3. Infinicache Implementation Analysis

Infinicache is a distributed system build over and for AWS Lambda framework which implements a in-memory cache capable of exploiting serverless advantages. It does so by deploying several lambda nodes which store in memory chunks of files which are managed by a proxy and a client. Current implementation available here and original paper here.

3.1 Infinicache Main Components Recap:

- **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**: Executes inside each Lambda instance and is designed to manage the cached object chunks in the function's memory. It will be also called Cache Node in this document.

For a deeper understanding about each component and their overall protocol, please refer to the original paper InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

3.2 Implementation Analysis

The implementation is written in Go and is composed by a `module github.com/mason-leap-lab/infinicache` which contains 10 `packages`. The project `go.mod` file shows that it uses 13 external libraries, of which 5 indirect.

The directly imported libraries are:

- github.com/ScottMansfield/nanolog v0.2.0 - nanosecond scale logger
- github.com/aws/aws-lambda-go v1.13.3 - tools to help Go developers develop AWS Lambda functions
- github.com/aws/aws-sdk-go v1.28.10 - AWS SDK for the Go programming language
- github.com/buraksezer/consistent v0.0.0-20191006190839-693edf70fd72 - provides a consistent hashing function
- github.com/cespare/xxhash v1.1.0 - Go implementation of the 64-bit xxHash algorithm
- github.com/cornelk/hashmap v1.0.1 - lock-free thread-safe HashMap
- github.com/google/uuid v1.1.1 - generates and inspects UUIDs
- github.com/klauspost/reedsolomon v1.9.3 - Reed-Solomon Erasure Coding in Go
- github.com/seiflotfy/cuckoofilter v0.0.0-20200106165036-28deee3eabd7 - bloom filter that efficiently allows also deletion
- github.com/mason-leap-lab/redeo v0.0.0-20200204234106-1e6f10c82f05 - for building redis-protocol compatible servers/services
- github.com/mgutz/ansi v0.0.0-20170206155736-9520e82c474b - allows to create ANSI colored strings and codes
- github.com/onsi/ginkgo v1.7.0 - testing
- github.com/onsi/gomega v1.4.3 - matcher lib for ginko

The packages which compose this implementation of infinicache are:

- `global` - contains procedures for global configuration (set ip address, ports, etc..) and is composed by the following files:
  - `infinicache-master/proxy/global/global.go`
  - `infinicache-master/proxy/global/ip.go` The global package does not depend on AWS
- `client` contains the client protocol and is composed by the following files:

- `infinicache-master/client/client.go`
- `infinicache-master/client/ec.go`
- `infinicache-master/client/ecRedis.go`
- `infinicache-master/client/log.go`

  Moreover the client implementation does not depend of AWS libraries.

- `server` implements the proxy server and contains files:
  - `infinicache-master/proxy/server/config.go` - Lambda runtime config
  - `infinicache-master/proxy/server/group.go` - Group - manages GroupInstances of LambdaDeployments
  - `infinicache-master/proxy/server/meta.go` - represents an object to be stored in cache (called `meta`)
  - `infinicache-master/proxy/server/metastore.go` - a store of metas
  - `infinicache-master/proxy/server/placer.go` - implements a Clock LRU for object/metas removal
  - `infinicache-master/proxy/server/proxy.go` - the main file of the Proxy server implementation which initializes the lambda group and manages client operations. It uses other local packages to manage things such as `infinicache/proxy/collector`, `infinicache/proxy/lambdastore`
  - `infinicache-master/proxy/server/scheduler.go` - controls the Group. And the Group controls the lambda cache nodes in this InfiniCache deployment.
- `main` contains files for deploying a lambda function in AWS, the handler for the function and the proxy entry point:
  - `infinicache-master/deploy/deploy_function.go` - deploys the function (Lambda) on AWS, hence uses `github.com/aws/aws-sdk-go`
  - `infinicache-master/lambda/handler.go` - handler file for Lamda Runtime, it is written for AWS Lambda runtime, hence imports AWS SDK, in particular `github.com/aws/aws-lambda-go/lambda and` and `github.com/aws/aws-lambda-go/lambdacontext`. Basically it handles the commands provenient from proxy.
  - `infinicache-master/proxy/proxy.go` - main handler file for proxy, it lunches/initializes the proxy server and all the components that it uses

- `collector` is a metrics collector. The package contains the following files
  - `infinicache-master/lambda/collector/collector.go` - is responsible for collecting some metrics such as latency breakdown in the lambda runtime side. This file is mainly focused on the performance evaluation part. (uses github.com/aws/aws-sdk-go/)
  - `infinicache-master/proxy/collector/collector.go` - is also responsible for collecting metrics like latency, but it is on the proxy and client side
- `storage` used by lambda runtimes to store data chunks. Contains files:
  - `infinicache-master/lambda/storage/storage.go` - chunk storage for cache nodes
- `migrator` implements the relay that allow to perform the backup protocol for maximizing data availability. Given that AWS Lambda does not inbound TCP/UDP connections, the authors of infinicache came out with a protocol wich overcomes this disadvantage. As described in the original paper this is highly related to AWS implementation. This package contains the following files:
  - `infinicache-master/lambda/migrator/client.go`
  - `infinicache-master/lambda/migrator/intercept_reader.go`
  - `infinicache-master/lambda/migrator/storage_adapter.go`
  - `infinicache-master/migrator/forward_connection.go`
  - `infinicache-master/migrator/migrator.go`
- `lifetime` - Cit. "Each Lambda runtime is warmed up after every T_warm interval of time" the authors use a T_warm value of 1 minute as motivated by observations made in the original paper. This package should manage Functions lifespan. This package contains the following files:
  - `infinicache-master/lambda/lifetime/lifetime.go`
  - `infinicache-master/lambda/lifetime/session.go`
  - `infinicache-master/lambda/lifetime/timeout.go` - this file uses `github.com/aws/aws-lambda-go/lambdacontext` but just to choose some constants based on the type of instance of lambda in therms of CPU power, hence it can be easily replaced with some other api call that does the same

- `lambdastore`: manages the pool of cache nodes and the connections lifecycles
  - `infinicache-master/proxy/lambdastore/connection.go`
  - `infinicache-master/proxy/lambdastore/deployment.go`
  - `infinicache-master/proxy/lambdastore/instance.go`
  - `infinicache-master/proxy/lambdastore/meta.go`
- `types` contains types definitions used by the packages. contains the following files:
  - `infinicache-master/common/types/types.go`
  - `infinicache-master/lambda/types/response.go`
  - `infinicache-master/lambda/types/types.go`
  - `infinicache-master/proxy/types/control.go`
  - `infinicache-master/proxy/types/request.go`
  - `infinicache-master/proxy/types/response.go`
  - `infinicache-master/proxy/types/types.go`

Notice that test files (which can be identified under `*_test.go` filename) are omitted from this description.

3.3 Migration from AWS Lambda to Knative Analysis

We want to analyze if moving Infinicache from AWS Lambda to Knative is doable. We want to do this in order to bring Infinicache power to an open infrastructure which would allow us to investigate other types of optimizations. Moreover we want to analyze if the migration could overcome actual limitations of infinicache due to the specific underlying platform. In order to analyze the feasibility of migration, the evaluation was abstracted by the following questions:

- Which are the theoretical dependencies of Infinicache from AWS Lambda?
- How these dependencies are translated in the Go implementation?
- Could Knative solve these limitations?

We list what emerged to be most evident limitations of AWS Lambda as Infinicache underlying implementation trying to answer in parallel the above questions.

1. **AWS Lambda do not offer TCP/UDP inbound connections**

   Hence Infinicache has a lot of code which has to go around this problem. In particular it has an entire part of the protocol designated to maintain a reliable network connection.

   Infinicache solves this problem by maintaining a TCP connection open between the Proxy and the Cache Nodes. However, given the nature of the nodes build over AWS Lambda, this do not ensures that the connection will not be reclaimed by the provider. Moreover, given the timeout mechanism introduced for optimizing billing, a node may return at any time. For these reasons, the proxy lazily validates the state of a cache node each time there is a request to send.

   This behaviour is embedded in the implementation of infinicache in the `lambdastore` package.

   Knative could solve this problem, because it is built over Kubernetes hence inbound connections are on the agenda. This will remove a lot of code from the project, but of course will require lots of integrations with some new connection management api.

2. **Anticipatory Billed Duration Control**

   AWS charges Lambda usage per 100 msv (a billing cycle). To maximize the use of each billing cycle, Infinicache's Lambda runtime uses a timeout scheme to control how long a cache node runs. From the paper emerges that this is a configurable option, hence the billing cycle value can be changed.
   Timeout mechanism: When a Lambda node is invoked by a chunk request, a timer is triggered to limit the function's execution time. The timeout is initially set to expire within the first billing cycle. The runtime employs a simple heuristic to decide whether to extend the timeout window. If no further chunk request arrives within the first billing cycle, the timer
   expires and returns 2–10 ms (a short time buffer) before the 100 ms window ends. This avoids accidentally executing into the next billing cycle. The buffer time is configurable, and is empirically decided based on the Lambda function's memory capacity. If more than one request

can be served within the current billing cycle, the heuristic extends the timeout by one more billing cycle, anticipating more incoming requests.

Timeout mechanism implemented by the `lifetime` package which allows also to configure such procedure.

If Infinicache would be ported over Knative, this option will highly depend on the cloud provider. Assuming that Knative would run over Google Cloud provider, this will highly depend on how K8s is configured to run our serverless workloads and which Google Cloud Products will be used to allow K8s to do that. This is true because Knative could be installed everywhere where K8s is installed, hence it could be also installed in AWS. A deeper analysis on Google Cloud Pricing for running serverless workloads using Knative has to be done.

* However it seems that this option is configurable, hence once we know how serverless workload is charged by the cloud provider, this option can be tuned.

3. **Eliminating Lambda Contention**

   Lambda functions are hosted by EC2 Virtual Machines (VMs). A single VM can
   host one or more functions. AWS seems to provision Lambda functions on the smallest possible number of VMs using a greedy binpacking heuristic. This could cause severe
   network bandwidth contention if multiple network-intensive Lambda functions get allocated on the same host VM.
   Authors choice: While over-provisioning a large Lambda node pool with many small Lambda functions would help to statistically reduce the chances of Lambda co-location, we find that using relatively bigger Lambda functions largely eliminates Lambda co-location. Lambda's VM hosts have approximately 3 GB memory. As such, if we use Lambda functions with ≥ 1.5 GB memory, every VM host is occupied exclusively by a single Lambda function, assuming INFINICACHE's cache pool consists of Lambda functions with the same configuration.

   Knative can help a lot in eliminating lambda contention. Of course once again the capacity of the host VM depends on the cloud provider, but Knative allows developers to fulfill needs and use Kubernetes functionalities whenever Knative is not enough. I believe the provision

algorithm of Kubernetes can however be better investigated than the ones from AWS.

4. **Preflight Message**

   While the proxy knows whether a Lambda node is running or has already returned, it does not know when a Lambda node will expire and return. Because of the billed duration control design that was just described, a Lambda node may return at any time.
   Authors solution: Instead of pooling the nodes, the proxy issues a preflight message (PING) each time a chunk request is forwarded to the Lambda node. Upon receiving the preflight message, the Lambda runtime responds with a PONG message, delays the timeout (by extending the timer long enough to serve the incoming request), and when the request has been served, adjusts the timer to align it with the ending of the current billing cycle. To further reduce overhead, the proxy can attach the PING message as a parameter of a Lambda function.

   The Go implementation just implements the Ping-Pong mechanism. However, in Knative, given that inbound TCP connections are allowed, the cache node could send itself a message to the proxy, notifying that it terminated due to no more requests received. This will for sure reduce the number of messages that transit in the network but also simplify the protocol in some ways.

3.4 Personal Conclusion

The migration of Infinicache in Knative is doable. For sure design changes have or would be better to be done in order to improve portability, efficiency and simplicity of the service. The actual implementation has many packages that seem to be reusable, but it seems that many of them are tightly coupled. A further analysis on their level of coupling and cohesion has to be done. Further discussion about possible re-implementation and possible re-utilization of already implemented components is needed.

Other topic related interesting web pages, articles and/or webinars:

Stop Calling Knative Serverless! - Doug Davis, IBM

Stop Calling Knative Serverless! - Doug Davis, IBM By now most people are familiar with what it means to be a serverless platform, features such as scale-to-...

https://www.youtube.com/watch?v=28CqZZFdwBY

Pricing pitfalls in AWS Lambda

Many companies, including one of my previous employers Yubl, were able to realise huge savings by moving to serverless. AWS Lambda itself is really cheap, but there are

https://blog.binaris.com/lambda-pricing-pitfalls/

Knative - IBM Developer

Run serverless containers on Kubernetes.

https://developer.ibm.com/components/knative/