CLOUDBURST : Stateful Functions-as-a-Service

Saanidhi Arora

Introduction

- Function-as-a-Service (FaaS) platforms and "serverless" cloud computing are gaining popularity
- Serverless target stateless functions with minimal I/O and communication
- Cloudburst: a stateful FaaS platform

FaaS & Serverless Computing

ADVANTAGES

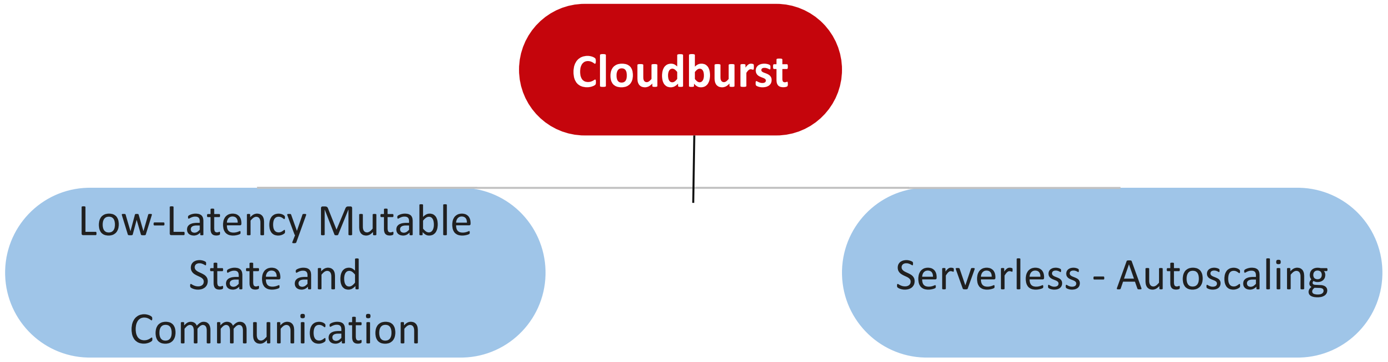
- Simplified Cloud Development
- Transparent Autoscaling using Disaggregation of Storage and Compute Services

ISSUES

- Ineffective Function Composition
- Lack of Direct Communication
- High Latency Access to Shared Mutable State

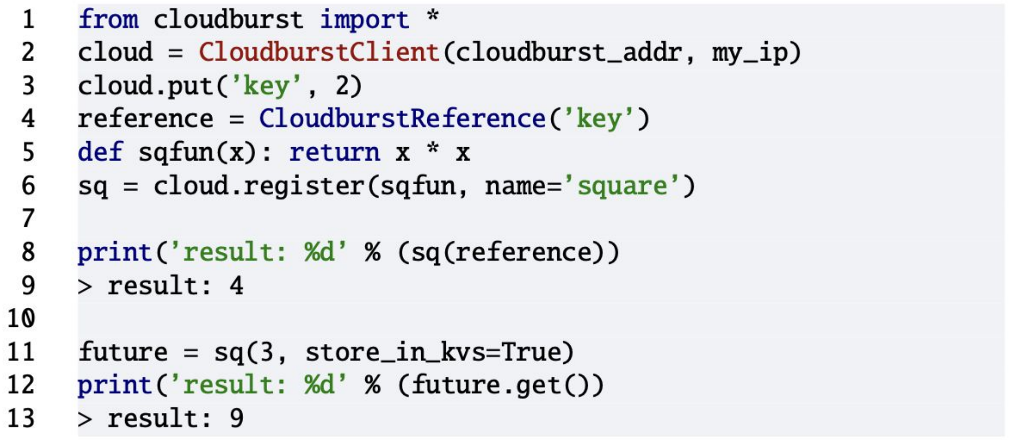
Solution

- Logical Disaggregation with Physical Colocation - distributed storage and local caching
- Coordination-free consistency - quorum expensive
- Programmability - easy for developers



Programming Interface

- Cloudburst functions - Regular python functions triggering remote computation in the cloud
- Results - directly to client or stored in KVS retrieved using CloudburstFuture object
- Function Arguments - KVS references or Python objects
- Function compositions as DAGs



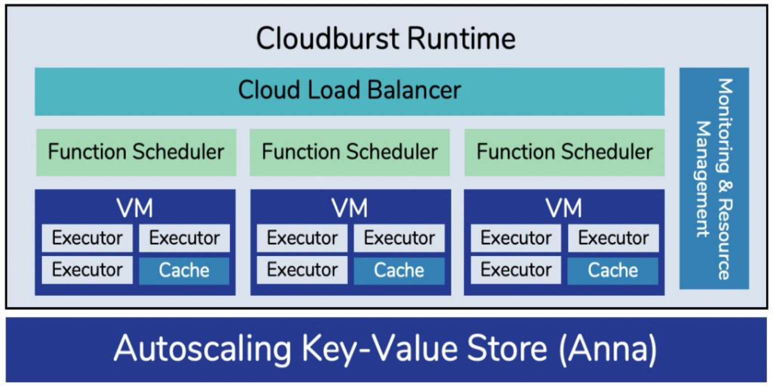
```python
from cloudburst import *
cloud = CloudburstClient(cloudburst_addr, my_ip)
cloud.put('key', 2)
reference = CloudburstReference('key')
def sqfun(x): return x * x
sq = cloud.register(sqfun, name='square')

print('result: %d' % (sq(reference))
> result: 4

future = sq(3, store_in_kvs=True)
print('result: %d' % (future.get())
> result: 9
```

Architecture

- **Function executors** - long running Python process
- **Caches** - frequently accessed data
- **Function Schedulers** - route invocation requests, packed into VMs
- **Monitoring and Resource Management System** - tracks system load and performance



Architecture

Function Scheduler

- Goal - low latency function scheduling
- Scheduling Mechanisms - register or invoke functions. New functions registered - store in Anna, update shared KVS list
- Scheduling Policy - heuristic decisions using metadata, prioritize data locality. Allocate or deallocate resources based on the workload

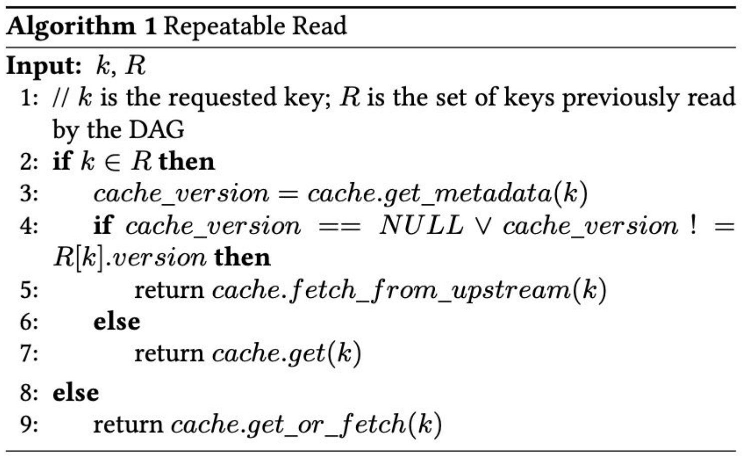
Fault Tolerance

- Anna's k-fault tolerance mechanism
- DAG re-execution when machine failure

Consistency Guarantees

- Every function in DAG - executed on different machines, causing inconsistencies
- **Repeatable Read Invariant** - read most update version of key within DAG, without updates - all functions same version
- **Causal Consistency Invariant** - reads and writes respect Lamport’s “happens before” relation. If $k_i \rightarrow l_j$ and l_j is read, subsequent functions must not see any version before k_i

Distributed Session Protocol

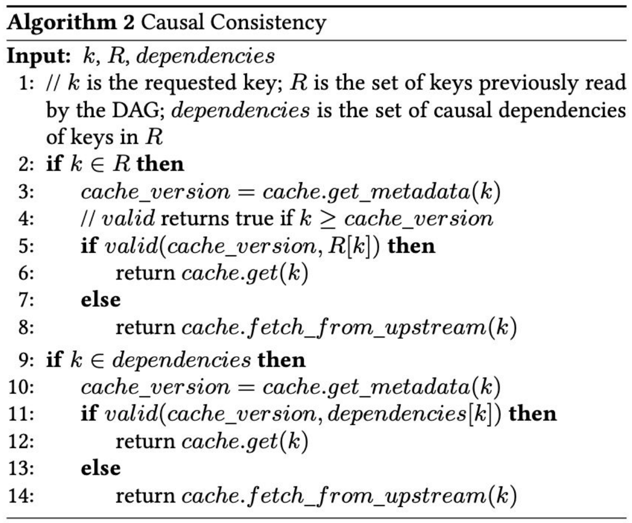


Algorithm 1 Repeatable Read

```
Input: $k$, $R$
1: // $k$ is the requested key; $R$ is the set of keys previously read by the DAG
2: if $k \in R$ then
3:     $cache\_version = cache.get\_metadata(k)$
4:     if $cache\_version == NULL \vee cache\_version\ != R[k].version$ then
5:         return $cache.fetch\_from\_upstream(k)$
6:     else
7:         return $cache.get(k)$
8: else
9:     return $cache.get\_or\_fetch(k)$
```

- Consistency across functions
- **Repeatable read**
- Cache creates snapshot version of locally cached objects on first read
- Propagate cache address list and version timestamp to downstream executors
- If read, version not stored locally - fetch from upstream cache
- If read, stored locally - returns cached value
- If not read, any version

Distributed Session Protocol



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

- **Causal Consistency**
- Causally consistent cache store
- Cache stores causal cut - stores key versions and dependencies
- Propagate Read Set Metadata + Causal Dependencies
- Check if local cached key's vector clock is causally concurrent
- If yes - return local value, else query upstream cache

Lattice Encapsulation

- Resolve conflicts from concurrent updates in Anna
- Cloudburst encapsulates python objects into lattices
- Last Writer Wins Lattice - eventual consistency, global timestamp and value. Use the last value
- Causal consistency Lattice - Key k in lattice - Anna vector clock-> k’s version, what all keys k depends upon. Choose which vector clock dominates

Evaluation

- **Setup** - us-east-1a AWS AZs, Schedulers - AWS c5.large EC2 VMs , and function executors - c5.2xlarge EC2 VMs
- **Mechanism Evaluation - Function Composition** square(increment(x:int))

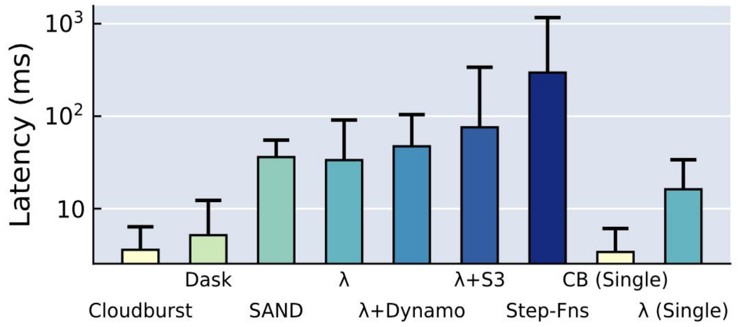


Figure 1: Median (bar) and 99th percentile (whisker) latency for `square(increment(x: int))`. Cloudburst matches the best distributed Python systems and outperforms other FaaS systems by over an order of magnitude (§6.1).

Cloudburst's function composition matches state-of- the-art Python runtime latency and outperforms commercial serverless infrastructure by 1-3 orders of magnitude.

Evaluation

- **Mechanism Evaluation - Data Locality**

  large input data but light computation: sum of all elements across 10 input arrays.

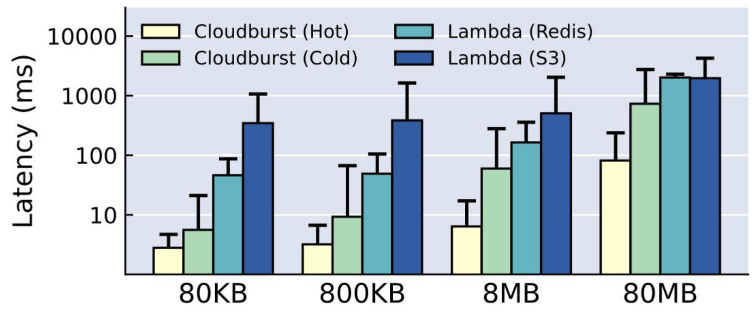


Figure 5: Median and 99th percentile latency to calculate the sum 10 arrays, comparing Cloudburst with caching and without and AWS Lambda over Redis and AWS S3. We vary array lengths from 1,000 to 1,000,000 by multiples of 10 to demonstrate the effects of increasing data retrieval costs.

While performance gains vary across configurations and data sizes, avoiding network round trips to storage services enables Cloudburst to improve performance by 1-2 orders of magnitude.

Evaluation

- **Mechanism Evaluation - Autoscaling**

  ability to detect and respond to workload changes.

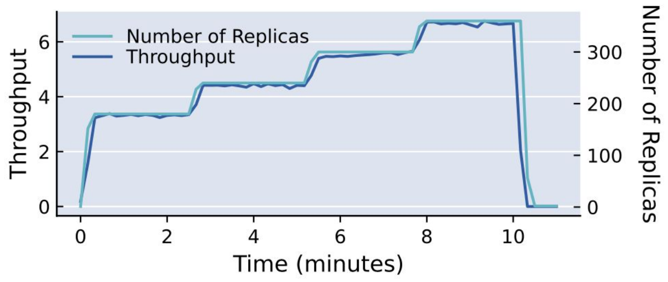


Figure 7: Cloudburst's responsiveness to load changes. We start with 180 executor threads, issue requests from 60 clients, and measure throughput. Cloudburst quickly detects load spikes and allocate more resources. Plateaus in the figure are the wait times for EC2 instance startup.

Cloudburst mechanisms for autoscaling enable policies that can quickly detect and react to workload changes. We are mostly limited by the high cost of spinning up new EC2 instances. The policies and cost of spinning up instances can be improved in future without changing Cloudburstś architecture.

Questions

Thank You!