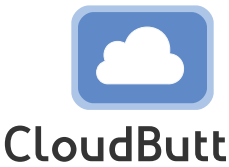

HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D4.1 Initial prototype for stateful serverless computation

Due date of deliverable: 30-06-2019
Actual submission date: 28-06-2019

Start date of project: 01-01-2019 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v1.0
Number of pages	24
WP/Task related to this document	WP4 / T4.1
WP/Task responsible	IMT
Leader	Pierre Sutra (IMT)
Technical Manager	Tristan Tarrant (RHAT)
Quality Manager	Marc Sánchez (URV)
Author(s)	Daniel Barcelona-Pons (URV), Anatole Lefort (IMT), Pierre Sutra (IMT), Gerard París Aixalà (URV), Pedro García (URV), Marc Sanchez (URV), Tristan Tarrant (RHAT)
Partner(s) Contributing	IMT, URV, RHAT
Document ID	CloudButton_D4.1_Public.pdf
Abstract	This document describes a prototype middleware for stateful serverless computing and preliminary experimental results on synthetic workloads.
Keywords	serverless, FaaS, distributed storage, machine learning

History of changes

Version	Date	Author	Summary of changes
0.1	19-06-2019	Pierre Sutra	First version
0.2	25-06-2019	Pierre Sutra	Revision
1.0	27-06-2019	Pierre Sutra	Final version

Table of Contents

List of Abbreviations and Acronyms

AWS	Amazon Web Services
DSM	Distributed shared memory
DSO	Distributed Shared Object
EC2	Amazon Elastic Compute Cloud
FaaS	Function as a Service
JVM	Java Virtual Machine
ML	Machine Learning
MPI	Message Passing Interface
NVRAM	Non-volatile random-access memory
SMR	State Machine Replication
SNS	Amazon Simple Notification Service
SOTA	State of the Art
SQS	Amazon Simple Queue Service
VPC	Virtual Private Cloud

1 Executive summary

Serverless computing is an emerging paradigm that greatly simplifies the usage of cloud resources and suits well to many tasks. Most notably, Function-as-a-Service (FaaS) enables programmers to develop cloud applications as individual functions that can run and scale independently. Yet, due to the disaggregation of storage and compute resources in FaaS, applications that require fine-grained support for mutable state and synchronization, such as machine learning and scientific computing, are hard to build.

The present document describes CRUCIAL, our initial prototype for stateful serverless computation. CRUCIAL allow to program highly-concurrent stateful applications atop serverless architectures. Its programming model keeps the simplicity of FaaS and allows to port effortlessly multi-threaded algorithms to this new environment. CRUCIAL is built upon the key insight that FaaS resembles to concurrent programming at the scale of a data center. As a consequence, a distributed shared memory layer is the right answer to the need for fine-grained state management and coordination in serverless.

Early validation results show that CRUCIAL is a promising approach for big data analytics with serverless architectures. In particular, we have implemented two common machine learning algorithms: *k*-means clustering and logistic regression. For both cases, CRUCIAL obtains superior or comparable performance to an equivalent Spark cluster.

2 Introduction

With the emergence of serverless computing, the cloud has found a paradigm that removes much of the complexity of its usage by abstracting away the provisioning of compute resources. This fairly new model was started by services such as Google BigQuery [1] and AWS Glue [2], and evolved into Function-as-a-Service (FaaS) computing platforms, such as AWS Lambda, Azure Functions, and Google's Cloud Functions, to name a few. With these platforms, a user-defined function and its dependencies are deployed to the cloud, where they are managed by the provider and executed on-demand.

Current practice shows that the FaaS model works well for applications that require a small amount of storage and memory due to the operational limits set by the cloud providers (see, for instance, AWS Lambda [3]). However, there are more limitations. While functions can initiate outgoing network connections, they cannot directly communicate between each other, and have little bandwidth compared to a regular virtual machine [4, 5]. This is because this model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier (e.g., uploading a photo to Amazon S3 [6]). Despite these constraints, recent works have shown how this model can be exploited to process and transform large amounts of data [7, 8, 9], encode videos [10], execute linear algebra tasks [11], and perform Monte Carlo simulations with large amounts of parallelism [12].

The above research projects, such as PyWren [7, 8] and ExCamera [10], prove that FaaS platforms can be programmed to perform a wide variety of embarrassingly parallel computations. Yet, these tools face also fundamental challenges when used out-of-the-box for many popular tasks. Although the list is too long to recount here, convincing cases of these ill-suited applications are machine learning (ML) algorithms. Just an imperative implementation of k-means [13] raises several issues: first, the need to efficiently handle a globally-shared state at fine granularity (the cluster centroids); second, the problem to globally synchronize cloud functions, so that the algorithm can correctly proceed to the next iteration; and finally, the prerogative that the shared state survives system failures.

Current serverless systems do not address these issues effectively. First, due to the impossibility of function-to-function communication, the prevalent practice for sharing state across functions is to use remote storage. For instance, serverless frameworks, such as PyWren [7, 8] and numpywren [11], use highly-scalable object storage services to transfer state between functions. Since object storage is too slow to share short-lived intermediate state in serverless applications [14], some recent works have opted to use faster storage solutions. For instance, this has been the path taken by Locus [9], which proposes to combine fast, in-memory storage instances with slow storage to scale shuffling operations in MapReduce. However, with all the shared state transiting through storage, one of the major limitations of current serverless systems is the lack of support to handle mutable state at a fine granularity (e.g., to efficiently aggregate small granules of updates). Such a concern has been recognized in various works [4, 15], but this type of fast, enriched storage layer for serverless computing is not available today in the cloud, leaving fine-grained state sharing as an open issue.

Similarly, FaaS platforms do not provide means to coordinate multiple functions. For instance, there should be abstractions for a function to signal another when a condition is fulfilled, or for multiple functions to synchronize, e.g., in order to guarantee data consistency, or just to ensure joint progress to the next stage of computation [15]. Of course, such fine-grained coordination should be also low-latency to not significantly slow down the application.

Contributions To overcome the aforementioned issues, we propose CRUCIAL, a system for the development of stateful distributed applications with serverless architectures. To simplify the writing of an application, CRUCIAL provides a thread abstraction that maps a thread to the invocation of a serverless function. To support fine-grained state management and coordination, our system builds a distributed shared object (DSO) layer on top of a low-latency in-memory data store. This layer provides out-of-the-box strong consistency guarantees, simplifying the semantics of global state mutation across serverless threads. Since global state is manipulated as remote objects, the interface for

mutable state management becomes virtually unlimited, only constrained by the expressiveness of the programming language (Java in our case). The result is that CRUCIAL can operate on small data granules, making it very easy to develop applications that have fine-grained state sharing needs. CRUCIAL also leverages this layer to implement fine-grained coordination. For applications that require longer retention of in-memory state, CRUCIAL ensures data durability using state machine replication (SMR), so that any acknowledged write can survive failures. This property is very compelling for serverless computing platforms since the in-memory storage instances offered by cloud providers, typically based on open-source projects such as Memcached or Redis, are not fault tolerant [15, 16].

Most importantly, CRUCIAL offers all of the above guarantees with almost no increase in the programming complexity of the serverless model. With the help of a few annotations and constructs, developers can run their single-machine, multi-threaded, stateful code in the cloud as serverless functions. CRUCIAL's programming constructs enable developers to enforce atomic operations on shared state, as well as to finely synchronize functions at the application level, so that (imperative) implementations of popular algorithms such as *k*-means can be effortlessly ported to serverless platforms.

Preliminary evaluation results show that, for representative applications that require fine-grained updates (e.g., *k*-means, logistic regression), CRUCIAL can rival, and even outperform, Spark running on a dedicated cluster.

Outline The remaining of the paper is structured as follows: We present CRUCIAL, our prototype for stateful serverless computation in Section 3. The softwares used to build this prototype are detailed in Section 4. Section 5 review the state of the art and how it compares to CRUCIAL. Section 6 covers the exploratory work which was conducted during this first period of the CloudButton project. We conclude in Section 7.

3 Current prototype

CRUCIAL is a framework to create and execute stateful serverless computation atop a Function-as-a-Service (FaaS) architecture. This section presents the programming model of CRUCIAL, its internals and preliminary performance results for machine learning (ML) applications.

3.1 Programming model

A CRUCIAL program is strongly similar to a regular multi-threaded, object-oriented Java one, besides some additional annotations and constructs. Table 1 summarizes the key programming abstractions available to developers that are explained hereafter.

Cloud threads To write a stateful application for serverless architectures, a programmer first builds its logic as a regular multi-threaded, object-oriented Java program. Then, two refinements are necessary to make it executable atop the FaaS model. First, each runnable object is associated with a `CloudThread`. An instance of this class hides the execution details of the remote cloud function to the developer. The second modification is to replace each mutable object shared between threads with its CRUCIAL counterpart.

State handling CRUCIAL already includes a library of base shared objects to support mutable shared data across cloud threads. This library consists of common objects such as integers, counters, maps, lists and arrays. By default, objects are *wait-free* and *linearizable* [17]. This means that each method invocation terminates after a finite amount of steps (despite concurrent accesses), and that concurrent method invocations behave as if they were executed by a single thread. CRUCIAL also gives programmers the ability to craft their own custom shared objects by decorating them with the

```
1 public class PiEstimator implements Runnable{
2  private final static long ITERATIONS = 100_000_000;
3  private Random rand = new Random();
4  private static crucial.AtomicLong counter = new crucial.AtomicLong(0);
5
6  public void run(){
7   long count = 0;
8   double x, y;
9   for (long i = 0L; i < ITERATIONS; i++) {
10   x = rand.nextDouble();
11   y = rand.nextDouble();
12   if (x * x + y * y <= 1.0) count++;
13  }
14  counter.addAndGet(count);
15 }
16 }
17
18 List<Thread> threads = new ArrayList<>(N_THREADS);
19 for (int i = 0; i < N_THREADS; i++) {
20  threads.add(new CloudThread(new PiEstimator()));
21 }
22 threads.forEach(Thread::start);
23 threads.forEach(Thread::join);
24 double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 1: Monte Carlo simulation to approximate π.

`@Shared` annotation. Annotated objects become globally accessible by any thread. CRUCIAL refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`.

Data Persistence Shared objects in CRUCIAL can be either *ephemeral* or *persistent*. Unless otherwise stated, shared objects are ephemeral and they only exist during the application lifetime. Once the application finishes, they are discarded. Ephemeral objects can be lost, e.g., in the event of a server failure in the DSO layer, since the cost of making them fault-tolerant outweighs the benefits of their short-term availability [14]. Nonetheless, it is also possible to make them persistent with the annotation `@Shared(persistent=true)`. In such a case, the annotated object outlives the application lifetime and is only removed from storage by an explicit call.

Table 1: Programming abstractions

ABSTRACTION	DESCRIPTION
`CloudThread`	Serverless functions are invoked like threads.
Shared objects	Linearizable (wait-free) distributed objects. `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, `Map`, ...
Synchronization objects	Shared objects providing primitives for thread synchronization (e.g., `CyclicBarrier`, `Semaphore`, `Future`).
`@Shared`	User-defined shared object. Methods are run on the DSO servers, allowing fine-grained updates and aggregates (`.add()`, `.update()`, `.merge()`, ...).
Data persistence	Long-lived shared objects are replicated. Use `@Shared(persistence=true)` to activate it.

Synchronization Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP) [15, 18]. To provide fine-grained coordination of cloud threads, CRUCIAL offers a number of primitives such as cyclic barriers and semaphores (see Table 1). These coordination primitives are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for serverless functions that is, as of today, non-existent.

3.1.1 Sample application

Listing 1 presents an application implemented with CRUCIAL. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. It draws a large number of

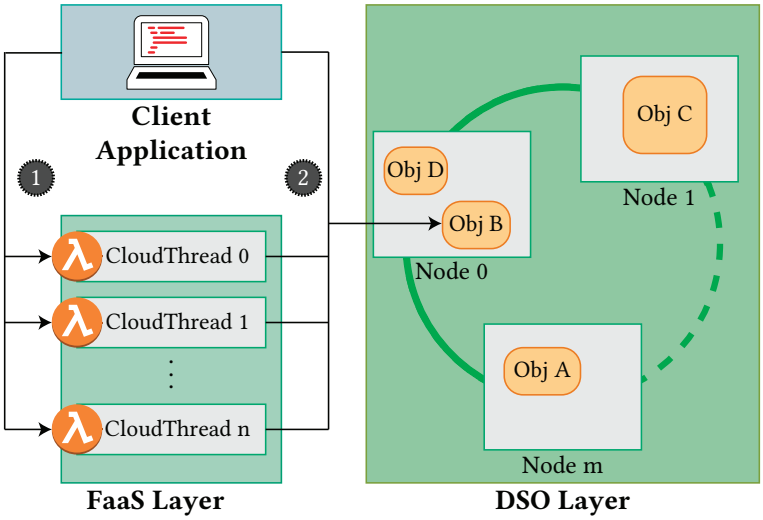

Figure 1: Overall architecture of CRUCIAL. A client application would run a set of threads in FaaS functions, and all the threads would have access to the same state (client included).

random points and computes how many fall in the circle enclosed by the unit square. The ratio of points falling in the circle converges with the number of trials toward $\pi/4$ (line 24).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1-18). To parallelize its execution, lines 22-23 run the fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 4). This counter maintains the total number of points falling into the circle, which serves to approximate π. It is updated by the threads concurrently using the `addAndGet` method (line 14).

3.2 Design

Figure 1 presents the overall architecture of CRUCIAL. In what follows, we detail its components and describe the lifecycle of an application in our system. The system encompasses three main components: 1) the DSO layer shared by all cloud thread instances; 2) the FaaS computing layer that runs the cloud threads (i.e., `CloudThread` instances); and 3) the client application. A client application differs from a regular JVM process on two aspects: threads are executed as serverless functions, and they access shared data using the DSO layer. In addition, CRUCIAL may use object storage (such as Amazon S3) to store the immutable input data of the application (not modeled in Figure 1).

3.2.1 The distributed object layer

In CRUCIAL, fine-grained updates to a data item are implemented as object methods. Internally, each object in the distributed object layer (for short, DSO) is uniquely identified by a reference. Given an object of type T, the reference to this object is (T, k), where k is either the field's name of the encompassing object or the value of the parameter *key* in the annotation `@Shared(key=k)`. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method. CRUCIAL constructs the DSO layer using consistent hashing [19], similarly to Cassandra [20]. Each storage node knows the full membership of the storage layer and thus the mapping from data to node. The location of a shared object o is then determined by hashing the reference (T, k) of o. This offers the following usual benefits: 1) no broadcast is necessary to locate an object; 2) disjoint-access parallelism [21] can be exploited; and 3) service interruption is minimal in the event of server addition and removal. The latter property is useful for persistent objects, as detailed next.

Persistence Objects marked as persistent are replicated *rf* (replication factor) times in the DSO layer. When a cloud thread accesses a persistent shared object, it contacts one of the server nodes via the proxy object. The operation is forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to caller. Notice that for ephemeral, non-persistent objects, *rf* is 1.

Consistency To help with the writing of stateful serverless applications, the DSO layer provides strong consistency. In particular, and as described in Section 3.1, objects shared with CRUCIAL are *linearizable* [17]. For persistent state, consistency across replicas is maintained with the help of state machine replication (SMR) [22]. To implement SMR, CRUCIAL relies on a variation of the view synchrony abstraction [23]. View synchrony also ensures consistency during membership changes. It is well-known that if a strongly-consistent shared object is updated very frequently by multiple processes, performance does not scale and CRUCIAL cannot help with it. Fortunately, we found that for multiple stateful applications, it is possible to achieve high performance with strong consistency. The study of other consistency models is planned in the upcoming Tasks T4.2 (M4-M34) and T4.3 (M9-M34).

Remote procedure CRUCIAL helps to alleviate perhaps one of the biggest downsides of FaaS platforms: its data-shipping architecture [18]. As functions are not network-addressable and run separate from data, applications are routinely left with no other choice but to "ship data to code". Fortunately, the DSO layer helps to resolve this design anti-pattern with minimal effort from the user side: it suffices to implement arbitrary computations as object methods. This feature is extremely useful for many applications that need to aggregate and combine small granules of data (e.g., machine learning tasks). As object methods are remotely executed on the DSO servers, applications can save significant communication resources. Without this property, each cloud function would need first to pull all the intermediate data from the remote storage service (e.g., S3) and then aggregate it locally (i.e., `AllReduce` operation). This would entail a communication cost of N^2 messages, where N is the number of functions. With CRUCIAL, however, this complexity reduces to $\mathcal{O}(N)$ messages. In particular, we exploited this feature in *k*-means clustering to calculate the final centroids from their partial updates.

3.2.2 Execution lifecycle

The execution lifecycle of a CRUCIAL application is similar to that of a multi-threaded Java application. Every time a `CloudThread` is started, a standard Java thread (i.e., instance of `java.lang.Thread`) is spawned in the client application with some extra logic. The basic role of this logic lies in calling a generic serverless function to execute the `Runnable` code attached to the `CloudThread`. During the execution of a cloud thread, each access to a shared object is mediated by a proxy. This proxy is created when a constructor is encountered in the code, and either the newly created object belongs to CRUCIAL's library, or it is tagged `@Shared`.

The Java thread remains blocked until the call to the serverless function terminates. Such behavior gives cloud threads the appearance of conventional threads; minimizing code changes and allowing the use of the `join()` method in the application's master thread to establish synchronization points (e.g., fork/join pattern). It must be noted, however, that as cloud functions cannot be canceled or paused, the analogy is not complete. If any failure occurs to the remote cloud function, the error is propagated back to the client application for further processing.

3.2.3 Fault tolerance

Fault tolerance in CRUCIAL is based on the disaggregation of the compute and storage layers. On the one hand, writes to the shared object layer can be made durable with the help of data replication. In

such a case, CRUCIAL tolerates the joint failure of up to $rf - 1$ servers.[1] On the other hand, CRUCIAL offers the same fault-tolerance semantics in the compute layer as the underlying FaaS platform. In AWS Lambda, this means that any failed cloud thread can be re-started and re-executed with the exact same input. Thanks to the cloud thread abstraction, CRUCIAL allows full control over the retry system. For instance, the user may configure how many retries are allowed and/or the time between them. If retries are permitted, the programmer should ensure that the re-execution is sound (e.g., it is idempotent). Fortunately, atomic writes in the DSO layer make this task easy to achieve. Considering the *k*-means example (as well as in other iterative algorithms), it simply consists of sharing an iteration counter. When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

3.3 Preliminary results

This section presents early results assessing that our approach is not only feasible but also desirable for certain types of applications, e.g., machine learning (ML). We first validate the general design of CRUCIAL with a series of micro-benchmarks. Next, we show that our system based on fine-grained updates to shared mutable data outperforms Spark at comparable cost in two instances of ML problems.

Setup All our experiments are conducted in Amazon Web Services (AWS), using the `us-east-1` region and a Virtual Private Cloud (VPC). Unless otherwise specified, we use `r5.2xlarge` EC2 instances for CRUCIAL's DSO nodes (usually, a single node is sufficient) and the maximum resource settings for AWS Lambda. Currently, the deployment of DSO layer of CRUCIAL is not automatic and has been done on a per-experiment basis. The provisioning of storage resources for serverless computing remains an open issue [15, 4], with just a couple of works appearing very recently in this area [14, 9].

3.3.1 Micro-benchmarks

For starters, we evaluate CRUCIAL's performance across a range of micro-benchmarks.

Latency Table 2 compares the latency to access a 1KB object sequentially in CRUCIAL, Redis, Infinispan and S3. Each function performs 30k operations and we display the average access latency. In this experiment, CRUCIAL exhibits a performance similar to other in-memory systems. In particular, the system is an order of magnitude faster than S3. This table also depicts the effect of data replication on the system. When replicating an object twice ($rf = 2$), the replicated state machine induces an additional round-trip that doubles the latency perceived at the client.

Table 2: Average latency comparison *1KB payload*

	PUT	GET
S3	34,868μs	23,072μs
Redis	232μs	229μs
Infinispan	228μs	207μs
CRUCIAL	231μs	229μs
CRUCIAL ($rf = 2$)	512μs	505μs

Throughput Figure 2a compares the performance of CRUCIAL and Redis for both simple and complex operations. In this experiment, 200 cloud threads access remotely 800 objects at random in closed loop. The experiment runs for 30s and we present the average performance. The storage layer consists of a two-node cluster for both CRUCIAL (with and without replication), and Redis (2 shards with no replicas).

The key observation in Figure 2a is that CRUCIAL is not affected by the complexity of the operation. For simple operations, Redis is 50% faster than CRUCIAL (without replication). On the other

[1] Synchronization objects (see Table 1) are not replicated. This is not an important issue due to their ephemeral nature.

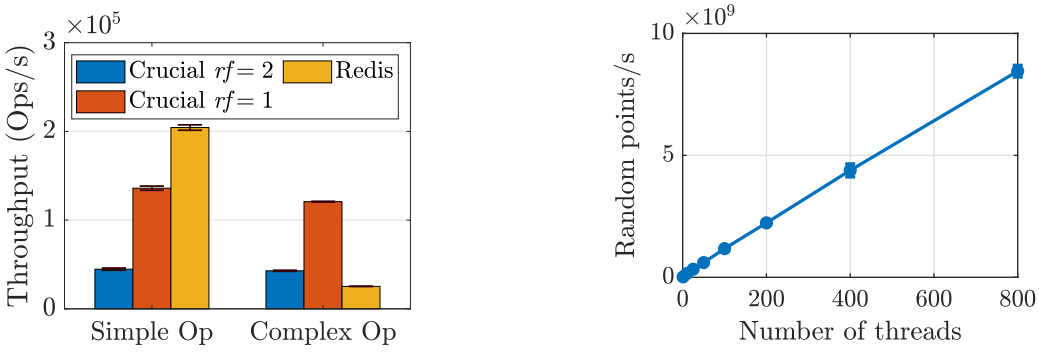

Figure 2: (a) Operations per second performed on CRUCIAL (with and without replication) and Redis. The simple operation is a multiplication of a value. The complex operation is 10k multiplications with that value. The experiment operates on a total of 800 different objects/keys concurrently. (b) Scalability of a Monte Carlo simulation to approximate π. CRUCIAL reaches 8.4 billion random points per second with 800 threads.

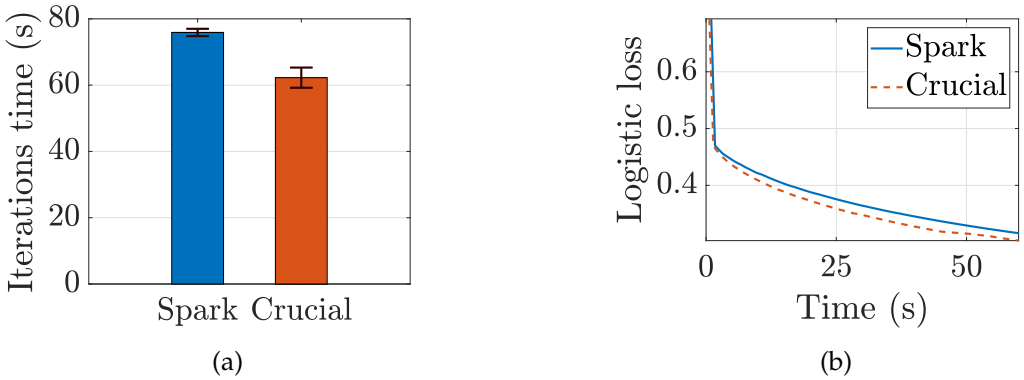

Figure 3: Comparison of Spark and CRUCIAL implementations of Logistic Regression. (a) shows the average completion time of the iteration phase (100 iterations). (b) shows a comparison of the performance of both systems.

hand, for complex operations, the performance of CRUCIAL is 70% better than Redis when replication is activated. Without data replication, CRUCIAL is almost five times faster. This large gap comes from the absence of disjoint-access parallelism in Redis. Indeed, calls to Lua scripts are executed sequentially in Redis, while CRUCIAL invokes the object methods in parallel.

Parallelism Our first application using CRUCIAL is the Monte Carlo simulation presented in Listing 1. This base algorithm is embarrassingly parallel, relying only on a single shared object (a counter). We run the simulation with 1 to 800 cloud threads and track the total number of points computed by them each second. The results presented in Figure 2b show that our system scales linearly and that it exhibits a 512x speedup with 800 threads.

3.3.2 Comparison with Spark

We compare CRUCIAL against Spark [24] using two machine learning algorithms: logistic regression and *k*-means. Both algorithms are iterative and share a modest amount of state that requires per-

iteration updates. They are a perfect fit to assess the efficiency of fine-grained updates in CRUCIAL against a current state-of-the-art solution.

Setup For this comparison, we provide equivalent CPU resources to both competitors. In detail, CRUCIAL experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1792MB and 2048MB of memory for logistic regression and *k*-means, respectively. These values are chosen to have the optimal performance at the lowest cost.[2] Spark experiments are run in an Amazon EMR cluster with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR's terminology), each having 8 cores. The Spark executors are configured to utilize the maximum resources possible on each node of the cluster.

Dataset As input data, both applications use a 100GB synthetic dataset generated with spark-perf [25]. The data comprises 55.6M elements. For logistic regression, each element is labeled and contains 100 numeric features. For *k*-means, each element corresponds to a 100-dimensional point. The dataset has been split into 80 equal-size partitions to ensure that all partitions are small enough to fit into the function's memory. Each partition has been stored as an independent file in Amazon S3.

Logistic regression We evaluate a CRUCIAL implementation of logistic regression against its equivalent counterpart in Spark's MLlib [26]: `LogisticRegressionWithSGD`. A key difference between the two implementations is the management of the shared state. At each iteration, Spark broadcasts the current weight coefficients, computes, and finally aggregates the sub-gradients in a MapReduce phase. In CRUCIAL, the weight coefficients are shared objects. At each iteration, a cloud thread retrieves the current weights, computes the sub-gradients, updates the shared objects, and synchronizes with the other threads. Once all the partial results are uploaded to the DSO layer, the weights are recomputed and the threads proceed to the next iteration.

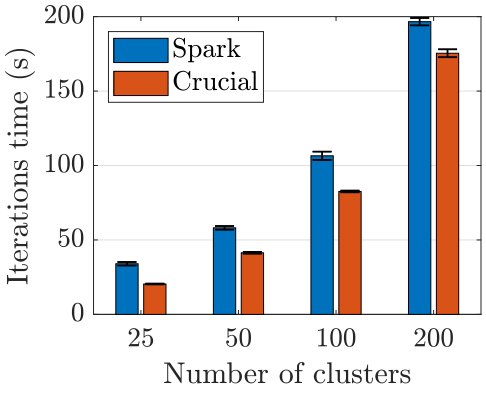

Figure 4: Average completion time of the iteration phase (10 iterations) of the *k*-means algorithm with varying number of clusters.

In Figure 3, we measure the running time of 100 iterations of the algorithm and the logistic loss after each iteration. Results show that the iterative phase is 18% faster in CRUCIAL (62.3s) than Spark (75.9s), and thus the algorithm converges faster[3]. This gain is explained by the fact that CRUCIAL aggregates and combines sub-gradients in the dedicated shared object layer. On the contrary, each iteration in Spark induces a reduce phase that is costly both in terms of communication and synchronization.

***k*-means** We now compare an implementation of *k*-means using CRUCIAL to the one available in Spark's MLlib. For both algorithms, centroids are initially at random positions. Figure 4 shows the completion time of 10 iterations of the clustering algorithm for different numbers of clusters. With $k = 25$, CRUCIAL completes the 10 iterations 40% faster (20.4s) than Spark (34s). The time gap is less noticeable with more clusters because the synchronization portion of each iteration is less representative as the number of clusters increases. That is, the iteration time becomes increasingly dominated

[2] Starting with a configuration of 1792MB, an AWS Lambda function has the equivalent to 1 full vCPU (`https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html`). Also, with this assigned memory, the function uses a full Elastic Network Interface (ENI) in the VPC.

[3] For Spark, Figure 3 does not consider the cluster provisioning time, nor the time to load and parse the dataset from S3.

Table 3: Monetary costs of the experiments

		Total time (s)	Total cost ($)	Iterations cost ($)
k-means ($k = 25$)	Spark	168	0.246	0.050
	CRUCIAL	87	0.244	0.057
k-means ($k = 200$)	Spark	330	0.484	0.288
	CRUCIAL	234	0.657	0.492
Logistic regression	Spark	192	0.282	0.111
	CRUCIAL	122	0.302	0.154

by computation. As in the logistic regression experiment, CRUCIAL benefits from computing centroids in the DSO layer, while Spark requires an expensive reduce phase at each iteration.

A note on costs Although one may argue that the programming simplicity of serverless computing justifies its higher cost [7], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

Table 3 details a cost comparison of Spark and CRUCIAL based on the above experiments. The first two columns list the time and cost of the entire experiments, including the time of loading and parsing the input data, but without considering provisioning times. The last column lists the costs that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we consider the pricing for on-demand instances and ignore AWS's free tier.

With the current pricing policy of AWS [27], the cost per second of the CRUCIAL setup is always higher than the Spark one: 0.25 and 0.28 cents per second for 1792MB and 2018MB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in the k-means clustering with $k = 200$, the cost of using CRUCIAL is logically higher. This difference is erased in experiments during which CRUCIAL is substantially faster than Spark (e.g., for k-means clustering with $k = 25$).

To give a proper picture of this cost comparison, there are two additional remarks to make here. First, the solution provided with CRUCIAL using 80 concurrent AWS Lambda functions employs a larger aggregated bandwidth from S3 than the solution with Spark. This reduces the cost difference between the two approaches. Secondly, CRUCIAL users only need to pay for the execution time of their functions, rather than the time the cluster remains active. This includes the bootstrapping of the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [28].[4]

4 Software

Our current prototype for serverless stateful applications is split into four distinct softwares: CRUCIAL, CRESON, the serverless executor service and INFINISPAN. CRUCIAL is built atop CRESON and the serverless executor service. CRESON implements a layer of distributed shared objects atop the INFINISPAN in-memory data grid.

Cloud threads are defined by implementing a `Runnable` and executed with the abstraction from Table 1. CRUCIAL uses AWS Lambda as computation engine for the cloud threads. Lambda functions are deployed with the help of the `lambda-maven-plugin`[5] and invoked through the AWS Java SDK. To gain control over the replay mechanism of the service, our prototype uses synchronous invocations

[4] Provisioning the 11-machine EMR cluster takes 2 minutes (not billed) and bootstrapping requires an extra 4 minutes. A CRUCIAL storage instance starts in 30 seconds.

[5] `https://github.com/SeanRoy/lambda-maven-plugin`

(`RequestResponse`). When an AWS Lambda function is invoked, it receives in the payload the name of the user-defined `Runnable` and a set of parameters to initialize it. We use the Java reflection API to instantiate the classes and provide them the initialization values. Before executing the user code, our generic function establishes the connection to the DSO layer. Since our prototype only accepts `Runnable`, the return payload is empty unless an error occurs. In case of error, the system interprets it and re-throws an exception.

Recently, we have started to modularize the FaaS code base into an independent serverless executor service. The service takes as input a Callable and executes it transparently either atop AWS Lambda or Kubernetes. Such a behavior is identical to the ExecutorService in the JDK and will simplify the portage of legacy application to serverless architectures.

The shared object (DSO) layer is written atop the Infinispan in-memory data grid [29] as a partial rewrite of the CRESON project [30]. The code of the client and the server of this layer weigh 2.5k and 9.2k SLOC, respectively. The prototype client of CRUCIAL includes a small library of shared objects and the proxies to access them. To wave proxies in the code of the client application and the functions, both are compiled with the help of AspectJ [31]. In the case of user-supplied shared objects, the aspects are applied to annotated instance fields (see Section 3.1). Such objects must be serializable and contain an empty constructor for marshalling purposes. For the servers to be able to manage user objects, a `.jar` package must be uploaded somewhere accessible by them (e.g., S3).

This package can be loaded dynamically, without having to restart the servers. The server is implemented using the interceptors API of Infinispan[6], which also allows the SMR logic. The approach follows the visitor pattern as commonly found in storage systems. Synchronization objects (e.g., barriers, futures, semaphores) follow the structure of their Java counterparts. These objects utilize Java's `synchronized` capabilities so that clients block on a remote connection, while the method execution on the server uses the combination of `wait()/notify()`. For instance, the barrier adopts the mechanism of a cyclic barrier, with an internal counter and a generation system.

4.1 Crucial

(Documentation) `https://github.com/danielBCN/crucial/README.md`

(Code) `https://github.com/danielBCN/crucial`[7]

4.2 Creson

(Documentation) `https://github.com/otrack/creson`

(Code) `https://github.com/otrack/creson/README.md`

4.3 Serverless Executor Service

(Documentation) `https://github.com/otrack/serverless-executor-service`

(Code) `https://github.com/otrack/serverless-executor-service/README.md`

4.4 Infinispan

(Documentation) `https://infinispan.org/documentation`

(Code) `https://github.com/infinispan`

[6] The interceptors API enables the execution of custom code in-between Infinispan's processing of data store operations.

[7] The CRUCIAL prototype is described in a scientific paper which was submitted to a conference. This repository is private due to double-blind review process of the conference.

5 State of the Art

5.1 Alternative programming models

Large datacenters hosting hundreds of thousands of commodity servers form the backbone of modern computing systems. Message passing is the native model for such infrastructures. This makes actor-based frameworks such as Akka [32] and microservices architectures popular for constructing distributed programs. In the high-performance computing world, MPI is a widely-employed programming interface [33]. MPI provides extensive messaging mechanisms such as unicast and broadcast as well as support for creating and managing remote processes in a distributed environment. The message passing model gives flexibility to the programmer by explicitly managing communication and state synchronization. On the other hand, it generally remains difficult to program with it and there is no incremental path to parallelize applications following this paradigm.

Distributed shared memory (DSM) systems such as Munin [34] and TreadMarks [35] implement the convenient shared-memory programming model over a cluster of machines. The programmer writes parallel programs with threads, synchronizing them with library routines that manipulates locks, barriers, and condition variables. Access to the shared memory is at the page level and fully transparent. This requires to map statically each shared variable to a specific shared segment at compile time. Furthermore, the approach ships data to code and thus needs complex protocols to arbitrate data races and false sharing problems.

OpenMP is an industry standard api for shared-memory programming on a single machine [36]. It allows the user to parllelize portion of her code using fork/join and doacross idioms. Most efforts to support OpenMP programs over a cluster of machines [37, 38] are based on software distributead shared memory. As a consequence, the resulting implementations suffer from the same limitations as DSM systems.

MapReduce [39] pioneered a model of cluster computing in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. In this programming model, the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention. While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. This includes two use cases for which MapReduce is deficient: iterative jobs and interactive analytics [24].

Spark focuses on distributed applications that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms [26], as well as interactive data analysis tools. Spark supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost [40].

5.2 Serverless architectures

Despite the increasing maturity and adoption of cloud technologies, programming a cloud application remains a complex error-prone task. In particular, managing the application state requires particular care to tolerate failures, scale to large amount of operations per second, and elastically adapt resources to demand. Besides that, it is desirable for application developers to have access to high-level interfaces, and manipulate complex data structures, while benefiting from the simplicity and safety associated with strong consistency guarantees.

Serverless computing is an emerging paradigm that pushes the principles of Platform-as-a-Service one step further. In this paradigm, the runtime is not started by the user, but already running at the service provider itself. The user ships only to the provider the functions to be executed. These functions are loaded in the platform and executed on-demand. A functions may access remote services

such as web services, storage backends or other serverless functions. This allows to solve complex tasks by decomposing them, as in classical service-oriented architectures. However, contrary to this last approach, programming time is reduced to the bare minimal: only the code that executes the actual logic of the application is required.

Typically, serveless computing platforms support multiple programming languages. For some target language, the platform requires to express user functions following a simple signature, e.g., `void execute(Stream<String> input, Context context)`, where `context` is an object that materializes the invocation context of the call. Once a serveless code is stored in the platform, it can be loaded dynamically on-demand. If the server is hot then this code is already in memory, reducing invocation time. To trigger the execution, the programmer may launch it by hand, or define some condition (e.g., a trigger at the storage level).

With the emergence of serverless computing, the cloud has found a paradigm that removes much of the complexity of its usage by abstracting away the provisioning of compute resources. This fairly new model was started by services such as Google BigQuery [1] and AWS Glue [2], and evolved into Function-as-a-Service (FaaS) computing platforms, such as AWS Lambda, Azure Functions, and Google's Cloud Functions, to name a few. With these platforms, a user-defined function and its dependencies are deployed to the cloud, where they are managed by the provider and executed on-demand.

Current practice shows that the FaaS model works well for applications that require a small amount of storage and memory due to the operational limits set by the cloud providers (see, for instance, AWS Lambda [3]). However, there are more limitations. While functions can initiate outgoing network connections, they cannot directly communicate between each other, and have little bandwidth compared to a regular virtual machine [4, 5]. This is because this model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier (e.g., uploading a photo to Amazon S3 [6]). Despite these constraints, recent works have shown how this model can be exploited to process and transform large amounts of data [7, 8, 9], encode videos [10], execute linear algebra tasks [11], and perform Monte Carlo simulations with large amounts of parallelism [12].

5.3 Dealing with state

All the major cloud providers offer a Function-as-a-Service platform (e.g., AWS Lambda, Azure Functions or Google Cloud Functions). These services have proven to be of little value to build stateful applications due to the lack of built-in support for mutable shared state and synchronization [15, 18]. As a consequence, serverless frameworks have had no other choice than developing their own mechanisms for this purpose.

A number of research works [7, 8, 11] opt to write shared data to slow, highly-scalable object storage. To hide high latency, these works have designed their systems to perform coarse-grained accesses. For instance, PyWren [7] replaces Amazon S3 with a Redis [41] cluster to sort 1TB. Nevertheless, the authors do not generalize this approach to other types of applications. A recent paper [9] takes this idea one step further by combining Redis with slow storage to scale the shuffling phase in MapReduce. Pocket [14] focuses on the scalability and the cost-efficiency to access ephemeral data for serverless analytics. Compared with CRUCIAL, none of the above works addresses the requirements for fast, fine-grained updates to shared mutable state necessary in stateful applications.

The problem of synchronizing cloud functions has followed a similar course. Current serverless systems do not provide means for fine-grained coordination of multiple functions [15, 18], but only via largely (uncoordinated) embarrassing parallelism [7, 8, 11, 10]. All these works mainly differ by the way they synchronize the map operator. While some of them use storage [7, 8, 11], other systems, such as ExCamera [10], have implemented its own notification systems using a VM-based rendezvous server. CRUCIAL, however, provides a broad suite of synchronization primitives (e.g., cyclic barriers, semaphores, futures) that permit fine-grained coordination, thereby going beyond the state of the art in that domain.

FaaS orchestration services may also be used for coordination purposes in stateful serverless applications. Services like AWS Step Functions, Azure Durable Functions, and OpenWhisk Composer can orchestrate function workflows using state machines. But as underlined recently [42], they are not designed for highly-parallel concurrent tasks and they show considerable overheads in that case.

Mutable shared state can be abstracted at different levels. Since CRUCIAL targets the simplicity of serverless computing for general stateful applications, we chose to represent state as objects. This simplifies the programming of fine-grained updates and/or aggregates by packing them in appropriate methods. To meet the latency and throughput requirements imposed by FaaS computing, we built the distributed shared object layer on top of the Infinispan in-memory data store [29]. Other systems available in the cloud, such as Memcached [43] or Redis [41], might be an alternative. Redis provides some data types (e.g., strings and lists) and support for server-side Lua scripting. Unfortunately, Memcached abstractions are too low-level for our context and building a distributed object-oriented model with strong consistency and data replication on top of Redis is far from being trivial. Furthermore, while CRUCIAL hides the latency of complex operations and offers parallelism, the single-threaded model of Redis for executing Lua scripts is not efficient in this context.

6 Exploratory work

Exploratory work is an important part of a European-funded RIA because it investigates and develops ideas that advance the state of the art both in research and industry. This section explains the exploratory work that has been conducted during the first six months. It also provides indications on how to decide whether this work will make it into a future version of CRUCIAL.

The exploratory work prepares Tasks T4.2 (M4-M34), T4.3 (M9-34) and T4.4 (M7-34), as well as the upcoming Deliverables D4.2 and D4.3 in CloudButton. This work is risky in the sense that not all of it will become part of the reference architecture. Such risk-taking is a necessary part of all successful research. Success of this work is to be measured not on how much of the work becomes part of the reference architecture, but on whether sufficiently innovative exploration is done and on whether the reference architecture itself is sufficiently innovative.

6.1 T4.2 - Degradable objects

In a distributed system, data is replicated for availability and to boost performance (typically, with more read replicas). When replicated data is mutable, it is necessary to maintain consistency with the help of a concurrency control mechanism. Due to the CAP and FLP impossibility results [44, 45], orchestrating data replicas is notably difficult and moreover subject to conflicting requirements. On the one hand, strong consistency maintains the application's sequential invariants and is well understood. On the other hand, performance and scalability suggest to use of a weaker consistency criterion, yet this requires considerable programming skills. A key challenge is thus to find a good balance between the programming model of the target distributed application, and its deployment constraints and performance requirements.

To reconcile programming model and data consistency, Task T4.2 of the CloudButton project investigates the notion of degradable object. A degradable object is a mutable shared data type whose behavior varies to match the requirements of an application. More precisely, a degradable object is a hierarchy of object types all having the same signature, but with varying pre- and post-conditions for their operations and that abide by different consistency criteria. Each level of this hierarchy is called a degradation level. The key principle is that the degradation level L+1 requires less synchrony to implement than the level L. Thus, it is more efficient and more scalable, but also less convenient to program with.

The programmer specifies the degradation level to use according both to the invariants of the application and its performance requirements. Finding the appropriate level for a given application pattern is an iterative process. At first glance, a programmer may use strong consistency, then later

refines her choices based on the fact that some interleavings and/or inconsistencies are acceptable. Our key insight here is that this iterative process will offer a principled and pedagogical approach to understand and use (weak to strong) data consistency in distributed applications.

A preliminary work in Task T4.2 has been conducted on the definition and implementations of degradable objects. Our effort have been conducted so far on three fronts.

- First, we are collaborating with the H2020 LightKone project [46] on introducing a new communication primitive in AntidoteDB [47]. AntidoteDB is a distributed database of conflict-free replicated data types (CRDTs). In the traditional CRDT approach, operations that are mutating a replica are executed in the background, outside the critical path. Their side effect (aka., the effector [48]) is then propagated eventually to all the replicas, for instance an epidemic protocol. Our new primitive will maintain this behavior, but will also offer better properties if needed (e.g., on the delivery order of effectors). The end goal is to allow some operations to execute under stricter consistency conditions than strong eventual consistency, the default criteria of CRDTs [49].
- Our second effort is on the specification and definition of degradable objects. We investigate the link between the specification of a sequential data type and the need for process synchronization. Typically, process synchronization is measured by the consensus power of a given data type. The consensus power is the largest number of processes that are able to solve consensus with this data type and registers. Starting from base shared objects, we are investigating how consistency degradation reduces the consensus power.
- The consensus power is formulated with linearizable objects, that is, in the classical shared memory model. As a consequence, this hierarchy does not fully capture the need for synchronization in a distributed message-passing system. To close this gap, we investigate alternative definitions to characterize process synchronization. In particular, our investigation covers the k-set agreement hierarchy and the link between failure detectors and quorums of data replicas [50].

6.2 T4.3 - Just-right synchronization

The classical way of maintaining shared objects strongly consistent is state-machine replication (SMR) [22]. In SMR, an object is defined by a deterministic state machine, and each replica maintains its own local copy of the machine. An SMR protocol coordinates the execution of commands at the replica, ensuring that they stay in sync. This requires to execute a sequence of consensus instances each agreeing on the next state-machine command. The resulting system is linearizable, providing an illusion that each command executes atomically throughout the system.

It is well-known that the above classical SMR scheme limits scalability. However, strong consistency is necessary to help transitioning legacy code from shared-memory to serverless architecture. Moreover, as underlined in Section 6.1, its semantics is well-understood by everyday programmers. To sidestep the performance problem of strong consistency, we have performed an exploratory work on (i) improving the scalability of SMR with leaderless consensus, and (ii) designing an efficient atomic multicast protocol to deal with partial replication. These two investigations are further detailed below.

6.2.1 Leaderless consensus

To date, SMR protocols do not scale, that is when more replicas are added to the system, the performance of the replicated service degrades. This situation results from the conjunction of several pitfalls:

- First of all, a large spectrum of protocols, e.g., Paxos [51], Raft [52] or Zab [53], funnel commands through a leader replica. This approach increases latency for clients far away from the leader and decreases availability because if the leader fails, the system halts to elect a new one.

To mitigate these drawbacks, leaderless approaches [54, 55, 56] allow each replica to contact a quorum of its peers to execute a command.
- A second concern is that many standard solutions rely on large quorums to make progress. For instance, in a system of n replicas, Fast Paxos [57] accesses at least $\frac{2n}{3}$ replicas, EPaxos [55] $\frac{3n}{4}$, and Mencius [54] contacts them all. Large quorums harms system reliability and scalability because more replicas have to participate to the ordering of each command.
- A last concern is the communication delay to execute a command. To minimize service latency SMR protocols should leverage non-conflicting commands, that is commands which are not concurrent to any other non-commuting command. These commands are frequent in distributed applications [58, 59] and can execute in a single round-trip [60].

Refining the above observations, we have introduced a set of desirable requirements for SMR: Reliability, Optimal Latency and Leaderlessness, We have shown that attaining all the ROLL properties is subject to a trade-off between fault-tolerance and scalability. More specifically, in a system of n processes, the ROLL theorem states that every leaderless SMR protocol that tolerates f failures must contact at least $(n - \frac{(n-f)}{2})$ processes to execute a command in a single round-trip.

Simultaneous failures and/or asynchrony periods are however a rare event. Leveraging this fact, we have proposed a novel SMR protocol which, based on the ROLL theorem, is optimal. In particular, this new leaderless SMR protocol offers two distinguishable unique features.
- First, it executes a command by contacting the closest $\lfloor \frac{n}{2} \rfloor + f$ processes. For small values of f, this implies that the protocol scales.
- The protocol applies commands using a fast path that completes after one round trip, or a slow path, which completes after two round trips. We introduce a new condition that allows commands to take the fast path even in the presence of conflicts. In particular, when $f = 1$, the protocol always takes the fast path.

We have experimentally compared our protocol against Paxos [51], EPaxos [55] and Mencius [54] on Google Cloud Platform. Our preliminary results show that our approach consistently outperforms these protocols. In particular, the protocol scales when f is small in the sense that adding more nodes close to the clients improves latency.

6.2.2 Atomic multicast

Atomic multicast is a communications primitive that allows a group of processes to receive messages in an acyclic delivery order. This primitive is a useful building block for distributed storage systems that enforce strong consistency properties. As an example, atomic multicast is used in Infinispan to implement distributed transactions. The main difference with atomic broadcast, which serves a similar purpose, is that a message can be addressed to a subset of the processes. To be scalable, atomic multicast protocols must be genuine, that is only the destination group of a message should be involved in its ordering.

The standard fault-tolerant genuine solution layers Skeen's multicast protocol on top of Paxos to replicate each destination group. Recent improvements decrease the latency of this standard solution by adding a parallel speculative execution path. Under normal operation, the standard protocol can deliver multicast message in 6 communication delays and such an optimized version in 4 communication delays.

Standard protocols employ the Paxos consensus protocol as a blackbox. Departing from this traditional way of guaranteeing fault-tolerance, we propose a new solution that weaves Paxos together with Skeen's multicast. The resulting multicast protocol embeds its own replication logic, enabling message ordering and delivery in 3 communication delays under normal operation.

Our protocol offers better theoretical performance. We have experimentally assessed that such characteristics pay-off in practice. We implemented our protocol in the same framework as Skeen's and its optimized variation and conducted a comparative performance analysis of the three protocols. Our protocol offer better latency than prior works (up to 2x faster than the optimized Skeen

variation). It also sustains a much higher number of concurrent client requests, thanks to its lower message complexity.

6.3 T4.4 - In-memory data storage

6.3.1 Ahead-of-time compilation

One of the concerns of using Java in high-density environments is the overhead of the Java Virtual Machine (JVM) both in terms of memory usage as well as in startup and warming-up time. Most of the blame for this doesn't actually lie in the JVM itself, which is still one of the best available optimizing virtual machines, but in the typical dynamic approach of many development frameworks which rely on runtime class reflection, annotation scanning and bytecode enhancements. However, the overhead of the JVM can still be drastically reduced by using ahead-of-time compilation (AOT), where Java source code can be directly compiled to machine code. Oracle has recently released the GraalVM project, which, among other things, delivers a "native-image" tool which generates a native binary from a Java application which does not require a JVM at runtime. This kind of native binary is ideal for applications which need very fast startup time with low memory overhead, which is typical in short-lived execution scenarios like FaaS. Real-world examples have demonstrated a 100-fold speedup in startup time for a microservice-style application (from 9 seconds to less than 1/10 of a second) and a 10-fold reduction in RSS (Resident Set Size) memory usage (from 250MB to 25MB). Because the native-image tool imposes some limitations on the kind of code that can be compiled and executed, traditional Java applications and libraries need to be altered. The Infinispan team has already implement the first set of changes to allow the Hot Rod client to be compiled ahead-of-time and more work is happening to be enable the same to be done with the embedded and server variants.

6.3.2 Support for non-volatile memory

Although processing occurs in main memory, big data stores such as Infinispan maintain the authoritative version of data on secondary storage (typically, SSD and disk). The existence of two versions of the data, one on-disk and another in-memory, poses several core problems. First, the slow access time of secondary storage represents a serious performance bottleneck that necessitates to persist data asynchronously at the expend of durability. Furthermore, starting and warming up data store, e.g., to recover after a failure, takes a very long time. A striking example is the September 2010 Facebook outage [61], in which the whole system was unavailable for 2.5 hours due to recovery. Moreover, the two representations of data have to be mutually consistent. This requires complex mechanisms to sustain the massively parallel access to the store, while minimizing the impact of a failure. At run-time, these mechanisms translate into a high overhead. For instance, some recent investigations show that Apache Spark is up to 6× faster without data persistence [62].

A key technology [63] that has the potential to remove the dual representation and greatly improve performance is the non-volatile byte-addressable memory (NVRAM). NVRAM is an emerging technology that combines the best features of traditional RAM and persistent storage. It is persistent upon power loss, provides fast and fine-granular access to data, and promises low latency (orders of magnitude faster than flash memory). Using NVRAM, the data may thus directly persist in memory. This removes the requirement of a secondary storage medium for persistence and allows the developer to focus on in-memory performance.

However, leveraging NVRAM requires to tackle several core challenges. A first challenge comes from the that algorithms managing the in-memory representation of data are not ready for persistent memory. A base example is an atomic map that serves key-value store operations, backed by several log-structured merge-trees for indexing, as found in many storage systems. In a nutshell, these algorithms are not designed to recover a consistent state after a failure. Upon recovery, threads may have to deal with a mix of data that was stored in persistent memory, while other parts of their state was not (e.g., processor cache, memory controller). Therefore, the state of the system may not be identical

before and after the failure. A related concern is that many key data structures in a big data store support concurrent accesses. Offering data persistent should not come at the price of performance by removing this parallelism A third challenge comes from the applicability of the approach as a whole.

In the context of Task T4.4, we have started to work on the introduction of NVRAM in Java-based big data stores. This work is currently at an early exploration stage, and we are investigating new approaches to offer data persistence in Java.

7 Conclusion

This document presents CRUCIAL, our initial prototype for stateful serverless computation. CRUCIAL allow to program highly-concurrent stateful applications on top of a Function-as-a-Service platform. It is built using an efficient disaggregated in-memory data store, and it can be used to construct demanding serverless applications that require fine-grained support for mutable state and synchronization. Early results show that CRUCIAL achieves superior or comparable performance to Spark for two common machine learning algorithms. In both cases, less than 3% of our code differs from a conventional solution using plain old Java objects.

References

[1] Google, "Google bigquery." https://cloud.google.com/bigquery/, 2019.

[2] Amazon, "Amazon glue." https://aws.amazon.com/glue/, 2019.

[3] Amazon, "Aws lambda limits." https://docs.aws.amazon.com/lambda/latest/dg/limits.html/, 2019.

[4] J. Carreira, P. Fonseca, A. Tumanov, A. M. Zhang, and R. Katz, "A case for serverless machine learning," in Workshop on Systems for ML and Open Source Software at NeurIPS, 2018.

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in 2018 USENIX Annual Technical Conference (USENIX ATC 18), (Boston, MA), pp. 133–146, USENIX Association, 2018.

[6] Amazon, "Amazon simple storage service." https://aws.amazon.com/s3, 2019.

[7] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, SoCC'17, 2017.

[8] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in Proceedings of the 19th International Middleware Conference Industry, Middleware '18, (New York, NY, USA), pp. 1–8, ACM, 2018.

[9] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), (Boston, MA), pp. 193–206, USENIX Association, 2019.

[10] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17), 2017.

[11] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," CoRR, vol. abs/1810.09679, 2018.

[12] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," CoRR, vol. abs/1710.08460, 2017.

[13] S. Lloyd, "Least squares quantization in pcm," IEEE Transactions on Information Theory, vol. 28, pp. 129–137, March 1982.

[14] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), (Carlsbad, CA), pp. 427–444, USENIX Association, 2018.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[16] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," Proc. VLDB Endow., vol. 12, pp. 624–638, Feb. 2019.

[17] N. A. Lynch, Distributed Algorithms. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[18] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings, 2019.

[19] 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, STOC, 1997.

[20] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," SIGOPS Oper. Syst. Rev., vol. 44, Apr. 2010.

[21] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'94, pp. 151–160, 1994.

[22] 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.

[23] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," ACM Comput. Surv., vol. 33, no. 4, pp. 427–469, 2001.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), (San Jose, CA), pp. 15–28, USENIX, 2012.

[25] Databricks, "spark-perf." `https://github.com/databricks/spark-perf`, 2014.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," Journal of Machine Learning Research, vol. 17, no. 34, pp. 1–7, 2016.

[27] Amazon, "Aws lambda pricing." `https://aws.amazon.com/lambda/pricing/`, 2019.

[28] H. Wang, D. Niu, and B. Li, "Distributed machine learning with a serverless architecture," in IEEE Conference on Computer Communications, INFOCOM 2019, 2019.

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

[30] P. Sutra, E. Riviere, C. Cotes, M. Sánchez-Artigas, P. García-López, E. Bernard, W. Burns, and G. Zamarreno, "CRESON: callable and replicated shared objects over nosql," in 37th IEEE International Conference on Distributed Computing Systems, ICDCS'17, 2017.

[31] 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, ECOOP, 2001.

[32] M. Gupta, Akka Essentials. Community experience distilled, Packt Publishing, 2012.

[33] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI-The Complete Reference, Volume 1: The MPI Core. Cambridge, MA, USA: MIT Press, 2nd. (revised) ed., 1998.

[34] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," SIGPLAN Not., vol. 25, pp. 168–176, Feb. 1990.

[35] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," in Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, WTEC'94, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 1994.

[36] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," IEEE Comput. Sci. Eng., vol. 5, pp. 46–55, Jan. 1998.

[37] L. Huang, B. Chapman, and Z. Liu, "Towards a more efficient implementation of openmp for clusters via translation to global arrays," Parallel Comput., vol. 31, pp. 1114–1139, Oct. 2005.

[38] H. Lu, Y. C. Hu, and W. Zwaenepoel, "Openmp on networks of workstations," in Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, (Washington, DC, USA), pp. 1–15, IEEE Computer Society, 1998.

[39] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Commun. ACM, vol. 51, pp. 107–113, Jan. 2008.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[41] "Redis." `https://redis.io/`, 2009.

[42] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, and D. Arroyo Pinto, "Comparison of faas orchestration systems," in 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 148–153, IEEE, 2018.

[43] B. Fitzpatrick, "Distributed caching with memcached," Linux J., vol. 2004, pp. 5–, Aug. 2004.

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

[45] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," J. ACM, vol. 32, pp. 374–382, Apr. 1985.

[46] LightKone. `https://www.lightkone.eu`.

[47] AntidoteDB. `https://www.antidotedb.eu`.

[48] M. Shapiro and P. Sutra, "Database consistency models," in Encyclopedia of Big Data Technologies., Springer International Publishing, 2019.

[49] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Convergent and commutative replicated data types," Bulletin of the EATCS, vol. 104, pp. 67–88, 2011.

[50] F. C. Freiling, R. Guerraoui, and P. Kuznetsov, "The failure detector abstraction," ACM Comput. Surv., vol. 43, pp. 9:1–9:40, Feb. 2011.

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

[52] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in USENIX Annual Technical Conference (USENIX ATC), pp. 305–320, 2014.

[53] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 245–256, 2011.

[54] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for wans," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 369–384, 2008.

[55] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in ACM Symposium on Operating Systems Principles (SOSP), pp. 358–372, 2013.

[56] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran, "Speeding up consensus by chasing fast decisions," in IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 49–60, 2017.

[57] L. Lamport, "Fast Paxos," Distributed Computing, vol. 19, pp. 79–103, 2006.

[58] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 335–350, 2006.

[59] 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," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 251–264, 2012.

[60] L. Lamport, "Generalized consensus and Paxos," Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[61] R. Johnson, "Facebook engineering.september 2010 outage, [online; accessed june 2019]."

[62] K. Ousterhout, "Making sense of spark performance [online, accessed june 2019]."

[63] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches," IEEE Transactions on Parallel and Distributed Systems, vol. 26, pp. 1524–1537, June 2015.