Sara

Cover

Hello! My name is Sara, my group colleagues are Ricardo and Rafael, and we will present to you the paper about Pocket: An Elastic Ephemeral Storage for Serverless Analytics.

Introduction

This work focuses on the ephemeral storage used on serverless computing, more specifically on serverless analytics. Serverless computing has become more popular for data intensive applications, like interactive analytics since it allows for fine-grain billing and high elasticity.

Unlike most serverless applications, analytics require the exchange of data across multiple stages of the tasks. The data exchanged between tasks or jobs, that is neither output nor input is referred as ephemeral data.

The availability of multiple storage medias increases the complexity of choosing a cluster configuration that balances performance and cost. Different jobs may present different requirements, so choosing the best configuration for each job is critical but complex.

So, the paper authors presented Pocket.

Why is this important?

Pocket is the first platform targeting data sharing in serverless analytics. It allows users of serverless computing to deploy their work without concerns regarding the configuration and resizing of the cluster.

The short-lived memory storage is a concern that multiple serverless analytics systems may face. Pocket brings a fast and well-defined approach that automatically balances the performance and cost of the system.

Storage for Serverless Analytics

We start by analysing what are the requirements of ephemeral storage and, identify the essential properties for an ephemeral data storage solution. Then we show that the current systems are not able to meet those requirements.

- ***Ephemeral Storage Requirements***
    - ***High performance for a wide range of object sizes***
      As presented in the left graph we see the object size distribution for a compilation of the cmake program called lambda-cc, a serverless video analytics job and a 100 GB MapReduce sort.
      We can see that the granularity of ephemeral data access varies greatly in size so an ephemeral data storage must deliver high bandwidth, low latency, and high IOPS (input and output operations per second) for the entire range of object sizes.
    - ***Automatic and fine-grain scaling***
      Scaling up or down to meet elastic application requires a storage solution capable of growing and shrinking in multiple resource at a fine time granularity. This should also be automatic since cluster configuration performance-cost trade-offs is a burden for users.

- ***Storage technology awareness***
  As mentioned before the availability of multiple storage systems increases the complexity of the configuration of the cluster, to users, so, an ephemeral data store must place application data on the right storage technology tier(s) for performance and cost efficiency.
- ***Fault-(in)tolerance***
  As we can see in the right graph the lifetime of ephemeral objects is very low. Unlike long-term storage, where the cost of data unavailability outweighs the cost of the fault-tolerance mechanisms, the ephemeral data is only valuable during the execution of a job and can be easily regenerated. For these reasons, an ephemeral storage solution does not have to provide high fault-tolerance as expected of traditional storage systems.

- ***Existing Systems***
  The table presents the characteristics of different storage systems compared to what is desired (presented in the last line). As we can see no system fully satisfies que desired properties and those that come close in some attributes present a high cost.
  This inspired the design of Pocket.

Pocket Design

Pocket is an elastic distributed storage service for ephemeral data that addresses the previously mentioned requirements. It has 3 key principals:

- ***Separation of responsibilities***
  Pocket divides responsibilities across three different planes: the control plane, the metadata plane, and the data plane. The control plane manages cluster sizing and data placement. The metadata plane tracks the data stored across nodes in the data plane.
- ***Sub-second response time***
  Pocket's storage servers are optimized for I/O operations only storying data, making them scalable. Also, the controller, from the control plane, can scale resources at second granularity. These characteristics make Pocket elastic.
- ***Multi-tier storage***
  Pocket ranks the different storage media into tiers that satisfies the I/O demands of the application while minimizing cost.

Pocket Architecture

The diagram bellow shows Pocket's system architecture. This consists of a centralized controller, one or more metadata servers, and multiple data plane storage servers.

- The controller allocates storage resources for jobs and dynamically scales Pocket up and down as the number of jobs and their requirements vary over time.
- Metadata servers enforce data placement policies generated by the controller by steering client requests to appropriate storage servers.
- Clients access data blocks on optimized storage servers equipped with different storage media.

Ricardo

Application Interface

Clients can interact with Pocket through its object storage API, adapted to the ephemeral storage use case.

- **Control functions**: Regarding the controller, there are 2 API calls that it exposes, which are issued only once per job.
    - The *register_job* which is called by the Application if it intends to start a job. This call returns a job identifier and the metadata server assigned to manage the job data. Optional hints can be passed by the app on this call, regarding latency, throughput or expected capacity that the job consumes.
    - The other controller API call is *deregister_job*, which notifies the controller that the job is complete, so that the controller releases resources.

- **Metadata functions:** As for API calls from metadata servers, unlike those from the controller, they can be issued several times per job.
    - The *connect* is issued to establish a connection to a metadata service.
    - The *close* is issued to close that connection.
    - Furthermore, Job data is organized into buckets so that an object of arbitrary size can be stored on multiple storage servers. Both are identified with names and there is a set of API calls that clients can use to create and delete buckets, enumerate objects in a bucket, verify that an object exists and delete it.

- **Storage functions**: Regarding storage servers, they expose 2 API calls.
    - The *put* is used to write an object. When its *PERSIST* flag is set to false, the object will be collected as garbage right after the job is completed, if its *PERSIST* flag is set to true, the object will remain even after the job is over.
    - The *get* is used to read an object. When its *DELETE* flag is set to true, right after reading the object it is deleted, in order to make garbage collection more efficient and make the most of the allocated resources.

Life of a Pocket Application

Now let's go through the steps that a Pocket app goes through.

Before starting the lambdas of the job, the application registers the job in the controller, which allocates the resources that it predicts the job will consume. It then returns the job identifier that must be provided in API calls with the storage servers and the IP address of the top-level metadata server that should be used to find out which storage servers to contact. Whenever the application wants to read from or write to the storage servers, it first contacts the metadata servers that forward it to the correct storage server. When the last lambda of the job ends, the application unregisters the job to free up resources.

Rightsizing Application Allocation

Pocket takes advantage of optional hints passed when the job is registered to conservatively estimate its latency, throughput, and capacity requirements, finding an economical allocation of resources.

- **Determining job I/O requirements**:
    - By default, Pocket assumes that a job is sensitive to latency, filling the job's resources with DRAM, if a hint is passed saying it is not sensitive to latency, it does not use DRAM, minimizing the cost.
    - Another hint that can be taken advantage of, is the maximum number of lambdas, that when they are not passed, Pocket conservatively assumes the limit of the cloud provider, on AWS, 1000, allocating more resources than it really needs.
    - In addition, when only either throughput or capacity is specified, the Pocket only allocates the proportional amount.
- **Assigning resources**:
    - Pocket allocates resources by generating a weight map for each job, that is, a map whose key is the IP and port of a storage server, and the value represents the fraction of the dataset of a job to place on that storage server. The controller sends this map to the metadata servers that enforce the data placement policy. When job requirements cannot be met by currently active storage servers, the controller launches new ones.

Rightsizing the Storage Cluster

In order for Pocket to be elastic, it continuously monitors the cluster nodes and decides when and how to scale the storage and metadata servers.

- **Mechanisms**:
    - Pocket contains two scale mechanisms, the horizontal and the vertical.
    - With regard to horizontal scaling, each node has a running process that sends statistics to the controller, which in turn processes it and decides whether it is necessary to launch new nodes.
    - As for vertical scaling, when the controller discovers that a node has high CPU Utilization and it has additional cores available, the controller instructs that node to use those additional cores.
- **Cluster sizing policy**:
    - The pocket sizing policy consists of keeping each type of resource within a configurable target range. It scales up when any type of resource exceeds the upper threshold defined for it and scales down when all types of resources are below its lower threshold, preferring vertical rather than horizontal scaling whenever possible, in order to avoid the delay in the launch of a new node.
- **Balancing load with data steering**:
    - To balance the load between the cluster, the controller assigns higher weights in the incoming jobs to the nodes that are underutilized, so that, based on the statistics it receives, the use of resources from that node remains within the acceptable range.

Rafael

Evaluation Methodology – Introduction

- To do a concrete evaluation**,** Pocket Storage, Metadata and Controller nodes were deployed on EC2 instances of AWS and this table resumes its configurations.
- For the storage nodes, different types of storage media were deployed: DRAM, NVMe, SSD and HDD.
- For the serverless computing platform it was used AWS Lambda.

Evaluation Methodology

For this evaluation 3 different serverless analytics applications were used that we briefly talked about at the beginning, which differ in the degree of parallelism, ephemeral object size distribution and throughput requirements.

The first is video analytics, which has two stages, the first that has 160 lambdas and the second that has 305 lambdas.

The second is MapReduce where it was executed a one hundred gigabytes sort, which generates one hundred gigabytes of ephemeral data. The job was executed using 250, 500, and 1000 lambdas.

Finally, lambda-cc, where the build job has a maximum parallelism of 650 tasks and generates a total of 850 megabytes of ephemeral data.

Evaluation – Microbenchmarks - Storage Request Latency

Let's first analyse the storage request latency, in this figure we have the latency times for different systems, S3, Redis and Pocket using different storage tears.

As you can see, Pocket using DRAM or NVMe, and Redis is over forty-five times faster than S3, and the increased latency for Pocket using SSD and HDD is because of their higher storage access times.

Pocket using DRAM has higher latency than Redis mainly because Redis cluster clients simply hash keys to nodes, where the Pocket clients must contact a metadata server. While this step increases latency, it allows Pocket to optimize data placement per job and dynamically scale the cluster without redistributing data across nodes.

Evaluation – Microbenchmarks - Metadata Throughput

To test the metadata throughput, it was measured the number of operations that a metadata server can handle.

The results showed that a single core metadata server supports up to ninety k operations per second and a quad core up to one hundred and five K operations per second.

The peak of metadata request rate observed for the applications studied was seventy-five operations per second per lambda, and so a multi-core server can handle jobs with thousands of lambdas.

Evaluation – Microbenchmarks - Adding or Removing Servers

For evaluating the addition and removal of servers it was made a breakdown of node start-up time.

The results are in this figure, where we can see that most of the amount of time produced is by the VM start-up and container image pull, but they only need to be done once, and so, after that, starting and stopping containers takes only a few seconds.

The time to terminate a VM is in the range of tens of seconds.

Evaluation - Rightsizing Resource Allocations - Rightsizing with application hints

To prove how Pocket leverages user hints to make cost effective resource allocations, the three applications were evaluated, and the results are present in this figure.

Having no knowledge of the requirements, makes it default to spreading data for a job across fifty nodes, filling DRAM first and then NVMe.

With knowledge of the maximum number of concurrent lambdas, pocket allocates a lower throughput than the default allocation, while maintaining a similar job execution time.

These jobs are not sensitive to latency, and so when the hint is given, it uses NVMe as opposed to DRAM which minimizes the cost.
When the application provides explicit hints for their capacity and peak throughput requirements, it obtains the least normalized resource cost.

Evaluation - Rightsizing Resource Allocations - Reclaiming capacity using hints

As ephemeral data in video analytics is written and read only once, the DELETE hint can be given, which enables Pocket to garbage collect an object as soon as it is read, which as shown in the figure allows the reclaiming of capacity more quickly and making a more efficient use of resources.

Evaluation - Comparison with S3 and Redis

In order to compare Pocket with S3 and Redis it was tested the average time per lambda on each of the applications.

The figure at the left shows the results for MapReduce, which as you can see both Redis and Pocket have similar performance, while S3 has lower performance and sometimes it does not even work.
For the video analytics the results in terms of performance were the same.
For the lambda-cc, because the job has limited parallelism, the overall execution time remained the same between the systems.

The table at the right shows the hourly cost of running Pocket nodes.
Using all these information it is possible to observe, that pocket achieves the same performance as Redis for all three jobs while saving fifty nine percent in cost. S3 is still cheaper, but although it has a similar performance on lambda-cc, for the others the execution time is forty to sixty five percent higher.

Conclusion

Concluding, this paper analysed the challenges associated with efficient data sharing as well as presented Pocket which provides a high elastic, cost-effective and fine-grained storage solution for analytics workloads.

The evaluation of Pocket demonstrated how it offers high performance on data access for different data sets sizes, combined with automatic fine-grain scaling and cost-effective data placement across storage tiers.

We hope you enjoyed our presentation. Thank you!