Profiling for Go AWS Lambda functions

- 2017 Spring CS263 Runtime System Project Summary

Group members: Yingchun Du (PERM: 4932646), Michael Zhang (PERM: 4908778)

Github repo: https://github.com/Heronalps/GO_AWS_Lambda

Presentation slides:
https://docs.google.com/presentation/d/1OAC_BOyWFwSKlB7g2hTP9asnaRFVgEC7YkvJcCuCZ74/edit?usp=sharing

1. Introduction

1.1 - AWS Lambda

AWS Lambda is a serverless compute service which makes users run code without any management of servers. AWS Lambda executes the code only when needed, and scales automatically and independently. Users only pay what they used, which means that there is no charge when their code is not running. With AWS Lambda, users can run code for virtually any type of application or backend service - all with zero administration. After uploading the code, Lambda takes care of everything required to run and scale the code with high availability. Users can set up their code to automatically trigger from other AWS services, such as Amazon DynamoDB, Amazon Kinesis and Amazon SNS, or call it directly from any web or mobile app.

AWS Lambda natively supports Node.js, Java, C# and Python. Although AWS Lambda does not officially support Go, there are many open source frameworks which support Go functions running on AWS Lambda. In this project, we executed and measured Go functions on AWS Lambda by using two open source frameworks.

1.2 - Golang

Go is an open source programming language to make programmers more productive. Its concurrency mechanisms, supported by goroutines, make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. It is a low-level system language with garbage collection and the power of run-time reflection. Go’s garbage collector is a concurrent, tri-color, mark-sweep collector. It's a fast, statically typed, compiled language. We can quickly compile Go code to machine code and run it.

1.3 - Docker vs VM

Docker is a software container platform. Developers use Docker to solve the problems related to the development environment when collaborating with co-workers, no matter in different R&D

centers or in different teams. Docker containers are an abstraction at the app layer that packages code and dependencies together. Using containers, everything required to make a piece of software run is packaged into isolated containers. Different containers run independently and share the kernels.

Virtual machines include not only the applications, binaries and libraries, but also the entire guest operating systems. Unlike VMs, containers do not include a full operating system - only the application and the necessary binaries and libraries. These features make Docker more efficient and storage saving, and easier to deploy on different platforms.

2. Frameworks and their mechanism

2.1 - Container reuse in AWS lambda

The first time a function executes after being created or having its code or resource configuration updated, a new container with the appropriate resources will be created to execute it, and the code for the function will be loaded into the container.

There is a background process running when the function finishes – what happens to it if the container is reused? In this case, Lambda will actually “freeze” the process and thaw it out the next time you call the function. So in the reuse case, the background processes will still be there, but they won’t have been executing while users were away. This can be really convenient if users use them as companion processes, since it avoids the overhead of recreating them.

2.2 - Sparta

At a high level, Sparta transforms a single Go binary’s registered lambda functions into a set of independently addressable AWS Lambda functions.A Sparta-compatible lambda is a Go function with a specific signature. Sparta uses the results of the http.ResponseWriter(both status and body) to determine the AWS Lambda response.

The event and context are mapped to event and context of native AWS Lambda. w http.ResponWriter is the response from AWS Lambda and logger records all logs. Content written to this logger will be available in CloudWatch logs.

Most Frameworks like Sparta use child process spawned by Node.js to call Go function, partially because Node.js is the first runtime on AWS Lambda. However, this method generates memory overhead and latency. When Go Lambda function executes, Node.js calls *child_process.spawn*, which follows the common async programming pattern by accepting a callback or returning an EventEmitter.

For Sparta, container reuse came in the form of a NodeJS HTTP proxying tier. For each Sparta-compatible AWS Lambda function, Sparta creates a unique NodeJS proxy route to forward the AWS request to a sidecar Go binary. The HTTP request/response semantics are a good fit for Lambda and allow applications to support one-time initialization as opposed to a process-per-request model.

However, we found the response of Go lambda function is not logged into CloudWatch, since Sparta uses Logrus and http.ResponseWriter to separate the output and log information. We spent a good amount of time to redirect the output to CloudWatch, but weren't able to do so. Another pain point is Logrus doesn't support character escaping, so fmt.Println function can't shift to another line and "\n" appears in the output without being interpreted.

2.3 - Eawsy

Unlike Sparta, the mechanism of eawsy is using a Docker image pulled out from dockerhub (eawsy/aws-lambda-go-shim:latest), which is based on docker image (amazonlinux:latest) Then, eawsy builds up the executable Go binary and Python proxy within running docker container and generates handler.zip that could be uploaded to Lambda. Therefore, docker has to run locally when eawsy builds up. In the Lambda runtime, Python proxy handles event and context to Go binary and receive return value from native Go empty interface.

Easwy only has two arguments, context and event, exactly as native Lambda runtime. Unlike Sparta, easwy doesn't segregate function log and response, therefore the response of function trigger by other AWS services will be available in CLoudWatch. Additionally, unlike Sparta, eawsy is compatible with AWS CLI command of event function mapping.

Since this advanced method of building up and run Go functions, Eawsy achieves high performance almost as same as native-supported runtime, like Python, according to our observed data. Based on its performance, we implemented following use cases and experiments under eawsy.

3. Implementation

3.1 - Malicious behaviors

To stress test Go Lambda function, we intentionally inserted malicious behavior instructions into the function and observed how it handles them.

With respect to duration limits, we inserted infinite loop and the function exits when the pre-configured time limit has been reached. Likewise, the Go function exits when it receives oversized "event bomb" that creates a memory leak. Interestingly, the error message under Sparta is "socket hang up" and stack traces are from NodeJS runtime, not from Go runtime. This is another disadvantage of using NodeJS child_process, because hiding stack trace information could be really hard for programmers to debug their functions.

Additionally, we tested Go Lambda function with unexpected crash by inserting panic and os.exit into the function. The function exits when it hits panic or os.exit instructions and log the error message like "errorMessage" : "RequestID: 234kbjb9askb1dfg0 Process exited before completing request".

The stress test shows Go Lambda function is really robust and handles malicious behavior instructions with appropriate stack trace information.

3.2 - Go Lambda function collaborating with other AWS Services

We implemented three use cases in Go Lambda function:

- Lambda function interacting with SNS
- Lambda function processing DynamoDB stream
- Lambda function processing Kinesis stream

There are two shortcomings that make these use cases not feasible under Sparta. First, since these functions are all triggered by another AWS service, user can't get http response from function and also can't get result from CloudWatch, because Sparta doesn't stream the result to CloudWatch log. Second, Sparta doesn't support event source mapping command in AWS CLI. Since the last two function are required to map DynamoDB and Kinesis event to Lambda function source, this significantly limits the usefulness of Sparta in the scenario of event source mapping. Based on above reasons, we implement these use case in Eawsy and it performed well under such scenarios.

In the use case of interaction with SNS, we implemented a Go function that interacts with SNS topic across two AWS account. When I publish a message from account A, the SNS topic in account B receives the message and triggers the Go function with such payload. As shown below, the message is directed to CloudWatch log.

An advantage of eawsy is it supports event source mapping command in AWS CLI. It greatly facilitates the Go function processing streaming data from sources like DynamoDB and Kinesis. In the use case of interaction with DynamoDB stream, We implemented a Go function processing DynamoDB stream data. Basically, we created a DynamoDB table and made three transactions: insert, modify and remove. Because the table's event is mapped to corresponding Go Lambda function, these transactions triggered the function, which recorded the log information in CloudWatch.

In the use case of interaction with Kinesis stream, we implemented a Go function processing the Kinesis Stream data. The procedure is we manually send a JSON with a base64 encoded data to kinesis. The kinesis streams the data to Go Lambda function, which decodes the data and logs the readable string to CloudWatch. The response is also recorded in the designated outputfile.txt, since the Go function return the message payload at the end. Go Lambda function polls the stream and, when it detects updates to the stream, it invokes your Lambda function by passing in the event data from the stream.

3.3 - Invocation latency experiment

In the effort of finding a way to record event dependencies of Lambda function, we think a universal timestamp across board could possibly be a solution. To verify this idea, we conducted the invocation latency experiment using one Go Lambda function and one EC2 instance. The Go Lambda function serves as a base function and the EC2 instance is configured as a web server running Go. When base function is triggered by another EC2 instance, it will take a timestamp of Unix epoch time and pass it to web server. Then the server

will take another timestamp and calculate the difference between two timestamps. The difference will be returned back to the triggering instance and recorded there.

The reason of constructing such architecture is because Lambda function cannot be constantly up and listen to a certain port. Also, since Go is not natively supported by AWS lambda, base lambda function can't use boto3 client to call another lambda function. I have to use HTTP call targeting API Gateway to trigger the destination, so that the delay on network might make the latency measurement uncomparable to experiment in Python.

We tried this experiment twice, 10,000 times each, which the first time we ran it on two Python Lambda functions and the second time we conducted it on Go Lambda and EC2 instance. Based on the data, we found the distribution of latencies is not normal. In addition, a certain amount of diff_timestamp are negative and the percentage of negative time_diff is much higher than result on Python. (Python: 669 / 10008 = 6.68%, Go: 4656 / 10038 = 46.38%)

We reason that it is because the time drift between EC2 instance and Lambda container is more serious than it between 2 Lambda containers. We tried to synchronize the time of Go Lambda function and EC2 instance by NTP client, but AWS Lambda doesn't allow the low-level system calls of NTP.

4. Conclusion

In this runtime system project, we found Go function is able to run on AWS Lambda under frameworks with more or less time and memory overhead. The function is also able to collaborate with other AWS services by event and function mapping.

In terms of frameworks, we analyzed the mechanism of both Sparta and Eawsy and tested their performance by certain Go functions. The result illustrates that Eawsy is a preferred framework with advanced performance and usefulness.

We also stress tested Go Lambda functions by malicious instructions and proved Go Lambda is robust and output useful error message for debugging.

We conducted an invocation latency experiment to profile the distribution of latency between Go Lambda container and EC2 instance. The distribution is not normal and the problem of time drift is serious. It also proves that timestamp is not a solution for recording event dependencies. In the future, we would like to explore another method to preserve event dependencies throughout the system.

Reference:

[1] AWS Lambda. (n.d.). Retrieved from https://aws.amazon.com/lambda.
[2] The Go Programming Language. (n.d.). Retrieved from https://golang.org/doc.
[3] Comparing Containers and Virtual Machines. (n.d.). Retrieved from https://www.docker.com/what-container.