Pikachu: Serverless Message Passing

Yang Zhou
Harvard Univeristy
yangzhou@g.harvard.edu

David Ralph Hughes
Harvard Univeristy
davidralphhughes@college.harvard.edu

Abstract

Serverless computing pioneered by AWS Lambda becomes popular due to its appealing fine-grain billing and auto-scaling properties. Recent research has demonstrated that moving big data processing (*e.g.*, data analytics, video processing) to serverless will bring huge benefits. However, this research heavily relies on non-serverless components such as S3 and Redis to pass messages between function instances during big data processing, which is non-efficient and costly.

In this paper, we argue that current serverless platforms lack the features of direct message passing and addressability that are necessary to implement distributed data processing algorithms in an efficient manner. To resolve these two missing features, we designed a new serverless platform, Pikachu , that allows function instances to find and communicate with each other directly. The experiment results show that Pikachu achieves up to 1070.4 times shorter messaging passing time than using Redis. We open source the code of Pikachu and evaluation scripts at GitHub [11].

1 Introduction

Serverless computing platforms offer a service to users that is often more intuitive, lower maintenance, and cheaper than cloud hosted VMs. Using a definition paraphrased from the 2019 Berkeley View on Serverless Computing [16], serverless computing is the combination of Functions as a Service (FaaS) and Backend as a Service (BaaS). This structure eliminates the needs for users to manage their own VMs, which abstracts away most of the complications of managing resources and auto-scaling.

Recent research has shown that big data processing jobs can be implemented on serverless platforms and reap the benefits of the fine-grained billing and auto-scaling. In particular, numpywren [19] thoroughly examines the benefits and drawbacks of implementing a system for linear algebra on AWS Lambda and using Redis, a remote key-value object store, for runtime state-store. The biggest constraint on numpywren's overall efficiency is network latency due to the necessity of using a remote object store, which leads to a much higher amount of data being read than their primary linear algebra comparison, ScaLAPACK.

Another example that seeks to utilize serverless' capabilities is Ex-Camera [13], which again uses AWS Lambda to process video using "thousands of tiny threads" to compute the highly parallel parts of video processing. Though Ex-Camera's performance results are quite good, their framework requires two long-running servers: a coordinator on an EC2 instance to send RPC requests to each worker, and a rendezvous server which relays messages from the workers to their destination on an S3 instance. Not only do these potentially complicated, long-running servers still require the management of VMs, but using S3 and the rendezvous server incurs significant latency.

Both numpywren and Ex-Camera's workarounds of using an in-memory or cloud store are due to the lack for function instances to be addressable, send or receive messages, or coordinate between multiple instances. The storage solutions (either in-memory or on-disk) not only make implementing algorithms potentially more complicated, but also introduce significant overhead for all data transferal, much of which comes from the double data copies during forwarding data from one function instance to another. Additionally, the attractiveness of fine-grained billing is significantly reduced, since the heavy usage of a storage system incurs costs, as well as utilizing a long running VM.

We argue that in current serverless platforms, two features are lacked: **direct message passing between function instances, and the ability to address function instances (*i.e.*, addressability)**. To implement these two missing features in current serverless platforms, we design and build a new serverless platform named Pikachu on top of Apache OpenWhisk [3], an open source serverless platform deployed in IBM Cloud. Pikachu utilizes the Docker overlay network to enable serverless function instances to directly communicate with each other. Further, Pikachu integrates a

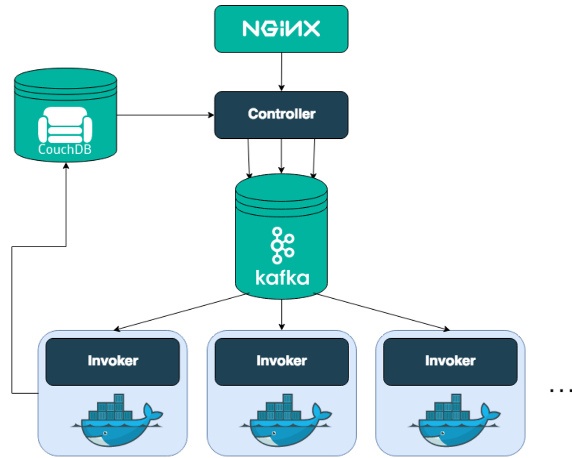

Figure 1. OpenWhisk components.

highly scalable directory service into serverless platform which enables instances to look up the address and survival status of other instances.

Our experiment results show that Pikachu incurs minimum overhead to the OpenWhisk serverless platform, while bringing enormous benefits. For instance, Pikachu increase the total completion time by $-2.25\% \sim 3.0\%$ compared with the original OpenWhisk. However, it achieves up to 1070.4 times shorter messaging passing completion time than the original OpenWhisk using Redis for messaging passing.

This paper proceeds as follows: In section 2, we provide a background for Apache OpenWhisk and two Docker features that are utilized to provide direct message passing and addressability. Section 3 summarizes the architecture of Pikachu, and in Section 4 we provide the details of Pikachu's design. Section 5 then describes the implementation, followed by an evaluation of Pikachu's performance in Section 6, and a discussion of future work on Pikachu in Section 7. In Section 8, we compare Pikachu with related work. We conclude this paper in Section 9.

2 Background

2.1 OpenWhisk

Pikachu is built on top of Apache OpenWhisk [3] – an open source serverless cloud platform which has been deployed in IBM Cloud as IBM Cloud Functions [9]. Here, we briefly describe the components and workflow of the OpenWhisk serverless platform.

OpenWhisk components: OpenWhisk stands on the shoulders of giants, including Nginx [10], Kafka [2], CouchDB [2], Zookeeper [4], Redis [12], Akka [1], and Docker [5]; OpenWhisk builds its own controller and invoker (developed in Scala) to glue these components and forms an event-based, serverless programming service, as shown in Figure 1. Note that all these components (including the controller and invokers) run in separate Docker containers.

Nginx serves as the SSL termination point and forwards appropriate HTTP requests to the controller. The controller load balances the incoming function instance invocation requests among all the alive invokers and evenly hashes each request to each invoker and pushes the invocation assignment to Kafka. Once Kafka has confirmed that it gets the invocation message, an ActivationId will be responded to user via an HTTP request; the user can later use this ActivationId to fetch the results of this specific invocation from the CouchDB. By default, OpenWhisk only uses one centralized controller, but it also supports deploying multiple controllers for scalability and fault-tolerance purpose.

Kafka (assisted by ZooKeeper) is a high-throughput, distributed, publish-subscribe messaging system. OpenWhisk uses Kafka to connect a controller to invokers: a controller pushes function instance invocation messages to Kafka, while invokers pull them. Kafka is essential to OpenWhisk when an invoker crashes, since Kafka can ensure the reliable communication with "exactly once" semantics for the messages sent between controller and invokers. OpenWhisk uses the concept of *topic* in Kafka to distinguish between message queues belonging to different invokers, *e.g.*, messages with topic "invoker0" will be read by invoker0.

Each invoker continually pulls invocation messages with specific topic from Kafka. Upon receiving a invocation message, normally the invoker will fetch the function code and parameters from CouchDB, create a container using Docker, inspect the IP addresses of the new created containers, and inject both the function code and parameters into the new created container via an HTTP connection. The container then will then start running the function code as a *cold start* invocation. After the function instances complete their executions, invokers will either destroy or pause the containers, the latter case results in a container waiting for new function invocations that own the same function code. The case where a container can be reuses is a *warm start* invocation, which removes the overhead of fresh container creation and function code initialization. The function execution results and logs (*i.e.*, the stdout and stderr of the instance) will be returned to invokers via

```
wsk action create fib ./fib.py
ok: created action fib

wsk action invoke fib -p number 39
ok: invoked /_/fib with id
↪ ee21c6a444fb401da1c6a444fb101d08
```

Figure 2. Function creation and direct invocation commands.

an HTTP connection and finally sent to CouchDB by the invoker. Thus, with CouchDB, the user can then access the results of this function instance. Additionally, each invoker frequently pushes heartbeat messages to Kafka, which is pulled by controller and used for validating the survival status of that invoker.

Redis, an in-memory data store is used to store configurations and some run states of OpenWhisk. Akka is an actor-based message-driven concurrency programming model, and extensively used in OpenWhisk controllers and invokers.

OpenWhisk workflow: OpenWhisk utilizes *namespaces* with private keys to isolate different users. (One user can also access multiple namespaces once that user gets the corresponding namespace keys). Within their own namespace, users can create triggers, rules, and actions; rules are used to connect triggers with actions. Users can specify how events (*i.e.*, git push) will set off triggers, and how fired triggers invoke action executions via a set of rules. OpenWhisk provides a CLI that allows for complete management of the system, which translates each command into a proper HTTP request and sends the request to Ngnix. Figure 2 shows the action (*i.e.*, function) creation and direct invocation commands, which will help the understanding of Pikachu 's design. `wsk action create` accepts the function name (*e.g.*, "fib"), function source code path (*e.g.*, "./fib.py"), and will eventually store the function in CouchDB. `wsk action invoke` accepts the function name (*i.e.*, "fib") and function parameter(s) (*i.e.*, "number 39") and will eventually push the function invocation message to Kafka and return an ActivationId to user.

2.2 Docker

OpenWhisk relies on Docker to create, destroy, pause, and resume containers running function instances. Our Pikachu explores two features – *overlay network* and *volume* – provided by Docker but not yet employed by

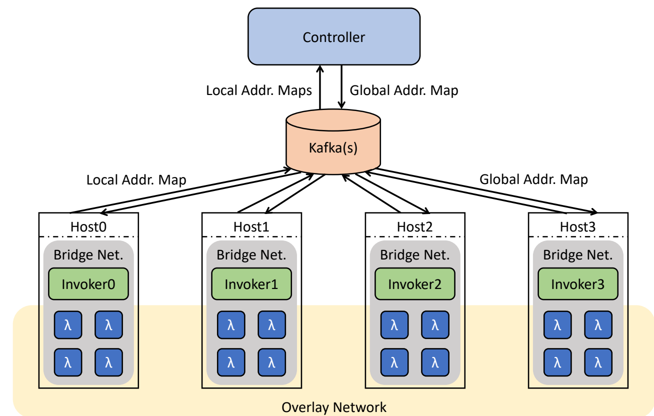

Figure 3. Pikachu architecture.

OpenWhisk to realize the message passing functionality. Thus, we briefly describe these two features here.

Container networking: Containers typically have four networking modes: bridge mode, host mode, macvlan mode, and overlay mode [6]. Bridge mode is used specifically for container communication within a single host, while the other three modes support cross-host container communication. Recent research (*i.e.*, SlimOS [20]) have shown that overlay network is currently the most popular choice for cross-host container networking in the cloud environment since it requires easier data center management and easier data center network routing schemes as compared with the host and macvlan modes.

The container overlay network creates a virtualized network address space where each container can get a unique virtual IP address. The overlay network supports cross-host routing using a vSwitch in the host kernel to do translations between the virtual IP address and the host IP address.

Docker volume feature: Docker supports mounting a directory within the container to a directory in the host via the volume feature [8]. The volume feature enables data generated by Docker containers to persist between container instances and also allows file sharing between the host and a container using this shared directory. Different containers on the same host share the same host directory.

3 Architecture

As shown in Figure 3, Pikachu is built on top of OpenWhisk, and introduces several changes to the existing OpenWhisk invoker and controller. When invoking function instances, the invokers in Pikachu attach the

created containers to the overlay network corresponding to this user. Each invoker then injects function code and parameters to its created local containers through an HTTP request over bridge network. Additionally, the invoker inspects the instance IDs and overlay network IP addresses of containers and constructs the *local address map* (*i.e.*, the <instance ID, IP address> mappings). The invokers then periodically send their local address maps to the controller via Kafka(s). Upon receiving the local address maps from invokers, the controller constructs a *global address map* and periodically send the map back to all the healthy invokers with Kafka. When invokers get the global address map, they share it to each running function instance in its local host. In this way, different function instances can address each other using the global address map provided by its local invoker and instances can communicate with each other over the overlay network.

4 Design

This section describes the system design details of Pikachu, including the usage of our system (§4.1), data transmission directed between containers on different hosts (§4.2), addressability support via a built-in directory service (§4.3), and security considerations (§4.5).

4.1 User Interface

In Pikachu, users can specify a function instance ID when invoking function instances by passing the instance ID string as an additional parameter to the function. The new function invocation command is shown in Figure 4. Users can invoke hundreds or thousands of function instances at the same time to do data processing jobs [13, 15, 17–19]. Previous work all use in-memory key-value stores (*i.e.*, Redis, Memcached) in long-running VMs or cloud storage services (*i.e.*, AWS S3) to perform inter-instance communication. In Pikachu, each running function instance can directly address others using the instance ID. Specifically, one instance can get the *<instance ID, IP address>* mapping of all survived instances belonging to the same user in the system. These mappings are exactly the *the local address map* mentioned in §3. The instance can then, for example, create a socket connection and transfer data with its peers. Besides this, running function instances can also get the survival status of other instances from this address map, avoiding the timeout cost of trying to connect to terminated instance.

```
wsk action invoke fib -p number 39 -p
↪ instanceID myname0
```

Figure 4. Function invocation commands in Pikachu.

4.2 Message Passing on the Overlay Network

The current OpenWhisk code only uses the Docker bridge network for invokers to inject function code and parameters into newly created or resumed[1] containers within the local host.

One function instance can use the IP addresses in the local bridge network to communicate with other instances on the same host; however, bridge network IP addresses do not support cross-host communication.

In order to provide cross-host communication in OpenWhisk, we open and utilize the overlay network feature provided by Docker. When a user registers in a Pikachu deployment, in addition to generating a namespace with a private key, Pikachu creates an overlay network bind with that namespace. When the user invokes a function instance, the invoker in Pikachu will attach the newly created container[2] to the overlay network corresponding to that user. In this way, once one function instance knows the overlay network IP addresses of its peers, it can directly communicate with those peers through the overlay network.

The next section will show how Pikachu lets one function instance knows its peers' IP addresses in the overlay network, which requires a directory service built into the serverless platform.

4.3 Build-In Directory Service

The built-in directory service in Pikachu provides the translation from Instance IDs to overlay network IP addresses and enables a fast query for the survival status of function instances. There is a natural division of labor in our directory service design: individual invokers collect overlay network IP addresses and construct the local address map, while a centralized controller gathers all the local address maps and constructs the global address map.

[1] These containers were paused with function code retained, serving for warm-start function instances with only parameters injected (for details, please refer to §2).

[2] Warm-start function instances are already attached to the overlay network.

Local address map construction: When inspecting container IP addresses in the bridge network and injecting function code and parameters into the container, the invoker in Pikachu additionally inspects the container IP address in the overlay network, and extracts the instance ID from the function parameters. Thus, the invoker obtains the <instance ID, IP address> mapping for this function instance, and constructs the local address map for this host. When the function instance completes its execution and returns, the invoker in Pikachu will remove the <instance ID, IP address> mapping from its local address map.

Encoding the local address map into heartbeat messages: One possible solution of sending a local address map to the controller is by creating a new topic in Kafka (*e.g.*, "addrMap"): invokers send their local address maps to this message queue and the controller pulls the maps from this queue. This is a workable solution, but it requires additional overhead to maintain the message queue in Kafka. Instead, to remove this overhead, invokers in Pikachu encode the local address map into the periodic heartbeat messages sent to the controller[3]. In this way, we efficiently send all local address maps to the controller with nearly no overhead.

Besides encoding the local address map into the heartbeat messages, Pikachu invokers only send changes of the local address maps to the controller. This means if there are no local address map updates since the last heartbeat message, no address map message is sent to the controller. In the case of a large number of function instances running in the cloud, sending only the deltas can enormously reduce the message sending overhead.

Global address map construction: The centralized controller gathers local address maps from invokers and constructs the global address map. The controller can determine the frequency of synchronizing the global address map to the invokers by considering the extent of local address map changes, current system load, *etc.*. By default, the controller sends back the global address map every second. It follows then that each function instance can know its peers' survival status with 1 second granularity. In the rare case where an invoker crashes, the controller removes all the <instance ID, IP address> mappings belonging to that invoker from the global address map. The controller then only needs to maintain the latest local address map for each invoker.

Distributing global address map: Pikachu employs Kafka's "group" feature to efficiently distribute the global address map to multiple invokers. Kafka supports dividing consumers into multiple consumer groups and each consumer group receives a copy of each message pushed by the producer. In Pikachu, we consider each individual invoker as a consumer group and the controller as the only producer. In this way, every global address map the controller pushes to Kafka can be received by every invoker. Note that we also only send the changes of the global address map from the controller to the invokers to reduce message sending overhead.

Invokers sharing global address map to function instances: After each individual invoker obtains the global address map from the controller via Kafka, it shares the address map to all the function instances running in its host. Pikachu employs the volume feature supported by Docker to share the global address map as a file to all the running containers. The main reasons we chose a shared file is that file systems exhibit generic behavior, are ubiquitous, and are easy to use. Serverless platforms run function instances written in a variety of programming languages and almost every language has sufficient support for file operations. In contrast, although a shared memory segment is more efficient than shared files, it is not generally supported by every language. For instance, Scala[4] runs on the JVM, and the JVM itself has no official API to manipulate a shared memory segment. The only way shared memory segments can be utilized in IPC on the JVM is by resorting to a helper library/DDL and JNI to use native libraries written in other languages such as C, C++, etc. Another difficulty of using shared memory for IPC appears when crossing the boundary of containers, since all the components in OpenWhisk, including invokers and function instances, are running in Docker containers.

Future work for sharing the global address map potentially would use RPC requests to reduce the overhead of file operations.

So far, we have described the complete design of the directory service in Pikachu. In the next section, we outline several potential alternative design choices and compare them with our current design.

4.4 Alternative Directory Service Designs

Using dedicated servers for directory service: In the cloud, we might be able to assign several servers the responsibility of handling the directory service. Every time a function instance is created or terminated, it

[3]The default interval between heartbeat messages is 1 second

[4]The development language of OpenWhisk

would contact the directory service and update its own <instance ID, IP address> mapping entry. Every time a function instance wants to get the IP addresses of its peers, it would contact the directory service to get the results. To reduce the directory service lookup overhead, each function instance could have a local cache for frequently used mapping entries, which would synchronize with the global address map in the directory servers periodically. Microsoft used this design to virtualize its own multi-tenant VM-based data center network [14] due to its ease of management and high scalability guarantee (*e.g.*, simply adding more directory servers when expanding data centers).

However, we argue that this might not be a good fit for serverless computing due to the following two reasons: 1) Function instances are much more ephemeral than VMs, which causes much more mapping entry updates on the directory servers, and requires much more timely synchronization. 2) In the context of serverless computing, this design requires more long-running dedicated directory servers and costs more. At the conceptual level, putting more long-running dedicated servers in serverless platforms runs in the opposite direction of "serverless".

Instead, Pikachu seek provides a scalable directory service built into serverless platform itself (*i.e.*, no need for additional long-running servers). Our design scales well because the directory service in Pikachu easily scales as the number of invokers (and controller) increases, due to the design only requiring the invokers and controller to do a small amount of additional work.

Using the embedded DNS service in Docker overlay network: Docker engine has a local DNS service maintaining the <container name, IP address> mappings for the overlay network, which could serve as directory service in OpenWhisk if we set the container name to the instance ID users specified.

We think this still might not be a good fit for serverless computing due to security complications, lowered genericity, and higher maintenance overhead. 1) The design using Docker's embedded DNS service exposes the entire container namespace to users, while the container namespace is shared among all the containers running in the overlay network (not just the containers running in a single host). We argue that exposing too many cloud details to users might cause potential security vulnerability, *e.g.*, Denial of Service attacks for a namespace. 2) Not every serverless platform has the local DNS service in its VM hypervisor or container engines. This would hurt the genericity of using embedded DNS service. 3) The DNS services in Docker engines must maintain a consistent global network view (*i.e.*, container name to overlay network IP mappings) across hosts. Given the ephemerality of serverless function instances, this might incur significant overhead for the hosts.

However, we do consider a purely network-based solution for the serverless directory service as a promising direction, but it would require a brand new design that targets the above three concerns instead of just using the existing Docker overlay network DNS service.

4.5 Security Considerations

In Pikachu, each user has its own overlay network (*i.e.*, virtualized network address space) which is created during user registration process (§4.2). Therefore, each user's networking is fully isolated. Further, Pikachu could support an access control interface in the centralized controller which has a global view of the network. Additionally, if we install traffic meters into the invokers and use the controller to gather the traffic statistics of different users, Pikachu could be extended to support rate limiting and adjusted quality of service. We leave the detailed design and implementation for the security aspect of Pikachu for future work.

5 Implementation

We implemented our Pikachu system on top of the existing OpenWhisk platform by adding over 500 lines of Scala code across 17 files. We hacked on the OpenWhisk commit version of 7ecae176c4c02fa789cee644dc24eee1317c0256, which was the latest version when we started our project. Our Pikachu's implementation runs successfully and stably on top of Ubuntu 16.04, and executes user-provided serverless functions correctly. We only use the Scala standard library, the Java NIO library for file operations, and the Akka library for efficient concurrent programming[5]. We expect Pikachu to run on other platforms without any change, since our implementation is purely based on the JVM without any OS-related dependencies. All the related source code and evaluation scripts can be found at GitHub [11].

Our implementation for maintaining local and global address maps is thread-safe, by employing the Akka actor-based, message-driven concurrency model with

[5] Akka is also written in Scala

```
case class IDIPpair(val id: String, val ip: String)
case class addAddrMsg(idip: IDIPpair)
case class rmAddrMsg(idip: IDIPpair)

class localAddrMap extends Actor {
  val addrMap: Set[IDIPpair] = Set[IDIPpair]()
  val lastAddrMap: Set[IDIPpair] = Set[IDIPpair]()

  def receive: Receive = {
    case addAddrMsg(idip) => {addrMap += idip}
    case rmAddrMsg(idip) =>{addrMap -= idip}
    case "getAddrMapMsg" =>{
      val rmAddrs = lastAddrMap diff addrMap
      val newAddrs = addrMap diff lastAddrMap
      lastAddrMap.clear()
      lastAddrMap ++= addrMap
      sender ! (rmAddrs, newAddrs)
    }
  }
}

object DockerContainer {
  ...
  val updateLocalAddrMap: ActorRef =
  ↪   actorSystem.actorOf(Props(new localAddrMap))

  def addAddr(idip: IDIPpair): Future[Unit] = {
    updateLocalAddrMap ! addAddrMsg(idip)
    Future.successful(())
  }
  def rmAddr(idip: IDIPpair): Future[Unit] = {
    updateLocalAddrMap ! rmAddrMsg(idip)
    Future.successful(())
  }
  def getDiffAddrMap(): (Set[IDIPpair], Set[IDIPpair]) = {
    val timeout = Timeout(Duration(1, TimeUnit.SECONDS))
    val futureRes = updateLocalAddrMap ? "getAddrMapMsg"
    val diffAddrs = Await.result(futureRes,
    ↪   timeout.duration).asInstanceOf[(Set[IDIPpair],
    ↪   Set[IDIPpair])]
    diffAddrs
  }
  ...
}
```

Figure 5. Address map maintenance in Pikachu.

the Scala Future library. Specifically, all necessary data structures and methods are encapsulated into a class inheriting from the Akka Actor class, and use Akka's message passing to invoke these methods and obtain results. Figure 5 shows how we maintain the local address map on individual invoker[6]. We declare the localAddrMap class for maintaining local address map, and use Akka symbols "!" and "?" to send methods invoking messages in the DockerContainer object.

[6]For conciseness, we use Set to denote scala.collection.mutable.Set in Figure 5

We retain most of the implementation logic of OpenWhisk, except that we make the following two changes:

1. **DockerClientWithFileAccess.scala:** We inspect the IP addresses of new created containers using Docker commands instead of reading from container configuration files. We make this change because these configuration files do not contain the overlay network IP addresses of containers.
2. **DockerClient.scala:** We explicitly connect all new created containers to the Docker overlay network that we created. The reason is that Docker only allows containers to attach to one network during creation time. We choose to first attach containers to the default bridge network during creation time for communicating with invokers, then explicitly (*i.e.*, the connectOverlay() function) attach them to the overlay network.

6 Evaluation

6.1 Experiment Setup

We conduct all the experiments on a cluster of five bare-metal servers from CloudLab, each with Ubuntu 16.04.6 LTS (4.4.0-145-generic kernel), 160GB memory, two Intel E5-2660 v3 10-core CPUs at 2.60 GHz, and a dual-port 10G Intel X520 NIC. We deploy Pikachu (or OpenWhisk) on this cluster: one server hosting controller, crouchDB, redis, zookeeper, nginx, and kafka; each of the other four servers hosting one invoker.

We write a Fibonacci calculation function in Python to test the overhead of Pikachu compared with the original OpenWhisk by varying the number of concurrent invocations (§6.2). We randomly choose function names from a name pool to make sure the function invocations are distributed to the four servers evenly, since OpenWhisk uses hashes on function name to distribute invocations among invokers. We calculate the average completion time of concurrent invocations. We run 10 trials, and show results in median value with 5th and 95th percentile error bars.

We compare socket-based message passing in Pikachu vs. message passing via Redis in the original OpenWhisk, by varying the message size and message number (§6.3). Both testing functions for message passing are written in Python using socket package and redis package, respectively. By message passing, we mean that one function instance as a client wants to send messages to another function instance as a server. We randomly generate the function names of function

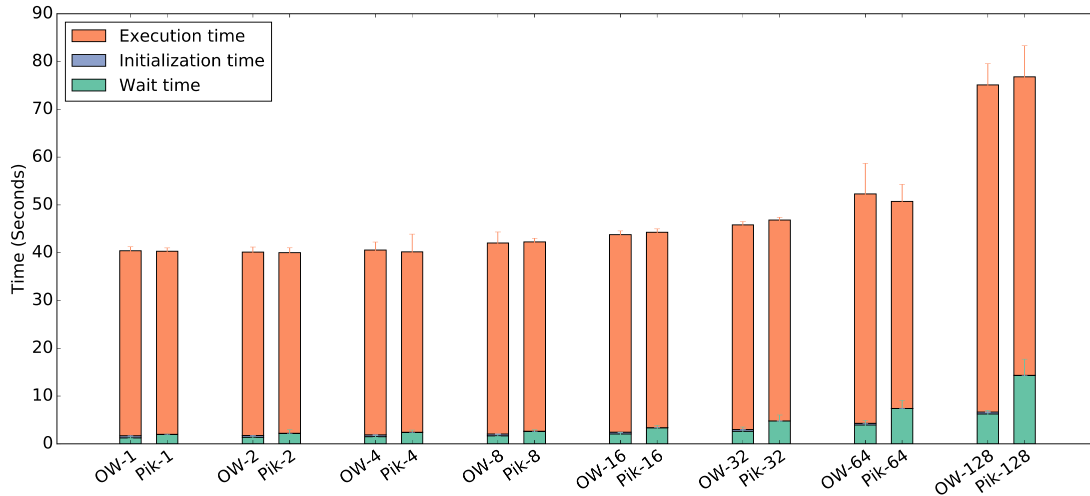

Figure 6. Function instance completion time breakdown, when varying the number of concurrent invocations. Here, the serverless function simply calculates the 39th Fibonacci number in Python. We compare the original OpenWhisk (*i.e.*, OW) with Pikachu (*i.e.*, Pik) in terms of invocation wait time, container initialization time, and function instance execution time.

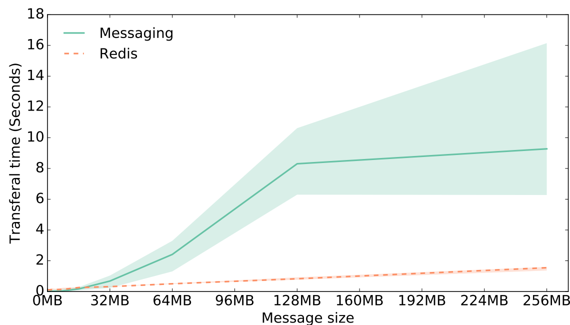

Figure 7. Data transferal time comparison of Pikachu vs. using Redis, when varying the message size.

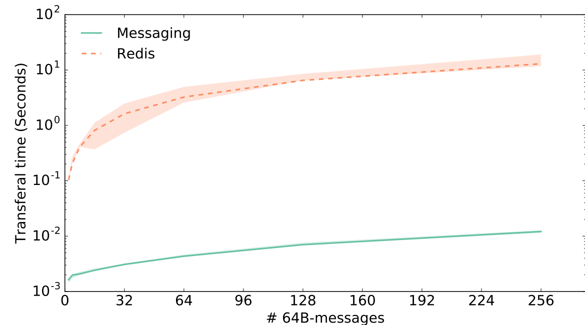

Figure 8. Data transferal time comparison of Pikachu vs. using Redis, when varying the number of 64B-messages sent.

pairs to test different co-location cases (*i.e.*, client and server are in the same server or different servers). All experiments are run in 10 trials, and results are shown in median value with 5th and 95th percentile error bars.

6.2 Overhead of Pikachu

Figure 6 shows the function completion time comparison of the original OpenWhisk vs. Pikachu, by varying the number of current invocations. Here, wait time refers to the time spent waiting in the internal OpenWhisk system, or more preciously, the time spent between the controller receiving the activation request and when the invoker begins to provision a container for this action. Initialization time refers to the time of creating containers, and injecting code and parameters (there is no container creation time in warm-start cases). Execution time refers to the time spent on executing the serverless function.

We find that *Pikachu increases function instance wait time by* 55.8$ ~ 128.9%, since attaching containers to overlay network and inspect their addresses bring additional overhead. We also find that *the function execution time remains almost the same or even slightly better* than the origin OpenWhisk (*i.e.*, 0.87% ~ 9.7% decremental). The reason is that the extent of overlap of concurrently running instances is decreased; each

function instance can get relatively more CPU time and thus the execution time will shrink. *Pikachu increase the total completion time by* $-3.0\% \sim 2.25\%$ compared with the original OpenWhisk.

6.3 Comparison to using in-memory store

Figure 7 shows the data transferal time comparison of Pikachu vs. using Redis, while varying the message size. We find that: when message size is small (*i.e.*, smaller than 16MB), the data transferal time of Pikachu is up tp 2 times shorter than the original OpenWhisk with Redis; when message size is large (*i.e.*, larger than 16MB), the data transferal time of Pikachu is up to 9.03 times longer than the original OpenWhisk with Redis. We think the reason comes from two parts: 1) overlay network relies on the host kernel to transfer data; larger volume of data would incur the buffer pressure in kernel, leading to worse performance; 2) Redis should have optimized the I/O operation during data transferal, while we only use simple socket send() and recv() due to project time limit.

Figure 8 shows the data transferal time comparison of Pikachu vs. using Redis, while varying the number of small messages sent. We find that *the data transferal time of Pikachu is* $62.4 \sim 1070.4$ *times shorter* than the original OpenWhisk using Redis. We consider the huge benefits as normal cases, since 1) Pikachu avoids the socket initialization overhead each time function instances send messages; 2) Pikachu avoids the double-copy overhead of message passing compared with using Redis. Note that Pikachu achieves these huge benefits without relying on any dedicated long-running server.

7 Discussion

We describe several points that currently are not supported by our system and serve as future work.

Exposing more function instance states via the directory service to serverless platform and applications, thus assisting instance scheduling and application coordinating. Currently Pikachu only supports exposing the survival status of each function instance; however, we believe that exposing more states about function instance could help the serverless platform schedule instances more efficiently and allow applications to coordinate different instances more effectively. For instance, one function instance might tell the platform that it is waiting for other instances and has no work to do currently. The platform then could pause the instance, store its state to the disk, and once the wait condition is satisfied, restore the instance to memory and wake it. An evaluation of the disk resource overhead of pausing instances would be required to determine billing adjustments (perhaps paused and stored instances would incur a smaller fee than running instances). Overall, though, this would make the fine-grained billing feature of serverless more flexible and let more applications fit in the serverless computing model. For another example, one function instance might announce its estimated time to be killed to its peers, which would give an application another vector to assign an appropriate amount of work to that instance (*i.e.*, better coordination for applications).

Generalizing Pikachu to other platforms. Pikachu is currently built on top of OpenWhisk, an open source project. We hope to materialize the spirit of direct message passing in more severless platforms, *e.g.*, Amazon Lambda, Azure Functions, and Google Cloud Functions.

Supporting multiple controllers. Pikachu currently only supports one centralized controller and we hope in the future to extend it to support multiple controllers. To this end, we might need to implement gossiping protocols for multiple controllers to synchronize their global address maps. We expect to see a (new) gossiping protocol with fast convergence speed, considering the ephemerality of serverless function instances.

Fault-tolerance when the controller crashes. Pikachu only considers the case of invoker crashing. However, in the context of multiple controllers, we need to consider the case of controller crashing. In particular, we need to guarantee the correctness of the global address map when some controller crashes.

Other fault-tolerance cases. Serverless platforms do not guarantee that every function instance survives for the intended duration given its code. numpywren discusses a fault-tolerance system with a task lease mechanism, but in numpywren, the "disaggregation of compute and storage" makes this easier to achieve [19]. With Pikachu , checkpointing and recovery becomes much more difficult, because intermediate data within an algorithm only kept in transient storage. Further, it becomes incredibly difficult or even impossible for a serverless system to generically determine if a function instance has made only idempotent external changes, meaning that blindly restarting crashed function instances is unsafe. Of course, it is possible for a user to implement an algorithm that includes manual checkpointing, but keeping with the spirit of serverless, we

believe a proper HPC serverless offering would have some level of automatic fault-tolerance and at the very least would also expose an API so the user could detect and handle crashed function instances. We leave this functionality for future work.

Handling the Docker overlay network bug. Docker currently has a bug that, when simultaneously attaching too many (around 100 according to our testing) containers within a host to the overlay network, will cause a "Context deadline exceeded error" [7]. We currently do not handle this Docker bug in Pikachu.

8 Related Work

8.1 Big Data Processing on Serverless Platform

PyWren [15], numpywren [19], and Ex-Camera [13] all similarly tackle the problem of moving a large amount of data between workers in highly parallel tasks. However, all of these solutions require the usage of storage on long-running servers to move data between function invocations.

PyWren uses S3 to deliver an entire pickled python program to an already-running function instance to extend the limitations of AWS Lambda. This means that PyWren can spin up many lambdas and send somewhat arbitrary Python code to them to be executed. PyWren sees significant time improvements to invocation, computation, S3 read/write, and Redis read/write as more lambdas and Redis shards are started prior to computation. The S3 and Redis read/write improvements are particularly interesting with regards to Pikachu 's goals, however we still believe that the departure from the spirit of serverless is a disappointing trade-off in PyWren. Further, the PyWren paper's section on the generality of PyWren is thorough, the design prohibits coordination among the various tasks, a problem easily solved when function instances can directly communicate with each other. Lastly, PyWren's functionality is dependant on Python's pickling ability, meaning this design cannot be easily extended to other languages or runtimes, which of course is extremely restrictive.

numpywren provides a linear algebra system for use on serverless platforms, in particular on AWS Lambda. numpywren uses Redis for application state storage, AWS' Simple Queue Service (SQS) for a task queue, and S3 for object storage. Although numpywren does not perform very favorably compared to the domain specific language (LAmbdaPACK) the paper's team also developed, the proof-of-concept that numpywren achieves is still promising for other high computation workloads on serverless platforms. Though numpywren's goals are not congruent with Pikachu 's goals, the overall design of that project has still been influential for Pikachu. Still, numpywren's design and usage of S3 and Redis in particular suffer from latencies that Pikachu seeks to resolve.

Ex-Camera utilizes S3, a long-running server called the coordinator, and a long-running rendezvous server. Although at the time of Ex-Camera's publication, the cost to encode a 15 minute movie is only $5.40, they do admit that some function instances are idle at the beginning of their invocation, and cost could be saved without sacrificing performance by delaying starting these instances. More importantly, however, is that Ex-Camera still requires the usage of S3 and and two servers running on an AWS EC2 instance. This means there are costs incurred with every operation on S3, which is the primary method of data transferal in Ex-Camera, and costs of running the EC2 instance. Not only does Pikachu seek to avoid requiring the user to implement a separate long-running server by extending the functionality of serverless functions, but the double-billing as a result of Ex-Camera's design given the current capabilities of serverless offerings is unappealing.

8.2 Storage System for Serverless Computing

Pocket [17] seeks an improvement for ephemeral storage which is more suitable for the very short-running functions on serverless platforms. Although S3 and Redis have been shown to be somewhat suitable to implement big data processing onto the current AWS Lambda platform, they incur large latency penalty (Redis-like inmemory store in the cloud, *e.g.*, ElasticCache, is quite expensive in serverless context). Pocket is able to achieve similar performance to Redis for serverless analytics applications while reducing costs by 60%. Pocket utilizes AWS EC2 instances to run their controller and cluster storage, and is probably the best way to replicate the behavior of Redis while being cost optimized for serverless workloads when the existing serverless platform cannot be modified. However, while Pocket shows an improvement in existing object stores for the ephemeral functions in serverless platforms, Pocket still has drawbacks that Pikachu seeks to resolve. For one, Pocket does not provide a way to send data directly between function instances, so a copy of the data has to be duplicated in the storage cluster, even if that data

is never used again. Secondly, Pocket still requires dedicated long-running servers for hosting storage system, although the paper argues that this overhead can be paid by the cloud. Last, because Pocket is just an object store, applications using Pocket would still have to translate the algorithm to utilize the data store, which isn't always natural, and also lacks the internal coordination abilities that message passing provides.

9 Conclusion

Pikachu opens the door for direct messaging passing among function instances in serverless computing. Pikachu utilizes the Docker overlay network to enable serverless function instances directly communicate with each other. It further integrates a highly scalable directory service into serverless platform itself to solve the addressability problem among function instances. The experiment results show that Pikachu incurs minimum overhead to serverless platform but brings enormous benefits for applications.

Acknowledgments

We would like to thank Eddie Kohler for his insightful comments and suggestions on the project along the semester, and also his great contributions to CS260r that make it full of enlightenment and fun. We also would like to thank Xinyin Song for helping setting up Ceph cluster, although we eventually have not got time to test on it.

References

[1] [n. d.]. Akka. https://akka.io/. Accessed May 12, 2019.
[2] [n. d.]. Apache Kafka. https://kafka.apache.org/. Accessed May 12, 2019.
[3] [n. d.]. Apache OpenWhisk. https://openwhisk.apache.org/. Accessed May 12, 2019.
[4] [n. d.]. Apache Zookeeper. https://zookeeper.apache.org/. Accessed May 12, 2019.
[5] [n. d.]. Docker. https://www.docker.com/. Accessed May 12, 2019.
[6] [n. d.]. Docker container networking. https://docs.docker.com/v17.09/engine/userguide/networking/. Accessed May 13, 2019.
[7] [n. d.]. Docker overlay network bug. https://success.docker.com/article/context-deadline-exceeded-error-observed-while-starting-container-on-drained-node. Accessed May 15, 2019.
[8] [n. d.]. Docker volume. https://docs.docker.com/storage/volumes/. Accessed May 13, 2019.
[9] [n. d.]. IBM Cloud Functions. https://www.ibm.com/cloud/functions. Accessed May 12, 2019.
[10] [n. d.]. Nginx. https://www.nginx.com/. Accessed May 12, 2019.
[11] [n. d.]. Pikachu open source. https://github.com/YangZhou1997/openwhisk-lambda-mpi. Accessed May 17, 2019.
[12] [n. d.]. Redis. https://redis.io/. Accessed May 12, 2019.
[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. 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)*. 363–376.
[14] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review*, Vol. 39. ACM, 51–62.
[15] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.
[16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).
[17] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 427–444.
[18] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 193–206.
[19] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).
[20] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim:{OS} Kernel Support for a Low-Overhead Container Overlay Network. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 331–344.