Punching Holes in the Cloud: Direct Communication between Serverless Functions Using NAT Traversal

Daniel Moyer

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

May 5, 2021

Blacksburg, Virginia

Keywords: Serverless computing, AWS Lambda, NAT traversal, TCP hole punching, communication framework

Punching Holes in the Cloud: Direct Communication between Serverless Functions Using NAT Traversal

Daniel Moyer

(ABSTRACT)

A growing use for serverless computing is large parallel data processing applications that take advantage of its on-demand scalability. Because individual serverless compute nodes, which are called functions, run in isolated containers, a major challenge with this paradigm is transferring temporary computation data between functions. Previous works have performed inter-function communication using object storage, which is slow, or in-memory databases, which are expensive. We evaluate the use of direct network connections between functions to overcome these limitations. Although function containers block incoming connections, we are able to bypass this restriction using standard NAT traversal techniques. By using an external server, we implement TCP hole punching to establish direct TCP connections between functions. In addition, we develop a communications framework to manage NAT traversal and data flow for applications using direct network connections. We evaluate this framework with a reduce-by-key application compared to an equivalent version that uses object storage for communication. For a job with 100+ functions, our TCP implementation runs 4.7 times faster at almost half the cost.

Punching Holes in the Cloud: Direct Communication between Serverless Functions Using NAT Traversal

Daniel Moyer

(GENERAL AUDIENCE ABSTRACT)

Serverless computing is a branch of cloud computing where users can remotely run small programs, called "functions," and pay only based on how long they run. A growing use for serverless computing is running large data processing applications that use many of these serverless functions at once, taking advantage of the fact that serverless programs can be started quickly and on-demand. Because serverless functions run on isolated networks from each other and can only make outbound connections to the public internet, a major challenge with this paradigm is transferring temporary computation data between functions. Previous works have used separate types of cloud storage services in combination with serverless computing to allow functions to exchange data. However, hard-drive–based storage is slow and memory-based storage is expensive. We evaluate the use of direct network connections between functions to overcome these limitations. Although functions cannot receive incoming network connections, we are able to bypass this restriction by using a standard networking technique called Network Address Translation (NAT) traversal. We use an external server as an initial relay to setup a network connection between two functions such that once the connection is established, the functions can communicate directly with each other without using the server anymore. In addition, we develop a communications framework to manage NAT traversal and data flow for applications using direct network connections. We evaluate this framework with an application for combining matching data entries and compare it to an equivalent version that uses storage based on hard drives for communication. For a job

with over 100 functions, our implementation using direct network connections runs 4.7 times faster at almost half the cost.

Acknowledgments

First, I would like to thank my adviser, Dr. Dimitrios Nikolopoulos, who patiently guided me through the process of the M.S. program despite the challenges of meeting remotely and by helping in all the times when I didn't have an idea of what to do. His direction for how to focus my research each week and overall, as well as his encouragement to keep going when I was overwhelmed were both essential, so I want to thank and fully recognize him for his role. Also, I would like to thank Dr. Godmar Back for answering my questions when I first considered applying to the Masters program and for his attention to detail in programming and computer systems. Additionally, I want to thank him for all of his time and work leading the Virginia Tech programming team and coaching me as part of it. For teaching my operating systems class and being flexible with the project in unforseen circumstances, I would like to extend my thanks to Dr. Ali Butt as well. Finally, I would like to thank Dr. Jamie Davis and Dr. Dongyoon Lee for encouraging me to get started with undergraduate research and giving me the opportunity and experience of contributing to a conference paper.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

Serverless computing is a service that allows developers to run programs directly without having to manage servers themselves. Also known as Functions as a Service (FaaS), serverless computing is offered by major cloud platforms, for instance Amazon Web Services (AWS) Lambda. Because serverless computing can scale rapidly, a growing trend is to use it to run large distributed workloads without needing to provision a cluster of machines. Recent works have used serverless for data analytics [28], machine learning [1, 7, 35], linear algebra [29], and video processing [11, 36]. However, since serverless programs, which are called functions, each run in their own isolated environment, a key problem for large-scale compute applications is transferring data between multiple instances of a function. Individual serverless functions have limited memory, computation power, and execution time, so inter-function communication is a requirement for many applications. Existing works have used object storage and in-memory databases for communication, but these techniques require trade-offs with regard to performance, cost, and scalability [17, 25]. Using direct network connections between functions is attractive since it would be fast, incur no extra cost, and scale with the number of functions; however, serverless providers use network firewalls to block inbound connections.

1.2 Novelty

Our work implements a method to bypass the firewall for AWS Lambda, and we develop the first communication framework that uses direct network connections between functions. Lambda uses a common networking configuration called Network Address Translation (NAT) to allow outbound connections from functions while blocking inbound connections. However, there are several standard techniques that allow endpoints behind NAT to establish a connection with each other by using an external server for setup. This process is called NAT traversal, and our communication library implements it in order to transfer data between serverless functions over TCP connections. We demonstrate that our communication library can scale to workloads with over 100 functions and show it achieves a significant speedup compared to exchanging data with object storage. There have been several previous serverless execution frameworks [5, 16, 38], but to our knowledge none of them use direct network communication. Also, prior studies [17, 25] on inter-function communication for large serverless workloads have found limitations in existing methods: object storage is slow and in-memory databases are expensive and do not scale easily. Fouladi et al. [12] mention that NAT traversal between functions is possible, but do not evaluate it or discuss it in detail. Our work demonstrates that serverless functions can communicate using network connections, which has significant advantages over current techniques.

1.3 Design and Contributions

We develop a communication library with function and server components so that serverless applications can use network communications easily. We specify an interface for serverless application code that runs on each serverless function. Our library supports multi-function jobs and manages communication between functions automatically. We also implement an orchestrator server to invoke functions and send control messages between them. An external server is necessary to perform NAT traversal, and we also use it for coordination. By using network connections, our library achieves high performance. We measure throughput of 680 Mbps between a pair of functions. We also evaluate it using a multi-stage reduce-by-key application with over 100 functions. Compared to an equivalent implementation using object storage, our library is 4.7 times faster and costs only 52% as much.

Chapter 2

Background

2.1 Serverless computing

Serverless computing, also called Functions as a Service (FaaS), is a paradigm in cloud computing where users can execute short-lived application code in a quickly scalable, stateless environment without needing to deploy or manage servers. A single instance of a serverless application is called a function. Serverless functions can be triggered automatically to respond to events or used to process data in parallel. As an example, uploading an image to a particular bucket in object storage might trigger a serverless function to resize it automatically. Major commercial FaaS providers include Amazon Web Services (AWS) Lambda [3], Azure Functions [21], Google Cloud Functions [13], IBM Cloud Functions [15], and Oracle Cloud Functions [23] and there are also several open source serverless platforms including OpenFaaS [22], Apache OpenWhisk [4] and Kubeless [19].

A key feature of serverless is that it scales quickly compared to traditional server-based computing where users provision virtual machine servers. It is possible to simultaneously launch hundreds of serverless functions and the initialization time is usually well under a second, whereas it may be up to a minute for server-based machines. However, each serverless function has a maximum execution time, which is 15 minutes in the case of AWS Lambda. Users can also configure the memory and CPU power allocated to each function, which ranges from 128–10,240 MB with AWS Lambda. Serverless computing uses a pay-as-yougo cost model where users are billed based on the total execution time of their functions proportional to the configured memory size as well as the number of function invocations. The rapid scalability and usage-based billing means that serverless is ideal for uneven or bursty workloads since users do not need to provision resources in advance.

Besides scalability, one of the main characteristic of serverless functions are that they run in a stateless, isolated environment called a container. Although containers have a small amount of storage space (512 MB for AWS Lambda) during their execution, it is not persisted so functions must use external storage mechanisms. Containers also block incoming network connections, so it is not possible to run web servers directly in functions or to access running function code in a remote shell for debugging. While different functions are guaranteed to run in separate containers, a FaaS provider may at its discretion reuse a container from a previous execution of the same function. This reduces startup time since the function does not need to reinitialize and is called a *warm start*, as compared to when a function runs in a new uninitialized container, which is called a *cold start*. Overall, serverless computing has several distinct properties, namely minimal management, rapid scalability, and isolated execution environments.

2.2 Data storage with serverless

Serverless applications must solve problems related to data transfer and limited function lifetime that do not occur with traditional server-based computing. One challenge in serverless is how to persist data and transfer it between multiple functions, since functions are ephemeral by nature. Although functions do have access to limited storage space during runtime, it is not persistent. Also, functions can be invoked with a small amount of metadata, but this is not sufficient for data processing applications. Thus, it is necessary for serverless functions to input and output data via one or more external data stores, such as database servers or object storage. Because functions are isolated from each other, using external data storage also allows functions to communicate for large computing workloads. This differs from traditional server-based computing frameworks where nodes can easily exchange data over the network. In addition, conventional server-based computing nodes do not have limited lifetimes, but serverless functions do. Serverless computations must account for limited function lifetime when scheduling functions and ensure that functions finish writing their output before they timeout.

There are several different external cloud computing services that serverless functions can use to send and receive data, such as object storage, memory- and disk-backed databases, and proxied network connections. Persistent object storage such as AWS Simple Storage Service (S3) can store large amounts of data for a few cents per gigabyte per month. A disadvantage is that it does not support high throughput and rate-limits requests to a few thousand per second [25]. While object storage is cheap, the fastest storage available is memory-backed key-value stores, such as AWS ElastiCache which provides managed Memcached or Redis instances. These memory-backed databases can support a very fast request rate (over 100,000 requests per second) and are efficient for small objects, but are significantly more expensive: ElastiCache is hundreds of times more expensive than S3 when storing the same amount of data [25]. Also, memory-backed stores must have servers provisioned to run on. Of course any other type of database can also be used in combination with serverless to provide different trade-offs with respect to cost and performance. Most databases still need provisioned servers, however, which limits scalability. In theory, it is also possible to use external proxy servers so serverless functions can communicate over the network. However, this would only be practical if the bandwidth each function needs is small compared to that of a server, since provisioning multiple servers just to act as proxies would defeat the purpose of using serverless in the first place. Overall, for large computing serverless workloads, there are several options for external storage including object storage, databases, and proxied connections, but they all require trade-offs between cost and performance.

2.3 NAT

2.3.1 Definition

Network Address Translation (NAT) [31] is a commonly used technique to connect an internal network with a different IP addressing scheme from its external network. A network address translator rewrites the IP packets that cross from one network to the other in order to translate between internal and external IP addresses and port numbers, modifying the packet headers to do so. It maintains connection state so that when a client on the internal network makes an outbound connection, it can correctly forward the return packets back to that client. Typically, NAT is used in combination with filtering rules to block unsolicited incoming packets from entering the internal network. In this case, two clients behind different internal networks are unable to connect to each other directly without using a special technique to bypass NAT, which is known as NAT traversal.

2.3.2 NAT Traversal

Transmission Control Protocol (TCP) hole punching is a technique for establishing a TCP connection between two endpoints that are both behind separate NAT networks and thus cannot connect directly [10]. The way TCP hole punching works is that both endpoints simultaneously make an outbound connection to the other endpoint. If the destination IP address and port of each connection matches the source IP address and port of the other, then each NAT device will allow the incoming connection from the other endpoint because it treats it as the response to the outbound connection from its own endpoint. For this to work, each endpoint must know the public source IP address and port of the other endpoint's outbound connection so it can make the destination IP address and port of its connection match. Thus, a prerequisite for hole punching is that both endpoints exchange their external source IP addresses and ports using an out-of-band method such as a relay server. However, the NAT device must use a known or predictable external source IP address and port for TCP hole punching to be possible. If the NAT device uses an unpredictable external port, then the opposite endpoint will not know what destination port to use and the NAT traversal will not work.

2.3.3 AWS Lambda NAT behavior

AWS Lambda creates each function container with a private IP address behind a NAT firewall that blocks incoming connections. The NAT firewall translates the function's private IP address into a public IP address so it can access the public internet. However, since the firewall does not allow functions to receive connections, this means that Lambda cannot be used to run externally-accessible servers and that two functions cannot directly communicate with each other over the network without NAT traversal. AWS does not explicitly state the behavior of the NAT firewall they use for Lambda function containers, but based on our experimentation, it rewrites the private IP address of the function to a public IP for outgoing connections, but preserves the TCP port used by the function. Also, the public IP address assigned to a container is static throughout the container's lifetime which means that the public IP address and port of an outgoing connection is predictable as required for TCP hole punching. RFC 3489 [27] classifies NAT implementations as *cone NAT* or *symmetric NAT* based on whether the mappings it assigns are consistent across multiple connections.

When a client makes outbound connections from the same private IP address and port, cone NAT consistently maps the connections to the same public IP address and port; however symmetric NAT assigns a different public IP address or port to subsequent connections. In general, it is not possible to perform hole punching with symmetric NAT since the public IP address and port number cannot be known in advance (except when symmetric NAT uses a sequential or other predictable assignment). However, AWS Lambda uses cone NAT, which makes hole punching feasible. We demonstrate that it is possible to perform NAT traversal to establish a TCP connection between two concurrently running Lambda functions and incorporate this ability into our communication library.

Chapter 3

Design

3.1 Communication library functionality

In order to use serverless functions to run a distributed application, there is a fair amount of setup involved. Each function must be started and then receive instructions for what input sources and output destinations to use. Also, for direct network communication, functions must exchange the network address information for the peer functions with which they will communicate. This paper presents a communication framework for performing this setup as well as establishing network connections between function. The implemented communications library consists of two main parts: an orchestrator that invokes functions and manages communication between them, and worker functions that perform NAT traversal and run application code for data processing. The orchestrator must be hosted on a machine that allows inbound network connections since this is a requirement for NAT traversal. When starting an application run, the orchestrator first generates a communication graph of all the functions to deploy that specifies which pairs of functions should establish direct network connections and provides a list of all the inputs and outputs for each function. The orchestrator uses the information in this graph to invoke the necessary functions and then starts a listening server. When each worker function starts, it connects to this server and exchanges information to learn which peer functions to connect to. After a function receives the address information for its peers, it performs NAT traversal to establish direct TCP connections with them. We use TCP instead of UDP to provide reliable transmission of data between functions. Once this setup is complete, the worker begins running the actual application code. This initialization process for the first TCP connection between two functions is shown in Figure 3.1. Worker applications that use the communication library must

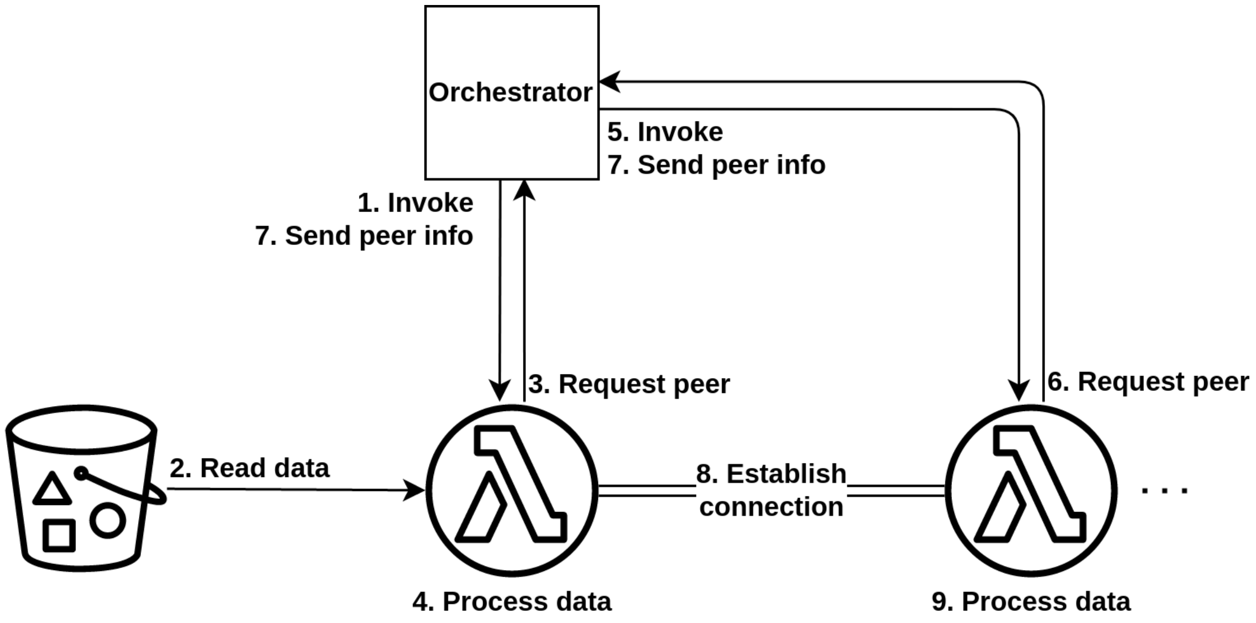

Figure 3.1: Communication library initialization procedure for the first two worker functions. The first worker starts reading and processing data while establishing a connection to the second in parallel. If the second worker has any other peers, it will repeat this setup with them once it starts reading data from the first worker.

implement interface methods to produce input data, consume output data, transfer data between functions, and process input data into output data.

3.2 Client-server design

Our framework uses a separate server for communication to bypass the restriction that Lambda functions do not allow inbound connections. Since AWS firewalls Lambda functions, it is not possible for one function to connect to another directly by themselves. However, with a server external to Lambda, it is possible to relay traffic between functions or to facilitate NAT traversal so functions can establish a direct connection. Because having a server is necessary to use NAT traversal in our communication library, we decided to also use it as an orchestrator to invoke functions and send control messages between them. The server starts a service using the WebSocket protocol [9] on a publicly accessible host and port, and includes this information as part of the function event data on invocation. Then, when each function starts, it makes a persistent connection to the orchestrator server so it can exchange small control-plane messages. At this point, the orchestrator sends peer network information for each pair of functions that need to perform NAT traversal.

While implementing the orchestrator on a single server may affect scalability, the per-function resource requirements in our design are small, so we believe that this is an acceptable trade-off. Because performing NAT traversal requires an external server so functions can exchange network address information, the only design variable is whether or not to have that server support additional functionality. We decided to use that server to implement a function orchestrator, which maintains a persistent connection with each function throughout its lifetime to send and receive control messages. However, an alternate design would be to have the central server only act as a NAT traversal relay and implement the orchestrator a part of the serverless functions themselves. Implementing the orchestrator on a traditional server has the benefit of allowing persistent connections to a centralized point. This saves the need for a more complicated peer-to-peer control approach, or running the orchestrator in a separate function and having to handle the case where it must run for longer than the 15-minute maximum function lifetime and having to use some replacement scheme. The main potential drawback with moving additional features to the server is that it might decrease the scalability of the design. For each function, the server must maintain some data structures in memory and a single persistent connection with that function. The difference is that in the NAT-traversal-only case, the server can free these resources for a function once it has completed NAT traversal, whereas in our design, the server must maintain them until the function exits. However, a single server should be able to scale to many thousands of concurrently-running functions since the per-function resources are fairly small, and running jobs of that scale would probably also require additional design improvements regardless.

3.3 NAT traversal protocol

Although there are multiple protocols that can be used to perform NAT traversal, we used a custom implementation of TCP hole punching because we wanted to use TCP for data reliability and to use the same system for NAT traversal and exchanging control messages. In general, there are many possible configurations for a network address translator, but our design only needs to work in the specific environment of AWS Lambda. One of our main design decisions was to use TCP instead of UDP. We chose TCP because it automatically detects and corrects transmission errors. Although UDP hole punching may be more common than TCP hole punching, Ford et al. [10] state the latter is just as “fast and reliable” as the former. However, TCP hole punching is slightly more complex to implement from an application perspective, since the application must both listen on and make outbound connections from the same socket. This is not a problem for our communication library since we control the implementation and can configure it to allow socket reuse in this way. Another potential concern with TCP is that its handshake and slow-start behavior may lead to slower data speeds than UDP soon after creating a connection. However, when sending large amounts of data, these startup costs should have minimal effect, but for applications that care about fine performance optimizations, we leave the investigation of UDP as future work. Another design decision we made was not to use existing protocols such as STUN

[27] or TURN [20] to help with NAT traversal. The STUN protocol uses a public server to help a client determine its public IP address, whether it is behind NAT, and the NAT type if so. However, since we already knew the NAT behavior for AWS Lambda, it was easier for us to find the public IP address using our orchestrator server that to set up a separate STUN server simply for that purpose. Also, the TURN protocol provides a way to implement relaying when NAT traversal is not possible, but this is not applicable for our case and would be very inefficient anyway. While our hole punching implementation is implemented specifically for AWS Lambda, the TCP hole punching technique works for most NAT configurations such as cone NAT as well as nested NAT when the outer NAT supports hairpin translation, although it does not work with symmetric NAT [10]. Based on our testing, AWS Lambda does not currently support IPv6, which could eliminate the need for NAT if its implementation assigned a public IP address directly to each function. However, there would likely still be a stateful firewall that blocked incoming connections, so a similar process to bypass this by making simultaneous outbound connections might still be necessary if AWS Lambda deployed IPv6 but sill prevented functions from communicating directly with each other. Although there are many possible behaviors for NAT, our communication library only needs to support the one used by AWS Lambda, so we decided to use TCP hole punching because it was simple enough to implement as part of our orchestrator server and because we wanted to use TCP for its built-in data reliability.

Chapter 4

Implementation

4.1 NAT traversal technique

In order to create a direct connection between Lambda functions, our communication library uses TCP hole punching, which is a known NAT traversal technique [10], to bypass how Lambda blocks incoming connections to functions. On startup, the function component of our library connects to the orchestrator server in order to receive control information. Since the orchestrator server runs outside of Lambda, this outbound connection is permitted by Lambda's firewall. Once the orchestrator accepts connections from both functions that plan to establish a mutual TCP connection, it sends the public IP and a TCP port number of each function to the other. Thus, both functions have the necessary IP address and port number information to make an outbound TCP connection, and repeatedly try to connect to the other function using this information. While trying to make the outbound connection, each function also opens a socket to listen on its own fixed port number in case the connection from its peer succeeds first. After both functions attempt outbound connections, eventually, the Lambda NAT firewall will allow traffic from the other function through as it treats this as return traffic from the outbound connection. At this point, from one function's perspective, either its outbound connection attempt succeeds or the listening socket accepts a connection from the other function. Which event succeeds first is non-deterministic, but either way the function has a valid TCP connection open to its peer function and can begin sending and receiving data.

We present an example sequence diagram of TCP hole punching between two AWS Lambda functions in Figure 4.1. In this example, Function 1 has public IP 3.0.1.1 and Function 2 has

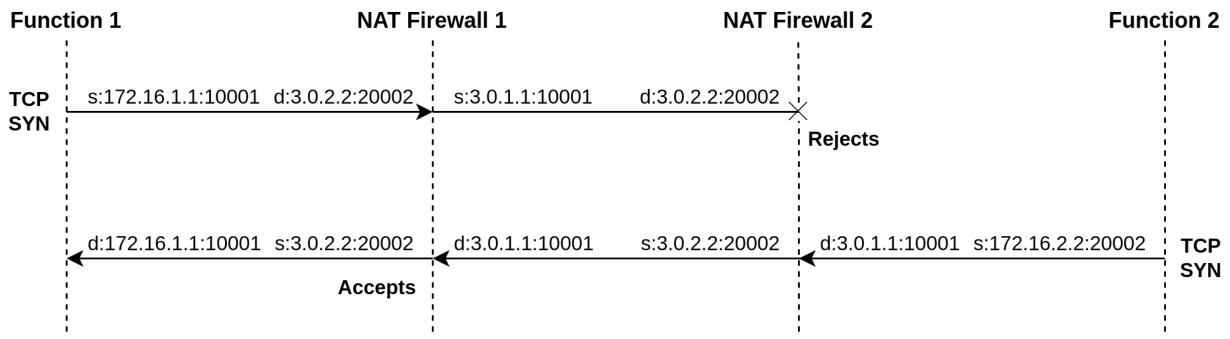

Figure 4.1: TCP hole punching example

public IP 3.0.2.2. Also, both functions start listening for incoming connections on their corresponding private IP addresses and ports: address 172.16.1.1 port 10001 for Function 1 and address 172.16.2.2 port 20002 for Function 2. At this point, both functions have exchanged public IP addresses and listening port numbers via the orchestrator server. Once they receive this information, both functions try to open an outbound TCP connection to the other. In the diagram, the first connection attempt is made by Function 1 to the public IP address of Function 2. Note that Function 1 sets the source port of the TCP SYN connection request packet to be the same as the port number it is listening on. The NAT firewall for Function 1 rewrites the private source port of this packet to the public IP of Function 1, namely 3.0.1.1, but it keeps the source port number the same. The fact that the AWS Lambda firewall preserves port numbers makes TCP hole punching possible. However, at this point the NAT firewall for Function 2 sees the incoming packet as unsolicited and drops it. Independently, Function 2 now sends a connection request to Function 1, and its NAT firewall rewrites the source IP address to the public IP address of Function 2. When the connection request reaches the firewall for Function 1, the source and destination addresses and ports are exactly switched compared to the connection previously initiated by Function 1. Thus, the Function 1 firewall treats the second connection request as a response to the first, and allows it. Now, Function 1 receives the connection SYN request from Function 2 on its listening port, and sends a SYN-ACK acknowledgment packet back to continue the TCP connection setup. The firewall for Function 2 will also accept the acknowledgment because it is a valid response to the SYN packet Function 2 sent. Thus, both firewalls will allow traffic through, so the functions can finish the handshake to establish a valid TCP connection between them.

4.2 Server-function communication

Our communications library uses persistent WebSocket connections implemented with the Python Socket.IO [14] library to exchange control messages between the orchestrator server and worker functions. As discussed in section 3.2, we use a client-server design for the communication library where the orchestrator server is responsible to send and receive commands and status updates between function. These control messages include the initial information needed for NAT traversal, when a function has finished, and notification of when a function has output available so the orchestrator can invoke functions that depend on it. As it invokes functions, the server creates a listening WebSocket service and each function establishes a connection to it on startup. The function keeps this connection open throughout its lifetime and only disconnects when it exits. One requirement of this design is that the orchestrator must be able to push messages to functions. Thus, since functions cannot receive inbound connections, they must either poll the server repeatedly for updates or establish a persistent full-duplex communication channel. Polling inherently has a higher latency and introduces more overhead since it uses multiple connections, so we opted for a single persistent connection. We decided to use the WebSocket protocol [9] because it supports full-duplex communication over a single TCP connection and is lightweight for small messages. Also, we wanted to use a preexisting message protocol with an available Python library instead of implementing some custom protocol directly on top of TCP. Thus, we chose the Python Socket.IO library for sending messages as both a client and server WebSocket implementation.

4.3 Communication library API overview

The serverless communications library presented in this work presents a high-level Application Programming Interface (API) that supports the development of serverless applications with inter-function communication over TCP. Applications using the library must specify two components: a topology graph of worker functions that dictates how they should communicate with each other and the data processing implementation itself. The communication library is responsible for invoking all of the functions and performing NAT traversal to establish all of the TCP connections for data transfer specified by the application topology. In the topology, applications must define the inputs and outputs for each worker function with two possible types for each: an internal TCP connection or an external application-implemented subroutine. Inputs and outputs with an internal TCP type are managed by the communication library itself where the output data of one function is sent over TCP to become the input data of another. These internal TCP connections form directed edges between functions in the topology graph. The application is required to make sure the function graph is acyclic so that there are no circular dependencies between functions. While TCP connections can be used to send data between functions, applications must be able to read initial data and write their results to an external source because serverless functions are ephemeral. Thus, the communication library also allows applications to define custom subroutines for input and output and then specify in the graph which functions should run these subroutines and with what parameters. As an example, Figure 4.2 defines a topology with three functions. The application's custom input and output subroutines could respectively read from and write to the S3 object specified in the `path` metadata field. Then, the first two functions, `worker-0` and `worker-1`, download data from S3, process it and send their immediate results over TCP to `worker-2`, which combines them and writes the final output to S3.

```
[ { "name": "worker-0",
    "inputs":  [ {"type": "s3", "path": "s3://example-bucket/input-0.csv"},
                 {"type": "s3", "path": "s3://example-bucket/input-1.csv"} ],
    "outputs": [ {"type": "tcp", "name": "worker-2"} ] },
  { "name": "worker-1",
    "inputs":  [ {"type": "s3", "path": "s3://example-bucket/input-2.csv"} ],
    "outputs": [ {"type": "tcp", "name": "worker-2"} ] },
  { "name": "worker-2",
    "inputs":  [ {"type": "tcp", "name": "worker-0"},
                 {"type": "tcp", "name": "worker-1"} ],
    "outputs": [ {"type": "s3", "path": "s3://example-bucket/result.csv"} ] } ]
```

Figure 4.2: Example function topology definition

In addition to the function topology, applications that use our communications library must implement the actual data processing logic that the serverless functions run. The communications library expects applications to write a Python class that implements a list of required methods. Each of these methods use Python generators to efficiently produce and/or consume a stream of data objects and run simultaneously in their own thread. As discussed before, the communication library expects the application to implement two methods to handle external inputs and outputs specified by the function topology. In addition, the application must write its main data processing method that transforms objects received as inputs into objects that can be sent as outputs. Finally, the application must implement two methods that can serialize and deserialize the data objects it uses into a binary format for transmission over TCP.

4.4 Function program design

The communication library uses a multi-threaded design to run application data processing subroutines and to perform network communication. A major advantage of using TCP for communication is that multiple functions running in parallel can use pipelining to improve throughput. So, as soon as one function is finished processing some block of data, it can immediately send that block to the next function in the pipeline while continuing to process the next data block. Thus, to implement pipelining efficiently, each function should be simultaneously reading input data, processing it, and writing output data. Our communication library uses separate threads for each of these tasks to implement parallelism within a function so that a series of functions can achieve data pipelining. In each function, the library uses one thread to run the application's data processing subroutine, as well as one thread for each input or output source or destination whether it is a TCP connection or a custom application subroutine. An alternative design would be to use an event-based approach using Python coroutines, but this requires non-blocking I/O, which is difficult to implement. Furthermore, our library's NAT traversal implementation must listen on a socket while making outbound connections at the same time, and using a separate thread for each is the most straightforward approach. Putting each application input and output subroutine in a separate thread means that it can block to read or write without affecting data processing. One disadvantage to our use of Python threads is that the Python interpreter uses a global lock so threads do not gain a benefit from running on a multi-core CPU [26]. For small function sizes, this does not matter since AWS Lambda will allocate less than one virtual CPU to the function, but it does prevent the communication library from fully utilizing multiple cores with larger function sizes. However, our communication library's use of multiple threads allows it to still support pipelining with other functions since it can process data at the same time as performing blocking input and output.

Chapter 5

Evaluation

5.1 TCP performance in Lambda

5.1.1 Throughput

In order to set some expectations for the performance of network communications, we ran several experiments to measure the performance of TCP connections between Lambda functions. The first metric we measured was the raw throughput for a single connection between two functions. We implemented a simple benchmark application using our communication library, which uses Python TCP sockets with default settings. The benchmark consisted of two components, a producer and a consumer, each of which ran in a separate function. The producer repeatedly transmitted a fixed block of data as fast as possible and the consumer read data from its connection and counted the total size of the data received. To calculate throughput, the consumer also recorded the time between the first and last block of data received. Each of our tests sent 1 gigabyte total and used functions with 3072 MB of memory. We tried different buffer sizes from 1 KB to 1 MB for reading and writing from the network socket and measured each with 5 runs. For each buffer size, Run 1 was a cold start due to the fact that the function code was modified and Lambda was thus forced to provision a new container for execution. The remaining Runs 2–5 were warm starts, but since our experiment only measured time while sending data after both functions had already initialized and established a TCP connection between them, we do expect this to make a difference. The results are graphed in Figure 5.1 with the full data listed in Table 5.1. The maximum

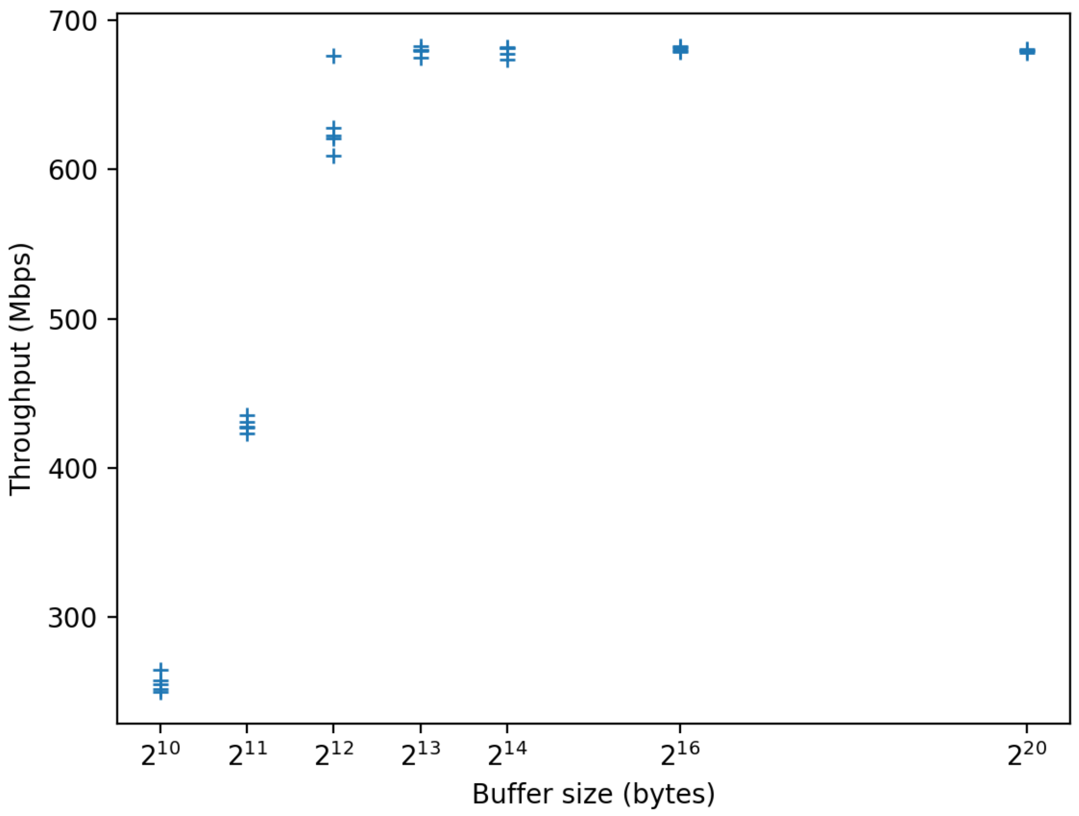

Figure 5.1: Throughput (Mbps) between two Lambda functions based on buffer size

throughput of about 680 megabits per second (Mbps) is achieved when using buffers of 8 KB or larger. We would expect smaller buffer sizes to increase overhead, and we see that the performance drops off with 4 KB buffers and smaller. While there was some variability in the results, this is understandable because AWS Lambda makes no guarantees on the performance or bandwidth available to functions. An additional comment is that Fouladi et al. [12] report observing speeds of 600 megabits per second for direct network communications

Table 5.1: Throughput (Mbps) between two Lambda functions based on buffer size

Buffer	1 KB	2 KB	4 KB	8 KB	16 KB	64 KB	1 MB
Run 1	264.965	435.219	676.116	682.517	673.219	678.374	679.434
Run 2	250.180	427.054	621.060	674.673	681.465	679.088	679.643
Run 3	257.868	423.257	609.191	680.248	677.673	682.674	680.327
Run 4	254.905	430.758	627.690	680.236	681.192	681.604	679.864
Run 5	252.256	427.881	622.661	679.362	681.917	680.902	677.918
Average	256.035	428.834	631.344	679.407	679.093	680.528	679.437

between functions in their future work discussion, which is similar if not quite as fast as our results.

We also experimented with two producers and with two consumers to see if a different function topology would increase the throughput. In each experiment, we used the same methodology as before with two functions, except adding either an additional producer or consumer. We halved or doubled the data sent by each producer so that each consumer would receive exactly 1 gigabyte of data as before. Based on our previous results, we used a buffer size of 8 KB since increasing the size further made no difference in throughput. The results for the first test with two producer functions sending to a single consumer function are listed in Table 5.2. Likewise, the results for one producer function sending to two consumers are presented in Table 5.3. This shows the throughput for each consumer separately as well as the combined total. In both experiments, the maximum total throughput was approximately 680 Mbps, so we conjecture that Lambda imposes a bandwidth cap on each function that limits the observed speed in each case.

After measuring the network throughput for direct connections, we extended the experiment to determine if the throughput would decrease when using a longer chain of functions. One use case for our communication library would be to implement a data pipeline with a chain of functions where each function receives data over TCP, processes it, and transmits its

Table 5.2: Throughput (Mbps) with 2 producer and 1 consumer functions

	Throughput
Run 1	679.197
Run 2	682.057
Run 3	677.897
Run 4	680.409
Run 5	681.373
Average	680.187

Table 5.3: Throughput (Mbps) with 1 producer and 2 consumer functions

	Consumer 1	Consumer 2	Total
Run 1	309.753	372.020	681.773
Run 2	369.452	307.845	677.297
Run 3	313.596	311.577	625.173
Run 4	306.886	354.791	661.677
Run 5	367.429	318.463	685.892
Average	333.423	332.939	666.362

results to be processed by the next function. To see how network speed would affect the performance of our communication library in this case, we increased the number of functions in our benchmark application. We used a simple copy operation for the data processing

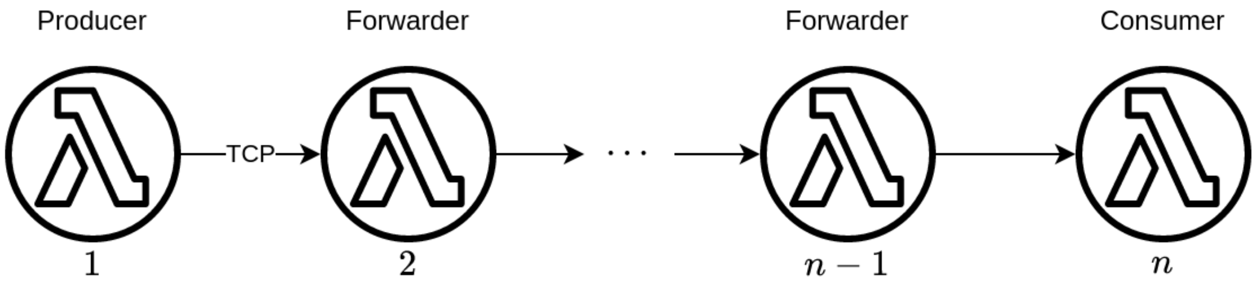

Figure 5.2: n-function chain for throughput experiment

stage, so for an execution with a chain of n functions there would be a single producer, $n-2$ forwarders, and finally a single consumer to measure throughput. This function graph is shown in Figure 5.2. Each forwarder in the chain simply copied the data it received on its input connection and wrote it to its output connection as fast as possible without doing any other processing. Note that the case $n = 2$ with no forwarders is identical to the setup for the previous throughput test between two functions. We used the same methodology as previous tests with the same sizes for function memory (3072 MB), read buffers (8 KB), and data sent (1 GB). We ran five trials for each value $n = 3, 4, 5, 10, 50$ and list the results in Table 5.4. This table also includes the results for $n = 2$ from the previous throughput test for comparison. Note that the performance is not very consistent between runs even for an

Table 5.4: Throughput (Mbps) for n-function chain

n	2	3	4	5	10	50
Run 1	682.517	613.047	679.856	604.428	679.200	629.934
Run 2	674.673	679.712	636.615	673.883	652.200	610.505
Run 3	680.248	681.710	667.813	620.689	647.970	631.217
Run 4	680.236	677.091	643.818	519.874	672.744	608.416
Run 5	679.362	679.981	665.965	646.269	677.361	618.527
Average	679.407	666.308	658.813	613.029	665.895	619.720

extremely simple application that does no actual computation. While there is substantial variance between runs and none of the larger cases are as consistent as $n = 2$, we see that it is still possible to get data throughput well over 600 Mbps using TCP communication at the end of a long function pipeline. One additional observation is that a complete trial with 50 functions took about 28 seconds of real time, whereas a trial with 2 functions only took about 12 seconds. This is due to the time it takes to initialize functions and perform NAT traversal to establish connections as well as the fact that the data simply has more functions to flow through. Our purpose with this experiment was to measure potential throughput, not latency. The results demonstrate the possibility of using our communication library to implement a multi-stage data processing pipeline with high throughput assuming a workload where CPU processing is not a bottleneck. This gives an indication of what throughput is possible with AWS Lambda using our communication library, but also shows the inherent variability in its performance.

5.1.2 Latency

While the throughput between Lambda functions is more important for data processing, we also tried to measure the network latency. There is variance to where Lambda provisions containers for different instances of a function—they could possibly be on the same machine or separate machines either close together or far apart. For the latency experiment, we started two instances of a function where one sent a small message over a TCP connection and the other waited to receive it and then responded immediately. The first function measured the total round-trip time between when it sent the original message and received a response. We chose a small message size of 16 bytes so it would be guaranteed to fit in a single packet and because using other sizes made no appreciable difference. For each trial with two functions, we sent 500 round trip messages at 50 millisecond intervals. We performed 10 trials and made sure they were cold starts so that Lambda used new containers for every function to get a broader sample. The round-trip times varied from 0.38 to 22.34 ms with an average across trials of 0.98 ms. We show box plots for each trial in figure Figure 5.3. A notable finding is high variability between trials. Since using cold starts for each trial forced Lambda to provision new containers each time, this shows that the latency between functions depends on AWS Lambda's internal placement of containers. The results also had a high number of outliers; note that Figure 5.3 only show outliers up to 3 ms, but the maximum overall time was 22.34 ms. Even though the round-trip time for TCP connections between functions has variability, the average latency of around 1 ms will be more than sufficient for most data processing applications as throughput is more important.

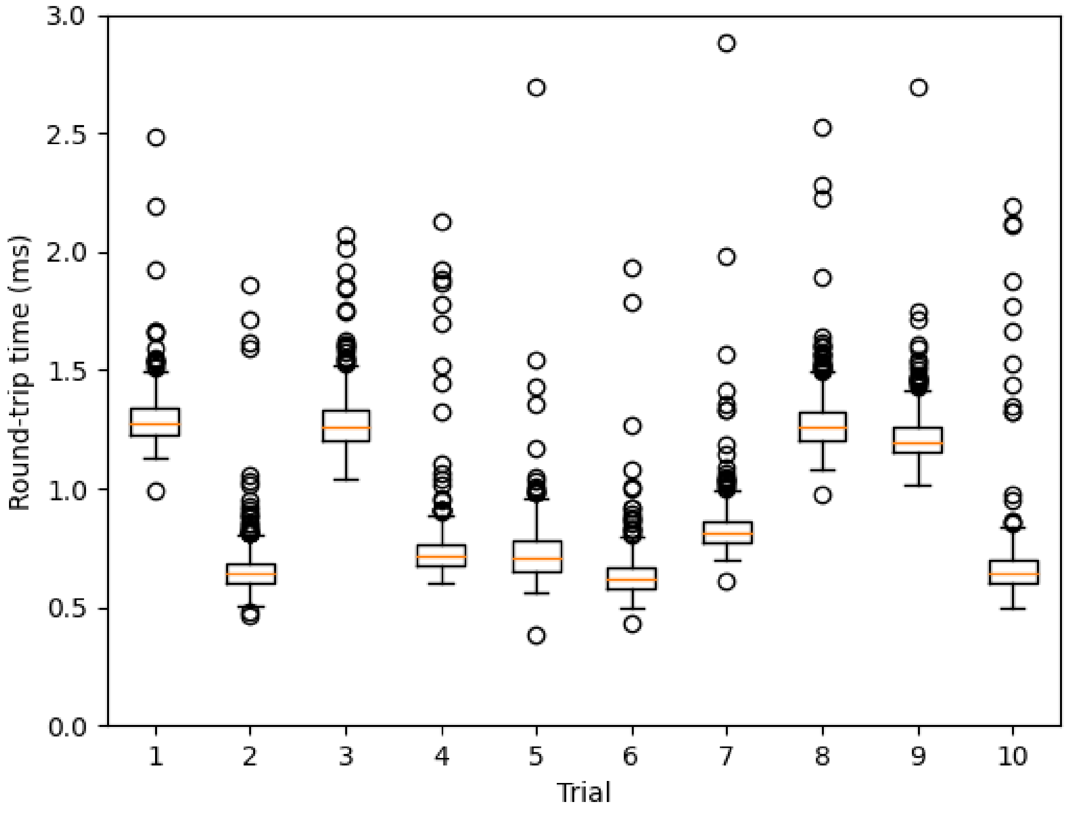

Figure 5.3: Round-trip time between pairs of function instances

5.2 TCP vs. S3 comparison

We developed a distributed serverless application using our communications library to compare the performance of using TCP networking for communication instead of S3 object storage. We implemented a large-scale reduce-by-key application using multiple stages of functions for merging based after a map-reduce computation paradigm. The input data set consisted of string keys and floating point values that were merged by summing the values for each distinct key. The application used a binary tree topology as shown in Figure 5.4. Each function receives data from two sources: either S3 objects for the first stage or TCP

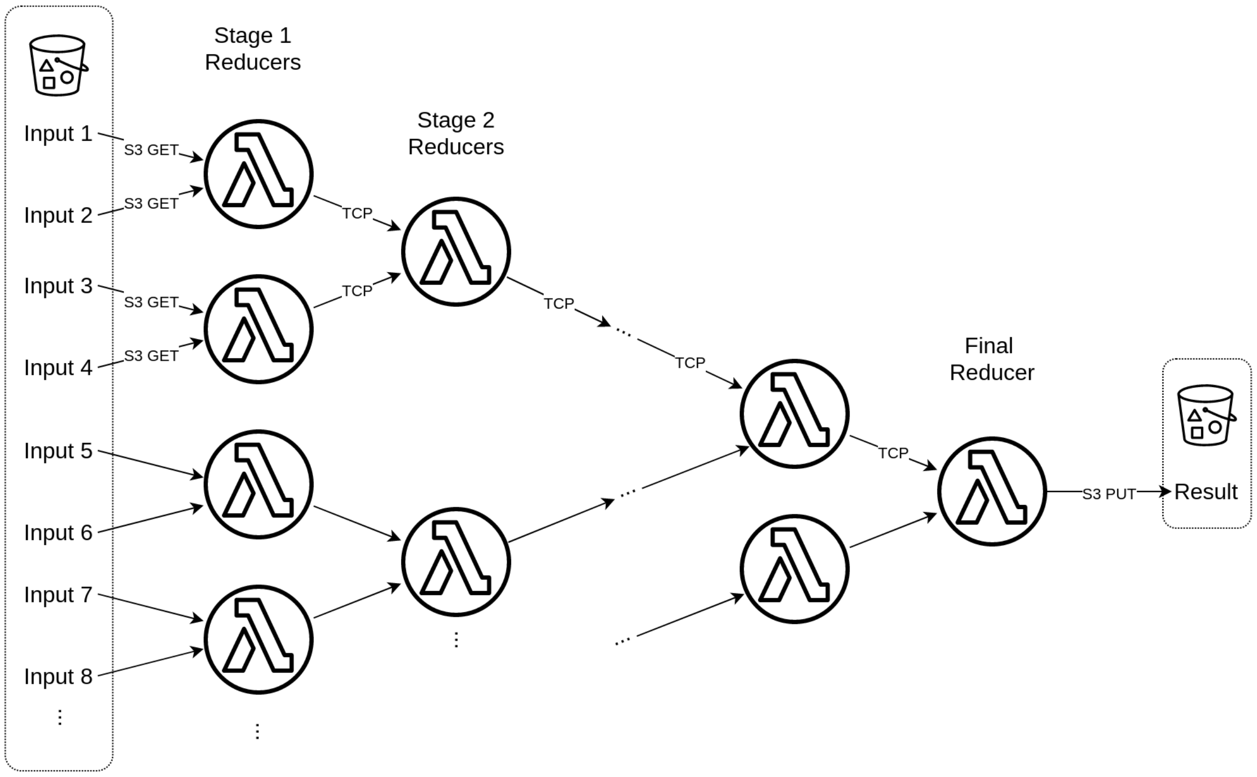

Figure 5.4: Function graph for reduce-by-key application

connections for subsequent stages. After processing its input, it sends it to a function in the next stage via a TCP connection, except in the case of the final function, which writes the completely reduced output to S3. Our experiments used 101 input files with a total of 44.5 million key-value pairs that reduced to 1.9 million unique keys. Each run used 104 Lambda functions configured with 3072 MB of memory each. We performed 6 executions, but discarded the first so that all runs used warm starts for consistency. We measured the total billed duration of all functions as well as the real time elapsed from when the communication library orchestrator started the first function to when the last function exited. For comparison, we ran an experiment using the same data and methodology but using S3 for inter-function communication instead of TCP. Instead of using TCP connections between pairs of functions to send data, the first function wrote all the data to an S3 object, and the receiving function read that object once it was completely written. Table 5.5 compares the real-time and cost measurement results for the TCP and S3 implementations. Using TCP connections is 4.7 times faster and costs only 52% as much based on Lambda execution time. One reason our communication library achieves a large improvement using network

Table 5.5: Communication method performance for reduce-by-key application

	S3		TCP	
Trial	Billed duration (s)	Real time (s)	Billed duration (s)	Real time (s)
Run 1	821.158	156.972	434.830	33.310
Run 2	758.467	157.013	420.921	33.631
Run 3	808.216	154.258	430.776	32.629
Run 4	837.696	153.305	419.542	32.907
Run 5	810.812	155.919	405.974	32.412
Average	807.270	155.493	422.409	32.978

connections is because functions send data incrementally so later stages can start processing data in parallel to previous ones. However, when using S3, an entire object must be created before it could be read, so our S3 implementation does not achieve this same parallelism. This could be somewhat optimized by writing multiple smaller S3 objects, but this would require significant implementation effort and is not done by Amazon Web Service's sample Lambda map-reduce implementation either [2]. Compared to a S3-only implementation, our communication library achieves significant time and cost savings by using direct TCP connections for inter-function communication.

Chapter 6

Related work

6.1 Serverless communication

A major challenge and topic of study with using serverless computation for data processing is exchanging data between functions. Many serverless frameworks either use object storage for its low cost and scalability [16, 29], or in-memory stores for their speed [5, 38]. However, Pu et al. [25] develop a hybrid storage system that combines S3 and Redis to balance cost and performance. They evaluate their implementation on analytics applications and their results are significantly faster than S3-only but use less resources than traditional server clusters. Klimovic et al. [17] evaluate different types of storage systems, specifically technologies based on disk (S3), memory (Redis), and NVMe flash (Apache Crail and ReFlex). They look at several workloads and observe that the storage properties serverless applications need for communication are scalability, high IOPS, high throughput, and low latency, while noting that high durability is not necessary because computations are short-lived. In addition, they find that NVMe flash storage can meet the performance requirements of most applications at a lower cost than memory-based solutions. Pocket [18] is a distributed elastic data store implementation designed for inter-function communication that uses a combination of storage based on DRAM, NVMe flash, and HDD. It scales automatically and chooses the most cost-effective storage type to meet bandwidth requirements, and matches the performance of AWS ElastiCache Redis at 59% of the cost. Based on these works, using a mixture of storage types for inter-function communication is necessary to balance cost, performance, and scalability, but our results demonstrate that network communication can achieve all of these properties itself.

6.2 Serverless execution frameworks

With different approaches to handle scheduling, communication, and failures, there are multiple prior works that implement serverless execution frameworks for running distributed applications. A commonality between the different frameworks is that for communication between functions they all use object storage such as S3, in-memory databases such as Redis, or a combination of the two. One of the foundational execution frameworks in the field is PyWren [16], which provides a simple interface to map data using parallel tasks on serverless functions. Functions read input data from and write their results to S3 object storage, which the authors note is a bottleneck. PyWren is designed to support embarrassingly parallel tasks and does not have built-in support for exchanging data between functions, although applications could implement this themselves. There are several works that extend PyWren, including IBM-PyWren [28], which ports it to IBM Cloud Functions and extends it to run Map-Reduce jobs. In addition, NumPyWren [29] implements parallel linear algebra algorithms on top of PyWren. Pérez et al. [24] develop a similar programming model and implementation that make it simple to process S3 files using parallel functions. While serverless functions are very well suited to run embarrassingly parallel applications, there are several works that support more general purpose computation. For example, the gg tool [12] supports jobs where the computation graph is not available before execution, while managing function failures and stragglers. The authors demonstrate that it can compile real-world software projects up to 5 times faster than a warm AWS EC2 cluster without the need for provisioning. To simplify application implementation, Kappa [38] presents a programming interface using futures similar to standard concurrency programming models. It also provides checkpointing of function results in order to allow failure recovery. Both the gg and Kappa frameworks support either S3 or Redis as storage backends for communicating data between functions. The Wukong [5] framework is distinguished in that it does not use a centralized scheduler. Its worker functions manage scheduling themselves by either becoming or invoking their successor functions, which improves scalability and data locality. Wukong also uses a Redis cluster for inter-function communication. There are a variety of serverless frameworks for parallel computation, but they all rely on object storage or memory-based stores for transferring data between functions.

6.3 Serverless applications

Serverless computing supports a wide variety of data processing and analytics applications. One strength of serverless computing is for embarrassingly parallel applications that need to scale quickly. Zhang, Krintz, Mock, and Wolski [35] implement an application to find the best hyperparameters for machine learning models while Zhang, Zhu, Zhang, and Liu [36] use serverless for video processing and consider the effect of function size and type. Ali et al. [1] use the inherent scalability of serverless combined with batching logic to efficiently perform machine learning inference with bursty workloads. While there are many other works that use serverless for straightforward data processing, some have more creative use-cases. Singhvi et al. [30] develop a network middle-box framework by using functions to operate on packet streams by dividing network flows into small time-limited chunks. By exploiting container reuse policies, Wang et al. [34] use serverless functions to implement an in-memory cache with a pay-per-use cost model. Overall, the scalability and pricing structure of serverless computing supports a broad range of applications.

6.4 FaaS platforms

Besides frameworks and applications that run on commercial serverless providers, there are several works that improve the FaaS platform itself. To avoid the problem of copying data to serverless functions, Zhang et al. [37] run functions directly on storage nodes to improve locality and reduce latency. Thomas et al. [33] implement an overlay network for serverless containers that reduces initialization time when many functions are started in parallel. Dukic et al. [8] reduce memory usage and cold starts by sharing the application libraries and runtime between multiple instances of the same function. In addition, Tariq et al. [32] provide quality-of-service for invoking and scheduling functions with a front-end framework to commercial FaaS platforms. To support large-scale scientific computing, Chard et al. [6] develop a distributed FaaS platform that uses heterogeneous clusters and supercomputers as function executors. All of these frameworks improve or extend the actual serverless execution platform, which demonstrates some of the current limitations of existing FaaS providers.

Chapter 7

Conclusion

7.1 Limitations and future work

A major area for future work with our communication framework is providing failure detection and recovery. Our library currently does not implement a way to recover from connection or function failures and requires restarting the entire job. We believe that rerunning a computation from the beginning will be sufficient for many serverless applications, especially since they tend to be short-lived due to the 15-minute maximum lifetime of Lambda functions. The elasticity of serverless computing makes spawning new functions to restart a job simple. Also, the delay in saving large amounts of intermediate results to durable object storage would be prohibitive and defeat the purpose of using direct connections for communications to begin with. Thus, using network connections prioritizes computation speed at the cost of some reliability. The main times for potential failures in our design are at function initialization, connection establishment, and during data processing. It would be simple to add recovery logic to our library for the first two cases by invoking a replacement function for the one that failed to connect. However, if a function failed during data processing whether due to a connection reset, runtime error, or exceeding a resource quota, whatever data was in its memory at the time is lost so it would be extremely difficult for the remaining functions to recover. This kind of mid-computation recovery would be application specific and difficult to implement without checkpoint to persistent storage. One approach that uses our communication library as-is would be to write an application in multiple stages that run in series where each stage completely writes its output to persistent storage before the next stage starts. This way, only a failed stage would need to be restarted instead of the whole application. In practice, the main source of failures we saw during development were caused by programming mistakes. At the scale we tested, we did not encounter any consistent failures related to network connections or Lambda functions themselves. A topic for further research would be to determine how larger or longer-running application are affected by these issues and to implement a checkpointing or other recovery method for them. Although our design using direct network connections loses some reliability because data is not persisted in the case of failures, we believe the elasticity, performance, and simplicity of only using network connections will outweigh this limitation for many applications, especially short-lived or data intensive ones.

While the design of our communications library does have some limitations, our results provide a lot of potential for future work. Primarily, we hope that our library or direct network connections in general can be used to improve the performance of serverless applications. Also, it would be useful to verify that NAT traversal is possible on serverless providers besides AWS Lambda and to port our communication library to them. A better solution would be for serverless providers to support direct networking between multiple instances of a function, which would eliminate the need for NAT traversal in the first place. One restriction with our framework is that the number of functions and their respective inputs and outputs must be defined in advance for an application before execution. A possible improvement would be to add the ability for functions to dynamically start successor functions and initiate TCP connections with them during runtime such as in the design used by the gg [12] or Wukong [5] frameworks. Despite these limitations, our communication library introduces a fast, cost-effective, and scalable way for serverless functions to exchange data.

7.2 Summary

Our work implements the first serverless communication library that uses direct network connections to transfer temporary computation data between functions. We use TCP hole punching to bypass the AWS Lambda firewall, which restricts incoming connections. Our communication library manages NAT traversal and connection setup and provides a interface for developing serverless applications with network connections. Based on our experiments, network communication is significantly faster and cheaper than writing and reading from object storage. Using TCP also eliminates the cost and provisioning overhead of in-memory databases. With its benefits for performance, cost, and scalability, we hope that our communication framework and NAT traversal technique can be used to improve existing serverless computing applications and open the potential for new ones.

Bibliography

[1] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[2] Amazon Web Services. Serverless reference architecture: Mapreduce. GitHub, 2021. URL https://github.com/awslabs/lambda-refarch-mapreduce.

[3] Amazon Web Services. AWS Lambda: Serverless compute, 2021. URL https://aws.amazon.com/lambda.

[4] Apache Software Foundation. Apache OpenWhisk is a serverless, open source cloud platform, 2021. URL https://openwhisk.apache.org.

[5] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 1–15, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421286. URL https://doi.org/10.1145/3419111.3421286.

[6] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. FuncX: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, pages 65–76, New York, NY, USA, 2020. Association

for Computing Machinery. ISBN 9781450370523. doi: 10.1145/3369583.3392683. URL https://doi.org/10.1145/3369583.3392683.

[7] Anthony S. Deese. Implementation of unsupervised k-means clustering algorithm within Amazon Web Services Lambda. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 626–632. IEEE Press, 2018. ISBN 9781538658154. doi: 10.1109/CCGRID.2018.00093. URL https://doi.org/10.1109/CCGRID.2018.00093.

[8] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 45–59, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421297. URL https://doi.org/10.1145/3419111.3421297.

[9] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. URL https://www.rfc-editor.org/rfc/rfc6455.html.

[10] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, USA, 2005. USENIX Association.

[11] 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)*, pages 363–376, Boston, MA, March 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[12] 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.

[13] Google Cloud. Cloud functions, 2021. URL https://cloud.google.com/functions.

[14] Miguel Grinberg. Python Socket.IO server and client, 2021. URL https://github.com/miguelgrinberg/python-socketio.

[15] IBM. IBM cloud functions, 2021. URL https://www.ibm.com/cloud/functions.

[16] 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, pages 445–451, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350280. doi: 10.1145/3127479.3128601. URL https://doi.org/10.1145/3127479.3128601.

[17] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

[18] 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, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/klimovic.

[19] Kubeless. Kubeless: The Kubernetes native serverless framework, 2021. URL https://kubeless.io.

[20] Rohan Mahy, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, RFC Editor, April 2010. URL https://www.rfc-editor.org/rfc/rfc5766.html.

[21] Microsoft Azure. Azure functions serverless compute, 2021. URL https://azure.microsoft.com/en-us/services/functions.

[22] OpenFaaS Ltd. OpenFaaS: Serverless functions made simple, 2021. URL https://www.openfaas.com.

[23] Oracle. Cloud functions, 2021. URL https://www.oracle.com/cloud-native/functions.

[24] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. A programming model and middleware for high throughput serverless computing applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 106–113, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359337. doi: 10.1145/3297280.3297292. URL https://doi.org/10.1145/3297280.3297292.

[25] 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, February

2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/pu.

[26] Python Software Foundation. Glossary: Python 3.8.8 documentation, 2021. URL https://docs.python.org/3.8/glossary.html#term-global-interpreter-lock.

[27] Jonathan Rosenberg, Joel Weinberger, Christian Huitema, and Rohan Mahy. STUN - Simple traversal of User Datagram Protocol (UDP) through Network Address Translators (NATs). RFC 3489, RFC Editor, March 2003. URL https://www.rfc-editor.org/rfc/rfc3489.html.

[28] 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. Association for Computing Machinery. ISBN 9781450360166. doi: 10.1145/3284028.3284029. URL https://doi.org/10.1145/3284028.3284029.

[29] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 281–295, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421287. URL https://doi.org/10.1145/3419111.3421287.

[30] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. SNF: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 296–310, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421295. URL https://doi.org/10.1145/3419111.3421295.

[31] Pyda Srisuresh and Matt Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, RFC Editor, August 1999. URL https://www.rfc-editor.org/rfc/rfc2663.html.

[32] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 311–327, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421306. URL https://doi.org/10.1145/3419111.3421306.

[33] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: Ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 16–29, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421275. URL https://doi.org/10.1145/3419111.3421275.

[34] 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.

[35] M. Zhang, C. Krintz, M. Mock, and R. Wolski. Seneca: Fast and low cost hyperparameter search for machine learning models. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 404–408, 2019. doi: 10.1109/CLOUD.2019.00071.

[36] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM Workshop*

on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV '19, pages 61–66, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362986. doi: 10.1145/3304112.3325608. URL https://doi.org/10.1145/3304112.3325608.

[37] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 1–12, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369732. doi: 10.1145/3357223.3362723. URL https://doi.org/10.1145/3357223.3362723.

[38] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 328–343, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421277. URL https://doi.org/10.1145/3419111.3421277.