Conference Paper

Boxer: Data Analytics on Network-enabled Serverless Platforms

Author(s):
Wawrzoniak, Mike; Müller, Ingo; Fraga Barcelos Paulus Bruno, Rodrigo; Alonso, Gustavo

Publication Date:
2021-01

Permanent Link:
https://doi.org/10.3929/ethz-b-000456492 →

Boxer: Data Analytics on Network-enabled Serverless Platforms

Michal Wawrzoniak[1] Ingo Müller[1] Rodrigo Bruno[2*] Gustavo Alonso[1]

[1]Systems Group, Dept. of Computer Science, ETH Zurich
{michal.wawrzoniak,ingo.mueller,alonso}@inf.ethz.ch

[2]Oracle Labs, Switzerland
rodrigo.b.bruno@oracle.com

ABSTRACT

Serverless is an attractive platform for a variety of applications in the cloud due to its promise of elasticity, low cost, and fast deployment. Instead of using traditional virtual machine services and a fixed infrastructure, which incurs considerable costs to operate and run, Function-as-a-Service allows triggering short computations on demand with the cost proportional to the time the functions are running. As appealing as the idea is, recent work has shown that for data processing applications (regardless of whether it is OLTP, OLAP, or ML) existing serverless platforms are inadequate and additional services are needed in practice, often to address the lack of communication capabilities between functions. In this paper, we demonstrate how to enable function-to-function communication using conventional TCP/IP and show how the ability to communicate can be used to implement data processing on serverless platforms in a more efficient manner than it was possible until now. Our benchmarks show a speedup as high as 11 × in TPC-H queries over systems that use cloud storage to communicate across functions, sustained function-to-function throughput of 621 Mbit/s, and a round-trip latency of less than 1 ms.

1. INTRODUCTION

Serverless (i.e., Function-as-a-Service, FaaS) platforms in the cloud [7, 16, 23] or for data centers [6] are becoming widely available as an alternative to running a fixed infrastructure based on either virtual machines (VMs) or X-as-a-Service (e.g., Query-as-a-Service systems, QaaS, like Amazon's Athena [3] or Google's BigQuery [15]). The appeal of serverless resides in its ease of use, extreme elasticity, and fine-grained usage-based pricing, thus promising overall lower costs [1, 2, 8, 9, 10, 11, 26]. These properties come to the fore especially for occasional or heavily fluctuating workloads, where the idle time of over-provisioned VMs can dominate the time spent on useful work.

*Work done while at ETH Zurich.

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.
DOI: 10.3929/ethz-b-000456492.

Even though serverless functions were not originally designed for this use case, a large number of potential uses have been proposed for job-oriented applications. In such a setup, serverless functions are treated as workers in a (virtually) infinite cluster of machines that can work at a massive scale (typically using several thousand cores). They can thus complete even larger jobs in an interactive time frame, while incurring cost only for the job execution itself. This promises to significantly outperform VM-based solutions, which would be either heavily under-utilized between jobs or too slow to start on a per-job basis [25]. Examples include MapReduce-like applications [19], video encoding and processing [14], distributed compilation [13], and data analytics [25, 27]. Similarly, stateful latency-sensitive applications such as web or model serving have also been explored [30], where the elasticity of serverless promises to reduce the need for over-provisioning by large factors.

However, current offerings of serverless functions come with limitations that make it impossible to exploit the full potential of serverless computing, in particular, for the applications mentioned above [17, 24]: (1) functions are short-lived and cannot be readily used for long-running computation such as complex analytical queries [13]; (2) functions cannot directly communicate with each other, making the exchange of intermediate data cumbersome [21, 25, 27]; (3) starting many functions induces a considerable delay as existing platforms either limit the speed at which new functions can be spawned [23] or have a limit on how many functions can be active at any point in time [22]; and (4) there are limited caching possibilities within functions, forcing data to be read over and over again from storage when processing complex queries unless an external caching service is provided [33].

Consequently, a large amount of effort has been spent to develop application-specific work-arounds for these limitations. In particular, several alternatives for direct network communication have been proposed, including (1) communication through cloud storage [25, 27], (2) purpose-built storage layers for ephemeral data [21], (3) dedicated virtual machines for exchanging intermediate results [19], and (4) rerouting messages through a central VM-based proxy [13, 14, 31]. While this makes direct network communication possible, these work-arounds incur a high latency and monetary cost, re-introduce fixed infra-structure, or do not scale to communication-intensive applications.

In this paper, we present *Boxer*, a system enabling direct function-to-function communication in existing public cloud infrastructure. To enable serverless applications to commu-

nicate through a conventional TCP/IP network stack, Boxer uses TCP hole-punching techniques related to Peer-to-Peer systems [12, 18] to circumvent the network constraints of current serverless offerings.

We show the potential of Boxer by extending our own work on data processing based on serverless functions, Lambada [25], with support for TCP/IP-based communication. The new implementation is not only simpler and less specific to a particular environment, it also significantly outperforms the original version based on data exchanges through cloud storage: Most TPC-H queries are both faster *and* cheaper by a factor $4\times$ and $6\times$, with some queries enjoying an improvement as high as $11\times$ and $15\times$ in running time and cost reduction, respectively.

These observations are encouraging and show the potential of network-enabled serverless platforms. Through Boxer, it becomes possible to explore many new architectural approaches to elasticity and large scale data processing. For instance, a database engine running on a fixed-size cloud deployment (a VM of a given size) could react to sudden load spikes quickly by moving part of the load to serverless functions. Such an approach exploits the conventional engine for the regular load and takes advantage of the elasticity of serverless to spawn occasional jobs that increase the overall capacity without requiring to provision the system for the peak load. We expect Boxer to enable many such designs, leading to hybrid approaches combining conventional cloud deployments and networked serverless functions.

2. BACKGROUND AND RELATED WORK

In what follows we focus on the use of serverless for data processing as discussed in, e.g., [17, 25, 27, 32], and do not consider other use cases that might impose different requirements and/or constraints.

2.1 Overview of Serverless Platforms

Serverless platforms allow developers to decompose applications in small logic units (functions) triggered when specific events occur, e.g., files are uploaded to storage, REST API calls, etc. Developers upload the function code/binary and configure how it should be activated. Then the serverless platform is responsible for instantiating the function execution runtimes, deploying them, and determining the scalability factor for each function, i.e., how many function execution environments to host for each function. Besides simplifying application development and deployment, serverless offers a lower startup time than virtual machine based deployments [1, 2, 8, 9, 10, 11, 26], thereby achieving high degrees of parallelism in a fraction of the startup time. This is useful for occasional, interactive use [25, 27].

2.2 Data Shuffling

Since functions cannot communicate directly with each other, applications that involve heavy data processing have to resort to a variety of mechanisms to shuffle data among functions: Amazon S3 [4], messaging queues such as Amazon SQS [5], using hand-tuned VM-based resources to exchange data [21, 28, 33], or implementing data exchange operators that rely on cloud storage to pass the data [25, 27]. Such solutions incur at least one of these two problems: i) cloud-based storage latency is at least two orders of magnitude higher than the sub-millisecond latency of point-to-point communication in data center networks (further aggravated by the fact that data shuffling requires at least two requests, one to read and one to write); ii) VM-based solutions force developers to manually manage cloud resources, partially defeating the purpose of using serverless.

Previous efforts have tried to ameliorate the data shuffling problem by tailoring storage services so that they can be used to exchange data. For example, Klimovic et al. [21] have designed an elastic, fast, and fully managed storage system for ephemeral data. Similarly, Pu et al. [28] have built a system for intermediate data that uses a combination of AWS ElasticCache and AWS S3. Built with a focus on transactional workloads, Anna [33] is an elastic caching system built on top of a pre-existing key-value store [32] that can be used to implement stateful functions [30] and transactional causal consistency for serverless functions [34]. In all these cases, the effort is complicated by the difficulty in determining what is the optimal storage strategy for data analytics [20] and the wide range of incompatible options among the offerings available [33].

For data analytics, the alternative to implementing such extra services is to exchange data through cloud storage using an exchange operator writing to and reading from permanent storage services. In Starling [27], intermediate data is exchanged between functions using Amazon's S3 [4]. The high latency of the service is hidden by using parallel read requests per function and, for actual data shuffling requiring *all-to-all* communication, the ability to read only a fraction of a file that S3 provides is used so that the amount of data to be moved is minimized. Lambada [25] also uses S3 but uses a more sophisticated exchange operator (see below). These two systems have the advantage of not requiring any external service but, despite their efforts to minimize the overhead, their performance is still limited by that of S3: both systems show that data analytics over serverless today is only cost-efficient at a low query throughput.

2.3 Networking in Serverless

The possibility of allowing direct communication between functions has been previously suggested [13] but without any concrete implementation nor details of how this could be achieved. Attempts at implementing such a service have been made [29] but using specialized APIs and network protocols. Existing examples of the proposed approach cover mostly communication between two functions rather than generalized networking capabilities.

Lacking the ability to establish direct communications, current systems exploit the ability of functions in Amazon Lambda to establish outgoing TCP connections. For instance, InfiniCache [31] provides functions with the address of a proxy at start time. The function then uses that address to establish a TCP connection to the proxy. When clients want to contact the function, they contact the proxy, which then relays the message to the function and eventually forwards the answer. A similar design, involving an external coordinator, is used in [13, 14] for the same purpose. Enabling communication through an intermediary service can become a scalability bottleneck, increases the round trip time for all requests, requires additional infrastructure, and does not allow direct communication between functions as needed for parallel query processing. For data processing, where the amount of data being exchanged can be large, such a multi-staged exchange operator is an inefficient approach due to the amount of data copying involved.

3. LAMBDA-TO-LAMBDA COMMUNICATION

In this section we describe *Boxer*, a system enabling network communication across serverless functions. Boxer leverages TCP/IP, a reliable data stream protocol, and standard sockets, to allow functions to communicate directly with each other without an intermediary service. The current implementation runs on AWS Lambda, which we have chosen for the first prototype as it supports more programming languages and has higher service limits than the other commercial offerings from Microsoft and Google [27].

3.1 Network Address Translation (NAT)

In AWS Lambda, functions execute behind a NAT. NAT (Network Address Translation) is a common networking technique used in data center networks composed of many isolated IP address spaces. NAT devices map addresses from one address space to another to provide transparent routing between hosts in different address spaces. In general, hosts behind a NAT can initiate connections to routable hosts outside of their NAT, however, they cannot accept new incoming requests. Lambdas are thus allowed to initiate new connections to outside services, but cannot accept incoming requests. Functions have their local private address spaces, and NATs provide a transparent mapping to externally routable addresses used for communicating with external services.

To enable direct communication between functions, functions must receive network requests initiated by other functions. New network requests to functions have to traverse their NAT, or otherwise, they will be discarded. We found two methods to traverse AWS Lambda NATs, enabling lambda-to-lambda TCP communication. They can be seen as variants of classic techniques developed more than a decade ago by the P2P research communities. The first one is a special case of *parallel TCP hole punching* [12], and the second one is similar to *sequential TCP hole punching* [18]. We currently use both, for different purposes as described below.

3.2 Boxer Hole-Punching Service

Boxer is composed of three sub-systems: (1) the Boxer hole-punching service, (2) the Coordinator service, and (3) the Transparent Compatibility module. All three are deployed together with the function code and execute alongside each Lambda instance.

The Boxer hole-punching service runs in every function instance. It provides a function-local API to request NAT holes to be punched in NAT(s) of other function instances. This API is used by the transparent compatibility module (described below) to provide function to function TCP connection setup capabilities to unmodified applications.

The hole-punching service instances maintain control connections with each other. These connections are set up during initialization using a version of the parallel TCP hole punching protocol [12]. The control connections are used by hole-punching service instances to forward remote hole punching requests to the appropriate remote instances.

When a hole-punching service instance receives a request from a remote peer, this request corresponds to a remote function attempting to make a new TCP connection to the local function. In the basic configuration, the receiving hole-punching service uses a form of sequential TCP hole punching [18] to punch a hole in the NAT and then send an acknowledgment to the requesting hole-punching service. The TCP connection instantiation is then allowed to proceed as it will now be possible to traverse the NAT. Via this local hole punching interface, functions can punch holes in remote NATs with the agreement of the remote function and instantiate TCP connections to a remote listening function.

3.3 Coordinator Service

The Coordinator Service is responsible for tasks related to control and orchestration. At Lambda function instantiation time, the local Coordinator in the function contacts its seed Coordinator to join the system. The seed Coordinator is the same as the local Coordinator running in the functions, except the seed Coordinator is not instantiated behind a NAT, e.g., in a VM,—it can accept initial network connections from lambdas without the need to perform hole punching. The seed replies with the observed external address and begins to stream function membership updates. At this point, the local Coordinator can start the local hole-punching service described above. Currently, we rely on one seed Coordinator for propagating membership updates, but other designs are conceivable. The way the seed Coordinator operates is similar to the way proxies are used in InfiniCache [31] except that it forwards addresses of other functions so that the function can establish direct connections to them without having to rely on the proxy for all communications as it happens in InfiniCache.

The seed Coordinator may also reject the join request. Currently, there are two reasons why a join request can be rejected: (1) if the joining lambda's external address is already registered, (2) if there is already a sufficient number of lambdas participating. Both of these decisions can be seen as very simple resource allocation decisions.

Every Coordinator provides a local membership service allowing the application to learn of other participating functions incrementally, just by receiving updates from a local network connection. This feature matches the elasticity expected of serverless systems.

The local Coordinator is also responsible for starting the application inside a Lambda function, and loading the compatibility module described below. Depending on the application, the required environment and start conditions may differ. For example, currently, Lambada requires the list of participating addresses to be provided at start time so that it can exchange intermediate data for query processing. In this case, the Coordinator delays the execution of the function until all necessary participants have joined and unique worker IDs have been assigned. It then starts the execution of the function application. This can be seen as a very simple distributed coordination service.

3.4 Transparent Compatibility Module

The AWS Lambda runtime environment is based on Linux. Typical Lambda functions (or the libraries they leverage) target that interface, including the socket interface for network functionality exposed by `libc` system library. To support the execution of such unmodified binaries that can perform function to function networking using Boxer, we deploy a small shim interposition library and instruct the dynamic linker to link all applications with it when they are started (using the `LD_PRELOAD` environment variable). The shim intercepts appropriate `libc` socket function calls. When the

	Median	Mean	Std Dev	Min	Max
Latency	531 µs	629 µs	318 µs	208 µs	1442 µs
Throughput	627 Mbit/s	621 Mbit/s	23.4 Mbit/s	552 Mbit/s	684 Mbit/s

Table 1: Function to function round-trip TCP latency (1 KiB message size) and throughput using Boxer.

	Lambda S3	Lambda Boxer-TCP	EC2 t2.large	EC2 t3.large	EC2 m4.large
Avg. latency	108 ms [17]	**629 µs**	335 µs	205 µs	186 µs
Slowdown	581 ×	**3.4 ×**	1.8 ×	1.1 ×	1 ×

Table 2: Network latency comparison across different cloud deployments and using S3 to exchange data.

function application wants to establish a TCP connection with another function, instead of directly calling the `libc` system library implementation that then issues system calls to the Linux kernel to perform the connection setup, the interposition module intercepts the request. It first uses the Boxer hole-punching service on behalf of the application to punch the necessary NAT holes and only then forwards the original request to the system libraries that then issue system calls to the Linux kernel. The application is unaware of this interception and proceeds to use the conventional socket interface.

3.5 Network Measurements

To explore the function to function TCP networking enabled by Boxer, we measured TCP throughput and latency between pairs of AWS Lambdas. The experiments were conducted in the AWS us-west-2 region with the Lambda function environment with 3008MB of memory.

We measured TCP throughput between pairs of Lambdas by running an unmodified iperf3[1] network measurement tool. We chose to use iperf3 because it is a popular and well-maintained tool aiming to measure throughput achievable by regular user-space applications. We invoked 48 concurrent Lambda instances, each running unmodified iperf3 using Boxer. Each Lambda instance was paired with one other Lambda instance; one running iperf3 in server mode and the other in client mode. Once an iperf3 pair established a TCP connection the server receive throughput benchmark was run for the 15 minutes. We recorded the measured unidirectional throughput at one-second granularity over the 15-minute period (the maximum lifetime of a Lambda function) of the 24 Lambda instance pairs. We observe an average sustained throughput of 621 Mbit/s without noticeable degradation over the entire 15-minute time period. Table 1 presents statistics of 1-second throughput measurements for all functions, discarding measurements from the first 15 seconds of the experiment. During the 15 minute experiment, for all active TCP connections, the minimum throughput achieved over any 1-second window was 552 Mbit/s showing the throughput was sustained over the lifetime of the connections. Throughput achieved during the initial period of the experiments was significantly higher, which is why the reported steady state statistics do not include first 15 seconds of the experiment. The average throughput during the initial 3 seconds of the experiment was 1008 Mbit/s, median was 644 Mbit/s and maximum throughput recorded over 1-second window was 1802 Mbit/s. After this initial burst of available throughput, the steady state reported in Table 1 is reached.

We also measure the TCP latency between pairs of Lambdas using a ping-pong message exchange program running in each Lambda instance. After instantiating 64 Lambda instances and pairing each instance to one other instance, the test program starts in server mode on one Lambda of the pair and in client echo mode in the other Lambda. After the echo client establishes a TCP connection to the server, the server starts a round of 1000 ping-pong exchanges with message size of 1 KiB. The server measures the total time of the round, repeats for 100 rounds and reports the final results, presented in Table 1. The average TCP round trip latency between two Lambdas is 629 µs and median latency of 531 µs. It should be noted that the level of dispersion in the latency measurement is not insignificant, and is contributed to by two main factors. First, latency outliers during the lifetime of one TCP connection, and second, the average latency between Lambda pairs varies between pairs, perhaps reflecting different placement of Lambda instances in the datacenter network.

Table 2 shows average TCP latencies measured using the same measurement tool described above, except not run between pairs of Lambda instances but between pairs of three EC2 virtual machine instance types. As above, the amount of data exchanged in all cases is 1 KiB. The comparison of the results shows that Lambda to Lambda TCP latency using Boxer has a function to function networking latency that is comparable to that of VM-to-VM networking, and significantly (172 ×) lower than that reachable when implementing communication through S3 as it is done in existing data processing systems over serverless [17, 25, 27]. For the numbers reporting the overhead of communication through S3, we use the figures provided in [17].

4. EVALUATION

4.1 Overview of Lambada

To evaluate Boxer in an end-to-end scenario, we use and extend Lambada [25], a data analytics system built for serverless functions that relies on S3 to exchange data between functions using an exchange operator that is among the most efficient published so far.

In Lambada, only a small component of the system referred to as *driver* resides on the laptop or workstation of the user. It consists of the user interface, query optimizer, and compiler, as well as part of the query execution layer. All other components are serverless components that are

[1] Version 4.19.76 fac957a0b5c3

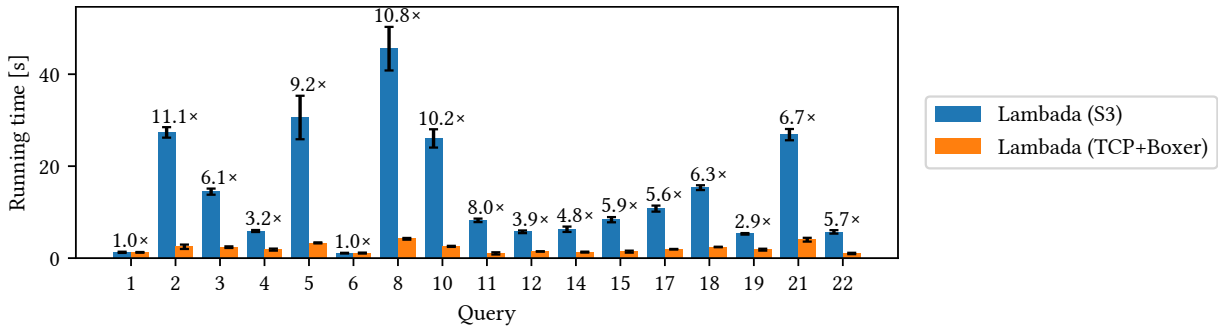

Figure 1: Running time of Lambada on TPC-H using TCP+Boxer or S3 for communication.

fully managed by the cloud provider. In particular, Lambada uses serverless functions as workers of the execution layer, which communicate among themselves and with the driver through shared serverless storage, i.e., a combination of message queues (*SQS*), cloud storage (*S3*), and key-value stores (*DynamoDB*).

The driver translates queries into executable plans and starts executing them. After some potential pre-processing, it launches a potentially large number of serverless functions in order to execute data-parallel plan fragments representing the bulk of the work. Queries typically read one or more input tables from cloud storage and either store the result again on cloud storage or return their intermediate results to the driver, which may post-process them before returning them to the user. For more details about Lambada, we refer to the original paper [25].

4.2 Baseline

In the original version, which we take as the baseline, Lambada uses serverless cloud services for indirect communication among the workers. In particular, it implements operators that need to shuffle data among the workers such as joins, grouping, reduction, and sorting, using a purpose-built exchange operator that communicates through files on cloud storage. The main idea is to write all data that needs to be sent from a particular worker to another one into a file whose name is based on the two worker IDs, such that the receiver can poll on that file until it exists and read its content. While this approach makes it possible for workers to communicate without a direct network connection, its basic version does not scale to a large number of workers: Since it involves a quadratic number of files (one for each pair of workers), it runs into service limits of the cloud storage system and incurs a quadratic request cost. To overcome this issue, Lambada does the exchange in two rounds, each within a sub-group of $\sqrt{P}$ of the P workers, which requires only $\mathcal{O}(P \cdot \sqrt{P})$ accesses to cloud storage, and combines all data sent by each worker into a single file, reducing the number of write accesses to just P.

4.3 Lambada + Boxer

To measure the impact of Boxer on query processing, we extend Lambada with a traditional TCP-based exchange operator that uses the interfaces provided by Boxer. This operator reads its upstream data, partitions it into buffers (one for each other worker), and sends each buffer to its target worker as soon as it is full using a regular TCP connection. In order to hide network latencies, we use asynchronous socket operations such that the latencies of many simultaneous messages overlap among themselves as well as with local processing. In order to reduce the number of connections, we reuse one connection among all operators in a query for every $\langle sender, receiver \rangle$ pair. Neither the design nor the implementation of this operator is specific or even aware of the serverless setup. In each worker, Boxer loads the interception library to Lambada's query processing engine with the mechanism described above, which transparently activates hole-punching for the connection setup and uses standard network sockets afterward.

We evaluate the two versions of Lambada on TPC-H at Scale Factor 10. As in the original paper [25], we generate the data in a dictionary-encoded form and formulate the queries against the dictionary codes in order to avoid implementing string support. We store the data in Snappy-compressed Parquet files on S3, totaling about 3.2 GB. For the workers, we use serverless functions of 2 GiB main memory, which gives them the timeslices of slightly more than one CPU. Lambada now supports more queries than in the original paper but still misses features to run the full benchmark.[2] We use 64 workers and a single-level exchange for both variants, which simplifies the setup and interpretation of the experiments. In order to isolate the effect of the communication mechanism, we add an artificial barrier at the beginning of each inner plan fragment (implemented using Boxer in both versions) and measure the time from the moment the first worker passed that barrier until the last worker completed its plan fragment. Except for using two different versions of the exchange operator, the plan fragments run by both versions are exactly the same. All numbers represent averages of five runs.

4.4 Query Execution Time

Figure 1 shows the running time of the two variants. As the plot shows, Lambada is consistently and significantly faster using direct communication via Boxer than using indirect communication via cloud storage. The numbers above the bars indicate the speed-up of the former over the latter. The performance difference is mostly in the order of $4\times$ and $6\times$ and has a strong correlation with the number of exchange operators in the data-parallel plan fragments: Queries 1 and 6 do not exchange any data and thus have the same running time with both variants as expected; Queries

[2]The missing features are unrelated to the network layer and are limitations of Lambada: Queries 9, 13, 16, and 20 need string or null support and Queries 7 and 20 need a broadcast operator or support for multiple stages.

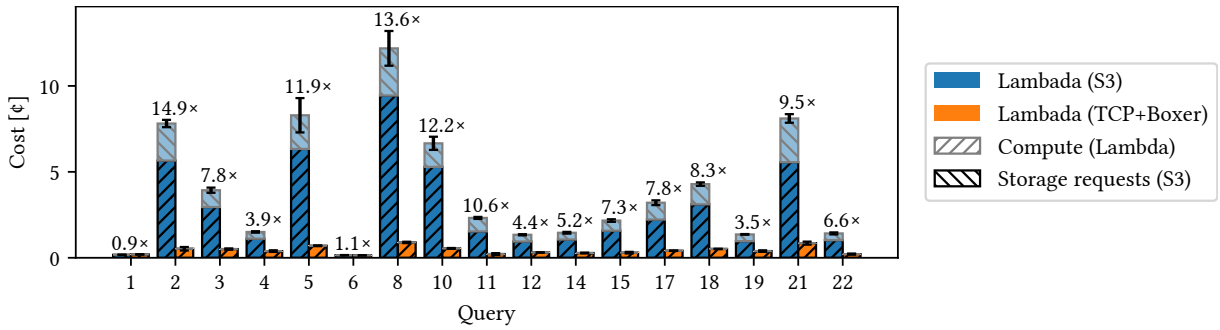

Figure 2: Monetary costs of Lambada on TPC-H using TCP+Boxer or S3 for communication.

4, 12, 14, 15, 19, and 22 have a single join, i.e., two exchange operators, or a join and an aggregation, each, so the speed-up of the two variants is mostly in the range of 4 × and 6 ×; Queries 3, 10, 11, 17, and 18 use four to seven instances of the exchange operator, so the TCP-based version tends to have a bigger advantage; and Queries 2, 5, 8, and 21 use ten or more exchange operators and are consequently the queries where the performance gap is the largest.

The performance difference comes from two main effects: First, while indirect communication requires the sender to complete *all* writing (per exchange operator) to cloud storage before the sender can start reading *any* data, the direct communication can be done in a streaming fashion using reasonably small chunks. Note that this is intrinsic to any storage system with a request limit or request cost (and thus to any real-world system with a usage-based pricing model we know of). This effect explains at most a performance difference of factor two assuming that sending and receiving can be perfectly overlapped. The second effect is due to the higher latency of accessing cloud storage compared to function to function communication via Boxer, which explains why the benefit of direct communication is larger for more complex queries. Note that this is also unlikely to change radically since cloud services typically have several layers of authorization, accounting, load balancing, etc. all of which increase the latency of accessing storage compared to sending the same amount of data through the network.

4.5 Monetary Query Cost

We also analyzed the costs of the CPU time used by the workers running in serverless functions as well as the request costs to cloud storage, which are mainly done by the exchange operator. The only other costs of running the query itself are (1) request cost to the serverless functions and (2) storage cost of temporary results on cloud storage, both of which are so small that they would not be visible on the plot. Except for the coordinator, there is no cost of setting up infrastructure or keeping it running between queries.

Figure 2 shows the costs of running the same queries as above. The numbers above the bars indicate the cost reduction of using communication through Boxer compared to using cloud storage. The numbers are strongly correlated to the running time: using Boxer reduces the costs by a factor between 4 × and 6 × for most queries and significantly more for the more complex ones. This is not surprising since the costs are dominated by the CPU time of the serverless functions. However, the request costs of accessing S3 have a significant share as well (GETs and PUTs to S3 are not for free and the total cost accumulates due to the many accesses needed to implement an exchange operator for parallel query processing); often in the order of a quarter and up to a third (for Query 21). Interestingly enough, the request costs alone are higher than the total costs using Boxer for almost all queries. Since the number of requests increases quadratically[3] with the number of workers, the cost-benefit of using Boxer will be even more apparent at larger scales.

5. FUTURE DIRECTIONS

Boxer enables building useful applications today at much better performance and higher efficiency than previous serverless approaches. As an enabler for research on future serverless systems, the following are interesting directions that Boxer opens up:

Deployment mechanism. How should individual components of a data-intensive application be deployed? Apart from VMs with relatively high (but steadily decreasing) start-up times, cloud providers also offer container-based deployments for arbitrary applications (e.g., AWS Fargate) and serverless functions (either deployed via special function packages or as containers as well). While the latter have by far the lowest start-up time, they currently come with the limitations discussed throughout the paper. Thanks to Boxer, all existing deployment mechanisms can be compared without having to purpose-built indirect communication primitives, which will help in deciding which one is more appropriate for a given application.

Systems for wide, short-lived jobs. As the abundant work on serverless computing has shown, the increasing elasticity of the cloud makes it possible for everyone to use thousands of cores for jobs that take just a few seconds to complete. These extremely wide but short-lived jobs create challenges for many system components, even if direct networking among the workers is possible. In particular, failures or at least delays become increasingly likely with the "width" of a job but must be dealt with at extremely short time scales. This is true for the start-up of large numbers of workers, the connection setup among these workers (which requires quadratic overall work for a fully connected network), the efficient handling of large numbers of connections at each worker, and many more. Solutions could include sparser connection topologies, such as the grid-based topology we used in Lambada's two-level exchange operator,

[3]As we have shown in prior work [25] and discussed above, sub-quadratic algorithms exist but at the expense of more phases which increase running time and, thus, costs as well.

or hedged computations and/or communication, i.e., redundant executions of latency-critical phases where only the first to complete contributes to the final result. Such optimizations can be easily combined with precise query plans to come up with better, less demanding approaches to deploy many functions to compute a query as the communication pattern within a query can be determined in advance and does not always need to involve a fully connected network topology (probably trees are a better, more natural option for query processing).

Elasticity for long-lived applications. While most work on serverless concentrates on one-off jobs [24], we envision the extreme elasticity of serverless to be useful also for long-running applications with fluctuating resource requirements. One example is a stream processing engine experiencing a load spike. This type of engine is usually able to adapt to changes in the input stream volume but typically at the time scale of minutes. With serverless functions, a large number of additional resources are ready to be used less than a second after detecting the spike, so systems could react more promptly. This could be done *only* using serverless functions or in a *hybrid* VM/serverless setup where serverless resources are used for short spikes or as a way to bridge the time until more cost-efficient resources are started. Both variants would allow applications to reduce over-provisioning, the traditional way to deal with sudden spikes.

Limitations of Boxer. Currently, Boxer is at the prototype stages and much remains to be done to increase its functionality, stability, and make it more efficient. There are also quite a few interactions with the underlying serverless platform that need further investigation. For instance, the time it takes to establish all connections among a group of functions varies greatly for reasons that are not yet clear. As another example, the bandwidth available is surprisingly stable for point-to-point connections but more experiments are needed to explore how the available bandwidth scales with the number of connections.

6. CONCLUSION

In this paper we have presented Boxer, a system enabling network communication for serverless functions. We have tested Boxer by implementing queries form TPC-H in a version of Lambada [25] and compared the performance when using the original S3-based exchange operator and when using network communication. We observe significant performance gains in most queries, especially in the most time-consuming ones by simply using Boxer without any further optimizations or careful query planning. Boxer is open source and can be developed further along several interesting directions to better support serverless data analytics.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *NSDI*. 2020.

[2] 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: *USENIX ATC*. 2018.

[3] *Amazon Athena*. URL: http://docs.aws.amazon.com/athena/ (visited on 08/17/2020).

[4] *Amazon S3*. URL: https://aws.amazon.com/s3 (visited on 08/17/2020).

[5] *Amazon SQS*. URL: https://aws.amazon.com/sqs (visited on 08/17/2020).

[6] *Apache OpenWhisk*. URL: https://openwhisk.apache.org/ (visited on 08/17/2020).

[7] *AWS Lambda*. URL: https://aws.amazon.com/lambda (visited on 08/17/2020).

[8] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. "Putting the "Micro" Back in Microservice". In: *USENIX ATC*. 2018.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. "SEUSS: Skip Redundant Paths to Make Serverless Fast". In: *EuroSys*. 2020.

[10] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *ASPLOS*. 2020.

[11] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. "Photons: Lambdas on a Diet". In: *SoCC*. 2020.

[12] Bryan Ford, Pyda Srisuresh, and Dan Kegel. "Peer-to-Peer Communication Across Network Address Translators". In: *USENIX ATC*. 2005.

[13] 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: *USENIX ATC*. 2019.

[14] Sadjad Fouladi et al. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *NSDI*. 2017.

[15] *Google BigQuery*. URL: https://cloud.google.com/bigquery/ (visited on 08/17/2020).

[16] *Google Cloud Functions*. URL: https://cloud.google.com/functions (visited on 08/17/2020).

[17] 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.

[18] Eppinger J.L. "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem". In: *Carnegie Mellon University, Tech. Rep, ISRI-05-104*. 2005.

[19] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. "Occupy the Cloud: Distributed Computing for the 99%". In: *SoCC*. 2017.

[20] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. "Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics". In: *USENIX ATC*. 2018.

[21] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *OSDI*. 2018, pp. 427–444.

[22] *Managing AWS Lambda Function Concurrency*. URL: https://aws.amazon.com/blogs/compute/managing-aws-lambda-function-concurrency/ (visited on 08/28/2020).

[23] *Microsoft Azure Functions*. URL: https://azure.microsoft.com/en-us/services/functions (visited on 08/17/2020).

[24] Ingo Müller, Rodrigo Bruno, Ana Klimovic, John Wilkes, Eric Sedlar, and Gustavo Alonso. "Serverless Clusters: The Missing Piece for Interactive Batch Applications?" In: *SPMA*. 2020. DOI: 10.3929/ethz-b-000405616.

[25] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". In: *SIGMOD*. 2020. DOI: 10.1145/3318464.3389758.

[26] 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: *USENIX ATC*. 2018.

[27] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. "Starling: A Scalable Query Engine on Cloud Functions". In: *SIGMOD*. 2020.

[28] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure". In: *NSDI 19*. 2019.

[29] *Serverless Networking SDK*. URL: http://networkingclients.serverlesstech.net/ (visited on 08/17/2020).

[30] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: Stateful Functions-as-a-Service". In: *PVLDB* (2020).

[31] Ao Wang et al. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache". In: *USENIX FAST*. 2020.

[32] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. "Anna: A KVS for Any Scale". In: *ICDE*. 2018.

[33] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Autoscaling Tiered Cloud Storage in Anna". In: *PVLDB* (2019).

[34] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. "Transactional Causal Consistency for Serverless Computing". In: *SIGMOD*. 2020.