O do send ⟨k, D(k), H(k)⟩ to q
46:              wait until ∀k ∈ I : received ⟨k, v_k, H_k⟩
47:              forall k ∈ I do (D(k), H(k)) ← (v_k, H_k)
48:              V ← V
```

Construction. We provide an elastic disjoint-access parallel implementation of the LKVS abstraction in Algorithm 1. This algorithm executes at each process a front-end and a back-end task. On the front-end, calls to the LKVS interface are both linearizable and wait-free. Wait-freedom ensures that any call issued by the front-end eventually returns. Linearizability satisfies that *(i)* every entry of the LKVS map behaves like an atomic register, and *(ii)* once a listener l is registered, l receives all the events $\langle k, handle(u, v) \rangle$ that pass the handler h_l in the order in which the corresponding calls $put(k, u)$ are linearized.

Algorithm 1 is presented as a sequence of effects (**eff**), each guarded by one or more preconditions (**pre**). When all the conditions in a **pre** clause are true, the instructions in the corresponding **eff** block are triggered. Conditions are mutually exclusive, thus their order of presentation does not matter.

Our construction employs the following three variables. $\mathcal{V}$ stores the current view; For some key k, $\mathcal{D}(k)$ indicates the value of the local copy of k; and Variable $\mathcal{H}(k)$ holds the set of handlers registered at key k. Initially, $\mathcal{D}(k)$ equals $\perp$ and $\mathcal{H}(k)$ equals $\varnothing$ for every key k. Internally, Algorithm 1 works as follows.

- When a front-end task executes an operation accessing a key k, it sends an appropriate message to $rep(k)_{\mathcal{V}}$ (e.g., at line 12). Upon receiving such a message, if the front-end asks for the current value of k (line 21), or wishes to register or unregister a listener (respectively at lines 33 and 37), the replica executes the corresponding action. On the other hand, if the front-end updates the shared map with a pair (k, u), a replica of k first updates $\mathcal{D}(k)$ appropriately (line 27), then notifies the handlers registered at key k (lines 28 to 31), before returning an acknowledgment to the caller.
- Now, upon receiving an event $viewChange(V)$, process p computes the set of keys for which it is newly in charge (line 43). For each key k that it replicates, p also computes all the pairs (k, q) for which process q is a new replica in view V (line 44). For every such pair (k, q), p sends to q the state of k together with the handlers registered in the mapping $\mathcal{H}(k)$. Process p then waits until this information is received for every new key it replicates (line 46).

Justification. The need to use the LKVS interface instead of a plain map-like key-value store interface stems from two reasons. First, we wish to ship operations and not the full object state between the client application and storage nodes. Second, we require the ability to implement any type of linearizable and deterministic object. This includes in particular objects that are able to solve wait-free consensus. To understand this last observation, let us consider a compare-and-swap object (CAS) implemented using CRESON. $CAS(a, b)$ checks whether the stored value is still a, and if so, replaces a with b; in any case the old value is returned. We may solve easily wait-free consensus with CAS by simply comparing the proposal of each client process to some initial blank value. As a consequence of Herlihy's hierarchy [63], this reduction tells us that a plain key-value store is not enough to implement CRESON with the guarantees listed in Section III-B. In other words, it is not possible to implement an atomic and wait-free CAS by using separate $put()$ and $get()$ calls. A solution to this problem is to add a synchronization primitive to the interface of the key-value store. Such a choice is not appealing as *(i)* it necessitates a helping mechanism to ensure progress under contention, and *(ii)* it requires to transfer the object state between the client and the server side. The addition of listening capabilities is an alternative solution that matches our requirements and allows for a simple and elegant design as we detail in the next section.

D. Object Management

We now describe how callable and replicated shared objects are managed using the LKVS. Each object o has a type T and a identifier id. This pair (T, id) must identify o uniquely. It is used as its *reference* ref_o in the LKVS. Let us note as well that each storage node hosting a replica of the value for key ref_o also maintains a copy of all handlers associated with that key. As calls to the LKVS are linearized, the observation of $put()$ calls by these handlers happens in the same order. Client proxies send operations through $put()$ calls that are intercepted and interpreted by the handlers.

Creation and Disposal. When the client application invokes the constructor of some object o with arguments $args$, CRESON instead instantiates a proxy, of the same type as o, that the client uses instead of the object itself.

The first call to some method of the proxy *opens* it. To open a proxy r, CRESON first registers r as a listener of ref_o together with its session handler h_r. This is performed with a $regListener(ref_o, r, h_r)$ call. Then, the proxy executes the call $put(ref_o, \langle \texttt{OPEN}, r, args \rangle)$ to request the instantiation of the object on the storage node(s). This $put()$ call triggers the $handle(u, v)$ call for session handlers associated with key ref_o, and in particular h_r. If h_r is the only session handler, it creates a new *object* handler h_o. It then calls $h_o.handle(u, v)$, forwarding the `OPEN` operation. Note that in this case, if there already existed a value under key ref_o, this value has been received as v and it is the serialized representation of the object. The object handler either (1) uses the serialized representation of the object in $v \neq \perp$ and maps it to a new object or (2) creates a new object using the constructor with parameters $args$. If other session handlers already exist for ref_o, h_r uses a map of known handlers to locate and link to the existing h_o.

The disposal of objects happen when all proxies are garbage-collected at the client applications side. Closing a proxy r of some shared object o consists in executing $put(ref_o, \langle \texttt{CLOSE}, r \rangle)$, then unregistering r as a LKVS listener. The `CLOSE` operation at the last session handler for ref_o triggers the closing of the object handler, the serialization of the object and its storage under key ref_o.

Method calls. Once the proxy r is opened at the client application side, a call to some method m *(i)* registers a future [64] for that call, *(ii)* executes $put(ref_o, \langle \texttt{INV}, r, m \rangle)$ on the LKVS to invoke the method on the shared object, then *(iii)* awaits until the future completes, which happens when the corresponding session handler calls back the proxy listener.

In more details and using Figure 2, consider a call $put(ref_o, \langle \texttt{INV}, r, m \rangle)$ sent to the LKVS interface (step ❶). All the replicas of ref_o receives this insertion in the same order (step ❷). The insertion triggers all session handlers for ref_o to execute $handle(u, v)$ (step ❸). Each of the session handler checks the identity of the calling proxy. If u does not mention r, h_r simply returns $\perp$. If u mentions proxy r, it means that the operation is from the corresponding client proxy and h_r calls $h_o.handle(u, v)$ (step ❹). Upon receiving the `INV` operation,

h_o executes the corresponding method call on its copy of the object. The result to the session handler h_r that called h_o is sent back to the proxy (step ❺). It includes the return value for the method call, if any. As all replicas send this result, the proxy receives it even if all but one of the replicas fails. Duplicate receptions are filtered at the proxy.

E. Composability

CRESON supports the composition of shared objects. To implement this functionality, we allow object handlers to use proxies. This means that when an object handler executes the code of a method, it may internally call another shared object.

Composition is not granted for free. As a method call executes synchronously at the object handler, the application should proscribe call loops across several shared objects. Call cyclicity can sometimes (but not always) be detected at compilation time. In practice, CRESON avoids the problem of entering call loops with the `hashcode` and `equals` methods as follows: Upon a call to `hashcode`, CRESON returns the hash code of the reference of the object held by the proxy. Now in the case of `equals`, if the operand is also a proxy, their references are compared; otherwise, `equals` is called on the operand with the proxy as argument.

F. Discussion

Thanks to the disjoint access parallelism property of the LKVS, concurrent accesses to different objects scale. On the other hand, accesses to the same object must be linearized by the replicas. This implies that, similarly to other stores offering linearizable writes [35], [39], [44]–[48], replicas must apply operations to a given object one after the other. As a result, the system does not scale if clients contend all on the same object. We also note that all method calls to a proxy translate into $put()$ operations at the LKVS interface. This includes calls to methods that only read the state of the object (e.g., getters). We discuss the pros and cons of application-side caching in Section V. In short, this approach can only provide *sequentially consistent* operations. Due to disjoint-access parallelism, accesses to different objects happen at different servers of the LKVS. Unlike with linearizability, sequential consistent objects are not composable. This means that the programmer must deal with inconsistencies related to the visibility of updates on multiple objects.

IV. Use Case

In this section, we illustrate the benefits of CRESON in a representative usage scenario. Our target use case is a *personal cloud* storage service. In what follows, we present this service, and then we cover two implementations: one relying on object-relational mapping, and another using CRESON. When presenting the latter, we also describe the integration of CRESON with the Java language.

Context. Personal cloud storage services such as Dropbox, Ubuntu One, Google Drive or Box are distributed systems that synchronize files across multiple devices with a back end operating in the Cloud. They typically rely on NoSQL databases for their data, and relational databases to store their metadata. Indeed, these services require strong consistency guarantees to keep their files up-to-date, and relational databases and transactions are the de-facto standard for these requirements.

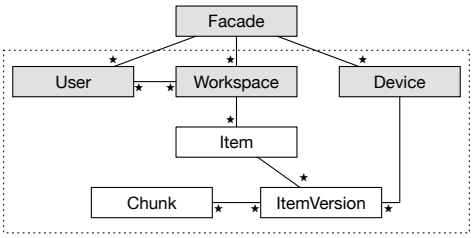


Fig. 3. SyncService UML class diagram – (PostgreSQL) dotted border enclose tables, (CRESON) grayed classes are `@Entity`

But relational databases also pose some core problems in this context. One important issue is the *expressiveness* of SQL to implement recursive data structures. Such services maintain metadata for the file system of every user, including directories, files, chunks and versions. These structures are, in general, complex to model in SQL to ensure queries efficiency.

Another relevant problem is *scalability*. It is generally addressed by using relational sharding. This requires non-trivial changes to the application. It also drops the ACID guarantees for transactions that touch multiple shards, necessitating great care when deciding on the sharding plan. The difficulty of sharding is also related to load balancing. A measurement study on existing personal cloud storage services [65] shows that a small fraction of very active users accounts for the majority of the traffic, while there is an important population of almost idle users (65% in the *Ubuntu One* service and 30% in Dropbox). Sharding inadequately may lead to serious load imbalance, something difficult to predict before the sharding plan is actually enforced.

A third important issue is regarding *elasticity*. Several past works [65]–[67] indicate that existing personal cloud storage services present strong diurnal seasonality. The problem of supporting elasticity with a sharded relational database is that adding or removing a store node requires stopping the service, re-sharding and restarting, which is not a viable option for a production service. In fact, the classical solution in this case is over-provisioning to avoid re-sharding.

Overview. Our use case application is StackSync [68], an open-source personal cloud storage service. StackSync is implemented in Java using an object storage service (OpenStack Swift) for its data, and a relational database (PostgreSQL) for its metadata. The StackSync synchronization service, or *SyncService* hereafter, uses ORM technologies to persist metadata in the relational database. In what follows, we present the ORM-based implementation and our portage to CRESON. These two approaches manipulate the same metadata objects detailed below.

Metadata. The SyncService operates several multi-threaded instances. Each instance processes incoming StackSync requests

to modify the application metadata. We depict the class schema of the SyncService in Figure 3. At some SyncService instance, a thread interacts with CRESON using a `Facade` object. Classes including the `Facade` and below differ from one persistence technology to another. Their portage is where we spent most of our effort.

A `Workspace` object models a synced folder. It is composed of files and directories (`Item` in Figure 3). For each `Item`, the SyncService stores versioning information as `ItemVersion` objects. Similarly to other personal cloud storage services, StackSync operates at the sub-file level by splitting files into chunks; this greatly reduces the cost of data synchronization. A `Chunk` is immutable and identified with a fingerprint. It may appear in one or more files.

Relational Approach. To scale up the original relational implementation, we followed conventional wisdom and sharded metadata across multiple servers. The key enabler of this process is PL/Proxy, a stored procedure language for PostgreSQL. PL/Proxy allows dispatching requests to several PostgreSQL servers. It was originally developed by Skype to scale up their services to millions of users.

Following this approach, we horizontally partition metadata by hashing user identifiers with PostgreSQL built-in function. As a result, all the metadata of a user is slotted into the same shard. Any request for committing changes made by the same user is redirected to the appropriate shard. In detail, we accomplish this with the following PL/Proxy procedure (we omit some parameters for readability):

```
1 CREATE OR REPLACE FUNCTION commitItem(client_id uuid, ...)
2 RETURNS TABLE(item_id bigint, ...) AS $$
3   CLUSTER 'usercluster';
4   RUN ON hashtext(client_id::text);
5 $$ LANGUAGE plproxy;
```

In the above code, the proxy first picks a shard with `RUN ON hashtext(client_id::text)`. Then, it runs a stored procedure named `commitItem` at the chosen shard. This procedure contains the logic for committing a `Workspace` change in PostgreSQL. It re-implements the original SQL transaction for committing changes to a `Workspace` (`commitRequest` in [68]). At a high level, committing a change requires several SQL insertions to different tables: one `INSERT` statement to create a new version of the file, and a variable number of SQL insertions to the table that records the chunks that constitute the file. We further detail how the stored procedure `commitItem` works in Section VI-B1.

At the time of this writing, there is no other mature technology to create a virtual table spread across multiple PostgreSQL servers. While the PL/Proxy approach has its pros, for example, having SQL statements operate close to the data, it also presents several caveats. First, it requires encapsulating all SQL statements in server-side functions. This adds significant complexity to the development and maintenance of the application logic. The amount of changes in lines of code (LoC) were several thousands, and it took us multiple weeks to obtain a stable system. Second, our implementation only scales up plain file synchronization. Partitioning is at odds with sharing. To allow two users to share a folder while maintaining consistency, the system would need to support cross-partition transactions. However, PL/Proxy currently does not provide such a feature.

```
1  @Entity(key = "id")
2  public class Workspace {
3
4   public UUID id;
5   private Item root;
6   private List<User> users;
7
8   /* ... */
9
10  public boolean isAllowed(User user) {
11   return users.contains(user.getId());
12  }
13 }
```

```
1  @Entity(key = "id")
2  public class Facade {
3
4   @Entity(key = "deviceIndex")
5   public static Map<UUID,Device> devices;
6
7   @Entity(key = "workspaceIndex")
8   public static Map<UUID,Workspace> workspaces;
9
10  @Entity(key = "userIndex")
11  public static Map<UUID,User> users;
12
13  public UUID id;
14
15  /* ... */
16
17  public boolean add(Device device) {
18   return deviceMap.putIfAbsent(
19      device.getId(),device) == null;
20  }
21 }
```

Fig. 4. Workspace and Facade classes

Portage to CRESON. As CRESON fully embraces the OOP paradigm, we greatly simplified the code of the SyncService. Starting from the PostgreSQL code, we removed the overhead of SQL and the low-level procedural PL/PostgreSQL constructions. The portage to CRESON was then split in two steps. First, we moved all the application logic back to the classes. As we benefited from the high-level construct from Java, this phase took only a couple of days. Overall, we wrote fewer lines of code, and in our honest opinion, this code is easier to understand. In a second step, we declared the shared objects in the SyncService application. To this end, we used the syntactic support for the Java language provided by CRESON. We detail this support and our design choices below.

Figure 4 illustrates the language support of CRESON for the `Workspace` and `Facade` classes. The language integration is on purpose close to the Java Persistence API (JPA) standard and is intended to be familiar to users of Object-NoSQL mappers such as Hibernate OGM or Objectify. In this figure, we observe the annotation `@Entity`. This annotation informs that an object or a field is shared. In more details, annotating

`Workspace` with `@Entity` indicates that instances of this class are shared. The value of the parameter `key` tells us that the field `id` defines the key of the object used in its reference.

The `Facade` class in Figure 4 provides another example of the Java language support provided by CRESON. This class offers basic end-user actions, such as the addition of a novel device (see lines 17-20 in Figure 4-bottom). The declaration of this class includes three collections: `devices`, `workspaces` and `users`. CRESON instantiates each collection remotely, thanks to the annotation `@Entity`. In this case, the value of `key` (at lines 4, 7 and 10) defines the key of each collection. A field annotated with `@Entity` is a unique instance accessible from all client application processes. In other words, each `@Entity` collection is transparently shared between all `Facade` objects. As a consequence, CRESON requires such fields to be class members, that is, the presence of the `static` keyword (see lines 5, 8 and 11).

In Figure 3, we list the key classes of StackSync that we ported to CRESON. The key question we had to address here was "What objects do we share across application instances?". In a basic design, we shared all application objects. As the performances were lower than those of the PostgreSQL implementation, we then embedded some classes into others. Namely, `Item`, `ItemVersion` and `Chunk` were all embedded in `Workspace`. We ended-up with the set of grayed classes in Figure 3 and a few static fields in `Facade` annotated with `@Entity`. This configuration has the advantage of maintaining operations across workspaces atomic. Another option is to share only the `Facade` objects which results in non-atomic cross-workspace operations as with PL/Proxy.

V. Implementation

We implemented CRESON on top of Infinispan [5], an industrial-grade open-source key-value store. The base code of CRESON is in Java and accounts for 4,000 SLOC. To implement the full support of the LKVS abstraction, we contributed a batch of modifications to Infinispan that weights around 13,500 SLOC.

In comparison to the implementation of the CRESON constituents we described in Section III, our code also includes a few additional practical features. We detail such features next, as well as interesting aspects of our implementation.

Language Support. We added the Java support of CRESON annotations with AspectJ [69]. As argued in Section IV, this support allows a developer to easily implement an application while distributing its persistent state on the NoSQL store. When it encounters a call to a constructor of an object declaring a CRESON annotation, the AspectJ compiler replaces this call with the creation of a proxy using aspect weaving. For performance reasons, shared maps such as `deviceIndex` in the `Facade` class (see Figure 4) are transparently stored as collection of key/value pairs in the LKVS.

Serialization. When an object is serialized, the proxy triggers a `writereplace` method that returns the object's reference. The actual serialization of the object in CRESON is however not weaved. When it occurs (e.g., upon persisting the object), CRESON needs to access all of the appropriate fields. In the current state of our implementation, we simply mark such fields public. Another possible approach would be to declare appropriate getters, similarly to what exist for JavaBeans.

Listener Mutualization. The cost of installing a listener per shared object can be high. In particular, for each shared object a naive implementation would use one open connection to each server hosting a replica. Instead, the implementation of CRESON multiplexes the listeners for the same application process, and thus only opens a few connections in total.

Chaining Calls. CRESON allows chaining calls between objects, provided that no cycles are created (see Section III-E). Starting from the application process, a call chain traverses multiple object handlers located on different LKVS machines. In this case, the storage nodes hosting replicas will call the same methods. For instance in Figure 4, executing `addDevice` on a facade object f induces that every replica of ref_f executes a call to `putIfAbsent` on the `devices` index. These calls may return inconsistent results, e.g., *true* for the first one, and *false* for the others. CRESON solves this problem by enforcing *call idempotence*. With more details, each call has a unique ID. Once a method returns, the object handler stores its return value r. If a call with the same ID is received, the handler simply returns r. The call ID is computed by the proxy with the help of a (deterministic) generator. The object handler creates this generator in a `ThreadLocal` variable when the object is instantiated for the first time.

Call Isolation. Execution of method calls on application objects at the server side may lead to system-wide performance issues; for instance when encountering an infinite loop. The solution proposed by the authors of [50] is to restrict loops to system-provided iterators with termination guarantees. This is required as their deployment context considers multiple applications accessing the same coordination service, and the system must enforce that one application does not impact the performance of the others. CRESON on the other hand recommends the deployment of one dedicated NoSQL store for one specific application. Limiting what can be implemented in objects methods would reduce the interest of the approach. We consider that careful application design is more adapted to our target context.

Client-side caching. In CRESON, every method invocation triggers a call to $put()$ at the LKVS interface. To improve performance of read-heavy workloads, CRESON offers an optional client-side caching mechanisms. Methods that are read-only can be annotated by the programmer with the `@Readonly` keyword. When a `@Readonly` method is called, the object handler returns not only the response value but also the full state of the object. This state is then stored locally at the proxy, and every call to a `@Readonly` method evaluates on this locally cached state.

The use of the client-side caching comes at the price of weaker consistency, namely sequential consistency [70]: updates to some unique object are seen in a common sequential order, but this order might not respect the real-time ordering. Sequential consistency allows to execute read-only methods locally, something not achievable under linearizability [57]. On the other hand, objects are not composable under sequential consistency [55], and thus assessing the correctness of non-sequential executions using several objects is left to the developer. Objects are indeed hosted by different servers of the LKVS and their accesses are disjoint-access parallel. While sequential consistency can allow seeing updates to several objects in a unique order when these accesses are made to a unique access point (e.g. as in ZooKeeper [44]), this is not the case when the accesses are handled in parallel as in CRESON. This means that, unlike with linearizability, it is possible for two clients using client-side caching to see updates to several objects in different orders, requiring extra care for reaching application-level consistency.

VI. Evaluation

In this section, we evaluate the performance of CRESON with micro-benchmarks and our use case application, StackSync.

A. Micro-benchmarks

We use a cluster of virtualized 8-core Xeon 2.5 Ghz machines. Each machine has 8 GB of memory and runs Ubuntu Linux 14.04 LTS. The machines are connected to a virtualized 1 Gbps switched network. The performance of the network, as measured by *ping* and *netperf*, is of 0.3 ms for a round-trip with a bandwidth of 117 MB/s. At an Infinispan server, a limited-size cache holds recently used values (i.e., serialized objects). Passivation to persistent storage of these values occurs in the background when 10^5 values are present in the cache, using the LRU strategy. The risk of losing serialized objects due to a server failure is already covered by replication, so synchronous writes of values to disk are not necessary. The (virtual) hard-drive read/write performance is 246/200 MB/s. Unless otherwise stated, the replication factor of an object is 2, and we use 3 Infinispan servers.

Raw Performance. We first evaluate the performance of Infinispan itself, without employing any of the LKVS features. We use the Yahoo! Cloud Serving Benchmark (YCSB) [71]. YCSB is a data storage benchmark that consists of four workloads with read and write calls (Infinispan does not support scan calls). Workload A is update-heavy, B is read-heavy, C is read-only, and D consists of repeated reads (95% of calls) followed by insertions of new values. In our evaluation, the four workloads access 10^6 items, and each item weights 1 KB.

Figure 5(a) shows the maximal throughput for the four different YCSB workloads. We observe that read-dominated workloads achieve around 17 to 20 Kops/sec, while the update-heavy workload is capped at around 8.5 Kops/sec. This performance is in-line with other NoSQL solutions [71].

Simple Distributed Objects. In Figure 5(b), we plot the per-key performance of CRESON when either calling $put()$ at the LKVS level, or when using simple shared objects. The first object we consider is a counter that clients increment (`inc`). The second object is a queue and clients execute back-to-back an en-queue (`enq`) followed by a de-queue (`deq`) of some random elements. These last two benchmarks are identical to the ones proposed in [50].

The maximal throughput of $put()$ over a single key is 1,134 calls/sec. This value upper bounds the performance a single distributed object may provide in CRESON. In Figure 5(b), we observe that CRESON processes 681 increments per second on a shared counter. This is around half the throughput of directly calling $put()$ at the LKVS level. On the other hand, when two calls are consecutive, this throughput is expected to be divided by 2. This is the case for an en-queue followed by a de-queue (column "`enq; deq`" in Figure 5(b)).

When parallelism increases, CRESON is able to sustain many more operations per second. We illustrate this in Figure 5(c). This figure depicts the latency of `inc` in relation to the contention between concurrent threads in accessing the shared counters. Due to the disjoint access parallelism property, performance improves with the number of shared counters. From top to bottom, we change the number of counters available in the system. With 24 threads and 100 counters, CRESON is able to sustain $3,400$ ops/sec; four times the speed at which these threads would access a single counter.

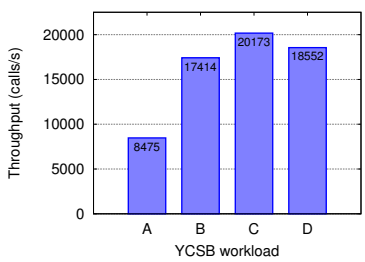
B. StackSync

This section comparatively evaluates the performance of StackSync using PostgreSQL and CRESON. To conduct this comparison, we use two different workloads, a synthetic workload consisting of randomly generated metadata, and a realistic workload using a trace from the Ubuntu One personal cloud service [72]. We use the former to evaluate the peak performance of the two systems, and the latter to analyze the behavior of the metadata back end in a realistic scenario.

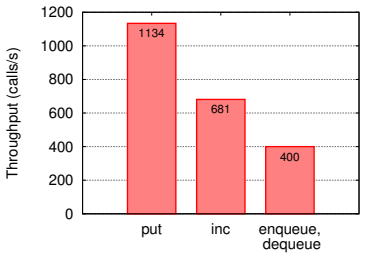
The core of the two workloads consists in users adding items to their workspaces. The entry point of this functionality is the `doCommit` method of the `Facade` object. This method receives as input a workspace, a user and a device, as well as a list of items to be added to the target workspace. Below, we briefly describe how the `doCommit` method is implemented in PostgreSQL. The CRESON implementation is similar following the object-oriented paradigm.

1) Workload Description and Settings: The `doCommit` method is implemented as a transactional stored procedure in PostgreSQL. This procedure loads from the database the workspace, the user and the device metadata, then it executes access control and base existence tests. Further, it iterates over the list of added items and calls the `commitItem` procedure on each of them.

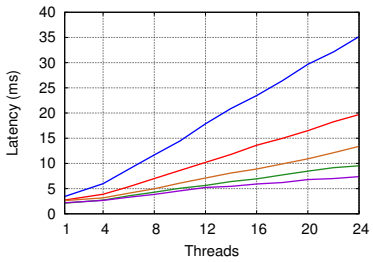
For each item, the `commitItem` procedure checks if it already exists in the database. If this is not the case, the first version of the item is created and added to the



(a) YCSB workloads on Infinispan



(b) Accessing a single key: Infinispan $put()$ and CRESON calls



(c) Concurrent increments – from top to bottom 1, 2, 5, 10 and 100 counters

Fig. 5. Throughput of Infinispan and performance of CRESON for single and multiple objects

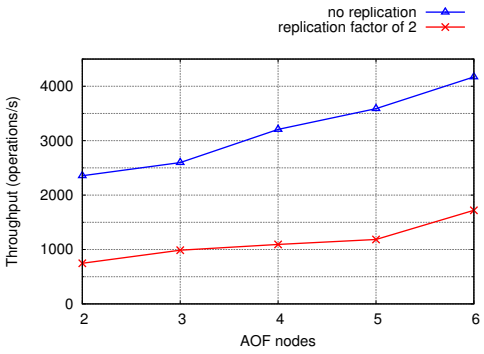


Fig. 6. Impact of replication (persistence is on)

database. Otherwise, `commitItem` produces a new version. In both cases, `commitItem` checks that the added version is appropriate according to the metadata provided by the client (i.e., no concurrent data item creation). If this test fails, the commit is rejected and the sync-service sends the latest version of the faulty item back to the client.

In all the experiments that follow, we fix the number of users of the personal cloud storage service to 4,000. We also assign a unique workspace to each user. A call to `doCommit` consists in the addition of 20 new items to the workspace.

Impact of replication. In our first experiment, we consider a single StackSync server that accesses the CRESON deployment. This server is multi-threaded and makes use of 240 concurrent threads. Figure 6 reports the results we obtained when the per-node cache contains at most 1,000 values (serialized objects) and values are persisted to disk. We vary the replication degree between 1 and 2 replicas. For each experiment, the StackSync server executes 10^6 calls, and we plot the average. Let us emphasize that, in order to produce Figure 6, we simply added CRESON nodes between each measurements, relying on the elasticity of the system.

As mentioned in Section IV, we opt for distributing all the classes that are grayed in Figure 3, in order to allow atomic cross-workspace operations in the CRESON portage. Coordinating more distributed classes is expected to be more expensive than when distributing a single one (e.g., only the `Facade`).

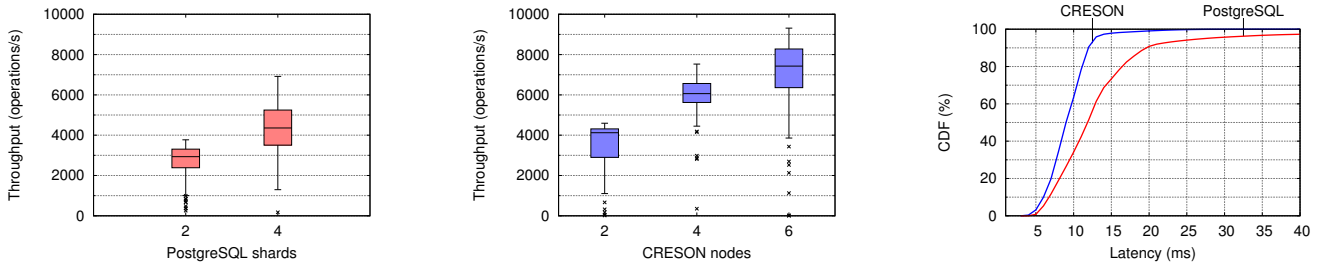
When distributing all the classes that are grayed in Figure 3, and when the replication factor equals two, the performance improvement of CRESON equals 1.74 when scaling the system from 2 to 6 servers. This is very close to the scale-up factor we obtain when CRESON uses a single replica per object (1.77 in that case). Additional results (not presented in Figure 6) indicate that if we only distribute the `Facade` objects and assign each workspace to a single `Facade`, CRESON with a replication factor of two sustains 2.7 Kop/sec with 2 nodes, and 5.6 Kop/sec with 6 nodes. This last design offers guarantees similar to those of the sharding approach with PL/Proxy. In particular, it does not provide atomicity on cross-workspace operations. These results also indicate that if we increase memory usage at the servers, moving the cache size from 10^3 to 10^5 values, CRESON throughput improves by around 30%.

To further understand the performance of CRESON and compare it to PostgreSQL, it is necessary to ensure that the StackSync server is not the bottleneck. We solve this problem with the help of a queue messaging service (RabbitMQ). Initially, this service is pre-loaded with either synthetic messages, or messages extracted from the Ubuntu One trace.

Based on the above architecture, the testbed we use for our remaining experiments consists in the following settings: *(i)* Storage: 2 to 6 servers consisting of a 4-core CPU and 8 GB of RAM, and running either PostgreSQL and PL/Proxy or CRESON with data persistence on in both cases; *(ii)* StackSync: up to 24 single-core CPU and 2 GB of RAM running the StackSync service; and *(iii)* RabbitMQ: one server with a 6-core CPU and 16 GB of RAM. We should note that in this setting PostgreSQL ensures a higher level of data persistence than CRESON, as we do not use replication in this experiment.

Synthetic workload. In this experiment, we pre-load RabbitMQ with 4 Million commit operations. Then, at the same time, we start the 24 StackSync servers, using 10 threads for each of them. This setting corresponds to the peak performance of PostgreSQL in our testbed. StackSync servers process commit operations concurrently at their maximum throughput.

Figure 7(a) depicts the peak performance of PostgreSQL when we use 2 and 4 shards. Notice that in this last case, we deployed 2 PL/Proxy nodes; otherwise with 4 shards a



(a) PostgreSQL *(2 proxies, stored procedures on, persistence on)*

(b) CRESON *(one replica per object, persistence on)*

(c) Ubuntu One trace: `doCommit` operation latency

Fig. 7. Comparing CRESON and PostgreSQL with the StackSync application

single PL/Proxy node is bottlenecking. The performance of CRESON using a single replica per item, and data persistence on is depicted in Figure 7(b). On average, CRESON is close or better than PostgreSQL in all cases. This comes from the fact that proxies are used solely to forward transactions and not to execute them. With 6 nodes, CRESON performance improvement is close to 50%.

Ubuntu One Trace. Figure 7(c) depicts the cumulative distribution of the latency of `doCommit` calls when following the Ubuntu One trace. In this experiment, we use 4 shards and 2 proxies for PostgreSQL, and CRESON makes use of 6 nodes with a replication factor equal to one and data persistence on.

The trace is executed without a speed-up factor, simply by adding all its operations to the RabbitMQ server. On average, the trace requires 300 `doCommit` operations per second and it lasts several hours. Figure 7(c) tells us that CRESON consistently responds quicker than PostgreSQL for this workload.

VII. Conclusion

We presented CRESON, a system offering support for callable and shared objects over NoSQL. Our framework is built upon a novel NoSQL abstraction, the Listenable Key Value Store (LKVS), that we contributed in this paper. CRESON allows using shared objects for designing applications in Cloud environments. It provides strong guarantees in terms of availability, through replication and wait-freedom. It also ensures strong consistency, offering linearizability even for composed objects. CRESON alleviates the need to use repeated conversions from an in-store and an in-memory representation of objects at the clients side. Instead, it ensures that these conversions and calls to object methods happen at the server side. This addresses an important limitation of current Object-NoSQL frameworks. Similarly to these frameworks, integration with the Java language allows application developers to simply mark which (composed) objects and fields must be hosted in the NoSQL storage.

Our comparison of the implementation of the metadata storage service for StackSync, a personal cloud application, using CRESON and object-relational mapping with sharding indicates that our approach is simpler. CRESON further outperforms the ORM-based implementation both in throughput and latency. In particular, we have shown that this result holds for a trace obtained with the real-world deployment of the Ubuntu One personal cloud storage service.

An interesting follow-up of this work is the addition of querying capabilities to the objects, to avoid the maintenance of explicit indexes in the common case. We envision this extension as our immediate future work, targeting an integration with technologies such as Hibernate Search.

Acknowledgments. We thank the anonymous reviewers and Maarten van Steen for their useful suggestions and for their help in preparing the final version of this paper. We are grateful to Valerio Schiavoni for comments on earlier drafts, and to Marc Shapiro for fruitful discussions on the topic. The work of P. Sutra, E. Rivière, E. Bernard, W. Burns and G. Zamarreño was supported by the LEADS project funded by the European Commission under the Seventh Framework Program (grant 318809). The work of C. Cotes, M. Artigas and P. Lopez has been partly funded by the EU project H2020 "IOStack: Software-Defined Storage for Big Data" (644182) and Spanish research project "Cloud Services and Community Clouds" (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] W. Vogels, "Eventually Consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[2] 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 *21st ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP, 2007, pp. 205–220.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[4] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A Distributed, Searchable Key-value Store," in *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM. ACM, 2012, pp. 25–36.

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

[6] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger, "Schemaless NoSQL data stores – Object-NoSQL Mappers to the rescue?" in *16th conference on Database Systems for Business, Technology, and Web*, ser. BTW, Hamburg, Germany, 2015.

[7] C. Bauer, G. King, and G. Gregory, *Java Persistence with Hibernate*. Manning Publications, 2015.

[8] V. Reniers, A. Rafique, D. Van Landuyt, and W. Joosen, "Object-NoSQL Database Mappers: a benchmark study on the performance overhead," *Journal of Internet Services and Applications*, vol. 8, no. 1, p. 1, 2017.

[9] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira, "Safe and Efficient Sharing of Persistent Objects in Thor," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD. ACM, 1996, pp. 318–329.

[10] "Objectify documentation," https://github.com/objectify/objectify/wiki/Caching.

[11] B. Liskov, "Distributed Programming in Argus," *Commun. ACM*, vol. 31, no. 3, pp. 300–312, Mar. 1988.

[12] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little, "The design and implementation of Arjuna," *Computing Systems*, vol. 8, no. 2, pp. 255–308, 1995.

[13] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190–205, Mar. 1992.

[14] S. Maffeis, "Adding Group Communication and Fault-tolerance to CORBA," in *1st USENIX Conference on Object-Oriented Technologies*, ser. COOTS, 1995.

[15] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Consistency of Partitionable Object Groups in a CORBA Framework," in *30th Hawaii International Conference on System Sciences: Software Technology and Architecture - Volume 1*, ser. HICSS, 1997.

[16] P. Felber, R. Guerraoui, and A. Schiper, "The implementation of a CORBA group communication service," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 93–105, 1998.

[17] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni, "System Support for Object Groups," in *13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, 1998.

[18] B. Liskov, M. Castro, L. Shrira, and A. Adya, "Providing Persistent Objects in Distributed Systems," in *13th European Conference on Object-Oriented Programming*, ser. ECOOP. London, UK, UK: Springer-Verlag, 1999, pp. 230–257.

[19] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM*, vol. 34, no. 10, pp. 50–63, Oct. 1991.

[20] M. Stonebraker and G. Kemnitz, "The POSTGRES Next Generation Database Management System," *Commun. ACM*, vol. 34, no. 10, pp. 78–92, Oct. 1991.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A Distributed SQL Database That Scales," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1068–1079, Aug. 2013.

[22] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[23] J. W. Stamos and D. K. Gifford, "Remote Evaluation," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 537–564, Oct. 1990.

[24] B. Garbinato, R. Guerraoui, and K. Mazouni, "Implementation of the GARF replicated objects platform," *Distributed Systems Engineering*, vol. 2, no. 1, pp. 14–27, 1995.

[25] M. van Steen, P. Homburg, and A. S. Tanenbaum, "Globe: A Wide-Area Distributed System," *IEEE Concurrency*, vol. 7, no. 1, pp. 70–78, Jan. 1999.

[26] S. Landis and S. Maffeis, "Building reliable distributed systems with CORBA," *Theory and Practice of Object Systems*, vol. 3, no. 1, 1997.

[27] P. Felber, "The CORBA Object Group Service: A Service Approach to Object Groups in CORBA," Ph.D. dissertation, École Polytechnique Fédérale de Lausanne (EPFL), 1998.

[28] S. Maffeis, "The Object Group Design Pattern," in *2nd Conference on USENIX Conference on Object-Oriented Technologies*, ser. COOTS. USENIX Association, 1996.

[29] P. Felber and P. Narasimhan, "Experiences, strategies, and challenges in building fault-tolerant CORBA systems," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 497–511, May 2004.

[30] H. Ishikawa, *Object-oriented database system - design and implementation for advanced applications*, ser. Computer science workbench. Springer, 1993.

[31] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik, "Building an Object-oriented Database System." Morgan Kaufmann, 1992, ch. The Object-oriented Database System Manifesto.

[32] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 1989.

[33] J. E. B. Moss, "Design of the Mneme Persistent Object Store," *ACM Trans. Inf. Syst.*, vol. 8, no. 2, pp. 103–139, Apr. 1990.

[34] A. V. Aho and J. D. Ullman, "Universality of Data Retrieval Languages," in *6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL. ACM, 1979, pp. 110–119.

[35] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A New Paradigm for Building Scalable Distributed Systems," in *21st ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP, 2007.

[36] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Boosting Database Replication Scalability Through Partial Replication and 1-Copy-Snapshot-Isolation," in *13th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC, 2007.

[37] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized STM algorithm," in *31st IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS, 2010.

[38] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *23rd ACM Symposium on Operating Systems Principles*, ser. SOSP, 2011.

[39] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally Distributed Database," *ACM Trans. on Comp. Sys.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[40] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication," in *32nd IEEE International Conference on Distributed Computing Systems*, ser. ICDCS, 2012.

[41] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Scalable Atomic Visibility with RAMP Transactions," in *ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD, 2014.

[42] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable Distributed Software Transactional Memory," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*, ser. PRDC, 2009.

[43] 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.

[44] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX Annual Technical Conference*, ser. USENIX ATC. USENIX Association, 2010.

[45] M. Burrows, "The Chubby Lock Service for Loosely-coupled Distributed Systems," in *7th Symposium on Operating Systems Design and Implementation*, ser. OSDI, 2006.

[46] D. Gelernter, "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan. 1985.

[47] A. N. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga, "DepSpace: A Byzantine Fault-tolerant Coordination Service," in *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, ser. Eurosys. ACM, 2008, pp. 163–176.

[48] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[49] B. Kalantari and A. Schiper, *14th International Conference Distributed Computing and Networking*, ser. ICDCN. Springer Berlin Heidelberg, 2013, ch. Addressing the ZooKeeper Synchronization Inefficiency.

[50] T. Distler, C. Bahn, A. Bessani, F. Fischer, and F. Junqueira, "Extensible Distributed Coordination," in *10th European Conference on Computer Systems*, ser. EuroSys. ACM, 2015, pp. 10:1–10:16.

[51] M. Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," in *IEEE International Conference on Distributed Computer Systems*, ser. ICDCS. IEEE, 1986.

[52] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[53] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos Replicated State Machines As the Basis of a High-performance

Data Store," in *8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI, 2011.
[54] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.
[55] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, Jul. 1990.
[56] M. Herlihy and N. Shavit, "On the Nature of Progress," in *15th International Conference on Principles of Distributed Systems*, ser. OPODIS, 2011.
[57] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, May 1994.
[58] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group Communication Specifications: A Comprehensive Study," *ACM Comput. Surv.*, vol. 33, no. 4, 2001.
[59] R. Guerraoui and A. Schiper, "Genuine Atomic Multicast in Asynchronous Distributed Systems," *Theor. Comput. Sci.*, vol. 254, no. 1-2, 2001.
[60] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[61] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, Apr. 2010.
[62] 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 *29th Annual ACM Symposium on Theory of Computing*, ser. STOC, 1997.
[63] M. Herlihy, "Wait-free Synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.
[64] H. C. Baker, Jr. and C. Hewitt, "The Incremental Garbage Collection of Processes," *SIGPLAN Not.*, vol. 12, no. 8, pp. 55–59, Aug. 1977.
[65] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *Internet Measurement Conference*, ser. IMC. ACM, 2013, pp. 205–212.
[66] R. Gracia-Tinedo, M. S. Artigas, A. Moreno-Martinez, C. Cotes, and P. G. López, "Actively Measuring Personal Cloud Storage," in *6th IEEE International Conference on Cloud Computing*, 2013.
[67] S. Liu, X. Huang, H. Fu, and G. Yang, "Understanding Data Characteristics and Access Patterns in a Cloud Storage System," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid, 2013.
[68] P. G. Lopez, M. Sanchez-Artigas, S. Toda, C. Cotes, and J. Lenton, "StackSync: Bringing Elasticity to Dropbox-like File Synchronization," in *15th International ACM/IFIP/USENIX Middleware Conference*, ser. Middleware, 2014.
[69] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *15th European Conference on Object-Oriented Programming*, ser. ECOOP, 2001.
[70] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.
[71] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *1st ACM Symposium on Cloud Computing*, ser. SoCC, 2010.
[72] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting UbuntuOne: Autopsy of a Global-scale Personal Cloud Back-end," in *ACM Internet Measurement Conference*, ser. IMC, 2015.