Lightweight and Fair Serverless Computing for Edge Clouds

K. K. Ramakrishnan
University of California, Riverside

Thanks to:
My students Shixiong Qi, Viyom Mittal, Elizabeth Liri, Ziteng Zeng, Leslie Monis, Ian-Chin Wang, and my collaborator Tim Wood and his student
Serverless Computing

What is Serverless computing?

• Paradigm for development and deployment of cloud applications to ease burden on users
  • Function as a service (FaaS): Users only provide application function code
  • Enabled by the shift of enterprise application architectures to containers and microservices.
  • Characteristics: Short running, Stateless, Event-driven

• Benefits of Serverless Computing
  • Remove need for traditional always-on server components
  • Reduce user cost and complexity, and greatly improve service scalability and availability
  • Provisioning and managing the infrastructure becomes the cloud providers’ job

• Challenges with serverless computing
  • Less optimized to have high-performance, resource-efficient, and responsive
  • Support both low latency processing and low resource consumption (‘Cold start’ vs. ‘Warm’ functions)
Serverless Computing

a typical architecture for a serverless cloud framework (based on Kubernetes and Knative)
Serverless Computing

An abstract functional view of a serverless cloud:
Serverless Edge Clouds

- Works well in highly scalable clouds
- Seamless scaling
- Quick provisioning
- Resource wastage
- Resource constrained
- Heavyweight serverless components
- Poor scalability of function chains

Current designs of serverless platforms are not yet a viable option for Edge environments.
Outline

• Supporting Serverless Computing in Edge Clouds: Mu
• Understanding the Needs of IoT in Edge Clouds
• High-Performance eBPF-based Event-driven, Shared-Memory Processing
Challenges to Using Existing Approaches in Edge Clouds

Challenge 1: Approximate Resource Provisioning

- **Existing autoscaling design parameters depends a lot on user input**
  - 😞 But, users typically have little knowledge or control over runtime features of functions
  - 😞 Potential for incorrect parameter configuration

- **Single metric-based autoscaling (RPS, Concurrency)**
  - 😞 Not comprehensive enough to achieve optimal autoscaling

- **Slow resource provisioning for traffic bursts**
  - 😞 Long response time, possibly leading to SLO violations
Challenges to Using Existing Approaches in Edge Clouds

Challenge 2: Unfair Function Placement

- **Existing placement engine design focuses on resource efficiency**
  - 😞 Does not explicitly consider fairness between functions
  - 😞 Unfair resource provisioning between functions

- 😞 If two functions that have the same SLO may be placed such that…
  - Function-1: **More** resources provisioned - **Better** SLO
  - Function-2: **Less** resources provisioned - **Worse** SLO
Challenges to Using Existing Approaches in Edge Clouds

Challenge 3: Lack of Awareness of Resource Heterogeneity and System Dynamics

- Resource Heterogeneity and System Dynamics can lead to poor load balancing decision
  - Least connection LB: Track the queue length at backend pods and distribute the request to the pod with minimum queue length
    - fails to deal with heterogeneity

![Diagram]

- Pod A needs 10s to respond
  - Node 1 has poor HW: Pods on Node 1 needs 10 seconds to serve one request (More response delay)

- Pod C needs 3s to respond
  - Node 2 has good HW: Pods on Node 2 needs 1 second to serve one request (Less response delay)
Challenges to Using Existing Approaches in Edge Clouds

Challenge 3: Lack of Awareness of Resource Heterogeneity and System Dynamics

- Resource Heterogeneity and System Dynamics can lead to poor load balancing decision
  - Least connection LB: Track the queue length at backend pods and distribute the request to the pod with minimum queue length
  - Random selection (power of 2 random choices) scheme:
    - Select pod A (from pod A and pod B), but pod D is the best choice
    - Existing designs fail to deal with dynamics

![Diagram showing the load balancing process with queue lengths for pods A, B, C, and D.]
Challenges to Using Existing Approaches in Edge Clouds

Challenge 4: Approximate metrics collection

• Serverless platform relies on pod metrics to guide resource management

• Existing design relies on approximate metrics collection to address scaling
  • 😞 an inaccurate view of system status
  • 😞 negative impact on resource provision, load balancing...

• Need a *precise, lightweight* and *scalable* metric collection mechanism
Mu: Efficient, Fair and Responsive Serverless Edge Cloud

Building blocks of Mu

Accounts for Heterogeneity and Dynamics

- Ingress gateway
  - Smart Load Balancer
  - Internal Metric Server

Incoming Rate Predictor

SLO-aware Autoscaler

Precise Resource Provision

Fair Function Placement

Function pods

- Queue proxy container
- User container

Accurate, Scalable, Metrics Collection

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predictor

Precise Resource Provision

Fair Function Placement

Incoming Rate Predator
SLO-aware Autoscaler

• Idea:
  • 😊 From user's perspective, providing SLO is more meaningful rather than internal configurations (Concurrency, RPS)

• How:
  • 1 Users only provide the target SLO of their function
  • 2 Provision resources by factoring in both the incoming request rate and the queue length
  • 3 Ensure SLOs by factoring in the average request execution time
    • ✔ Avoid over-allocation of resources to ensure performance with just the right amount of resources
Incoming Rate Predictor

• Our Goal:
  • 😞 Provision adequate resource in case of traffic bursts

• Idea:
  • 😄 Mu uses a simple online linear regression model to predict the incoming rate based on previous observations
  • 😄 Mu uses multi-armed bandit to improve accuracy

• Features:
  • 1. lightweight and fast
  • 2. Accurate prediction - dynamically select the model with minimum error
Placement Engine

2-stage heuristic algorithm:

• **Resource fairness** between serverless functions
  • Function selection based on Dominant Resource Fairness (DRF)

• **Resource efficiency** between nodes
  • Node selection based on scoring
    • Alignment [1], WorstFit [2], and BestFit [2].
    • reduce the resource fragmentation, minimize unfairness

• *Call two stages iteratively until there are no resources left or all functions are placed


Load Balancer

Our Goal: 😞 to be aware of resource heterogeneity and system dynamics

- 😞 use extra metrics to estimate response time of each pod
  - 😞 differentiate “fast” and “slow” pods in the system

\[ R_i = \frac{Q_i + 1}{Cap_i} \]

- 😞 use “piggybacked” metrics of each pods instead of “two random choices”
  - 😞 track system dynamics

queue length of pod \( i \)
estimated response time of pod \( i \)
account for the cost of processing the new request
Service capacity: number of requests can be processed by pod \( i \) per second
Metric collection

Our goal:

• Precise metric collection that can reflect latest system state and achieve good scalability

• Solution 1 - providing sufficient metrics to support better load balancing and autoscaling
  • departure rate, confidence ratio, execution time, queue length

• Solution 2 - piggybacking the metrics in each response header

• Precise, Dynamic, Heterogeneity, Scalable
Summary of Mu

Mu achieves better

01 latency performance
02 resource fairness
03 resource efficiency
04 SLO performance

compared to Knative
Internet-of-Things (IoT)

Why IoT?

• Build smarter world with connected sensors
  • Traffic management, smart home, security

Features of IoT

• IoT needs to access backends with low latency
• IoT needs an efficient low-cost backend to host the service on-demand
• IoT offloads computation to the cloud for power-saving

Serverless edge cloud is an ideal fit for IoT applications
How do we build a suitable serverless edge cloud for IoT?

Service mesh for serverless functions

- Managing the service can be challenging when the number of elements increases
  - Use sidecar proxies to build up a service mesh: decoupled from service instances
- Main functions of sidecar proxy
  - Control plane: monitoring
  - Data plane: routing, load balancing

A service mesh is a configurable infrastructure layer that handles interactions between functions.
Existing approach and its limitations

Building blocks of Serverless Edge Cloud for IoT

🤔 We have to really be aware of resource usage…

- Ingress gateway
- Protocol adaptor
- Gateway
- Function pod
- User container
- Queue proxy
- User container
- Queue proxy
- Function pod
- User container
- Queue proxy
- Function pod
- User container
- Queue proxy
- Service Mesh
- Metrics server
- Scrape metrics
- Autoscaler
- Scaling decision

Expensive

Constantly running

IoT devices

Networked Systems Group
Enhancement: Event-driven Interaction via eBPF

Event-driven proxy (EPROXY)

Event-based interaction works naturally with serverless
Enhancement: Event-driven Interaction via eBPF

Features of eBPF

- **In-kernel** execution
- naturally event-driven
- Various **hook** points in kernel
  - Used for packet processing, packet filtering, traffic monitoring
- Userspace-kernel interaction
  - eBPF Maps
- Limitations
  - run to completion
  - limited instructions in a single program

The eBPF code needs to be carefully written by the developer
Enhancement: Event-driven Interaction via eBPF

The monitoring service provided by EPROXY

- **User space**
  - User Container
  - TX
  - RX

- **Kernel space**
  - veth (pod)

- **Host network stack**
  - Egress traffic
  - Ingress traffic

- **Function Pod**

- **Metrics Agent**
  - Metrics storage
  - Report metrics to control plane (for auto-scaler)

- **eBPF maps**

- **Request per second, execution time...**
Outline

• Supporting Serverless Computing in Edge Clouds
• Understanding the Needs of IoT in Edge Clouds
• High-Performance eBPF-based Event-driven, Shared-Memory Processing - SPRIGHT

SPRIGHT: Extracting the Server from Serverless Computing! High-Performance eBPF-based Event-driven, Shared-Memory Processing, Sigcomm 2022 (to appear)
Auditing the Overheads of Serverless Computing: KNative

Processing involved in a typical serverless function chain setup: network protocol, copies, interrupts, context switches etc. abound

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② total</td>
<td>③ ④ ⑤ total</td>
<td></td>
</tr>
<tr>
<td># of copies</td>
<td>1 2 3</td>
<td>4 4 4</td>
<td>12 15</td>
</tr>
<tr>
<td># of ctx switches</td>
<td>1 2 3</td>
<td>4 4 4</td>
<td>12 15</td>
</tr>
<tr>
<td># of irqs</td>
<td>3 4 7</td>
<td>6 6 6</td>
<td>18 25</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1 2 3</td>
<td>3 3 3</td>
<td>9 12</td>
</tr>
<tr>
<td># of serialization</td>
<td>1 1 2</td>
<td>2 2 2</td>
<td>6 8</td>
</tr>
<tr>
<td># of deserialization</td>
<td>0 1 1</td>
<td>2 2 2</td>
<td>6 7</td>
</tr>
</tbody>
</table>
**Overhead auditing**

Key takeaways

Takeaway#1: Excessive data copies, context switches, and interrupts.

Takeaway#2: Excessive, duplicate protocol processing.

Takeaway#3: Unnecessary serialization/deserialization.

Takeaway#4: Individual, constantly-running heavyweight components.

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>① ② total</td>
<td>③ ④ ⑤ total</td>
<td>Total</td>
</tr>
<tr>
<td># of copies</td>
<td>① 1 ② 2 ③ 3</td>
<td>④ 4 ⑤ 4 total</td>
<td>⑥ 12 ⑦ 15</td>
</tr>
<tr>
<td># of ctxt switches</td>
<td>① 1 ② 2 ③ 3</td>
<td>④ 4 ⑤ 4 total</td>
<td>⑥ 12 ⑦ 15</td>
</tr>
<tr>
<td># of irqs</td>
<td>① 3 ② 4 ③ 7</td>
<td>④ 6 ⑤ 6 total</td>
<td>⑥ 18 ⑦ 25</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>① 1 ② 2 ③ 3</td>
<td>④ 3 ⑤ 3 total</td>
<td>⑥ 9 ⑦ 12</td>
</tr>
<tr>
<td># of serialization</td>
<td>① 0 ② 1 ③ 2</td>
<td>④ 2 ⑤ 2 total</td>
<td>⑥ 6 ⑦ 7</td>
</tr>
</tbody>
</table>
**Overheads: Understanding impact of Sidecar Proxies**

**Key takeaways**

**Takeaway#4: Individual, constantly-running heavyweight components.**

Sad face: Having a sidecar proxy results in a 3×–7× reduction in throughput, 3×–7× higher latency, and a significant increase in CPU cycles per request.

Sad face: CPU overhead breakdown: 50% of CPU cycles are consumed by the kernel stack for the sidecar proxy.

---

**Figure 2: Performance and overhead breakdown of different sidecar proxy implementations.**
Overhead auditing: Summary

• The **loosely coupled** construction of a serverless cloud by pulling together existing cloud environment components: Kubernetes; containers; CNI plugins; Linux OS

• Each one was built to serve its own designed purpose in a large-scale cloud assuming a resource-rich environment – can be inefficient, resource hungry.
  • 😞 Poor dataplane design: poor throughput, large latency
  • 😞 Individual, constantly-running components in the function chain use up resources

• A cloud with lots of resources can accommodate such ‘free-wheeling’ use of resources; serverless computing performance requirements are still not stringent, and the user-view of cost 'savings' has dominated so far. We expect that this will change.

  **Substantial unnecessary processing overhead**

  The ‘server’ is still entrenched in serverless computing

**More streamlined, responsive serverless framework needed**
Extracting the Server out of Serverless Computing!

An overview of our design

- eBPF-based event-driven capability
- Shared memory processing

Optimization#1: Event-driven, shared memory function chain processing

Optimization#2: Direct Function Routing (DFR)

Optimization#3: Event-driven proxy

Optimization#4: eBPF-based dataplane acceleration for external communication

Optimization#5: Event-driven protocol adaptation (e.g., IoT)
Design details

Event-driven, Shared memory processing

Shared memory processing

• Zero-copy data sharing between functions
  • Relies on packet descriptors to pass the location of data in a shared memory pool
• Use shared HugePages to reduce memory access overhead
• Supports queueing to help sustain traffic bursts

Deliver packet descriptor using eBPF’s socket message (SKMSG)

• An eBPF program attached at the socket interface
• The function sends/receives SKMSG through regular socket
• Using eBPF’s socket map to map the function instance and its socket interface, which helps route SKMSG between functions in the kernel
  • 16-byte socket message to minimize overhead (instance ID of next function and a pointer to data in shared memory)
• Purely even-driven
  • No overhead when there are no events (packets)
  • Resource-efficient compared to polling-based DPDK RTE Ring
Design details

Event-driven Shared Memory processing vs. Polling-based Shared memory

Why did we choose to use SKMSG?
• Socket-to-socket data transfer between functions
  • Bypass the protocol stack processing in kernel
  • Avoid unnecessary processing overhead compared to traditional socket communication
  • Suitable for delivering small event messages
• Strictly load proportional compared to polling-based DPDK RTE ring shared memory processing
  • Polling-based descriptor delivery constantly consumes CPU resources - wasteful
    • Polling-based approach has better overload/livelock behavior
• eBPF’s SKMSG is triggered by kernel interrupts
  • Receive livelock [1] at gateway when interrupts are frequently generated within the function chain
  • The consolidated protocol processing offered by our gateway supports flow control to alleviate overload/livelock behavior of SKMSG

**Design details**

**Direct Function Routing (DFR)**

Having our gateway perform invocations between functions is unnecessary

- SKMSG overhead, routing overhead

DFR optimizes invocations within a function chain

- The upstream function in the chain directly invokes the downstream function: bypass the gateway
  1. Function/Gateway extracts the ‘topic’ from the payload in shared memory
  2. Use ‘topic’ and current function ID to lookup the routing table in shared memory and get the next function ID
  3. Use next function ID for load balancing, find the instance with maximum residual service capacity
  4. Write the instance ID into packet descriptor. SKMSG transmits packet descriptor to target function pod with gateway bypassed

- DFR can reduce end-to-end latency of the function chain and improve the scalability
Design details

E/S-PROXY

In Knative, a queue proxy runs as an additional container in a function pod distinct from the user container
• Buffering, metrics collection, health check
• Existing sidecar proxy designs are too heavyweight

We built a lightweight, event-driven eBPF based E/S-PROXY instead
• Buffering/queueing is offloaded to shared memory
• eBPF programs used for metrics collection
  • We create a metrics map with eBPF maps to store collected metrics
  • The metrics map can be accessed by a user space metrics agent to report metrics to the control plane
• Internal event-driven metrics collection hooks inside Gateway to provide fine-grained L7 metrics as an enhancement of EPROXY
• Health check is offloaded to kubelet

No overhead if no events: strictly 'load proportional'
Much less overhead when handling events
Resource-savings compared to current 'sidecar' design
Design details

eBPF-based Gateway for Dataplane

- eBPF forwarding programs attached to XDP/TC hooks at network interfaces (host-veth, NIC)
  - Packet redirect features are offered by eBPF (‘XDP_REDIRECT’ and ‘TC_ACT_REDIRECT’)
  - Pass raw packets between network interfaces and bypass iptables
    - Save CPU cycles, benefit dataplane performance (reduce latency, improve throughput)
  - Price: Loss of full-featured iptables network policy support
    - Suitable for users only requiring higher dataplane performance
Design details

Event-driven protocol adaptation

- Running a protocol adapter as a separate component introduces extra overhead (just like a sidecar proxy)
- Application can use the payload in shared memory directly,

It would be ideal to run a protocol adapter as an internal, lightweight event-driven component

- Predefined ‘protocol adaptation hook points’
  - For protocol adaptation support. Even to support different protocols
    - Implemented on the packet datapath inside the Gateway
    - Before the gateway sends messages to the function pod

- Dynamic code injection
  - Adapter programs can be attached to the hook during runtime (no need to recompile environment)

- Stateless/Stateful protocol conversion
  - Our adapter works seamlessly with stateless protocols, e.g., HTTP
  - Gateway handles the application layer protocol control plane (e.g., connection establishment)
  - Gateway delivers only the application layer data packets (i.e., MQTT PUBLISH message) to the protocol adapter by following the CloudEvent specification.
Overhead auditing of our design

Compared to the same function chain setup in Knative, our design achieves

- 0 data copies, 0 additional protocol processing, and no serialization-deserialization overheads within the chain.
- Although the use of SKMSG generates context switches and interrupts, and these do add latency for processing, the total number of context switches and interrupts for our design is still much less (< ½) than that of the base Knative design.

<table>
<thead>
<tr>
<th>Data Pipeline No.</th>
<th>External</th>
<th>Within chain</th>
<th>Total</th>
<th>Total of Knative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>①</td>
<td>② total</td>
<td>③</td>
<td>④ total</td>
</tr>
<tr>
<td># of copies</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td># of ctx switches</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td># of irqs</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td># of proto. processing</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td># of serialization</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td># of deserialization</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The event-based shared memory processing brings substantial reduction of overheads for communication within the serverless function chain.
EVALUATION: Across multiple serverless workloads

1. Online Boutique from Google [1]
   • Intensive web traffic
   • 10 functions, heterogeneous CPU service time for each (from 0.6 ms to 260 ms)
   • 6 different sequences of function chains

2. Motion detection
   • MERL motion detector dataset [2]
   • Intermittent IoT traffic (a burst every few seconds)
   • 2 simple functions, 1ms CPU service time each

3. Parking: image detection & charging
   • CNRPark+EXT image dataset [3]
   • Intermittent & Periodic IoT traffic (once every 240 seconds)
   • 5 functions, heterogeneous CPU service time for each (from 1 ms to 435 ms)

Compared with: (i) baseline Knative setup, using NGINX as front-end proxy/message broker to coordinate the communication within function chains (ii) DPDK-based zero-copy design

Performance with Realistic Workloads

1. Online boutique

For different alternatives: configure different concurrency levels (i.e., # of concurrent users) at the load generator

- Knative: 4K concurrency (we stop at 4K since it’s the maximum load that Knative can handle)
- Our designs using event-driven shared memory processing (SKMSG) and our design using polling-based shared memory processing (DPDK): 12K concurrency

Throughput & latency:

- Knative’s RPS is highly variable over time (~890 req/sec)
- Both DPDK and SKMSG maintain a stable RPS of ~2600 req/sec (3× higher than Knative)
- Knative shows clear overload behavior, e.g., from 50s to 72s, response time increases significantly, due to large queueing at the Knative’s gateway ⇒ large tail latency
- Shared memory processing reduces communication overhead within function chains, achieving better RPS and latency than Knative, even at much higher traffic load
Performance with Realistic Workloads

1. Online boutique

Resource efficiency:

- entire Knative setup (including the gateway and queue proxies, which are constantly running) consumes \(~30\) CPU cores (\(46\%\) of the total CPUs (a 64-core CPU) available on the physical node)
- DPDK consumes \(10\) cores (one core for Gateway, 9 cores for functions)
- SKMSG consumes in total only \(~0.94\) CPU cores

The event-driven SKMSG mechanism makes our design’s shared memory processing more resource-efficient
Performance with Realistic Workloads

1. Online boutique

Concerns with SKMSG:

- SKMSG generates context switches and interrupts for descriptor delivery
  - additional latency in SKMSG’s shared memory processing
- SKMSG is slightly worse than DPDK in terms of tail latency
  - The 4th function chain -- the Checkout service function with a higher CPU service time (260ms)
  - The additional delay for SKMSG’s descriptor delivery, adds to the transient queueing and hence longer tail latency

- the higher CPU service time of the functions still dwarfs the extra latency introduced by SKMSG in relative terms
- Limited impact on the throughput: the SKMSG’s RPS is very close to DPDK as we can see
Performance with Realistic Workloads

2. IoT - Motion Detection

For intermittent traffic, Knative can use zero-scaling with “cold start” to save resources

- Without incoming requests, Knative will scale functions down to zero to save resources and reduce costs. BUT:
  - Knative has 30 seconds ‘grace period’ before scaling to zero

Functions in our design are kept ‘warm’

- event-driven processing doesn’t consume CPU resources when idle

Observations:

- Knative has large response times that possibly render the motion detection application ineffective, and violate SLOs
- Cold start latency has cascading effect during start-up of chain
- Our design shows much lower response consistently as it keeps functions ‘warm’
- The higher resource usage of Knative’s queue proxy under load more than offsets any benefit of zero-scaling

Our event-driven design sidesteps the need for cold start by keeping functions warm at minimum cost
Performance with Realistic Workloads

3. IoT - Parking: image detection and charging

For periodic traffic, it’s ideal to “pre-warm” functions in Knative

- avoid penalty of cold start, while trading-off a 'small amount' of the resource savings for shutting down serverless functions with zero-scaling (let's see if it is true)
- pre-warm functions 20 seconds before the next burst (to allow for function to be ready)

In our design: functions are kept ‘warm’

Observations:
- Expensive startup: Knative’s function startup consumes more CPU than function execution (see ‘pre-warm’ spikes)
  - Inefficient for scaling functions down to zero
- Our design consumes a small amount of CPU, but is load-proportional
  - Our design has lower latency

Our design’s event-driven features make it more efficient even if we keep functions warm compared to ‘pre-warming’ Knative functions
Summary

• Resource management in current open-source serverless framework needs to be improved for edge clouds

• Our MU design is one example:
  • Using SLO-aware autoscaler and incoming rate predictor to achieve precise resource provision
  • Using DRF-based placement engine to achieve resource fairness
  • Factoring resource heterogeneity and system dynamics in load balancing to achieve optimal request distributing
  • Using piggyback to achieve precise and scalable metrics collection

• We achieve higher dataplane performance, while reducing the inefficiencies in current open-source serverless environments
  • Using shared memory processing to optimize the data pipeline of current serverless function chains
  • Using event-driven processing to improve resource efficiency of current serverless function design