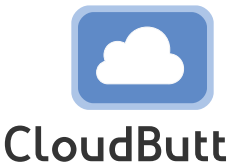


HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D4.2 Specification and partial support for degradable objects

Due date of deliverable: 30-06-2020
Actual submission date: 31-07-2020

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

Summary of the document

Document Type	Report
Dissemination level	Public
State	v2.0
Number of pages	41
WP/Task related to this document	WP4 / All tasks
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_D4.2_Public.pdf
Abstract	This document describes in full details CRUCIAL, a complex distributed system to program efficient (both stateful and stateless) serverless applications.
Keywords	serverless, FaaS, distributed storage, big data, data analytics, machine learning

History of changes

Version	Date	Author	Summary of changes
0.1	26-06-2020	Pierre Sutra	Initial version
0.2	28-06-2020	Pierre Sutra	Sections 1-4 added
0.3	29-06-2020	Pierre Sutra	Section 5 added
0.4	07-07-2020	Pierre Sutra	Section 6 added
1.0	08-07-2020	Pierre Sutra	Complete draft for internal review
1.2	15-07-2020	Tristan Tarrant	Edits section 6
2.0	23-07-2020	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 greatly simplifies the use of cloud resources. In particular, Function-as-a-Service (FaaS) enables programmers to develop applications as individual functions that can run and scale independently. The CloudButton project aims at leveraging this new infrastructure to bring closer to the end user the vast amount of data and computing power available in the cloud.

In this document, we present CRUCIAL, a system to program and execute efficient applications that target serverless platforms. The programming model of CRUCIAL keeps the simplicity of serverless computing and allows to write effortlessly parallel code for this new environment. Our system is structured into a compute and a storage tier, while providing a convenient user-facing library to the programmer. The compute tier runs atop a FaaS platform, while storage consists of an efficient in-memory distributed storage layer. CRUCIAL is built upon the key insight that serverless computing resembles to concurrent programming at the scale of the cloud. As a consequence, a distributed shared memory layer is the right answer to the need for fine-grained state management and coordination in serverless.

We validate CRUCIAL with the help of micro-benchmarks and various data analytics applications. To update shared data and synchronize cloud functions, CRUCIAL has better performance than alternative solutions. We show that it also simplifies the writing of parallel tasks (e.g., Monte Carlo) and ML algorithms (such as k-means clustering and logistic regression). Compared to Apache Spark, it obtains close or better performance at similar cost. We also use CRUCIAL to port a state-of-the-art multi-threaded ML library to serverless, as well as classical shell scripts. CRUCIAL brings elasticity and on-demand capabilities to these traditional single-machine programs. The performance of the serverless versions of these two programs are on par with the one offered by a dedicated cluster of high-end servers. A key feature of CRUCIAL is that it also allows to port programs to serverless in a few lines of code. All the above applications were ported while modifying less than 4% of the original code base.

This document describes in full length the CRUCIAL prototype, its programming API and the underlying distributed storage and compute tiers. It also details upcoming features (e.g., NVRAM-backed data persistence, leaderless consensus and consistency degradation) which will be added to future releases during the second half of the CloudButton project.

2 Introduction

Serverless computing is a paradigm that removes much of the cloud's usage complexity by abstracting away the provisioning of compute resources. This fairly new model was started by services such as Google BigQuery [93] and AWS Glue [13], and evolved into Function-as-a-Service (FaaS) computing platforms, such as AWS Lambda and Google Cloud Functions. In these services, a user-defined function and its dependencies are deployed to the cloud, where they are managed by the provider and executed on demand at scale.

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 [11]). 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 [25, 119]. 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 file to Amazon S3 [10]). Despite these constraints, recent works have shown how this model can be exploited to process and transform large amounts of data [63, 98, 101], encode videos [38], execute linear algebra tasks [103], and perform Monte Carlo simulations with large amounts of parallelism [57].

The above research projects, such as PyWren [63, 101] and ExCamera [38], 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 distributed stateful computations such as machine learning (ML) algorithms. Just an imperative implementation of *k*-means [81] 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 and numpywren [103], 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 [74], some recent works use faster storage solutions. This has been the path taken by Locus [98], 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 [25, 64], 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 orchestration services (such as AWS Step Functions [12] or OpenWhisk Composer [40]) offer limited capabilities to coordinate serverless functions [44, 64]. They have no abstraction to signal a function when a condition is fulfilled, or for multiple functions to synchronize, e.g., in order to guarantee data consistency, or to ensure joint progress to the next stage of computation. Of course, such fine-grained coordination should be also low-latency to not significantly slow down the application. Existing stand-alone notification services, such as AWS SNS [22] and AWS SQS [45], add significant latency, sometimes hundreds of milliseconds. This lack of efficient cloud coordination tools means that each serverless framework needs to develop its own mechanisms. For instance, PyWren enforces the synchronization of map and reduce stages through object storage, while ExCamera has built a notification system using a long-running VM-based rendezvous server. As of today, there is no general way to let multiple functions synchronize via abstractions hand-crafted by users, so that fine-grained coordination can be truly achieved.

2.1 Contributions

To overcome the aforementioned issues, we propose CRUCIAL, a system for the development of efficient (both stateful and stateless) serverless applications. To simplify the writing of an application, CRUCIAL provides a thread abstraction that maps a thread to the invocation of a serverless function: the *cloud thread*. This abstraction can be extended to build task management systems with serverless thread pools. 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 cloud 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 through replication. To ensure the consistency of replicas, CRUCIAL uses state machine replication (SMR), so that any acknowledged write can survive failures.

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.

Our evaluation shows that *(i)* 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; *(ii)* CRUCIAL demonstrates its ability to scale traditional parallel jobs to hundreds of workers with its simple code abstractions (e.g., Monte Carlo simulation and Mandelbrot computation); and *(iii)* the DSO layer is better suited for the fine-grained management and interaction with high-performance state than other systems like Redis.

With more details, the present work makes the following contributions:

- We provide the first concrete evidence that stateful applications with fine-grained data sharing can be efficiently built using stateless serverless functions and a disaggregated shared object layer.
- We design CRUCIAL, a system for the development and execution of serverless stateful distributed applications. CRUCIAL supports fine-grained semantics for both mutable state and coordination. Moreover, the programming model of CRUCIAL keeps the simplicity offered by current serverless architectures. These properties make CRUCIAL a great tool to easily move multi-threaded applications to the cloud.
- CRUCIAL is suited for many stateful distributed applications such as traditional parallel computations, machine learning algorithms, and complex concurrency situations. We show the scalability of a Monte Carlo simulation and a Mandelbrot computation with CRUCIAL over on demand serverless resources. Using extensive evaluation of *k*-means and logistic regression over a 100 GB dataset, we show that CRUCIAL can lead to $18-40\%$ performance improvement over Spark running on dedicated instances at similar cost. CRUCIAL is also within 8% of the completion time of the Santa Claus problem running on a local machine.
- Using CRUCIAL, we port to serverless part of Smile [79], a state-of-the-art multi-threaded ML library. The portage impacts less than 4% of the original code base. It brings elasticity and on-demand capabilities to a traditional single-machine program. Its performance are on par with

a dedicated high-end server: using a random forest classification algorithm, the portage with 200 cloud threads is up to 30% faster than a 4-CPU 160-threads dedicated server solution.

- Leveraging CRUCIAL, we implement the serverless shell. The serverless shell brings the massive copmutation power of the cloud to regular shell scripts. We validate the interest of this new software with the Common Crawl data set, a very large data set (PBs) freely available on the Internet. Our results show that CRUCIAL offers in a few lines of code, on-demand and instantaneously the performance of a dedicated cluster of high-end servers.

Complementary to the above works, we also investigated several innovative ideas to improve the CRUCIAL prototype. This includes multiple research results on data distribution, replication and persistence. These works appeared (or are being submitted) to top tier conferences and journals in their respective domains [14, 19, 34, 42, 48, 70, 112].

Outline The remaining of this document is structured as follows: We present the CRUCIAL prototype 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 that was conducted in WP4 during this first half of the CloudButton project. Part of this work will be included in future releases of CRUCIAL. Section 7 concludes this document.

3 Serverless programming with CRUCIAL

CRUCIAL is a powerful framework to create and execute complex serverless tasks atop a Function-as-a-Service (FaaS) architecture. This section presents the programming model and internals of our system, then evaluate its performance for multiple (big) data analytics applications.

3.1 Programming model

The programming model of CRUCIAL is object-based and can be integrated with any concurrent object-oriented language. As Java is the programming language supported in our implementation, the following description considers its jargon.[1]

Overall, 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 the developer that are detailed hereafter.

Cloud threads A `CloudThread` is the smallest unit of computation in CRUCIAL. Semantically, this class is equivalent to a `Thread` in conventional parallel computing. To write an application, each task is defined as a `Runnable` and passed to a `CloudThread` that executes it. This class hides the execution details of the cloud function in the FaaS platform to the developer.

Serverless executor service Both `Runnable` and `Callable` may run in the cloud using the serverless executor service (`ServerlessExecutorService`). This class implements the `ExecutorService` interface, allowing the submission of both individual tasks and fork-join parallel constructs (`invokeAll`). The full expressiveness of the original JDK interface is retained. Moreover, this executor also includes a distributed parallel *for* to run n iterations of a loop across m workers. To use this feature, the user specifies the in-loop code (through a functional interface), the boundaries for the iteration index, and the number of workers m. We have also implemented a `HybridExecutorService` that combines local threads and remote serverless functions. This hybrid version is transparent from the application point-of-view: `Callable` tasks are submitted to the `HybridExecutorService`, which decides whether to run them as local threads or remote serverless functions using a naive approach depending on

[1] The programming model of CRUCIAL is also recalled in Deliverable D5.2 [97, Section 4].

the current local load: if the local thread pool queue is not empty, tasks are scheduled to be run as serverless functions.

State handling CRUCIAL includes a library of base shared objects to support mutable shared data across cloud threads. The library consists of common objects such as integers, counters, maps, lists and arrays. These objects are *wait-free* and *linearizable* [82]. 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 a field declaration with the `@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)`. Our framework supports distributed references, permitting a reference to cross the boundaries of a cloud thread. This feature helps to preserve the simplicity of multi-threaded programming in CRUCIAL.

Data Persistence Shared objects in CRUCIAL can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and 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 [74]. Nonetheless, it is also possible to persist them using annotation `@Shared(persistent=true)`. Persistent objects outlive the application lifetime and are only removed from storage by an explicit call.

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.

Table 1: Programming abstractions

Synchronization Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP) [51, 64]. To provide fine-grained coordination of cloud threads, CRUCIAL offers a number of primitives such as cyclic barriers and semaphores. These coordination primitives are semantically equivalent to those in the `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 applications

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 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 25).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1–17). To parallelize its execution, lines 23–24 run a fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 5). 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 15).

The previous fork-join pattern can also be implemented using the `ServerlessExecutorService`. In this case, we simply replace lines 19–24 in Listing 1 with the content of Listing 2.

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

Listing 1: Monte Carlo simulation to approximate π.

```
1 ServerlessExecutorService se = new ServerlessExecutorService();
2 List<Callable> tasks = IntStream.range(0, N_THREADS).mapToObj(i -> Executors.callable(new
      PiEstimator())).collect(Collectors.toList());
3 se.invokeAll(tasks);
```

Listing 2: Using the `ServerlessExecutorService` to perform a Monte Carlo simulation.

```
1 public class Mandelbrot implements Serializable {
2   @Shared(key = "mandelbrotImage")
3   private MandelbrotImage image = new MandelbrotImage();
4
5   private static int[] computeMandelbrot(int row, int width, int height, int maxIters) {...}
6
7   private void doMandelbrot() {
8     image.init(COLUMNS, ROWS);
9     ServerlessExecutorService se = new ServerlessExecutorService();
10    se.invokeIterativeTask((row) -> image.setRowColor(row, computeMandelbrot(row, COLUMNS, ROWS,
       MAX_INTERNAL_ITERATIONS)), N_TASKS, 0, ROWS);
11    se.shutdown();
12  }
13 }
```

Listing 3: Mandelbrot set computation in a distributed parallel *for*.

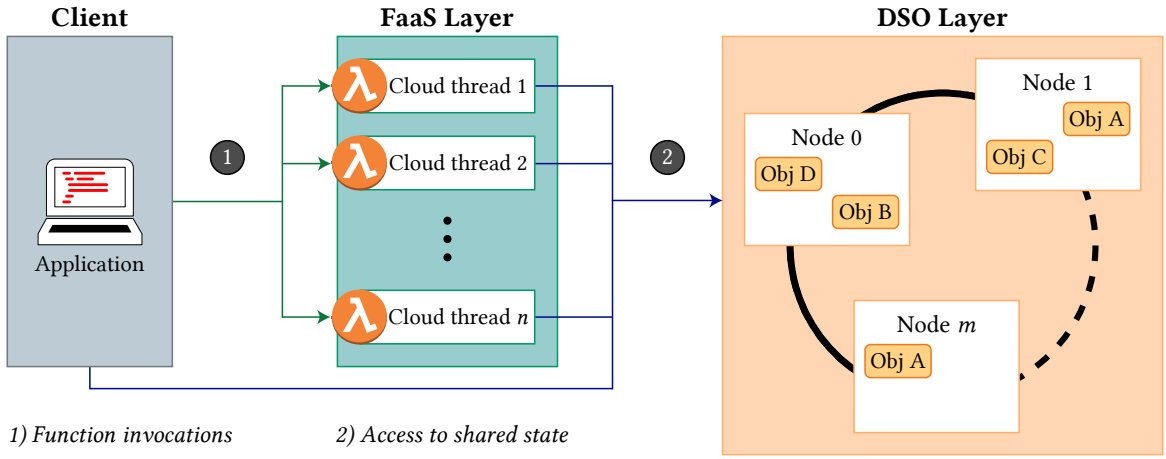


Figure 1: CRUCIAL's overall architecture. A client application runs a set of cloud threads on the FaaS layer, and all of them have access to the same shared state at the DSO layer (client included).

A second application is shown in Listing 3. This application outputs an image approximating the Mandelbrot set (a subset of $\mathbb{C}$) with a gradient of colors. The output image is stored in a CRUCIAL shared object (line 3).

To create the image, the application computes the color of each pixel (line 5). The color indicates when the pixel escaped from the Mandelbrot set (after a bounded number of iterations). The rows of the image are processed in parallel, using the `invokeIterativeTask` method of the `ServerlessExecutorService` class. As seen at line 10, this method takes as input a functional interface (`IterativeTask`) and three integers. The interface defines the method to apply on the index of the *for* loop. The integers define respectively the number of tasks among which to distribute the iterations, and the boundaries of these iterations (`fromInclusive`, `toExclusive`).

This second example illustrates the expressiveness and convenience of our framework. In particular, as in multi-threaded programming, CRUCIAL allows to express concurrent tasks with lambdas and pass them shared variables defined in the encompassing class.

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.

CRUCIAL encompasses three main components: 1. the FaaS computing layer that runs the cloud threads; 2. the DSO layer that stores the shared objects; and 3. the client application. A client application differs from a regular JVM process in two aspects: threads are executed as serverless functions, and they access shared data using the DSO layer. In addition, CRUCIAL applications may use other services in the cloud. For instance, an object storage (such as Amazon S3) to store input data (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 DSO layer 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 [71], similarly to Cassandra [75]. Each storage node knows the full storage layer membership and thus the mapping from data to node. The location of a shared object o is 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 [58] 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 One interesting aspect of CRUCIAL is that it can ensure durability of the shared state. This property is appealing, for instance, to support the different phases of a machine learning workflow (training and inference). Objects marked as persistent are replicated *rf* (replication factor) times in the DSO layer. They reside in memory to ensure sub-millisecond read/write latency and can be passivated to stable storage using standard mechanisms (marshalling). When a cloud thread accesses a shared object, it contacts one of the server nodes. The operation is then forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to the caller. Notice that for ephemeral—non-persistent—objects, *rf* is 1.

Extensions Traditionally, data is persisted on disk asynchronously to avoid the cost of having a disk access in the critical path. The release of Intel Optane DC offers the promise to reduce drastically this cost while ensuring better durability guarantees. We are currently investigating a new framework to access NVRAM in Java (see Section 6.3.2. It will be used to create a new persistent backend for the upcoming versions of CRUCIAL.

Consistency CRUCIAL provides linearizable objects and developers can reason about interleavings as in the shared-memory case. This greatly simplifies the writing of stateful serverless applications. For persistent objects, consistency across replicas is maintained with the help of state machine replication (SMR) [102]. To handle membership changes, the DSO layer relies on a variation of virtual synchrony [27]. Virtual synchrony provides a totally-ordered set of views to the server nodes. In a given view, for some object x, the operations accessing x are sent using total order multicast. The replicas of x deliver these operations in a total order and apply them on their local copy of x according to this order. A distinct replica (primary) is in charge of sending back the result to the caller. When a membership change occurs, the nodes re-balance data according to the new view.

Extensions Weaker consistency models are currently being investigated through the notion of degradability – the full details appear in Section 6.2. Degradable objects will later be included in our CRUCIAL prototype.

3.2.2 Fast aggregates through method call shipping

CRUCIAL helps to alleviate perhaps one of the biggest downsides of FaaS platforms: its data-shipping architecture [51]. 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. For instance, we exploited this feature in k-means clustering to calculate the final centroids from their partial updates. The performance benefits are detailed in Section 3.3.2.

3.2.3 Lifecycle of an application

The lifecycle of a CRUCIAL application is similar to that of a standard multi-threaded Java one. Every time a `CloudThread` is started, a Java thread (i.e., an instance of `java.lang.Thread`) is spawned on

the client. This thread pushes the `Runnable` code attached to the `CloudThread` to the FaaS platform. Then, it awaits for the result of the computation before it returns.

Access to some shared object `S` at cloud threads (or at the client) are mediated by a proxy. This proxy is instantiated when a call to "`new S()`" occurs, and either the newly created object of type `S` belongs to CRUCIAL's library, or it is tagged `@Shared`. As an example, consider the counter used in Listing 1. When an instance of `PiEstimator` is spawned, the field `counter` is created. If this object does not exist in the DSO layer, it is instantiated using the constructor defined at line 5. From thereon, any call to `addAndGet` (line 15) is pushed to the DSO layer. These calls are delivered in total order to the object replicas where they are applied before sending back a response value to the caller.

The Java thread remains blocked until the serverless function terminates. Such a behavior gives cloud threads the appearance of conventional threads; minimizing code changes and allowing the use of the `join()` method at the client 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.

The case of the `ServerlessExecutorService` builds on the same idea as `CloudThread`. A standard Java thread pool is used internally to manage the execution of all tasks. In the case of a callable task, the result is accessible to the caller in a `Future` object.

3.2.4 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.[2] 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 depicted in Listing 4 (or another iterative algorithm), it simply consists in sharing an iteration counter (line 6). When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

3.3 Evaluation

This section presents several experimental results assessing the benefits of our approach to write stateful serverless applications. In particular, we show that CRUCIAL allows to port machine learning (ML) applications in a convenient and efficient manner.

Outline. Section 3.3.1 validates the general design of CRUCIAL with a series of micro-benchmarks. Section 3.3.2 shows that our system based on fine-grained updates to shared mutable data outperforms Spark at comparable cost in two instances of ML problems. In Section 3.3.3, we outline the benefits of CRUCIAL when coordinating serverless functions. Section 3.3.5 details the (partial) portage of the Smile library [79], a conventional single-machine ML library written in Java. Section 3.3.7 explains quantitatively how CRUCIAL simplifies the programming of multi-threaded stateful applications over a serverless infrastructure.

Experimental setup. All the experiments are conducted in Amazon Web Services (AWS), within a Virtual Private Cloud (VPC) located in the `us-east-1` region. Unless otherwise specified, we use `r5.2xlarge` EC2 instances for the DSO layer and the maximum resources available for AWS Lambda.[3] Experiments with concurrency over 300 cloud threads are run outside the VPC due to

[2] Synchronization objects (see Table 1) are not replicated. This is not an important issue due to their ephemeral nature.
[3] 3008MB of memory at the time of writing.

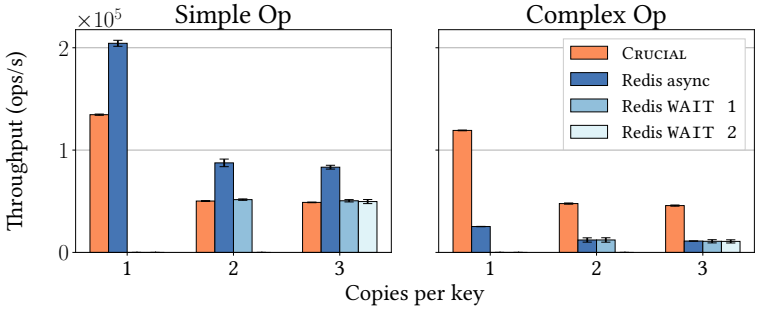


Figure 2: Operations per second performed in CRUCIAL and Redis (with and without replication). The simple operation is a multiplication. The complex one is the sequential execution of 10K multiplications. Cloud threads access uniformly at random 800 different keys/objects.

service limitations. All the benchmarks presented below are available online [6].

3.3.1 Micro-benchmarks

First, we evaluate the performance of CRUCIAL across a range of micro-benchmarks.

Latency Table 2 compares the latency to access a 1 KB object sequentially in CRUCIAL, Redis, Infinispan and S3. We chose Redis because it is a popular key-value store available on almost all cloud platforms, and it has been extensively used as storage substrate in prior serverless systems [63, 74, 98]. Each function performs 30K operations and we display the average access latency. In this test, CRUCIAL exhibits a performance similar to other in-memory systems. In particular, it is an order of magnitude faster than S3. This table also depicts the effect of object replication. When data is replicated, SMR adds an extra round-trip, doubling the latency perceived at a client. The number of replicas does not affect this behavior, as shown in the next section.

Throughput Next, we measure the throughput of CRUCIAL and compare it against Redis. The two systems are configured to provide availability and durability thanks to replication and (asynchronous) data persistence.

In Figure 2, 200 cloud threads access remotely 800 objects picked at random in closed loop for 30 s. Each object consists of an integer offering basic arithmetic operations. We consider simple and complex operations. Complex operations are implemented in Redis with the help of Lua scripts. To replicate data, Redis uses a master-based asynchronous mechanism. As a consequence, clients may observe stale values even during synchronous periods. To avoid such inconsistencies, and offer guarantees closer to CRUCIAL, clients may issue a `WAIT` command after a write [100]. In Figure 2, Redis `wait` indicates that clients execute such synchronous operations.

We compare the average throughput of the two systems when the replication factor (rf) of a datum vary as follows: ($rf = 1$) Both CRUCIAL and Redis (2 shards with no replicas) are deployed over a 2-node cluster; ($rf = 2$) We keep the same 2-node cluster, configuring Redis to use one master and one replica; ($rf = 3$) We add a third node to the cluster, Redis employing one master and two replicas.

As expected, the experimental results reported in Figure 2 indicate that CRUCIAL is not sensitive to the complexity of operations. Redis is 50% faster for simple operations because its implementation is optimized and written in C. However, for complex operations, CRUCIAL is almost five times better than Redis.

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

Table 2: Average latency comparison *1KB payload*

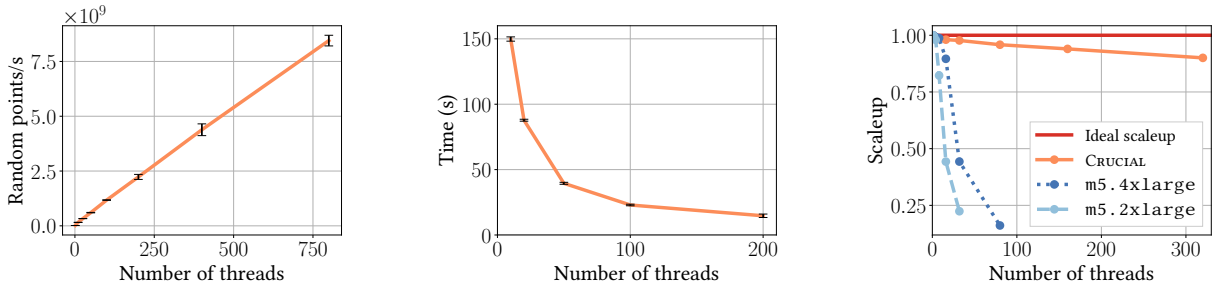


Figure 3: (a) Scalability of a Monte Carlo simulation to approximate π. CRUCIAL reaches 8.4 billion random points per second with 800 threads. (b) Scalability of a Mandelbrot computation with CRUCIAL. (c) Scalability of the *k*-means clustering algorithm with VM threads versus FaaS with CRUCIAL.

Again, implementation-specific details are responsible for this behavior: while Redis is single-threaded—so concurrent calls (Lua script) run sequentially—, CRUCIAL benefits from disjoint-access parallelism [58]. With replication, we observe a similar behavior. In particular, Figure 2 shows that CRUCIAL and Redis have similar performance when Redis operates in synchronous mode.

As expected, this experiment also verifies that the performance of CRUCIAL is not proportional to the number of replicas rf. That is, throughput is equivalent for all $rf \geq 2$, because CRUCIAL requires one extra RTT to replicate the operation to all the replicas.

Extensions In Section 6.2, we present different lines of work to the performance of SMR, in particular through the use of leaderless consensus.

Parallelism Our first application using CRUCIAL is the Monte Carlo simulation presented in Listing 1. This algorithm is embarrassingly parallel, relying 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 per second. The results presented in Figure 3a show that our system scales linearly and that it exhibits a 512x speedup with 800 threads.

Our second scalability experiment relies on the code in Listing 3. This experiment computes a a 30K×30K projection of the Mandelbrot set, with (at most) 1000 iterations per pixel. As shown in Figure 3b, the execution time reduces from 150 s with 10 threads to 14.5 s with 200 threads, that is, a speedup factor of 10.2x over the 10-thread execution. The non-linear speedup for this experiment is due to the skew in the coarse-grained, row partitioning of the image, rather than an artifact of CRUCIAL.

The example in Figure 3b underlines a key benefits of CRUCIAL. If this experiment is run in a cluster, the cluster is billed for the entire job duration, even if some of its resources are idle. Running on serverless resources, our system ensures that the row-dependent tasks are billed for their exact duration.

3.3.2 Fine-grained state management

This section shows that CRUCIAL is efficient for parallel applications that access a shared state at fine granularity. To this end, we describe the implementation of two machine learning algorithms in CRUCIAL and compare them to a single machine solution and Spark.

```
1 public class KMeans implements Runnable{
2   private CyclicBarrier barrier = new crucial.CyclicBarrier();
3   @Shared(key = "delta")
4   private GlobalDelta globalDelta = new GlobalDelta();
5   @Shared(key = "iterations")
6   private AtomicInteger globalIterCount = new AtomicInteger();
7   // Wraps a list of @Shared centroids
8   private GlobalCentroids centroids = new GlobalCentroids();
9
10  public void run(){
11    loadDatasetFragment();
12    int iterCount = globalIterCount.intValue();
13    do {
14      correctCentroids = globalCentroids.getCorrectCoordinates();
15      resetLocalStructures();
16      localDelta = computeClusters();
17      globalDelta.update(localDelta);
18      centroids.update(localCentroids, localSizes);
19      barrier.await();
20      globalIterCount.compareAndSet(iterCount, iterCount++);
21    } while (iterCount < maxIterations && !endCondition());
22  }
23 }
```

Listing 4: *k*-means implementation with CRUCIAL.

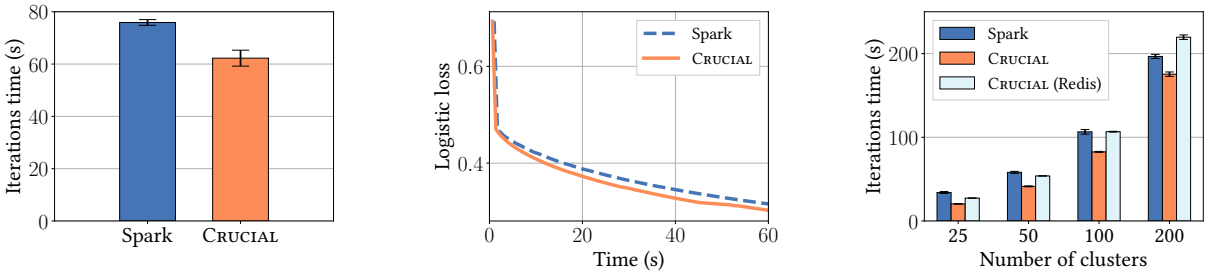
A serverless *k*-means Listing 4 details the code of a *k*-means clustering algorithm written with CRUCIAL. This program computes k clusters from a set of points across a fixed number of iterations, or until some convergence criterion is met (line 21). The algorithm is iterative, with recurring synchronization points (line 19), and it uses a small mutable shared state. Listing 4 relies on shared objects for the convergence criterion (line 4), the centroids (line 8), and a synchronization object to coordinate the iterations (line 2). At each iteration, the algorithm needs to update both the centroids and the criterion. The corresponding method calls (at lines 14, 17 and 18) are executed remotely in the DSO layer.

Figure 3c compares the scalability of CRUCIAL with FaaS against two EC2 instances: `m5.2xlarge` and `m5.4xlarge`, with 8 and 16 cores respectively. In this experiment, the input increases proportionally to the number of threads. We measure the *scale-up* computed with respect to that fact: $\textit{scale-up} = T_1/T_n$, where T_1 is the execution time of Listing 4 with one thread, and T_n when using n threads.[4] Accordingly, $\textit{scale-up} = 1$ means a perfect linear scale-up, i.e., the increase in the number of threads keeps up with the increase in the workload size (top line in Figure 3c). The scale-up is sub-linear when $\textit{scale-up} < 1$. Non-surprisingly, the single machine solution quickly degrades when the number of threads exceeds the number of cores. The solution using CRUCIAL is within 10% of the optimum. For instance, with 160 cloud threads, the scale-up factor is ≈ 0.94. This lowers to 0.9 for 320 threads due to the overhead of thread creation.

Comparison with Spark We compare CRUCIAL against Spark [121] 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. So they are a perfect fit to assess the efficiency of fine-grained updates in CRUCIAL against a current state-of-the-art solution. To complement this analysis, we also run the *k*-means application with a modified version of CRUCIAL that uses Redis for in-memory storage.

Setup For this comparison, we provide equivalent CPU resources to all competitors. In detail, CRUCIAL experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1792 MB and 2048 MB of memory for logistic regression and *k*-means, respectively. These values are chosen to have the optimal performance at the lowest cost (see

[4] In Figure 3c, threads are AWS Lambda functions for CRUCIAL, and standard Java threads for the EC2 instances.



(a) (b) (c)

Figure 4: Comparison of Spark and CRUCIAL. (a) Average completion time of the logistic regression iterative phase (100 iterations). (b) Logistic regression performance. (c) Average completion time of the *k*-means iterative phase (10 iterations) with varying number of clusters.

Section 3.3.2).[5] Spark experiments are run in an Amazon EMR cluster with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR terminology), each having 8 cores. Spark executors are configured to utilize the maximum resources possible on each node of the cluster. The DSO layer and Redis run on a `r5.2xlarge` EC2 instance.

Dataset The input is a 100 GB dataset generated with spark-perf [31] that contains 55.6M elements. For the logistic regression use case, 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 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 counterpart available in Spark's MLlib [87]: `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.

In Figures 4a and 4b, 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 (62.3 s) than with Spark (75.9 s), and thus the algorithm converges faster.[6] 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 the *k*-means implementation described in Section 3.3.2 to the one in MLlib. For both systems, the centroids are initially at a random position and the input data is evenly distributed among tasks. Figure 4c shows the completion time of 10 iterations of the clustering algorithm. In this figure, we consider different values of k to assess effectiveness of our solution when the size of the shared state varies. With $k = 25$, CRUCIAL completes the 10 iterations 40% faster (20.4 s)

[5] Starting with a configuration of 1792 MB, 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.

[6] This includes neither the provisioning time of the Spark cluster, nor the time to load and parse the dataset from S3 (the same for both systems). FaaS cold starts are also excluded due to a global barrier before measurement.

		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

Table 3: Monetary costs of the experiments

than Spark (34 s). The time gap is less noticeable with more clusters because the time spent synchronizing functions is less representative. In other words, the iteration time becomes increasingly dominated 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. We also see in Figure 4c that the implementation that uses Redis as the storage tier is always slower than CRUCIAL. This aligns with the results in Section 3.3.1.

A note on costs Although one may argue that the programming simplicity of serverless computing justifies its higher cost [63], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

Table 3 offers a cost comparison between Spark and CRUCIAL based on the above experiments. The first two columns list the time and cost of the entire experiments, including the time loading and parsing input data, but not the resource provisioning time. The last column lists the costs that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we only take into account the pricing for on-demand instances.

With the current pricing policy of AWS [11], the cost per second of CRUCIAL is always higher than Spark: 0.25 and 0.28 cents per second for 1792 MB and 2048 MB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in 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., $k = 25$).

To give a complete 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. Second, as pointed in Section 3.3.1, CRUCIAL users only need to pay for the execution time of each function, and not the time the cluster remains active. This includes bootstrapping the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [118].[7]

3.3.3 Fine-grained synchronization

This section focuses on assessing the capabilities of CRUCIAL to coordinate serverless functions.

Synchronizing a map phase Many algorithms require synchronization at various computation stages. In MapReduce, synchronization happens during the shuffle between the map and reduce phases. Starting the reduce phase requires to wait that all the appropriate data is output in the map phase. This is a costly operation, even if the reduce phase is short.

[7] 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.

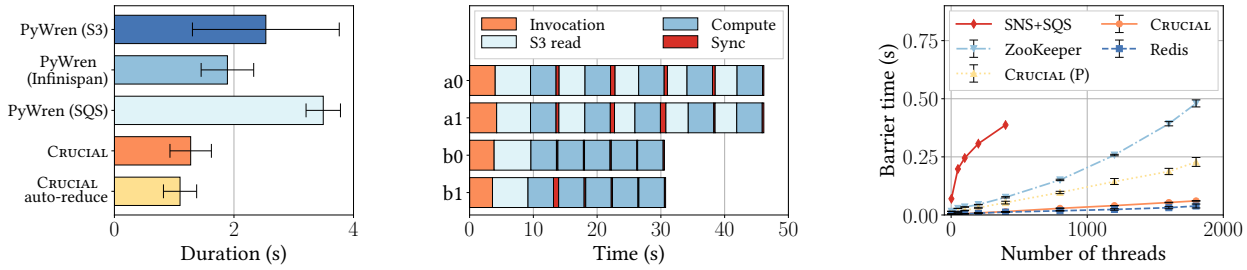


Figure 5: (a) Synchronizing a map phase in MapReduce with PyWren, Amazon SQS and CRUCIAL. (b) Performance breakdown of an iterative task using either multiple stages (a0/a1), or a single stage with a CRUCIAL barrier (b0/b1). (c) Average time threads spend waiting on a barrier.

When data is small and the reduction operation simple, aggregating the map phase output directly in the storage layer is faster [32]. The DSO layer of CRUCIAL allows to implement such an approach. To attest this fact, we compare different techniques to synchronize at the end of a map phase. 1. the original solution in PyWren, based on S3; 2. the same mechanism but above an in-memory key-value data store (Infinispan); 3. using Amazon SQS, as proposed in some recent works (e.g., [73]); and 4. two techniques based on the `Future` object available in CRUCIAL. The first solution outputs a future object per cloud thread, then runs the reduce phase. The second aggregates all the results directly in the DSO layer (auto-reduce).

We compare the above five techniques by running back-to-back the MonteCarlo simulation in Listing 1. The experiment employs 100 cloud threads, each doing 100M iterations. In a run, we measure the time spent in synchronizing the threads. On average, this time accounts for 23% of the total time spent in a run.

Figure 5a presents the results of this comparison. The solution based on Amazon SQS is the slowest. It employs a polling mechanism that actively reads messages from the remote queue. Using Amazon S3 is also slow, and the approach presents a high variability, some experiments being far slower than others. This is explained by the combination of high access latency, eventual consistency, and the polling-based mechanism. Infinispan is faster, but still being based on polling, the approach induces noticeable overheads. The solution based on `Future` objects allows to immediately respond when the results come up. This reduces the number of connections necessary to fetch the result and thus translates into better synchronization times. When the results of the each map phase are aggregated directly in the storage layer, the proposed solution based on CRUCIAL achieves even better performance, being twice faster than the polling-solution with S3.

Synchronization objects In what follows, we evaluate the capabilities of CRUCIAL to synchronize cloud threads. Figure 5b studies the performance of the barrier primitive when executing iterative tasks. In Figure 5c, we draw a comparison with existing state-of-the-art solutions.

With more details, Figure 5b presents the performance for iterative tasks that need to fetch data from an object store. This figure provides a breakdown of the time spent in each phase (Invocation, S3 read, Compute and Sync) for a selection of 2 cloud threads (out of 10). We report the breakdown for the following two approaches. The first approach launches a new stage of cloud threads (a0 and a1) for each iteration and does not use the barrier primitive. The second one launches a single stage of cloud threads (b0 and b1) that run all the iterations and use the barrier primitive for synchronization. In the first approach, input data must be fetched from Amazon S3 at each iteration, while in the second approach the threads only need to fetch it once, resulting in a lower total execution time.

Figure 5b shows that the overall time spent in synchronizing cloud threads with a barrier is low.

This lower overhead comes from the fact that function invocations and S3 accesses are not in the critical path. Both contribute to a higher variability, as seen in Figure 5a.

In Figure 5c, we presents a comparison between five different barrier implementations (including state-of-the-art approaches): 1. A non-resilient barrier using the `BLPOP` command ("Block Left Pop") and a single Redis server; 2. A ZooKeeper cyclic barrier based on the official double barrier [41] in a 3-node cluster; 3. A pure cloud-based barrier, which combines Amazon SNS and SQS services to notify threads; 4. The default cyclic barrier available in CRUCIAL, with a single server instance; and 5. A resilient, poll-based (P) barrier implementing the algorithm in [53].

To establish the comparison, we measure the time needed to exit 1000 barriers back-to-back for each approach. The experiment is run 10 times Figure 5c reports the average time to cross a single barrier for a varying number of cloud threads.

The results in Figure 5c show that non-resilient barriers exhibit close performance. With 1800 threads, these barriers are passed after waiting 68 ms on average. Resilient barriers create more contention, incurring a performance penalty when the level of parallelism increases. With the same amount of threads, passing the poll-based barrier of CRUCIAL takes 287 ms on average. ZooKeeper requires twice that time. The solution using Amazon SNS + SQS is an order of magnitude slower than the rest.

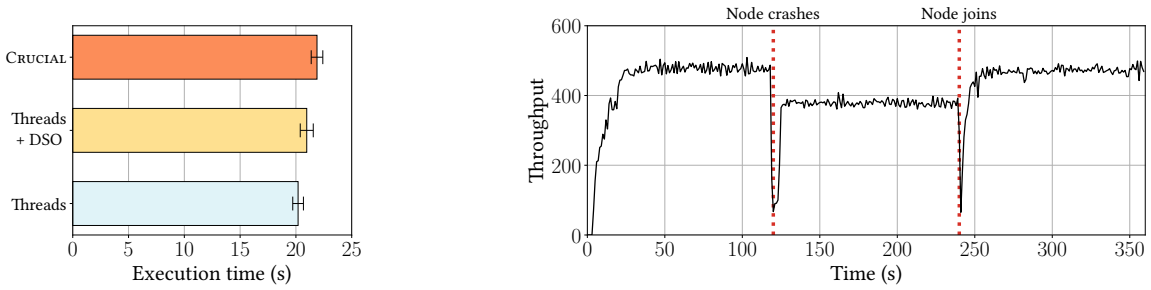
It worths noting here the difference between the programming complexity of each barrier. Both barriers implemented in CRUCIAL takes around 30 lines of basic Java code. The solution using Redis has the same length, but it requires a proper management of the connections to the datastore as well as the manual creation/deletion of shared keys. ZooKeeper substantially increases code complexity, as developers need to deal with a file-system-like interface and carefully set watches, requiring around 90 code lines. Finally, the SNS + SQS approach is the most involved technique of all, necessitating 150 lines of code and to use two complex cloud services APIs.

A concurrency problem CRUCIAL can also be used for complex task coordination. To demonstrate this feature, we consider the Santa Claus problem [116]. This problem is a concurrent programming exercise in the vein of the dining philosophers, where processes need to coordinate in order to make progress. Common solutions employ semaphores and barriers, while others, actors. [20]

In detail, the Santa Claus problem involves three sets of *entities*: Santa Claus, nine reindeer and a group of elves. The elves work at the workshop, making toys, until they encounter an issue. The reindeer are on vacation until Christmas eve, when they gather at the stable. Santa Claus sleeps, and can only be awakened by either a group of three elves to solve a workshop issue, or by the reindeer to go delivering presents. In the first case, Santa solves the issues and the elves go back to work. In the second, Santa and the reindeer go delivering toys. The reindeer have priority if the two situations occur concurrently.

Let us now explain the design of a common solution [20] to this problem. Each entity (Santa, Elves, and Reindeer) is a thread. They communicate using two types of synchronization primitives: *groups* and *gates*. In essence, these objects act like barriers and semaphores. Elves and reindeer try to join a group when they encounter a problem or Christmas is coming, respectively. When a group is full—either by three elves or nine reindeer—, the entities enter a room and await Santa. A room has two gate objects: one for entering and one for exiting. Gates act like barriers, and all the entities in the group wait for Santa to open the gate. When Santa receives a call, he looks whether a group is full (either of reindeer or elves, prioritizing reindeer). He then opens the gate and solves the workshop issues or goes delivering toys. This last operation is repeated until enough deliveries, or *epochs*, have occurred.

We implemented the above solution in three flavors. The first one uses plain old Java objects (POJO), where groups and gates are monitors and the entities are threads. Our second variation is a refinement of this base approach, where synchronization objects are stored in the DSO layer. The conversion is straightforward using the API presented in Section 3.1. In particular, the code of the objects used in the POJO solution is not changed. Only the `@Shared` annotation is required. The



(a) (b)

Figure 6: (a) Santa Claus problem's execution time on a single-machine vs. CRUCIAL. (b) Inferences per second performed on a *k*-means model during 6 minutes. Up to 100 concurrent FaaS functions connecting to the shared model on up to 3 DSO nodes with $rf = 2$. Note the FaaS cold start at the beginning.

last refinement consists in using cloud threads instead of Java ones—leveraging the `CloudThread` abstraction.

We consider an instance of the problem with 10 elves, 9 reindeer, running the simulation for 15 *toy deliveries* (epochs of the problem). We take the average time to complete the problem for each solution and plot the results in Figure 6a.

Storing the objects in CRUCIAL induces an overhead of around 8% of the running time. This low penalty shows that the synchronization model of our framework is efficient. When cloud threads are used in the last solution, there is almost no difference in the completion time. The difference in Figure 6a is due to the remote calls to the FaaS infrastructure to start the computation (for consistency, we do not include cold starts, which add 1 to 2 seconds of invocation delay). Overall, these results assess that our approach fits well for this kind of application.

3.3.4 Data availability

To assess the availability of shared objects stored with CRUCIAL, we carry out an experiment using the *k*-means algorithm. Figure 6b shows a 6-minutes run during which inferences are executed with the model trained in Section 3.3.2. The model is stored in a cluster of 3 nodes with $rf = 2$. The inferences are performed using 100 cloud threads. Each inference executes a read of all the objects in the model, i.e., the 200 centroids.

During the experiment, at 120 s and 240 s, we crash and add, respectively, a storage node to the DSO layer. Figure 6b shows that our system is elastic and resilient to such node failures. Indeed, changes in the DSO layer's membership affect performance but do not block the system. The (abrupt) removal of a node lowers performance by 30%. The initial throughput of the system (490 inferences per second) is restored 20 s after a new storage node is added.

Extensions Consistent hashing [71] is the common approach to distribute data among machines in a storage tier. When a new machine is inserted in the cluster, this algorithm may move data around. As a consequence, the system often experiments a slowdown upon the insertion – second performance drop in Figure 6b. We are currently investigating new data placement algorithms to sidestep this problem (see Section 6.3.3).

3.3.5 Smile library

This section details the portage to serverless of the random forest classification algorithm found in the Smile library. Smile [79] is a multi-threaded library for machine learning, similar to Weka [54]. It is widely employed to mine datasets with Java and contains around 165K SLOC. In what follows,

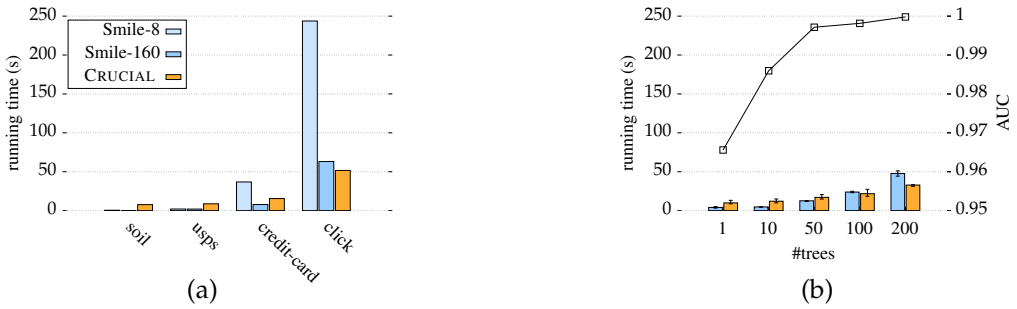


Figure 7: Smile portage. (a) Performance per dataset using 50 trees. (b) Varying the number of trees for the credit-card dataset [96].

we first describe the steps that were taken to conduct the portage using CRUCIAL. Then, we present some evaluation results against the vanilla version of the library.

Porting `smile.classification.RandomForest` The portage consists in adapting the random forest classification algorithm [23] with the help of our framework. In the training phase, this algorithm takes as input a structured file (commonly, `.csv` or `.arff`) which contains the dataset's description. It outputs a random forest, i.e., a set of decision trees. During the classification phase, the forest is used to predict the class of the input items. Each decision tree is calculated by a training task (`Callable`). The tasks are run in parallel on a multi-core machine during the training phase. During this computation, the algorithm also extracts the out-of-bag (OOB) precision, that is the forest's error rate induced by the training dataset.

To perform the portage, we take the following three steps. First, a proxy is added to stream input files from a remote object store (e.g., Amazon S3). This proxy lazily extracts the content of the file, and it is passed to each training task at the time of its creation. Second, the training tasks are instantiated in the FaaS platform. With CRUCIAL, this transformation simply requires to call a `ServerlessExecutorService` object *in lieu* of the Java `ExecutorService`. Third, the shared-memory matrix that holds the OOB precision is replaced by a DSO object. This step requires to change the initial programming pattern of the library. Indeed, in the original application, the `RandomForest` class creates a matrix using the metadata available in the input file (e.g., the number of features). If this pattern is kept, the application must load the input file to kick off the parallel computation which is clearly inefficient. In the portage, we instead use a barrier to synchronize the concurrent tasks. The first task to enter the barrier is in charge of creating the matrix in the DSO layer.[8]

For performance reasons, Smile uses Java arrays (mono or multi-dimensional) and not object-oriented constructs (such as `ArrayList`). Unfortunately, it is not possible to build proxies for such objects in Java without changing the bytecode generated during compilation. As a consequence, it is necessary to transform these arrays into high-level objects. Then, these objects are replaced with their CRUCIAL counterparts.

Overall, the portage modifies 378 SLOC in the Smile library (version 1.5.3). This is less than 4% of the original code base to run the random forest algorithm. We also added scripts to deploy and run the serverless solution in AWS Lambda, and performance benchmarks (see below), for a total of around 1K SLOC. Notice that the portage does not preclude local (in-memory) execution, e.g., for testing purpose. This is possible by switching a flag at runtime.

Evaluation results In Figure 7, we compare the vanilla version of Smile to our CRUCIAL portage. To this end, we use 4 datasets: (*soil*) is built using features extracted from satellite observations to

[8]This pattern is reminiscent of a Phaser object in Java.

categorize soils [47]; (*usps*) was published by Le Cun et al. [86] and it contains normalized handwritten digits scanned from envelopes by the U.S. Postal Service; (*credit-card*) is a set of both valid and fraudulent credit card transactions [96]; (*click*) is a 1% balanced subset of the KDD 2012 challenge (Task 2) [56].

We report the performance of each solution during the learning phase. As previously, CRUCIAL is executed atop AWS Lambda. The DSO layer runs with $rf = 2$ in a 3-node (4 vCPU, 16 GB of RAM) Kubernetes cluster. For the vanilla version of Smile, we use two different setups: an hyperthreaded quad-core Intel i78550U laptop with 16 GB of memory (tagged Smile-8 in Figure 7), and a quad-Intel CLX 6230 hyperthreaded 80-core server with 740 GB of memory (tagged Smile-160 in Figure 7).[9]

As expected for small datasets (*soil* and *usps*), the cost of invocation out-weights the benefits of running over the serverless infrastructure. For the two large datasets, Figure 7a shows that the CRUCIAL portage is up to 5x faster. Interestingly, for the last dataset the performance are 20% faster than with the high-end server.

In Figure 7b, we scale the number of trees in the random forest, from a single tree to 200. The second y-axis of this figure indicates the area under the curve (AUC) that captures the algorithm's accuracy. This value is the average obtained after running a 10-fold cross-validation with the training dataset. In Figure 7b, we observe that the time to compute the random forest triples from around 10 to 30s. Scaling the number of trees helps improving classification. With 200 trees, the AUC of the computed random forest is 0.9998. This result is in line with prior reported measures [96] and it indicates a strong accuracy of the classifier. Figure 7b indicates that training a 200-trees forest takes around 30 s with CRUCIAL. This computation is around 50% slower with the 160-threads server. It takes 20 minutes on the laptop test machine (not shown in Figure 7b).

Overall, the above results show that the portage is efficient, bringing elasticity and on-demand capabilities to a traditional monolithic multi-threaded library. We focused on the random forest classification algorithm in Smile, but other algorithms in this library can be ported to FaaS with the help of CRUCIAL.

3.3.6 Serverless shell

Our last application of CRUCIAL is a shell for serverless. The shell is a classical program in data science (see, e.g., [62], for a list of examples). The serverless shell brings the pipelining capability and powerfulness of its compact syntax to the serverless world.

The code base of the serverless shell [7] is short, only 300 SLOC (including the deployment scripts). This software takes as input shell command(s) and pipelines them into a cloud thread where they are executed.

The key idea behind the serlessshell is that the programmer may use the simplicity of shell commands to harvest public data set in a few lines of code. Typically, she will download a data sample on her local machine, program her code logic on the sample, then execute the same code in parallel over a massive amount of data in the cloud.

Some code samples are provided in Figure 9. All of these examples make use of the Common Crawl data set. In what follows, we present this data set, detail the code presented in Figure 9 and cover empirical evaluation results.

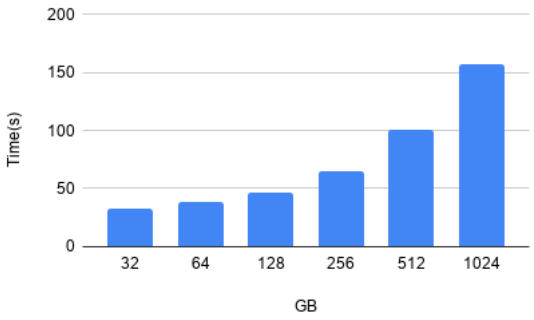


Figure 8: Distribution of IP addresses.

Dataset Common Crawl [1] is a very large data set (PBs) publicly available in AWS S3. This data set consist of billions of Internet web pages. Pages are stored in the web archive format (WARC). Every

[9] In this last case, the JVM executes with additional flags (+XX:+UseNUMA -XX:+UseG1GC) to leverage the underlying hardware architecture.

```
1 CCBASE="https://commoncrawl.s3.amazonaws.com"
2 CCMAIN="CC-MAIN-2020-16" # march 2020
3 curl -s ${CCBASE}/crawl-data/${CCMAIN}/warc.paths.gz
      | zcat | head -n ${INPUT} > ${TMP_DIR}/index
4
5 average(){
6   while read l; do
7   sshell "curl ${CCBASE}/${l} | zcat -q | grep
        ^Content-Length " &
8   done < ${TMP_DIR}/index | awk '{ sum += $2 } END {
      if (NR > 0) print int(sum / NR) }'
9 }
```

```
1 average_stateful(){
2   sshell "counter -n average reset"
3   while read l; do
4   sshell "counter -n average increment -i
        \$(curl ${CCBASE}/${l} | zcat | grep
        ^Content-Length | awk '{ sum += \$2 }
        END { if (NR > 0) print int(sum / NR)
        }')" 1> /dev/null &
5   done < ${TMP_DIR}/index
6   wait
7   local lines=$(wc -l ${TMP_DIR}/index | awk
      '{print $1}')
8   local total_average=$(sshell "counter -n
      average tally")
9   echo $((total_average/lines))
10 }
```

Figure 9: Average web page size – stateless (left), stateful (right)

month the Common Crawl organization executes a massive crawl to renew the content of the pages. Each crawl consists of around 100 TB of (compressed) data.

Code In Figure 9, we present a code that extracts the average web page size from a monthly Common Crawl data set (March 2020 in this example). The code on the left of this figure is stateless. This means that calling a cloud thread has no side effect, as in the Monte Carlo simulation in Listing 1. On the right, we present the same computation in a stateful manner, where each cloud thread increments a shared counter object.

With more details, a serverless shell is non-interactive. To call it, one uses the command `sshell` followed by the shell commands. One cloud thread is created per call. For instance, each time line 7 in Figure 9 (left) executes, a cloud thread fetches a chunk of the crawled data and extracts the average page size from the HTTP headers. The standard output of the function is sent back to the caller. At line 8, the output from all the cloud thread is aggregated `awk` to compute the final result.

The example of the right of Figure 9 is a bit more involved. The previous approach is limiting in the sense that the final computation (a reduce in the MapReduce parlance) is executed at the caller. In some cases, this can be costly and instead one would like to execute the reduction also in the cloud. To do this, we use the DSO layer of CRUCIAL and shell bindings for shared objects. For instance, calling `counter -n average tally` at line 8 in Figure 9(right) returns the value of the counter named `average` in the DSO layer. Now, at line 4, each cloud thread increments the counter with the average size of the web pages it encountered.

Evaluation results In Figure 8, we present a set of experiments in which we compute the distribution of IP addresses in the web pages of a Common Crawl data set. We scale from 32 GB to 1 TB the amount of data parsed during the computation. One cloud thread executes the computation per GB of data. A shared map object is used in the DSO layer to aggregate the final result.

In Figure 8, we observe that executing the computation with 32 GB takes 33s. It takes 100s for 512GB and 157s for a terabyte. This increase in time is due to scalability limit in the FaaS infrastructure (here AWS Lambda). There are also limitations coming from the storage system itself, the aggregation being expensive. We will investigate means to improve scalability in the second half of the CloudButton project. Nevertheless, it worths noting that the processing speed of the solution is 10GB/s. This is around 20x the sequential read speed of an average SSD drive. Hence, as in Section 3.3.5, CRUCIAL allows to execute on-demand large computation. Its performance are comparable to a dedicated cluster of high-end servers.

Application	Total lines	Changed lines	
Monte Carlo	44	2	(4.5%)
Mandelbrot	88	3	(3.4%)
Logistic Regression	430	10	(2.3%)
k-means	329	8	(2.4%)
Santa Claus problem	255	15	(5.9%)
Random Forest [10]	9882	378	(3.82%)

Table 4: Lines of code changed in each application to move it to FaaS with CRUCIAL.

3.3.7 Programming simplicity

Table 4 details the modifications that were necessary to port each Java application to CRUCIAL. For the shell applications, there is almost no difference with a local execution using, e.g. GNU parallel.

In Table 4, we observe that the difference between single-machine, parallel code and its serverless counterpart is small. In the case of Smile, as mentioned earlier, the difference mainly comes from the use of low-level non-OOP constructs in the library (e.g., Java arrays). For the other complex programs, e.g., logistic regression detailed in Section 3.3.2, changes account for less than 3%. Hence, starting from a conventional object-oriented program, CRUCIAL requires a handful of changes to port it to a FaaS platform. We believe that this smooth transitioning can help everyday programmers to start harvesting the benefits of serverless computing.

4 Software

Our current prototype is open-source and fully available online [6]. To run cloud threads, the prototype implementation relies on AWS Lambda. Support for additional FaaS platform can be easily added. Lambda functions are deployed with the help of the `lambda-maven-plugin`[11] and invoked with the AWS Java SDK. To control the replay mechanism, calls to Lambda are synchronous.

The `ServerlessExecutorService` implements the base `ExecutorService` interface. It accepts `Callable` objects and task collections. The invocation of a `Callable` returns a (local) `Future` object. This future is completed once a response from AWS Lambda is received. For `Runnable` tasks, the response is empty unless an error occurs. In that case, the system interprets it and throws an exception at the client machine, referencing the cause.

To create a distributed parallel *for*, the programmer uses an instance of `IterativeTask` (as illustrated at line 10 in Listing 3). This functional interface is similar to `java.util.function.Consumer`, but limited to iteration indexes (i.e., the input parameter must be an integer). Internally, the iterative task creates a collection of `Callable` objects. In our current prototype, the scheduling is static and based on the number of workers and tasks given by parameter.

When an AWS Lambda function is invoked, it receives a task, that is a user-defined `Runnable` (or `Callable`) object. The task and its content are marshalled and shipped to the remote machine, where they are re-created. Initialization parameters can be given to the constructor. As pointed out in Section 3.1, a distributed reference is sent *in lieu* of a shared object.

The implementation of CRUCIAL is open source and available online [6]. It consists of around 10K SLOC, including scripts to deploy and run CRUCIAL applications in the cloud. The DSO layer is written atop the Infinispan in-memory data grid [85] as a partial rewrite of the CRESON project [113].

Proxies for the shared objects are waved into the code of the client application using AspectJ [72]. In the case of user-defined objects, the aspects are applied to the annotated fields (see Section 3.1). Such objects must be serializable and they should contain an empty constructor (similarly to a Jav-

[10] Transitive closure of the dependencies of `smile.classification.RandomForest` in the Smile library.
[11] `https://github.com/SeanRoy/lambda-maven-plugin`

aBean). The `jar` archive containing the definition of the objects is uploaded to the DSO servers where it is dynamically loaded.

Synchronization objects (e.g., barriers, semaphores, futures) follow the structure of their Java counterparts. They rely internally on Java monitors. When a client performs a call to a remote object, it remains blocked until the request responds. The server processes the operation with a designated thread. During the method invocation, that thread may suspend itself through a `wait` call on the object until another thread awakes it. For instance, the cyclic barrier is implemented thanks to an internal counter and a generation system. All parties except the last one entering the barrier invoke `wait` after they increment the counter. The last thread to reach the barrier wakes up the waiting threads before starting a new generation.

State-machine replication (SMR) is implemented using Infinispan's interceptors API. This API enables the execution of custom code during the processing of a data store operation. It follows the visitor pattern as commonly found in storage systems. Infinispan [85] relies on JGroups [50] for total order multicast. The current implementation uses Skeen's algorithm [21].

In our prototype, the deployment of the storage layer is explicitly managed (like, e.g., AWS ElastiCache). Automatic provisioning of storage resources for serverless computing remains an open issue [25, 64], with just a couple works appearing very recently in this area [74, 98].

5 State of the Art

Serverless computing brings cost-efficiency and elasticity to software development. This new paradigm has gained traction recently and many works have been proposed in this area. In what follows, we cover runtimes (Section 5.1), programming frameworks (Section 5.2) and storage (Section 5.3) for cloud functions. The bottom of this section (Section 5.4) focuses on (serverless and non-serverless) solutions to the problem of stateful distributed computation. Table 5 outlines our survey, comparing CRUCIAL to other existing serverless systems that address the problem of state sharing and coordination.

5.1 Runtimes

Serverless computing has appealing characteristics, based on simplicity, high scalability and fine grained execution. It has seduced both industry [11, 88, 94] and academia [52]. This enthusiasm has also led to a blossom of open-source systems (e.g., [2–5, 52] to cite a few).

At core, a cloud function runtime is in charge of maintaining the user-defined functions, executing them upon request. It must ensure strong isolation between function instances and low startup latency for performance. Many works propose to tackle these two central challenges.

Micro-kernels [83] offer a solid basis to quickly kick-off a function and attain sub-millisecond startup time. Catalyser introduces the `sfork` system call to reuse the state of a running sandbox. Similarly, Firecracker [8] makes containers more lightweight and faster to spawn. SOCK [91] is a serverless-specialized system that uses a provisioning mechanism to cache and clone function containers. SAND [9] exploits function interaction in FaaS to improve the performance applications. The system relaxes isolation at the application level, enabling functions from the same application to share memory and communicate through a hierarchical message bus. This allows better latency and resource efficiency when combining functions. Faasm [107] offers similar guarantees using a language-agnostic runtime built atop WebAssembly. User functions use all the same substrate to execute, allowing fast initialization. They can access a distributed key-value store cached locally and shared across functions located on the same physical machine.

Two recent works [51, 64] coincide with our view that existing runtimes do not support *mutable shared state* and *coordination* across cloud functions. Hellerstein et al. [51] underline that the model is a data-shipping architecture that imposes indirect communication and hinders coordination. Jonas

System	Shared state	Synchronization	Durability	Consistency
PyWren	Object store	coarse-grained	replication	weak
ExCamera	rendezvous	rendezvous	—	—
Ripple	composition	composition	—	—
Pocket/Crail	multi-tiered	coarse-grained	ephemeral	—
Cloudburst	FaaS + cache	coarse-grained	replication	weak
Faasm	shared memory	fine-grained	—	—-
CRUCIAL	DSO	fine-grained	replication	strong

Table 5: Serverless solutions for state sharing and coordination.

et al. [64] highlight the lack of adequate storage for fine-grained operations and the inability to coordinate functions at fine granularity.

5.2 Programming frameworks

Several works that address the above two challenges confront them from a function composition perspective: a scheduler orchestrates the execution of stateless functions and shares information between them.

Many public cloud services support function composition. AWS allows creating state machines with Step Functions [12]. IBM Composer [40] offers a similar solution. Google Cloud Composer [95] allows to easily create and run a DAG of tasks in the cloud. Azure Durable Functions [88] enables to programmatically coordinate function calls. AWS has its own Amazon States Language to define the state machines. Unfortunately, the JSON-based language may become complicated for complex workflows. IBM's solution facilitates coding with a JavaScript API that is later on transformed into a state machine. While both enable state combinations, the expressiveness is very limited. Google Cloud Composer is based on Apache Airflow. It runs a per-user dedicated server that acts as a scheduler. Composition is limited to a DAG of tasks, which is inherently less expressive than state machines. Azure Durable Functions is the most complete solution among all, allowing to directly write imperative code. Asynchronous calls to functions are expressed in C# permitting a function to wait explicitly prior results.

All the above services struggle to execute embarrassingly-parallel tasks [18, 44]. To sidestep this limitation, PyWren [63] pioneered the idea to use FaaS for bulk synchronous parallel (BSP) computations. The paper shows the elasticity and scalability of FaaS and demonstrates with a base Python prototype how to run MapReduce workloads. PyWren uses a client-workers architecture where stateless functions read and write data to cloud storage (mainly Amazon S3). Numpywren [103] is a framework for linear algebra computations over cloud functions. Like with PyWren, functions are managed as a pool of stateless workers and tasks are managed in a queue. IBM-PyWren [101] evolves the PyWren model with new features and enhancements. Locus [98] enhanced PyWren for analytics computation on top of cloud functions. It focuses on the shuffling phase of the MapReduce scheme and combines cheap but slow storage with fast but expensive storage to explore a cost-performance trade-off. ExCamera [38] is another system atop FaaS, more focused on video encoding and low latency. Its computing framework (*mu*) is designed to run thousands of threads (as an abstraction for cloud functions) and manages inter-thread communication through a rendezvous server. *gg* [39] keeps *mu*'s line for running serverless parallel threads but taken to a broader audience.

Ripple [65] is a programming framework to take single-machine applications and allow them to benefit from serverless parallelism. Users rely on a simple interface to express the high-level dataflow of their applications. Ripple automates resource provisioning and handles fault tolerance by eagerly detecting straggler tasks. With the user definition, the framework is able to apply heuristics to abstract data partitioning, task scheduling, resource provisioning and fault tolerance. Before the full computation run, the framework performs a series of dry runs to test and find the best resource provisioning for the job.

Some recent works attempt to build theoretical foundations for programming with cloud functions. Jangda et al. Jangda et al. [61] propose a concise calculus to capture the operational semantics of serverless computing. Baldini et al. [17] detail the serverless trilemma: functions should be black boxes, composition should be also a function and invocations should not be double-billed. They present a solution to the trilemma for sequential compositions called IBM Sequences.

5.3 Storage

Many frameworks focus on cloud function scheduling and coordination, while using disaggregated storage to manage data dependencies. In particular, they opt to write shared data to slow, highly-scalable storage [63, 101, 103]. To hide latency, they perform coarse-grained accesses, resort to in-memory stores, or use a combination of storage tiers [98].

Pocket [74] is a distributed data store that scales on demand to tightly match the space needs of serverless applications. It leverages multiple storage tiers and right-sizes them offline based on the application requirements. Crail [111] presents the NodeKernel architecture with similar objectives. These two systems are designed for ephemeral data, which are easy to distribute across a cluster. They do not use a distributed hash table that would require data movement when the cluster topology changes, but instead use a central directory. Both systems scale down to zero when computation ends.

InfiniCache [117] is an in-memory cache built atop cloud functions. The system exploits FaaS to store objects in a fleet of ephemeral cloud functions. It uses erasure coding and a background rejuvenation mechanism to maintain data available despite the churn. Similarly to a traditional distributed in-memory cache, InfiniCache is designed to hold objects but not to facilitate their update.

The above works do not allow fine-grained updates to a mutable shared state. Such a feature can be abstracted in various ways. CRUCIAL chooses to represent state as objects, and keeps the well-understood semantics of linearizability. This approach is in-line with the simplicity of serverless computing.

Existing storage systems such as Memcached [37], Redis [99], or Infinispan [85] cannot readily be used as a shared object layer. They either provide too low-level abstractions or require server-side scripting. Coordination kernels such as ZooKeeper [55] can help synchronizing serverless functions. However, their expressiveness is limited and they do not support partial replication [33, 68]. We show these problems in Section 3.3.

CRUCIAL borrows the concept of callable objects from CRESON [113]. It simplifies its usage (`@Shared` annotation), provides control over data persistence and offers a broad suite of synchronization primitives. While CRUCIAL implements strong consistency, some systems [105, 109, 115] rely instead on weak consistency, trading ease of programming for performance. Weak consistency has been used to implement distributed stateful computation in FaaS, as detailed in the next section.

5.4 Distributed stateful computation

Cloudburst [110] is a stateful serverless computation service. State sharing across cloud functions is built atop Anna [120], an autoscaling key-value store that supports a lattice put/get CRDT data type. Cloudburst offers repeatable read and consistent snapshot consistency guarantees for function composition—something that is not achievable, for instance, when using AWS Lambda in conjunction with S3 (i.e., computing $x + f(x)$ is not possible if x mutates).

Cirrus [26] is a machine learning framework that leverages cloud functions to efficiently use computing resources. This system specializes in iterative training tasks and asynchronous stochastic gradient descent. The initial motivation for Cirrus is much in line with CRUCIAL, however the solution is quite different. Cirrus relies on a distributed data store that does not allow custom shared objects and/or computations. Furthermore, distributed workers cannot coordinate as they do in CRUCIAL.

Besides serverless systems, there exist many frameworks for machine clusters that target stateful distributed computation.

Ray [90] is a recent specialized distributed system mainly targeting AI applications (e.g., Reinforcement Learning). It offers a unified interface for both stateless task-parallel and stateful actor-based computations. Applications use both types combined and the system runs a single dynamic execution engine. Ray achieves high-scalability with a bottom-up distributed scheduler and fault-tolerance using a chain-replicated key-value store. Its architecture is based on cluster provisioning and does not fit the serverless model. CRUCIAL shares Ray's motivation for the need of a specialized system that combines stateful and stateless computations. However, Ray couples both models in the same system and is built for a provisioned resource environment where stateless tasks and actors live co-located. CRUCIAL is built with serverless in mind and separates the two types of computation. Our system uses the highly scalable capabilities of FaaS platforms for stateless tasks and a layer of shared objects for data sharing and coordination. The programming model is also consequently different: while Ray exposes interfaces to code tasks and actors, CRUCIAL uses a traditional shared-memory model where concurrent tasks are expressed as threads.

Other systems with a focus on stateful computations, such as Dask and PyTorch, usually build on low-level technologies (e.g., MPI) to communicate among nodes. These frameworks rely on clusters with known topology and struggle to scale elastically. Such a design is at odds with the FaaS model, where functions are forbidden to communicate directly.

Specialized distributed big data batch processing frameworks, like MapReduce, are available as a service in the cloud (e.g. AWS EMR). We explore such alternatives in the evaluation section (Section 3.3.2), where we compare against Apache Spark.

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 half of the project. It also provides indications on how to decide whether this work will make it into a future version of CRUCIAL.

This work is risky in the sense that not all of it will become part of the reference architecture. Such risk-taking is necessary in 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 [36, 46], orchestrating data replicas is notably difficult and moreover subject to conflicting requirements. On the one hand, strong consistency maintains the sequential invariants of the applications 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 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.

With more details, our efforts have been conducted so far on three fronts.

- First, we are collaborating with the H2020 LightKone project [80] on introducing a new communication primitive in AntidoteDB [16]. 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 [104]) 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 [106].
- 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 [43].

Results We are currently writing a research article covering the notions of consistency degradation and its interest wrt. both strong and weak consistency [70]. A prototype library of degradable objects is also being implemented [69]. Preliminary results show a up to 10x improvement over regular objects in the Apache Cassandra key-value store.

6.2 T4.3 - Just-right synchronization

The classical way of maintaining shared objects strongly consistent is state-machine replication (SMR) [102]. 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.

Strong consistency is necessary to help transitioning legacy code from shared-memory to serverless architecture. As pointed out in Section 3.2.1, it also helps the programmer to use a distributed programming framework by providing a familiar semantic. For both of these reasons, it was a key concern when developping CRUCIAL.

On the other hand, it is well-known that the above classical SMR scheme limits scalability. To sidestep this performance problem and further scale the size of the data sets that CRUCIAL is able to process, we investigated (i) how to improve the scalability of SMR with leaderless consensus, and (ii) the design of an efficient atomic multicast protocol to deal with partial replication. These two lines of work are 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 [76], Raft [92] or Zab [66], funnel commands through a leader (aka. primary) 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 [84, 89] 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 [78] accesses at least $\frac{2n}{3}$ replicas, EPaxos [89] $\frac{3n}{4}$, and Mencius [84] 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 [24, 30] and can execute in a single round-trip [77].

Refining the above observations, we have introduced a set of desirable requirements for SMR: Reliability, Optimal Latency and Leaderlessness (ROLL), 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 named Atlas which, based on the ROLL theorem, is optimal. In particular, Atlas 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 Atlas against Paxos [76], EPaxos [89] and Mencius [84] on Google Cloud Platform using the YCSB benchmark [29]. Our 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.

Results The Atlas protocol is detailed in the proceedings of the Eurosys 2020 conference [34]. Its code base [35] is available upon request. In a recent work, that appeared in the Information Processing Letter [112], we study the correctness of the Egalitarian Paxos protocol. The work on the ROLL theorem [42] was accepted to DISC 2020.

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, it 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 white-box 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.

Results The work on white-box atomic multicast [48] was presented at DSN 2019. Its implementation is open source [49].

6.3 T4.4 - In-memory data storage

Infinispan caches data in memory and overflows to a secondary storage. Task T4.4. is devoted to improving its startup time and running speed. To this end, we investigated the use of ahead-of-time compilation and recent advances in non-volatile memory.

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.

Results Every variants of Infinispan (e.g., embedded and server), as well as the Java-based clients now compiles ahead-of-time. This work is now part of the main development tree [114]

6.3.2 Support for non-volatile memory

Non-volatile main memory (NVMM) is a byte-addressable memory that preserves its content after a power lost. It provides durability and offers the promise of a dramatic increase in storage performance. NVMM is order of magnitudes faster than a flash disk, while only slightly slower than a

volatile memory (between 1 to 4 times slower). Optane DC is a recent NVMM offer by Intel, available on the public market since Q4 2019 [60].

Efficiently using NVMM in Java remains an open challenge. The first solution consists in using NVMM as a mass storage accessible through a file system interface [67]. This simple solution improves IO performance for legacy applications. However, this solution does not fully leverage the performance potential of NVMM because of a dual data representation in the volatile Java heap and in a file stored in the NVMM. Because Java object contains object references that can dynamically change during the compaction phase of a garbage collection, we cannot simply copy the state of a volatile object directly in the file. Instead, when we use the NVMM as a mass storage, a data has to be marshalled/unmarshalled when it is copied from/to the NVMM, which leads to a large overhead that could be avoided if only a single representation of the data would have leaved in the NVMM.

The second solution, explored in [28, 108], solves the problem of the dual data representation by fully unifying the volatile Java heap and the NVMM. In this case, a persistent object directly leaves in the persistent heap and the application directly accesses its field by using the byte-addressability of the NVMM. These persistent objects are fully compatible with the Java language specification, which makes the use of these systems easy for the developer. However, in counterpart, fully unifying the volatile and the persistent heap comes with several limitations: *(i)* this unification leads to important modifications in the JVM, which are hard to integrate and maintain in a production JVM; *(ii)* the persistent objects have to be garbage collected, which may drastically slow down the application[12]; and *(iii)* each write has to be instrumented in order to transparently migrate an object from the volatile to the persistent heap when the object becomes reachable through another persistent object, which also slows down the application.

In the context of Task T4.4, we work on introducing NVMM in Java-based big data stores. We propose a new framework that avoids the dual data representation in the JVM. Our framework trades the simplicity of the fully unified design for better performance. In detail, we propose a programming interface along with a library, called J-NVM, to build and manage proxies in the volatile heap that represent persistent objects. At a low level, as with the fully unified design, the state of a persistent object only leaves in the persistent heap, and J-NVM accesses the object's state through proxies, which leverage the byte-addressability of the NVMM to be efficient. However, contrary to the unified approach, the application does not directly manipulate a persistent object. Instead, the application manipulates a volatile proxy that represents the persistent object, and that serves as an interface to access the persistent heap.

Our framework makes the development of applications slightly more complex because our system does not expose persistent objects as plain old Java object. It exposes proxies, which makes the use of the persistent heap non-transparent for the developer. However, because J-NVM does not unify the persistent heap and the volatile heap, it also avoids the limitations of the fully unified approach: *(i)* J-NVM does not require important modifications to the JVM (only the addition of an intrinsic function to flush a cache line); *(ii)* J-NVM avoids the garbage collection cost by considering that, as it is done with a file in a file system or a tuple in a database the developer has to explicitly destroys the persistent objects; and *(iii)* J-NVM does not require write instrumentation because J-NVM prevents object migration from/to the persistent heap by leveraging the Java type system.

To assess the benefits of J-NVM, we implement an NVMM-backend for Infinispan. Our backend is a few hundreds lines of code long, yet offers linearizable durable operations [59]. Using the YCSB benchmark [29], we show that its performance are on par with a pure volatile memory implementation. It is also an order of magnitude faster than layering a DAX filesystem atop NVMM, while offering better guarantees.

Results The work on J-NVM was submitted to the ASPLOS 2021 conference [14]. The code of our framework is available upon request [15].

[12] Many systems advise their user to keep a small Java heap of less than 8 GB (e.g., Apache Cassandra [75]). However the smallest Intel Optane DC already contains 128 GB of persistent memory.

6.3.3 Anchored keys

Infinispan's horizontal scaling has always been driven by three main goals: *(i)* high-availability, i.e. the ability to recover from the loss of one ore more nodes by keeping multiple copies of data across the cluster; *(ii)* balancing load evenly across all nodes by ensuring that data is evenly distributed across the cluster, and *(iii)* allowing intelligent clients to retrieve data from the nodes owning it in a single hop.

As pointed out in Section 3.2.1, the key to achieving the above goals has been consistent-hashing: Nodes which own a particular item of data are computed based on a hashing algorithm known by both the server and the clients. When a topology change happens (nodes joining/leaving the cluster), the consistent-hash mapping of data to nodes (called ownership) is recomputed according to the available nodes. If necessary, entries are moved around to ensure high-availability and even distribution and load-balancing.

While the rebalancing algorithm tries to perform as little moving of data as possible and in a non-blocking fashion, it still may have a non-negligible impact on throughput and latency while it is running. For this reason a new optional data distribution algorithm, named "Anchored Keys", has been implemented since Infinispan 11. This algorithm has been designed to minimize the data which is exchanged by nodes in case of a topology change. When anchored keys are enabled, Infinispan no longer uses consistent-hashing to determine data ownership but uses the last (most recently added) node in the cluster. However, in order for all nodes to know which node owns a specific key, the ownership information needs to be propagated to all nodes. This information is stored in the extended metadata of each entry, which is then replicated to all nodes.

The current implementation of anchored keys sacrifices availability by not maintaining multiple copies of each item, but this will be amended in a future version by maintaining one or more symmetrical copies of each node.

Results The new anchored-keys module is now part of Infinispan 11 and will evolve to add more capabilities, such as high-availability [114].

7 Conclusion

This document presents CRUCIAL, a powerful system to write efficient serverless programs. CRUCIAL offers a simple interface to serverless, allowing to write (or port) effortlessly parallel code for this new environment. It is structured into a compute tier running atop a FaaS platform and a dedicated in-memory distributed storage tier.

We demonstrate how to use CRUCIAL in the context of large data analytics applications (e.g., bulk processing, parallel processing, ML). CRUCIAL allows to move to serverless existing parallel shared-memory code bases in a few modifications. The performance (and costs) are on par with a cluster of high-end servers running a dedicated complex software (such as Apache Spark).

This document also presents different directions to further improve our initial prototype during the second half of the CloudButton project. These works includes several research results on data distribution, replication and persistence that have appeared in top tier conferences and journals.

References

[1] Common crawl. https://commoncrawl.org, 2007.

[2] Serverless functions for kubernetes - fission, 2016. URL https://fission.io/.

[3] Kubeless, 2016. URL https://kubeless.io/.

[4] Openfaas, 2016. URL https://www.openfaas.com/.

[5] Apache openwhisk is a serverless, open source cloud platform, 2016. URL https://openwhisk.apache.org/.

[6] https://github.com/crucial-project, 2020.

[7] https://github.com/crucial-project/shell, 2020.

[8] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 923–935, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL http://dl.acm.org/citation.cfm?id=3277355.3277444.

[10] Amazon. Aws simple storage service. https://aws.amazon.com/s3, 2008. retrieved Aug. 2019.

[11] Amazon. Aws lambda. https://states-language.net/spec.html, 2014. retrieved Aug. 2019.

[12] Amazon. Aws step functions. https://aws.amazon.com/step-functions, 2016.

[13] Amazon. Aws glue. https://aws.amazon.com/glue/, 2017. retrieved Aug. 2019.

[14] Pierre Sutra Gaël Thomas Anatole Lefort. J-nvm: Free data persistence in java. https://github.com/jnvm-project/jnvm-paper, 2020. retrieved Aug. 2020.

[15] Pierre Sutra Gaël Thomas Anatole Lefort. The J-NVM Project. https:///github.com/jnvm-project/jnvm, 2020. retrieved Aug. 2020.

[16] AntidoteDB. https://www.antidotedb.eu.

[17] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 89–103, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355308. doi: 10.1145/3133850.3133855. URL https://doi.org/10.1145/3133850.3133855.

[18] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 25–30, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370387. doi: 10.1145/3366623.3368137. URL https://doi.org/10.1145/3366623.3368137.

[19] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, pages 41–54, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7009-7. doi: 10.1145/3361525.3361535. URL http://doi.acm.org/10.1145/3361525.3361535.

[20] Mordechai Ben-Ari. How to solve the santa claus problem. *Concurrency: Practice and Experience*, 10, 2001. doi: 10.1002/(SICI)1096-9128(199805)10:63.0.CO;2-2.

[21] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computers Systems*, 5(1):47–76, January 1987. ISSN 0734-2071. doi: 10.1145/7351.7478. URL http://doi.acm.org/10.1145/7351.7478.

[22] Stephen Blum. Amazon sns vs pubnub: Differences for pub/sub. https://www.pubnub.com/blog/2014-08-21-amazon-sns-pubnub-differences-pubsub/, 2014.

[23] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. URL https://doi.org/10.1023/A:1010933404324.

[24] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[25] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew M Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, 2018.

[26] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369732. doi: 10.1145/3357223.3362711.

[27] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[28] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 259–269, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357999. doi: 10.1145/3210377.3210400. URL https://doi.org/10.1145/3210377.3210400.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[31] Databricks. spark-perf. https://github.com/databricks/spark-perf, 2014.

[32] David J. DeWitt and Michael Stonebraker. MapReduce: A major step backwards, 2008. DatabaseColumn Blog. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

[33] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 10:1–10:16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741954. URL http://doi.acm.org/10.1145/2741948.2741954.

[34] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387543. URL https://doi.org/10.1145/3342195.3387543.

[35] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. https://github.com/imdea-software/VCD-broadcast, 2020. retrieved Aug. 2020.

[36] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. doi: 10.1145/3149.214121. URL http://doi.acm.org/10.1145/3149.214121.

[37] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[38] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith 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.

[39] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/fouladi.

[40] The Apache Software Foundation. Openwhisk composer. https://github.com/apache/openwhisk-composer, 2017.

[41] The Apache Software Foundation. Zookeeper barrier recipe, 2019. URL https://zookeeper.apache.org/doc/current/recipes.html#sc_recipes_eventHandles.

[42] Tuanir França and Pierre Sutra. Leaderless State-Machine Replication: Specification, Properties, Limits. 2020. to appear.

[43] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, February 2011. ISSN 0360-0300. doi: 10.1145/1883612.1883616. URL http://doi.acm.org/10.1145/1883612.1883616.

[44] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of faas orchestration systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 148–153. IEEE, 2018.

[45] Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. Technical Report TR-08-07, Harvard Computer Science Group, 2007. URL http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829568.

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

[47] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLOS ONE*, 11(4):1–31, 04 2016. doi: 10.1371/journal.pone.0152173. URL https://doi.org/10.1371/journal.pone.0152173.

[48] Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast (extended version). *CoRR*, abs/1904.07171, 2019. URL http://arxiv.org/abs/1904.07171.

[49] Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast. https://github.com/imdea-software/atomic-multicast, 2020. retrieved Aug. 2020.

[50] Red Hat. Reliable group communication with jgroups. http://jgroups.org/manual/#TOA, 2015.

[51] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang 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. URL http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[52] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association. URL http://dl.acm.org/citation.cfm?id=3027041.3027047.

[53] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, February 1988. ISSN 0885-7458. doi: 10.1007/BF01379320. URL https://doi.org/10.1007/BF01379320.

[54] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. In *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*, pages 357–361, 1994.

[55] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, USENIX ATC. USENIX Association, 2010.

[56] Tencent Inc. Kdd cup - 2012. https://www.openml.org/d/1220, 2014.

[57] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017. URL http://arxiv.org/abs/1710.08460.

[58] Amos Israeli and Lihu 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, pages 151–160, 1994. doi: 10.1145/197917.198079.

[59] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2016. doi: 10.1007/978-3-662-53426-7_23. URL https://doi.org/10.1007/978-3-662-53426-7_23.

[60] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[61] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360575. URL https://doi.org/10.1145/3360575.

[62] Jeroen Janssens. *Data Science at the Command Line: Facing the Future with Time-Tested Tools*. O'Reilly Media, Inc., 1st edition, 2014. ISBN 1491947853.

[63] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC'17, 2017. doi: 10.1145/3127479.3128601.

[64] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[65] Shannon Joyner, Michael MacCoss, Christina Delimitrou, and Hakim Weatherspoon. Ripple: A practical declarative programming framework for serverless compute, 2020.

[66] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks (DSN)*, 2011.

[67] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359631. URL https://doi.org/10.1145/3341301.3359631.

[68] Babak Kalantari and André Schiper. *14th International Conference Distributed Computing and Networking*, chapter Addressing the ZooKeeper Synchronization Inefficiency. ICDCN. Springer Berlin Heidelberg, 2013.

[69] Boubacar Kane and Pierre Sutra. Degradability: a Simplified Approach to Data Consistency. https://git.overleaf.com/5e25ea7dcf314c000109a114, 2020. retrieved Aug. 2020.

[70] Boubacar Kane and Pierre Sutra. A library of degradable objects for Apache Cassandra. https://github.com/BoubacarKaneTSP/Application, 2020. retrieved Aug. 2020.

[71] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel 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.

[72] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, ECOOP, 2001.

[73] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. *CoRR*, abs/1803.06354, 2018. URL http://arxiv.org/abs/1803.06354.

[74] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/klimovic.

[75] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[76] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 1998.

[77] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[78] Leslie Lamport. Fast Paxos. *Distributed Computing*, 2006.

[79] Haifeng Li. Smile. `https://haifengl.github.io`, 2014.

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

[81] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2): 129–137, March 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

[82] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

[83] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132763. URL `https://doi.org/10.1145/3132747.3132763`.

[84] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[85] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[86] Ofer Matan, Henry S. Baird, Jane Bromley, Christopher J. C. Burges, John S. Denker, Lawrence D. Jackel, Yann Le Cun, Edwin P. D. Pednault, William D. Satterfield, Charles E. Stenard, and Timothy J. Thompson. Reading handwritten digits: A zip code recognition system. *Computer*, 25(7):59–63, July 1992. ISSN 0018-9162. doi: 10.1109/2.144441. URL `https://doi.org/10.1109/2.144441`.

[87] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL `http://jmlr.org/papers/v17/15-237.html`.

[88] Microsoft. Azure durable functions. `https://functions.azure.com`, 2016.

[89] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[90] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL `http://dl.acm.org/citation.cfm?id=3291168.3291210`.

[91] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/oakes.

[92] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[93] Google Cloud Platform. Bigquery. https://cloud.google.com/bigquery/, 2010. retrieved Aug. 2019.

[94] Google Cloud Platform. Cloud functions. https://cloud.google.com/functions/, 2016. retrieved Aug. 2019.

[95] Google Cloud Platform. Cloud composer. https://cloud.google.com/composer, 2018. retrieved March. 2020.

[96] A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. Calibrating probability with undersampling for unbalanced classification. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 159–166, 2015.

[97] The CloudButton project. D5.2 - CloudButton Prototype of Abstractions, Fault-tolerance and Porting Tools, 2020.

[98] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/pu.

[99] Redis. https://redis.io/, 2009.

[100] Redis. Replication, 2019. URL https://redis.io/topics/replication.

[101] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6016-6. doi: 10.1145/3284028.3284029. URL http://doi.acm.org/10.1145/3284028.3284029.

[102] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. ISSN 0360-0300. doi: 10.1145/98163.98167.

[103] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018. URL http://arxiv.org/abs/1810.09679.

[104] Marc Shapiro and Pierre Sutra. Database consistency models. In *Encyclopedia of Big Data Technologies*. 2019. doi: 10.1007/978-3-319-63962-8_203-1. URL https://doi.org/10.1007/978-3-319-63962-8_203-1.

[105] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, June 2011.

[106] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011. URL http://eatcs.org/beatcs/index.php/beatcs/article/view/120.

[107] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *CoRR*, abs/2002.09344, 2020. URL https://arxiv.org/abs/2002.09344.

[108] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 316–332, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314608. URL https://doi.org/10.1145/3314221.3314608.

[109] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ISBN 978-1-4503-0977-6. doi: http://doi.acm.org/10.1145/2043556.2043592. URL http://doi.acm.org/10.1145/2043556.2043592.

[110] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service, 2020.

[111] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 767–782, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/stuedi.

[112] Pierre Sutra. On the correctness of egalitarian paxos. *Inf. Process. Lett.*, 156:105901, 2020. doi: 10.1016/j.ipl.2019.105901. URL https://doi.org/10.1016/j.ipl.2019.105901.

[113] Pierre Sutra, Etienne Riviere, Cristian Cotes, Marc Sánchez-Artigas, Pedro García-López, Emmanuel Bernard, William Burns, and Galder Zamarreno. CRESON: callable and replicated shared objects over nosql. In *37th IEEE International Conference on Distributed Computing Systems*, ICDCS'17, 2017. doi: 10.1109/ICDCS.2017.239.

[114] [The Infinispan Team.

[115] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[116] John A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994. ISSN 0097-8418. doi: 10.1145/187387.187391.

[117] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/wang-ao.

[118] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE Conference on Computer Communications, INFOCOM 2019*, 2019.

[119] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX*

ATC 18), pages 133–146, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[120] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, February 2019. ISSN 2150-8097. doi: 10.14778/3311880.3311881. URL https://doi.org/10.14778/3311880.3311881.

[121] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*, pages 15–28, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.