8. Infinicache Lazy Migration from AWS to Kubernetes

8.1 Introduction

Infinicache is a distributed system build upon AWS Lambda framework, which implements a in-memory cache capable of exploiting serverless advantages [1]. It does so by deploying several lambda nodes which store in memory chunks of files which are managed by a proxy and a client.

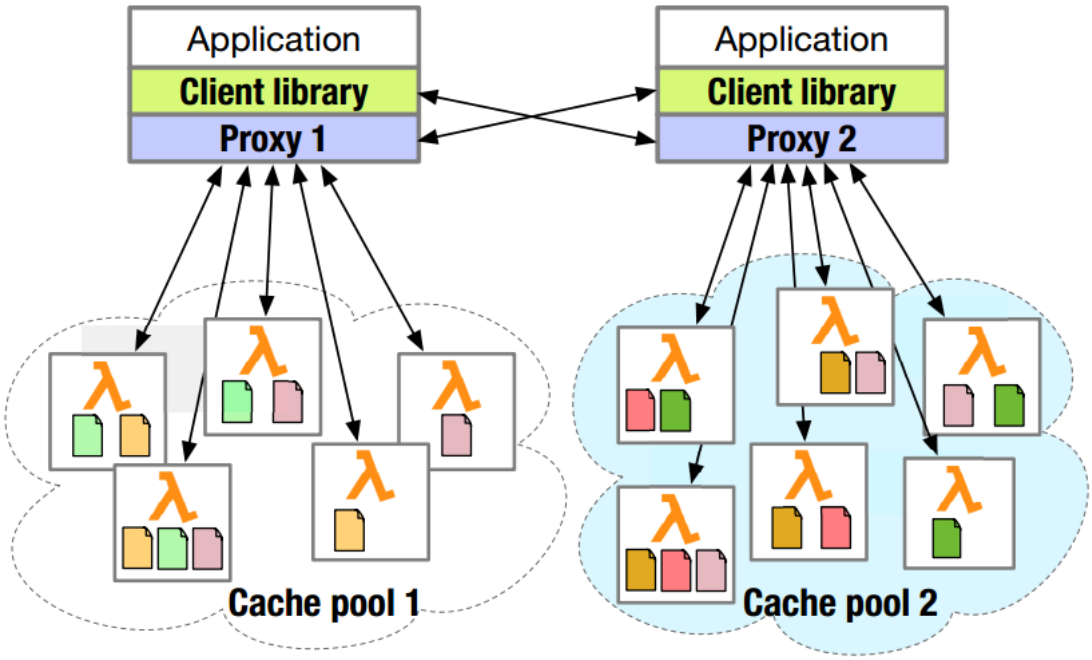

Figure 1: Taken from [1]. INFINICACHE architecture overview. Colored Icon denotes EC-encoded object chunks. Chunks with same color belong to the same object

Briefly,the system architecture can be seen in figure 1 and the protocol responsibilities are distributed over the components following the description below.

- **Client**: Exposes to the application a clean set of GET(key) and PUT(key, value) APIs. The client library is responsible for: (1) transparently handling object encoding/decoding using an embedded EC module, (2) load balancing the requests across a distributed set of proxies, and (3)

determining where EC-encoded chunks are placed on a cluster of Lambda nodes.

- **Proxy**: Responsible for: (1) managing a pool of Lambda nodes, and (2) streaming data between clients and the Lambda nodes. Each Lambda node proactively establishes a persistent TCP connection with its managing proxy.
- **Lambda Function Runtime (Cache Node)**: Executes inside each Lambda instance and is designed to manage the cached object chunks in the function's memory.

We noticed that Infinicache has introduced many interesting mechanisms specifically tuned AWS Lambda which allow features such as Anticipatory Billed Duration Control [1], but AWS Lambda comes also with some limitations such as Banned Inbound Connections which the original authors had to solve with the introduction of the Proxy component which allows clients to keep connections open with the Lambdas untill are not reclaimed by the provider. This brings overhead and complexity to the protocol. For this reason we want to migrate the system from AWS Lambda to Kubernetes (K8s). Kubernetes will allow a more fine grained management of the cluster and of the containers running on it. In order to run the lambda nodes as serverless workloads on K8s we use Knative [2]. Knative extends K8s through a set of Custom Resource Definitions (CRDs) which allow developers to seamlessly deploy containerized application as serverless workloads. Basically Knative adds to K8s all those typical serverless features such as scale-to-zero and scale-to-infinity but with a big set of customizable configurations. For more about Infinicache limitations related to its dependence on AWS Lambda please refer the article [3] Infinicache Implementation Analysis.

8.2 Lazy Migration

As stated in the previous chapter and by article [3] Infinicache Implementation Analysis, there are many protocol simplifications which can be done on a more versatile version of Infinicache. But in order to perform protocol modifications we first need to have Infinicache running over K8s. Our first goal is to migrate Infinicache from AWS to K8s without modifying the protocol, but only remove the AWS dependencies from the system and replace them with K8s dependencies. Therefore we want to modify the original version of Infinicache code repository in such way that we are able to run Infinicache over a Kubernetes cluster and we call such process Lazy Migration.

8.3 Original Deployment

The code base of Infinicache contains some documentation for the installation of the system. However does not contain much documentation about the implementation itself. In order understand the implementation, we analyze the code and support such activity with the deployment of the original version of Infinicache over AWS using a free tier account for testing purposes. The deployed system is shown in figure 2. In order to simulate the original deployment is enough to follow the instructions provided by the original authors.

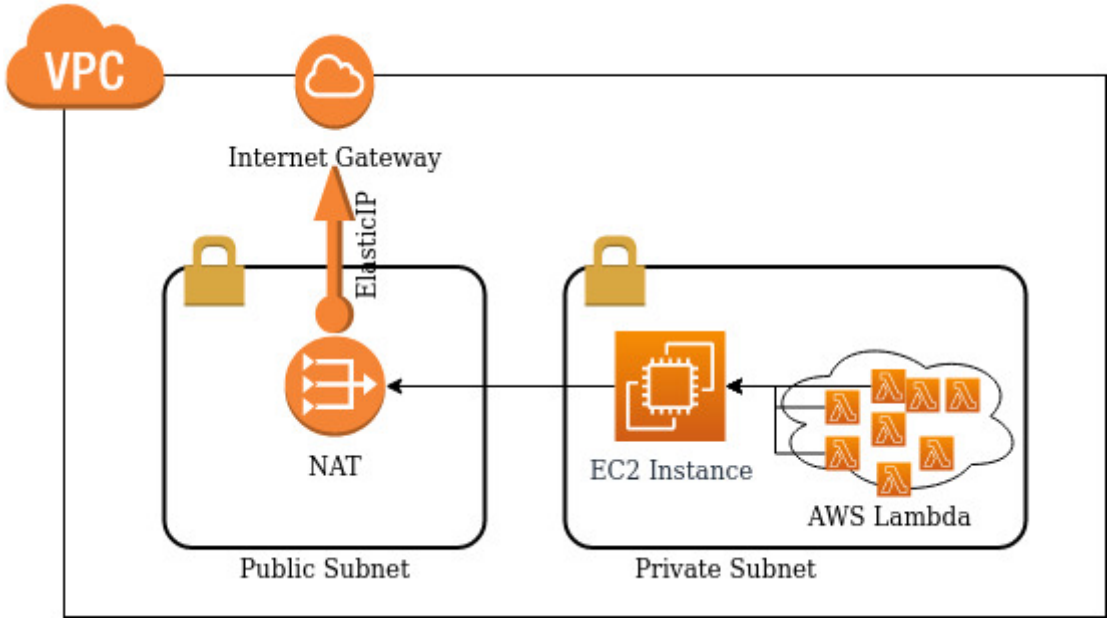

Figure 2: Deployment of original version of Infinicache. EC2 Instance hosts ubuntu-xenial-16.04 on which the Proxy. runs The Lambda Nodes and the EC2 instance belong to the same private network. The Proxy is made publicly available using a Network Address Translator (NAT) and a public Elastic IP. The entire deployment resides in a Virtual Private Cloud.

8.4 Kubernetes Lazy Deployment

As mentioned before in order to lazily migrate Infinicache over K8s, we need to understand the protocol and how this is translated in the code base. A fully description of Infinicache implementation analysis can be read in[3] Infinicache Implementation Analysis., but for the purpose of this document we will only highlight the modifications that we have made to the code base in order to eliminate AWS dependencies and deploy the system in k8s.

8.4.0 Environment

In order to accomplish the migration of Infinicache from AWS to Kubernetes we absolutely need a Kubernetes cluster. For this project we have used Google Kubernetes Engine on which we installed Knative for managing serverless workloads.

8.4.1 Code Base Reorganization

In AWS the proxy was running in a standalone Ubuntu Xenial 16.04 EC2 Instance and the Cache nodes in AWS Lambda Functions. Although AWS Lambda Functions run inside containers, the developer is not aware about that, and in fact when he wants to deploy a new Function, he just has to upload the code to AWS. On the other side, the proxy is just a program which is compiled, build and run on the VM. We would like instead to manage containers, which fit perfectly Kubernetes ecosystem.

We do not consider the client because is a component which can run wherever in internet. In our case we run it always from the proxy VM, therefore it does not necessitate extra environment configuration.

Both programs are written in Go and the original repository structure is composed by a 1 Go module (`module github.com/mason-leap-lab/infinicache`) and 10 packages. All the code that manages Infinicache is contained in the Go module, therefore packages that belong to the cache node are siblings of packages that belong to the proxy. If we want to deploy our system in Kubernetes we need containers, and this code structure doesn't allow us to compile 2 different programs from the same Go module. Moreover we want to keep the cache node container as light as possible, so we don't want to build proxy code together to cache node code. For these reasons we divide the repository in 2 modules: `proxy` and `node`. That way we can easily have 2 different Dockerfiles which we can test individually and which allow us to create 2 distinct containers for the 2 components.

8.4.2 Delete direct AWS dependencies

The original Go module relies on AWS SDK for performing more operations than only deploying Lambda functions and storing data on them. More specifically:

- uses AWS session data instead of generating its own
- stores metrics data on AWS S3, and

- interacts with AWS SDK to understand Lambda's hardware and dynamically select some timeouts

In order to eliminate these dependencies we modified the code by deactivating the metrics storage, generating our own session IDs and selecting a default timeout.

8.4.2 From AWS Lambda to Knative Service

AWS Lambda requires the developer to write a handler function which is triggered each time the function is invoked, therefore the original cache node contains such function. We modified the handler file, and substituted the AWS Lambda handle function with an HTTP Handle function which is served by an HTTP server.

In order to understand this choice we have to explain how the protocol behave. The lambdas are the first to be deployed. Once they are online, the proxy (previously configured with how many nodes it should manage) connects using AWS SDK to AWS Lambda and triggers all the Lambdas by sending them its IP. Once this is done the proxy waits that the Lambdas initialize an outbound TCP connection which will then be used for transmitting further messages. Therefore the AWS Lambda Handle function is triggered only when the cache node is suspected to be slipping or not initialized. For this reason, substituting the handlers this way should keep the other Cache Node functionalities intact, while changing only the triggering method.

After having changed the triggering method of the Cache Node we want to transform the program in a Knative Serverless Service. In order to do that we first containerize the Lambda, publish the image, create a Knative Service manifest file in which we link the Cache Node image and apply the manifest to K8s. Further we also changed the visibility of the service to be local, that way it is accessible only within the cluster. Once we have the image, we can deploy how many cache nodes we want, but a characteristic of Infinicache is that it has to be statically configured with the number of nodes that we want in the cluster. Essentially does not easily scale horizontally.

8.4.3 Proxy Deployment

Until now the proxy was using AWS SDK to invoke the Lambdas by name, but if we assume that our previous modifications on the Node are correct, the proxy should now use HTTP instead. In order to do that the proxy node should have knowledge of the nodes addresses. The proxy uses a scheduler which

manages a Group of Lambda Deployments. We have simply statically configured the proxy with the HTTP addresses of each deployment and changed the invocation method by serializing the same information in an JSON object and sending it to the node through an HTTP POST. After that, we containerized the application, created a Kubernetes deployment with 1 replica and a Kubernetes ClusterIP service to expose the proxy within the cluster.

8.4.4 Testing

The main test the Lazy Migration was subjected to was to check if the system deployed over K8s and Knative is able to successfully perform a `SET(k,v)` operation and then a `GET(k)`. The system was able to pass the test, but we cannot ensure that every mechanism is working as expected. However the Lazy Migration represents a base for future research developments which have the objective to investigate further such area and maybe bring Infinicache over a fully-managed serverless framework and potentially improve it. The diagram of the Lazy Migration deployment upon K8s can be visualized in figure 3.

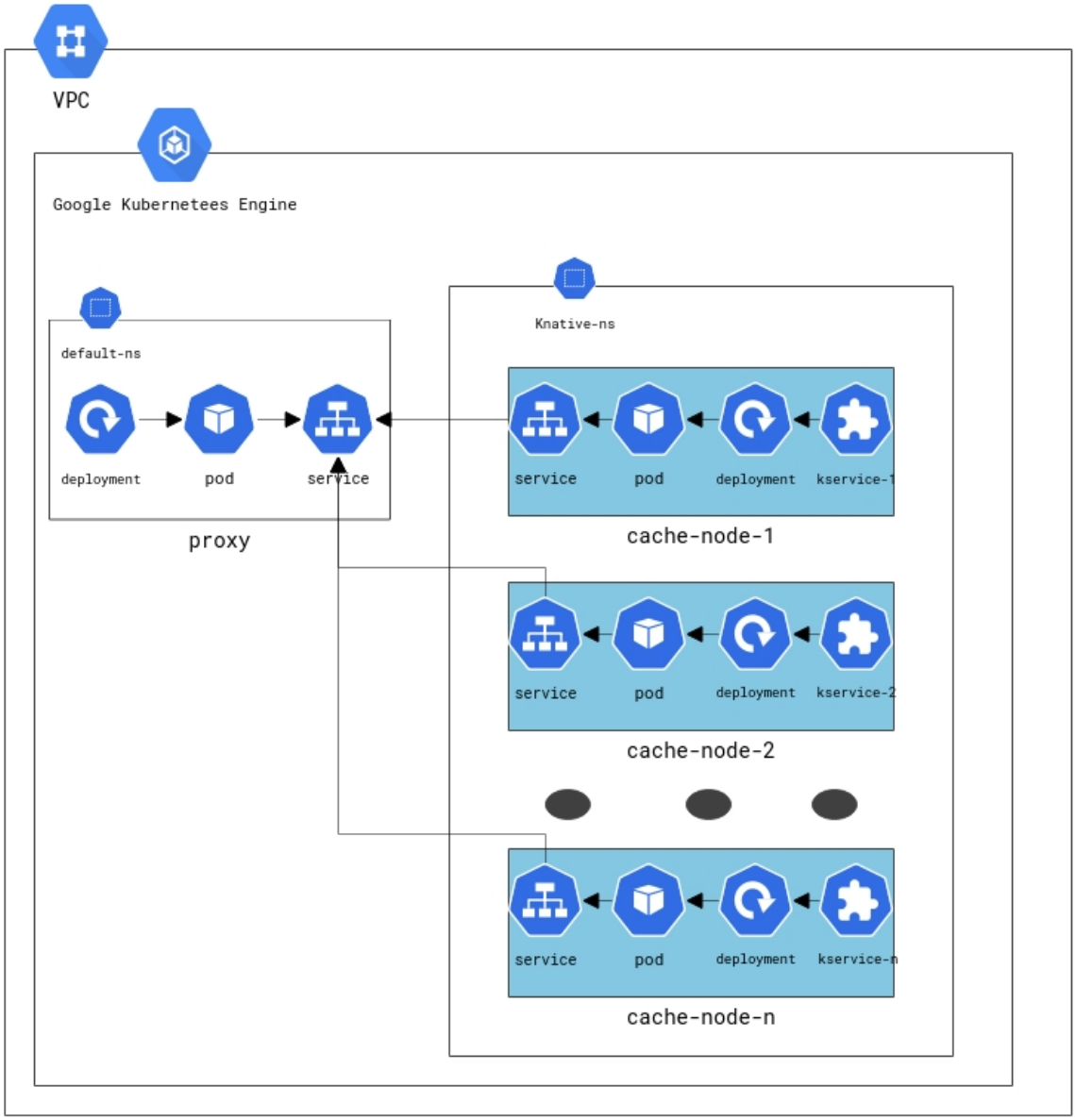

Figure 3: The diagram shows how the Infinicache components are mapped over a Kubernetes cluster. We have on the left the proxy powered by a deployment and a service in the default namespace and on the right we have n serverless Knative cache nodes powered in a similar way as the proxy but completely managed by Knative,

Conclusion

Migrating a system from an infrastructure to another is challenging, especially if the two infrastructures are different in therms of APIs. In this paper we present how we faced a first migration phase where we move Infinicache from AWS to K8s. We show the approach we have taken in analyzing the implementation and applying changes in order to reach a functioning version of Infinicache upon Kubernetes. We also show which are the differences

between the original deployment of Infinicache over AWS and the deployment on K8s.

References:

[1] INFINICACHE: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache - https://arxiv.org/abs/2001.10483

[2] Knative - https://knative.dev/

[3] Infinicache Implementation Analysis.