Cloudburst: Stateful Functions-as-a-Service

Han Liang Wee Eric

Slides at https://eric-han.com/cloudburst.pdf
Other papers – Anna: A KVS For Any Scale

4 Sept 2020

Serverless Computing

How we got here?

On-demand Software (SaaS)
Gmail - Shopify

Serverless (FaaS)
AWS-Lambda - Gcloud-Fn

Container (PaaS/CaaS)
GAE / GKE

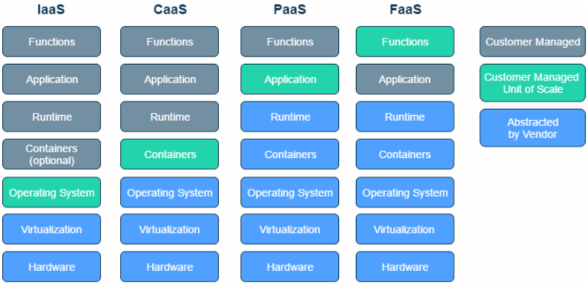
Virtual Machine (IaaS)
EC2 - AzureVM

Bare-Metal (On-Premises)
AWS - GCloud

Function as a Service (FaaS)

What exactly is it?

The ultimate abstraction? – A function.



[1] Img from serverless.zone

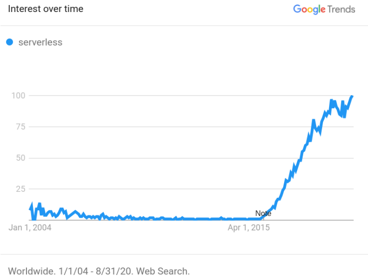
Function as a Service (FaaS)

Why use it?

Reduced administrative burden:

- **Fast**: Reliable deployment
- **Elastic**: Autoscaling handled by design
- **Easy**: Difficult to write /maintain services at scale
- **Abstraction**: Operations handled externally

Case Study: Netflix



[1] Img from Google Trends

Motivation – Function as a Service (FaaS)

What is the problem?

Leading FaaS providers: ▶ AWS λ ▶ GCloud Fn ▶ MS Azure Fn ▶ IBM/Apache OpenWhisk ▶ Oracle Cloud Fn

Limitations due to 'disaggregation taken to an exteme' (isolated, stateless functions):

1. Limited execution behavior
2. High Latency
3. Cannot communicate between fn
4. (Embraced as general computing)

Applications: ▶ ExCamera ▶ numpywren

Motivation – Function as a Service (FaaS)

What is the problem?

Leading FaaS providers: ▶ AWS λ ▶ GCloud Fn ▶ MS Azure Fn ▶ IBM/Apache OpenWhisk ▶ Oracle Cloud Fn

Problems tackled via shared state management:

- **Function composition** – #4, #1
- **Direct communication** – #3
- **Shared mutable storage** – #2

Applications: ▶ ExCamera ▶ numpywren

Cloudburst: Stateful FaaS Platform

Intuition & Idea & Goal – Stateful Serverless via LDPC

Stateful serverless via

Logical *Disaggregation* with Physical *Colocation* (LDPC)

Disaggregation: Provision, scale.
Colocation: Deploy resources to compute in close proximity.

Contributions:

- Implement serverless with LDPC: Cloudburst
- Distributed session consistency (DSC) – repeatable read, casual consistency
- Ease of use, abstraction of coordination storage
- Performance and consistency evaluation

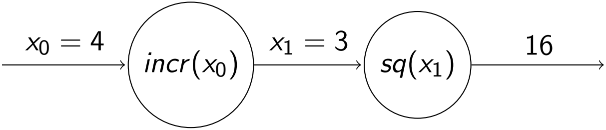
Cloudburst: Stateful FaaS Platform

Using cloudburst

$$f(x) = sq(incr(x)) : sq(x) = x^2, incr(x) = x + 1$$

```
>>> from cloudburst.client.client import CloudburstConnection
>>> local_cloud = CloudburstConnection('127.0.0.1', '127.0.0.1', local=True)
>>> cloud_sq = local_cloud.register(lambda _, x: x * x, 'sq')
>>> cloud_sq(2).get()
4
>>> local_cloud.register(lambda _, x: x+1, 'incr')
>>> local_cloud.register_dag('f', ['incr', 'sq'], [('incr', 'sq')])
>>> local_cloud.call_dag('f', {'incr': [3]}).get()
16
```

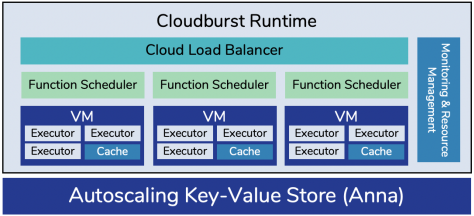
Arbitrary compositions as DAG (like Spark, Tensorflow etc...):



[1] My forked repo eric-vader/cloudburst

Architecture

Roles & Responsibilities



Built upon Anna Key-Value-Store (KVS) with 4 key components:

- **Fn schedulers** – Route fn invocation requests
- **Fn executors** – Resolve KVS references & DAG
- **Caches** – Ensure that frequently used data is local, fresh (L1)
- **Resource management system** – Optimize based on metrics

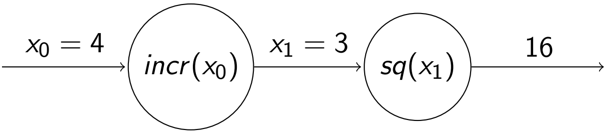
Component Responsibilities

Resource management system

- Uses Kubernetes to manage and scale cluster
- Metrics collected over Anna
- Optimize based on current hueristics (dynamic?)

Component Responsibilities

Fn schedulers



- **Scheduling Mechanism** – Fn/DAG aware
  - Register/invoke functions
  - Decide schedule for DAG
- **Scheduling Policy** – Metrics from executors
  - Data locality – Colocation
  - High utilization of executors

DAG can execute across nodes ($\because$ Distributed session consistency)

Component Responsibilities

Fn executors

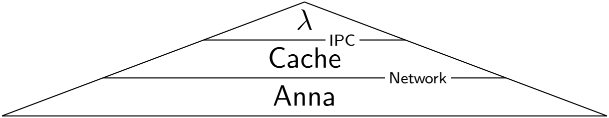
$$\xrightarrow[\text{Cache}]{x_0 = 4} \; incr(x_0) \; \xrightarrow[\text{Cache}]{x_1 = 3}$$

Executor is an independent, long-running Python process:

- **Before** (Parallel)
  - Retrieve & deserialize function
  - Resolve function arguments
- **After**
  - Triggers downstream fn
  - Caches result
- **Collect Metrics** – CPU stats, cache stats, execution latencies

Component Responsibilities

Caches



Executor will only interact with cache (ie. memory controller):

- **Ensures frequently-use data is local**:
  - Executor updates to cache, cache async updates Anna
  - Executor requests from cache, cache async fetches from Anna
- **Ensures frequently-use data is fresh**:
  - Cache publish snapshot of cache keys to Anna
  - Anna uses the index to propagate key updates (conflict?)

Distributed session consistency needs to be addressed

Distributed session consistency

We denote k_i influences l_j with $k_i \rightarrow l_j$, ie. if a read of k_i happens before a write of l_j.

For the flexibility of DAG to be executed across nodes:

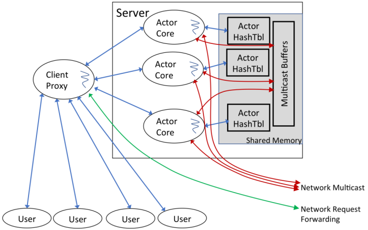
- **Repeatable Read**: In a linear DAG, when any function reads k, it sees the most recent, updated k in the DAG.
  ie. $f(x, g(x))$ – f and g sees the same initial version x_0
- **Casual Consistency**: The version of k read by a function f must be concurrent to or newer than any version of k in D, where $D = \{d_i : d_i \rightarrow l_j \in V\}$ and V are the versions previously read by f or its ancestors.

Distributed session consistency

Anna: A KVS For Any Scale

Each actor in an epoch:

1. Compile changeset – incoming requests from clients
2. Multicast changeset to relevant masters
3. Merge incoming multicast messages into its local state



Built upon: ▶ Coordination-free Actors ▶ Lattice-Powered, Coordination-Free Consistency ▶ Cross-Scale Validation

Distributed session consistency

Anna: A KVS For Any Scale

Bounded join semilattice (aka Lattice):

- **Commutativity**: $\sqcup(a, b) = \sqcup(b, a)$
- **Associativity**: $\sqcup\Big(\sqcup(a, b), c\Big) = \sqcup\Big(a, \sqcup(b, c)\Big)$
- **Idempotence**: $\sqcup(a, a) = a$

Domain $a, b, c \in S$, Bin op. $\sqcup$ least upper-bound, Min value $\perp$

Anna is able to leverage on them to achieve:

- Insensitive to merge updates - updates merged at sender

$$\sqcup\Big(\sqcup(s, u_1), u_2\Big) = \sqcup\Big(s, \sqcup(u_1, u_2)\Big)$$

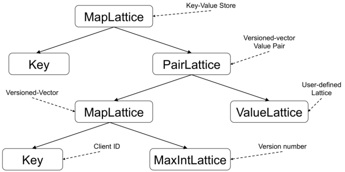
- Configurable to various consistency levels

Distributed session consistency

Anna: A KVS For Any Scale

Lattice Composition for casual consistency

- **Global Timestamp** – $< id, clock >$
- **PairLattice** – Last Writer Wins lattice



For every key, PairLattice merge ($\sqcup$) wrapping:

- Where casual ordered, always returns the most recent version.
- Where mulitple concurrent version exist, one version is chosen via arbitrary tie-breaking. (Gotcha!)

Distributed session consistency

Algorithm to address the gotcha

Algorithm 2 Causal Consistency

Input: k, R, $dependencies$

// k is the requested key; R is the set of keys previously read by the DAG; $dependencies$ is the set of causal dependencies of keys in R

Repeatable Read

if $k \in R$ **then**
    $cache\_version = cache.get\_metadata(k)$
    // $valid$ returns true if $k \geq cache\_version$
    if $valid(cache\_version, R[k])$ **then**
        return $cache.get(k)$
    else
        return $cache.fetch\_from\_upstream(k)$

Casual Consistency (Gotcha: consider dependency)

if $k \in dependencies$ **then**
    $cache\_version = cache.get\_metadata(k)$
    if $valid(cache\_version, dependencies[k])$ **then**
        return $cache.get(k)$
    else
        return $cache.fetch\_from\_upstream(k)$

Experiments

Setup

Hardware, all in AWS us-east-1a:

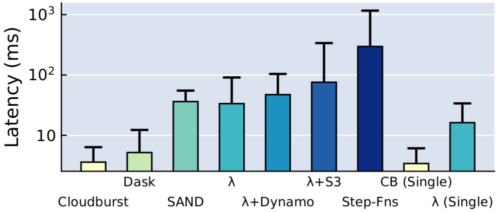
- c5.large EC2 (1C) – Fn schedulers
- c5.2xlarge EC2 (4C) – Fn executors(3), Cache(1)
- Clients are on seprate machines

Detailed Evaluation:

- Individual Mechanisms
- Consistency Models
- Case Studies

Experiments

Individual Mechanisms

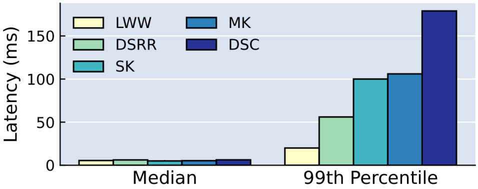


- **Fn Composition** $sq(incr(x))$ – Better by 1-3 orders of mag
- **Data Locality** – Cloudburst better than AWS λ w (Redis/S3)
- **Low-Latency Comm.** – Better than AWS λ w (Redis/S3)
- **Autoscaling** – Cloudburst is responsive to load changes

Note that AWS λ w (Redis/S3) are workarounds.

Experiments

Consistency Models' overheads



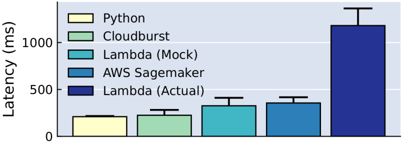
Different consistency models evaluated

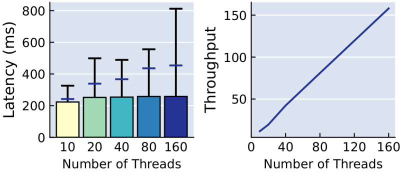
- **Latency** – Increases tail latencies, but median acceptable
- **Inconsistencies** – Able to detect and prevent these anomalies

Cloudburst: DSC - Distributed session consistency

Experiments

Case Studies



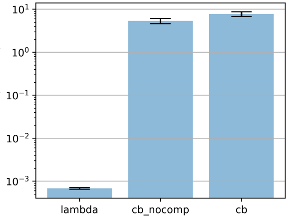


- **Prediction Serving** – Comparable with custom app. - AWS Sagemaker, closest to native Python
- **Retwis** – Ease of implementation, converting Retwis to Cloudburst, a little slower than Redis

Ignore the poor color on the right diagram

Experiments

My Experiment – For Repeatability's Sake



100 experiments, where X_i is 100 randomly selected ints w seed i:

$$\forall x \in X_i : f(x) = sq(incr(x)) : sq(x) = x^2, incr(x) = x + 1$$

- Repeatability
- Ease of use
- Protocol Overhead
- Intel i7-6600U (4T), 32 GiB, conda env
- Timeit with 10 iterations

[1] In my repo eric-vader/cloudburst/my_test

Summary

Conclusion

Demonstrated the feasibility of general-purpose stateful serverless computing

- ✓ Implement serverless with LDPC: Cloudburst
- ✓ Distributed session consistency (DSC) – repeatable read, casual consistency
- ✓ Ease of use, abstraction of coordination storage
- ✓ Performance and consistency evaluation... (*is it fair?*)

Notable Future work (but not limited to):

- Autoscaling Policy
- Fault Tolerance

Summary
Criticism

"It is unfortunately common for many in academia to overweight the value of ideas and underweight bringing them to fruition. For example, the idea of going to the moon is trivial, but going to the moon is hard."

— Elon Musk

- **Wild CB performance** – Against the wild AWS, SAND
  - Operational cost considerations
  - Production overheads
- **Network multicast performance** – 60 nodes, multicast small
- λ **Applications** – Web app, backends
  - Which means better engineering can be done

Q & A