SIGCOMM 2022
Amsterdam

SPRIGHT:
Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing

Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan
University of California, Riverside

Visit us at: https://kknetsyslab.cs.ucr.edu/

Networked Systems Group
UC RIVERSIDE

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
  - Remove need for traditional always-on server components
  - Provisioning and managing the infrastructure becomes the cloud providers' job

➢ Reduce user cost and complexity, and greatly improve service scalability and availability

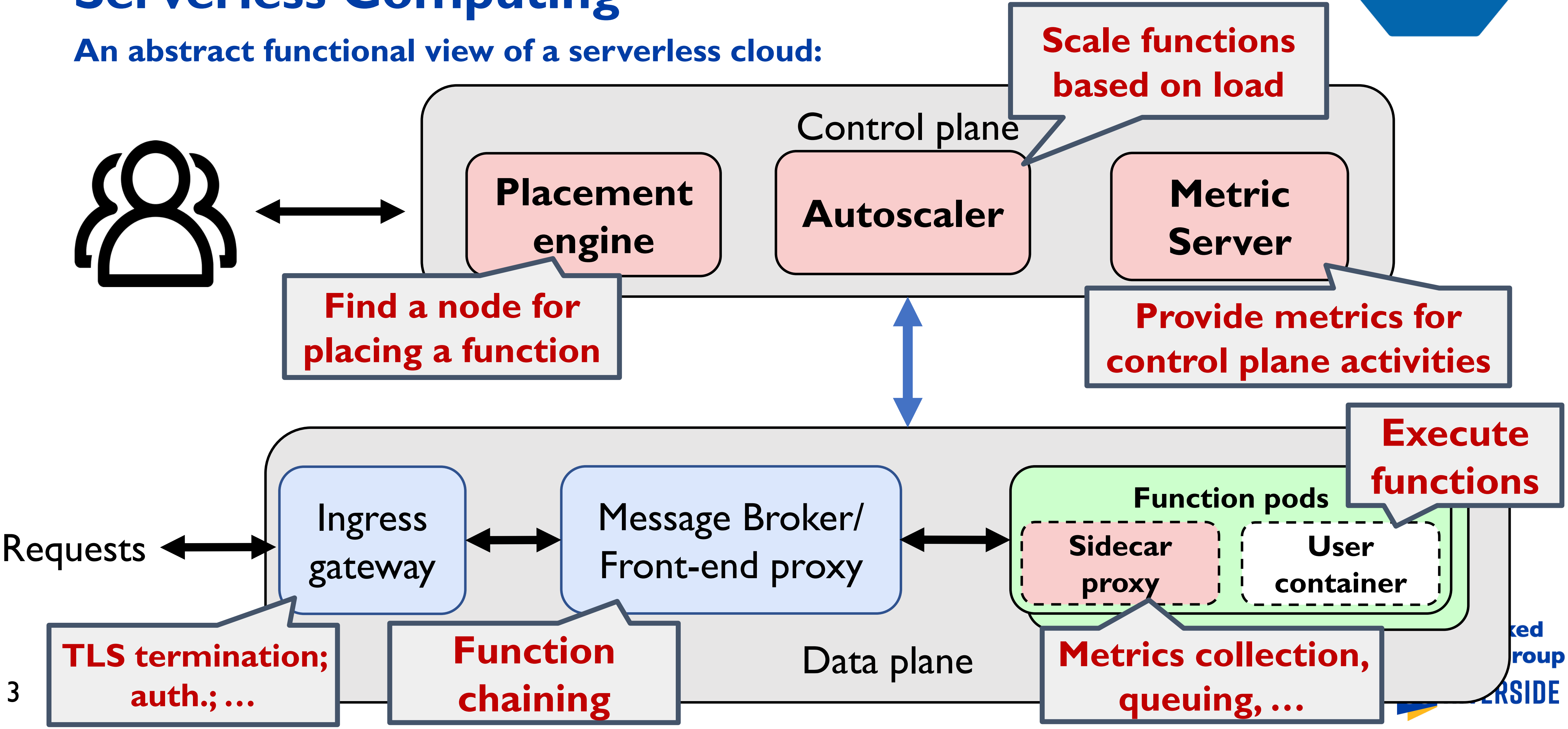
- Challenges with serverless computing
  - Less focus on optimizing for high-performance, resource-efficiency, or being responsive
  - Need better support for both **low latency processing** and **low resource consumption**

SPRIGHT: achieve high-performance, resource-efficient serverless function chains through ***shared memory and event-driven processing***

Networked Systems Group
UC RIVERSIDE

Serverless Computing

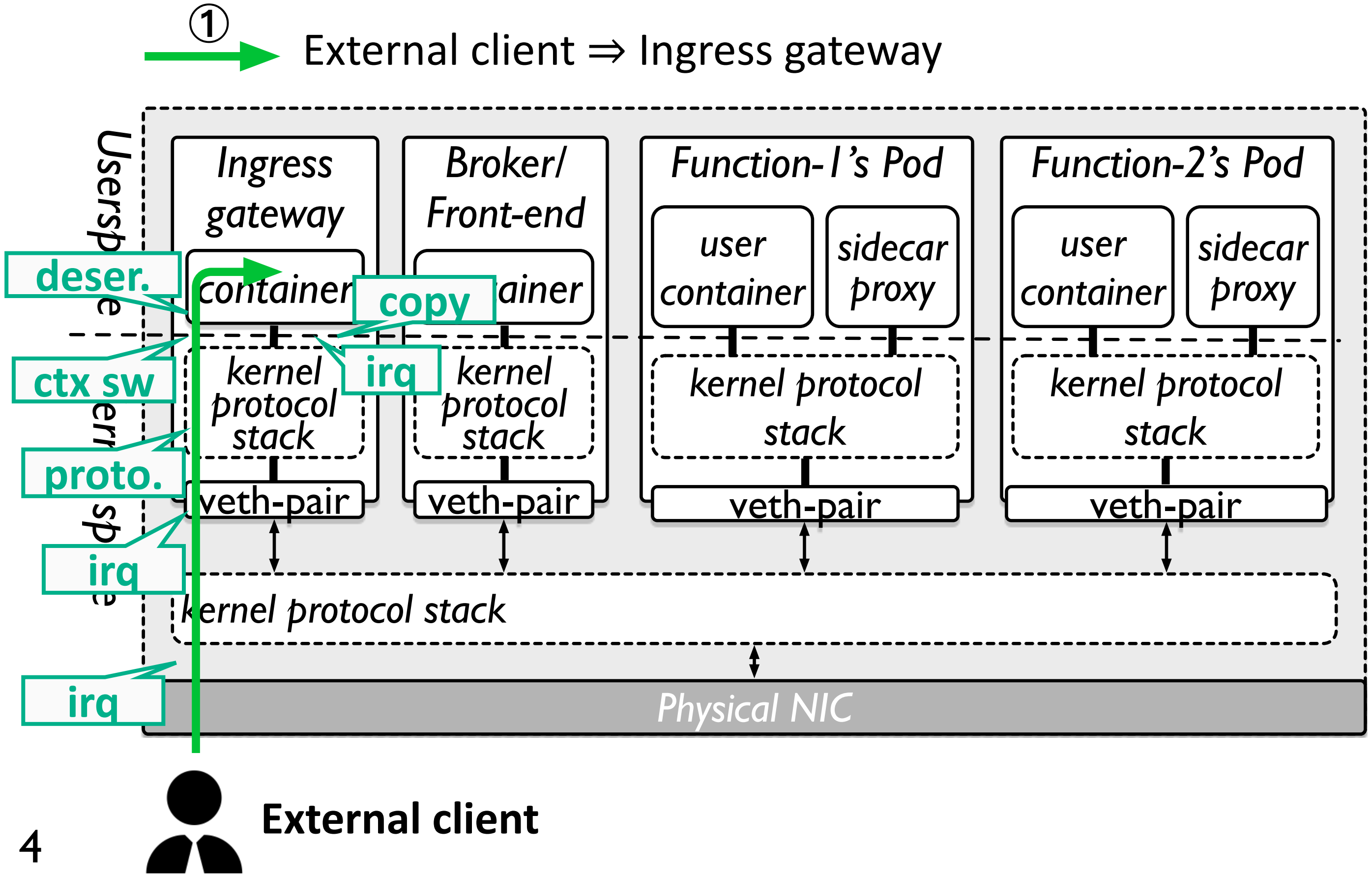
An abstract functional view of a serverless cloud:



Auditing the Overheads of Serverless Computing (I)

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

Ingress gateway: Intercept external requests; TLS Termination, authentication, etc

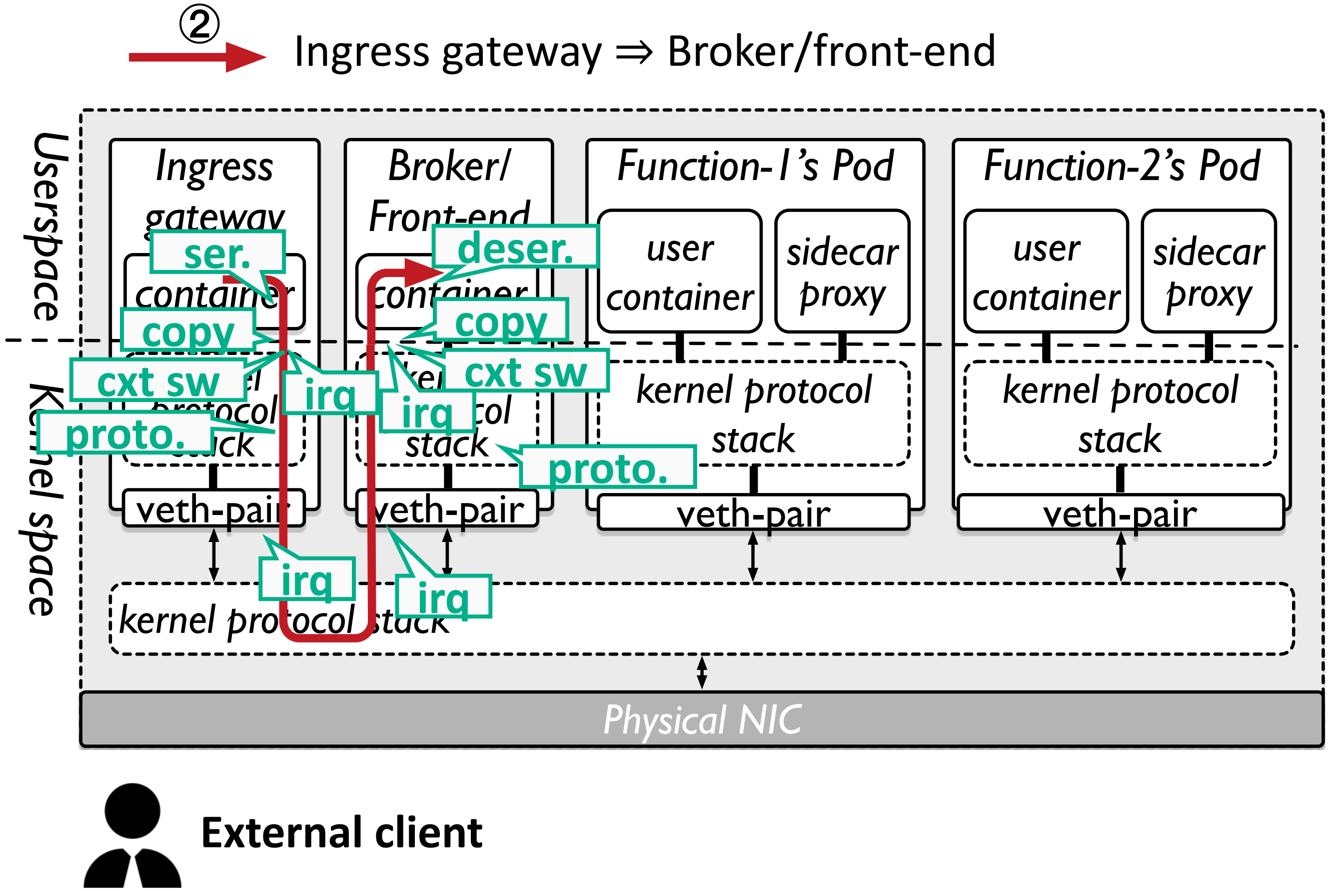


Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1							
# of ctxt switches	1							
# of irqs	3							
# of proto. processing	1							
# of serialization	0							
# of deserialization	1							

Auditing the Overheads of Serverless Computing (2)

Broker/front-end: an intermediate component for coordinating invocations within the function chain

Broker: 1 copy, 1 context switches, 2 interrupts, 1 protocol processing, and 1 deserialization



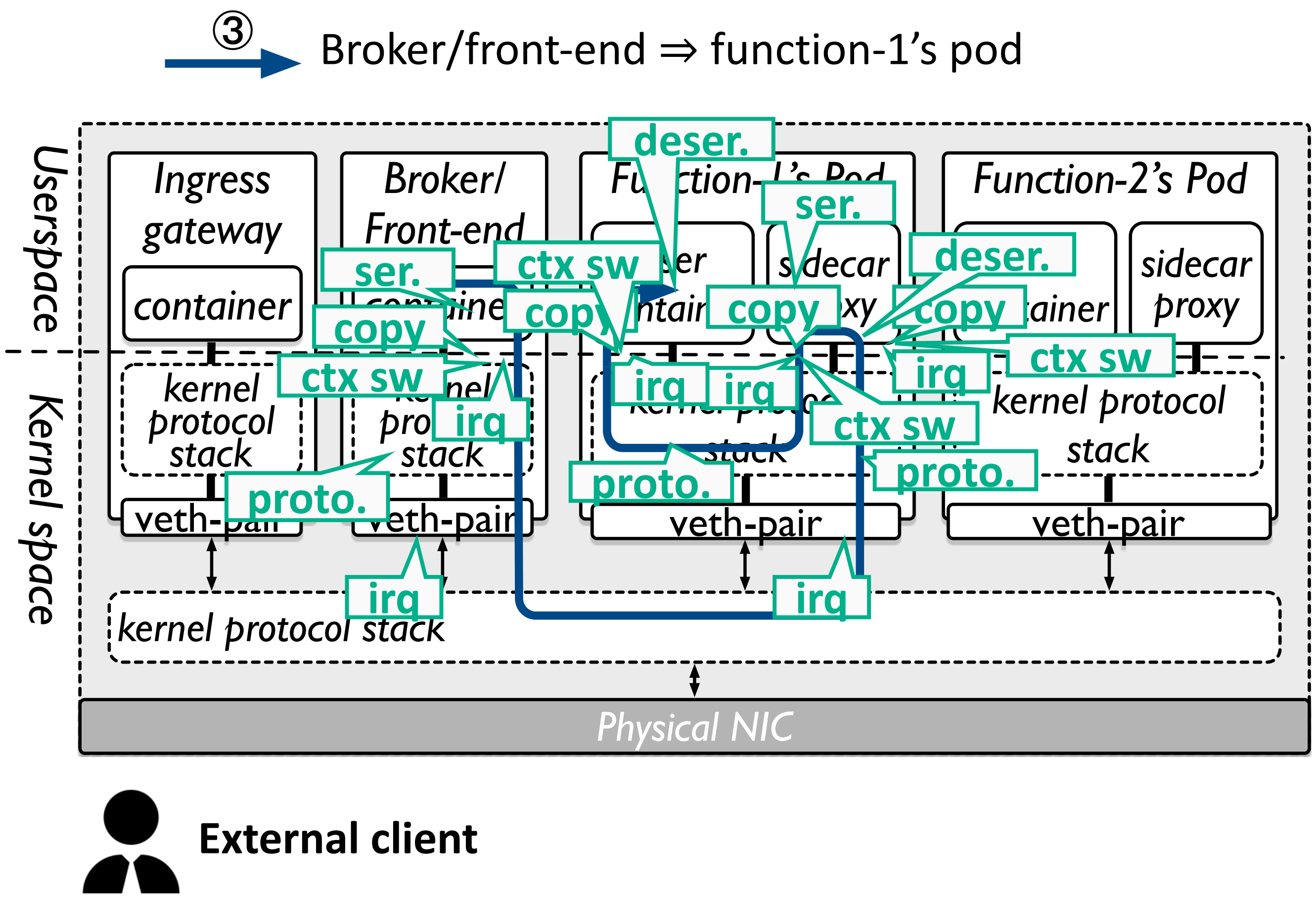
Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3					
# of ctxt switches	1	2	3					
# of irqs	3	4	7					
# of proto. processing	1	2	3					
# of serialization	0	1	2					
# of deserialization	1	1	1					

Networked Systems Group
UC RIVERSIDE

Auditing the Overheads of Serverless Computing (3)

Total: 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

Sidecar: 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization



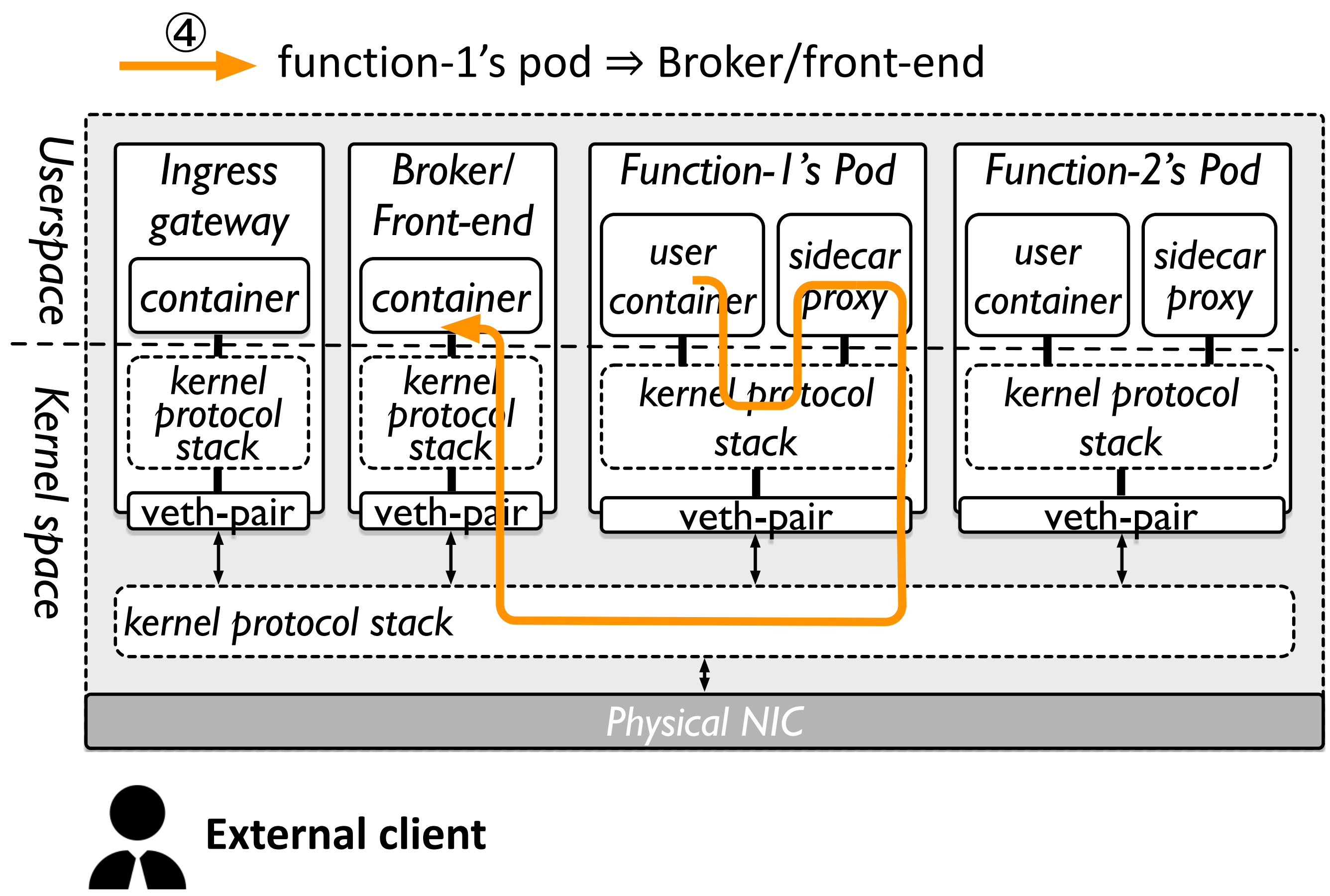
Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4				
# of ctxt switches	1	2	3	4				
# of irqs	3	4	7	6				
# of proto. processing	1	2	3	3				
# of serialization	0	1	2	2				
# of deserialization	1	1	1	2				

Networked Systems Group
UC RIVERSIDE

Auditing the Overheads of Serverless Computing (4)

Total: 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

Sidecar: 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization



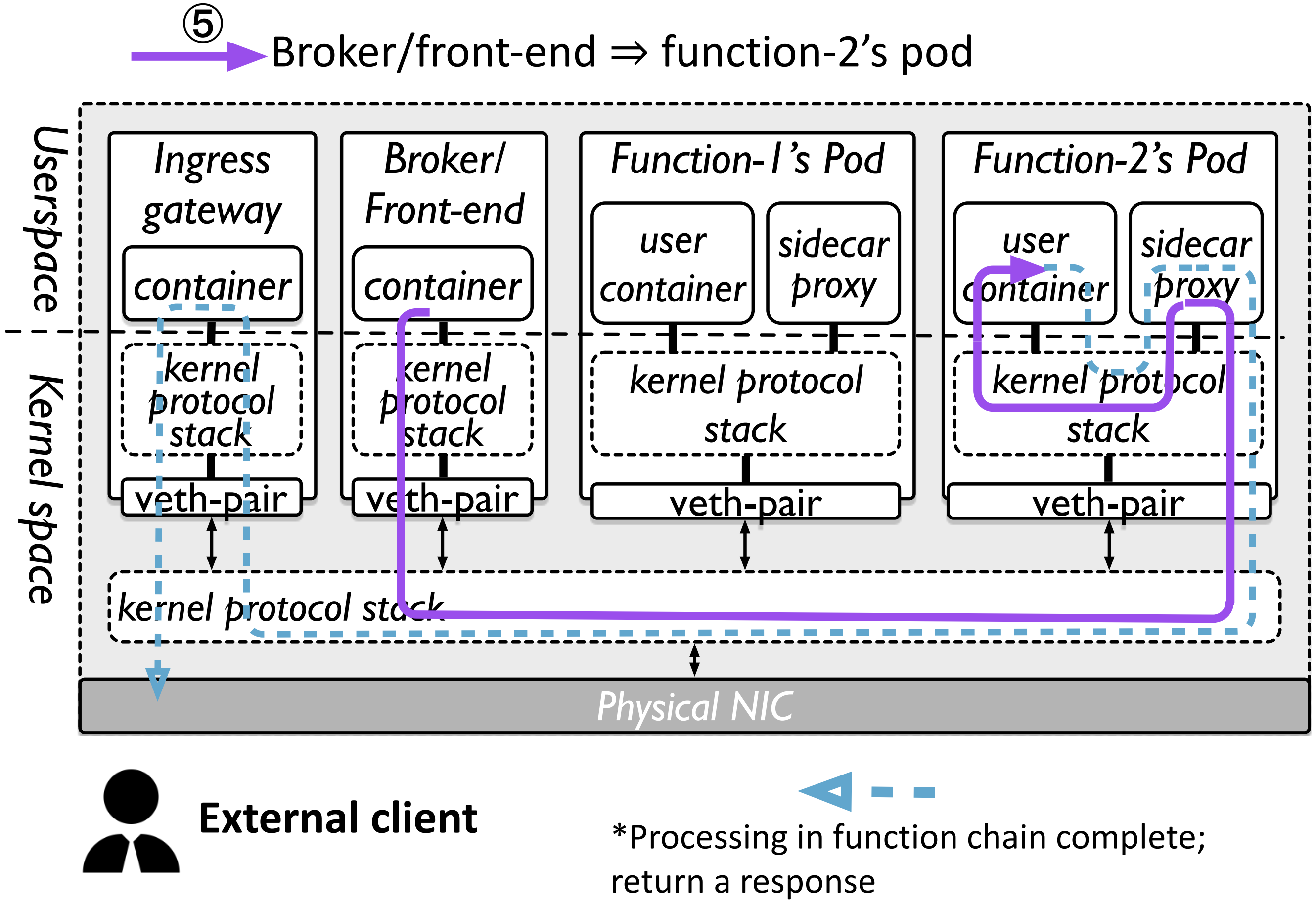
Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4	4			
# of ctxt switches	1	2	3	4	4			
# of irqs	3	4	7	6	6			
# of proto. processing	1	2	3	3	3			
# of serialization	0	1	2	2	2			
# of deserialization	1	1	1	2	2			

Networked Systems Group
UC RIVERSIDE

Auditing the Overheads of Serverless Computing (5)

Total: 4 copies, 4 context switches, 6 interrupts, 3 protocol processing, 2 serializations, and 2 deserializations

Sidecar: 2 copies, 2 context switches, 2 interrupts, 1 protocol processing, 1 serialization, and 1 deserialization



Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4	4	4	12	15
# of ctxt switches	1	2	3	4	4	4	12	15
# of irqs	3	4	7	6	6	6	18	25
# of proto. processing	1	2	3	3	3	3	9	12
# of serialization	0	1	2	2	2	2	6	8
# of deserialization	1	1	1	2	2	2	6	7

Networked Systems Group
UC RIVERSIDE

Auditing the Overheads of Serverless Computing

Key takeaways: Excessive overhead within the function chain

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

Takeaway#2: Excessive, duplicate protocol processing for communication within the function chain

Takeaway#3: Unnecessary serialization/deserialization.

Takeaway#4: Individual, constantly-running components.

Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4	4	4	12	15
# of ctxt switches	1	2	3	4	4	4	12	15
# of irqs	3	4	7	6	6	6	18	25
# of proto. processing	1	2	3	3	3	3	9	12
# of serialization	0	1	2	2	2	2	6	8
# of deserialization	1	1	1	2	2	2	6	7

Networked Systems Group
UC RIVERSIDE

What is SPRIGHT?

eBPF-based event-driven capability + Shared memory processing

Optimization#1: Event-driven proxy

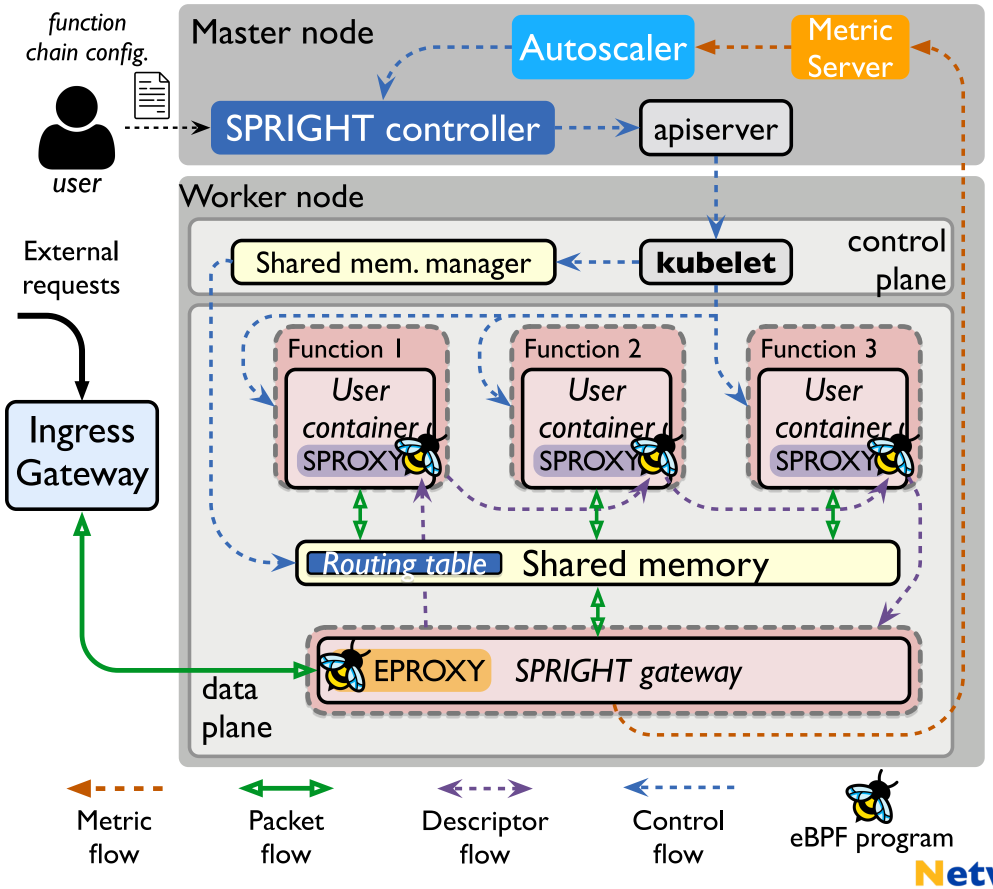
Replacing individual, constantly-running sidecar

Optimization#2: Shared memory processing

Reduce data movement overhead

Optimization#3: Direct Function Routing (DFR)

Simplify inter-function invocations

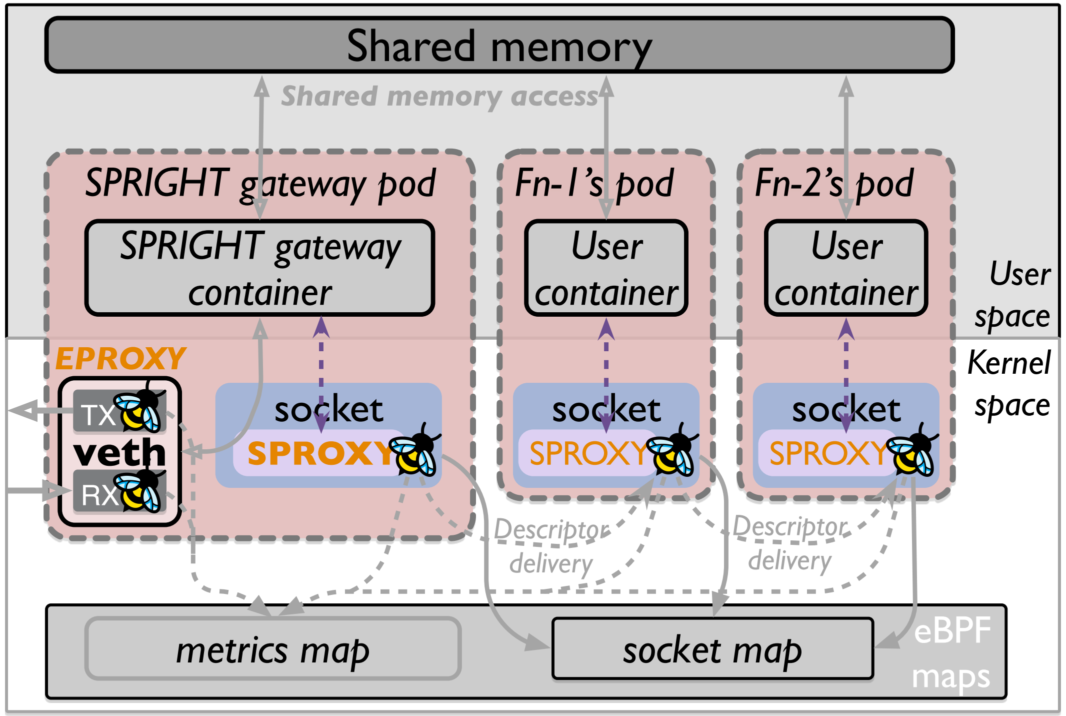


Networked Systems Group
UC RIVERSIDE

SPRIGHT: Lightweight Serverless Function Chains

Optimization#1: eBPF-based Event-driven proxy (EPROXY and SPROXY)

- In-kernel eBPF-based "sidecar"
  - **EPROXY**: @Veth of SPRIGHT GW pod
    - Monitoring; iptables acceleration
  - **SPROXY**: Sidecar being injected at the socket level
    - Monitoring; Security; Routing
- Purely event-driven
  - No CPU overhead when there are no requests
- All in the kernel
  - Avoid extra user-kernel boundary crossings

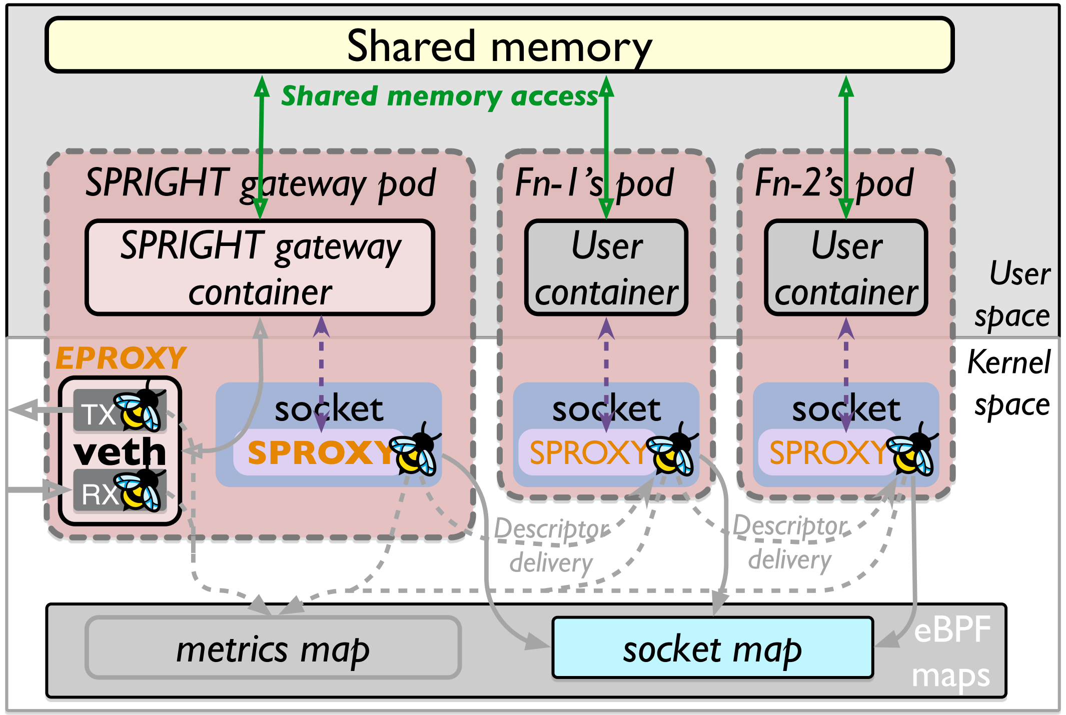


Networked Systems Group
UC RIVERSIDE

SPRIGHT: Lightweight Serverless Function Chains

Optimization#2: Shared memory processing

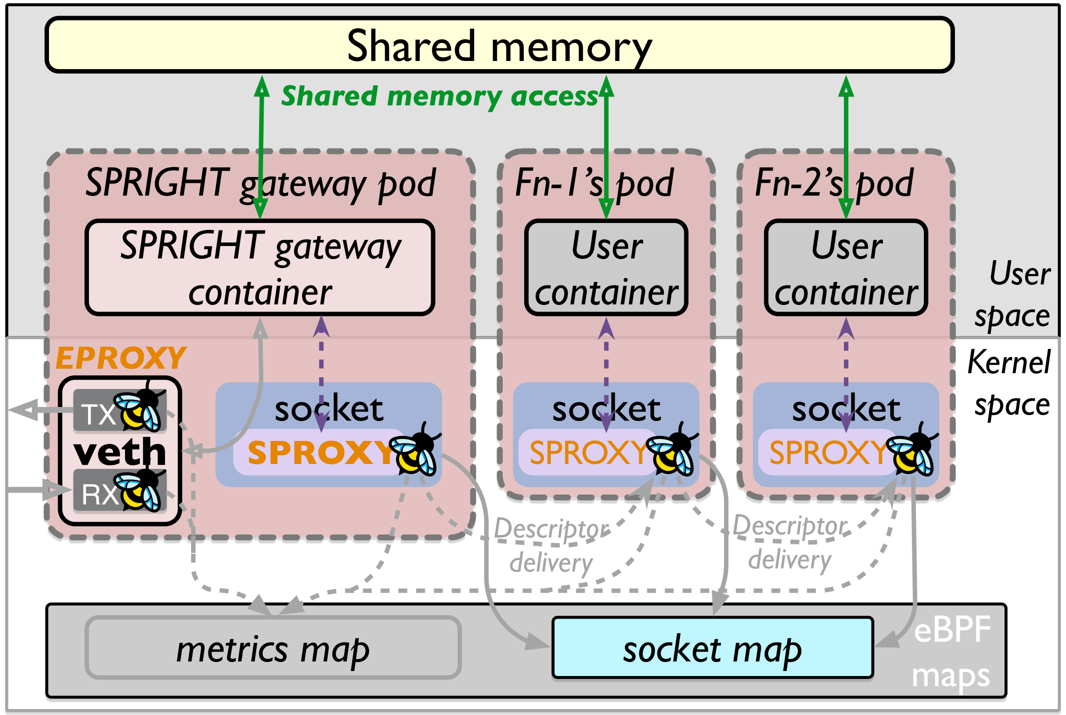
- How to handle protocol processing?
  - SPRIGHT Gateway: Entry-point of a function chain
    - **Consolidate** kernel protocol processing
    - Move payload into shared memory



SPRIGHT: Lightweight Serverless Function Chains

Optimization#2: Shared memory processing

- How to handle protocol processing?
  - SPRIGHT Gateway: Entry-point of a function chain
    - **Consolidate** kernel protocol processing
    - Move payload into shared memory
- Shared memory based data sharing **between** functions
  - **NO** copy, protocol processing, serialization, …
  - Packet descriptor delivery: eBPF's socket message
    - reside in Event-driven proxy (**SPROXY**)
    - Socket-to-socket data transfer; Routing using eBPF's socket map
    - Strictly load-proportional compared to polling-based packet descriptor delivery (**DPDK RTE Ring**)

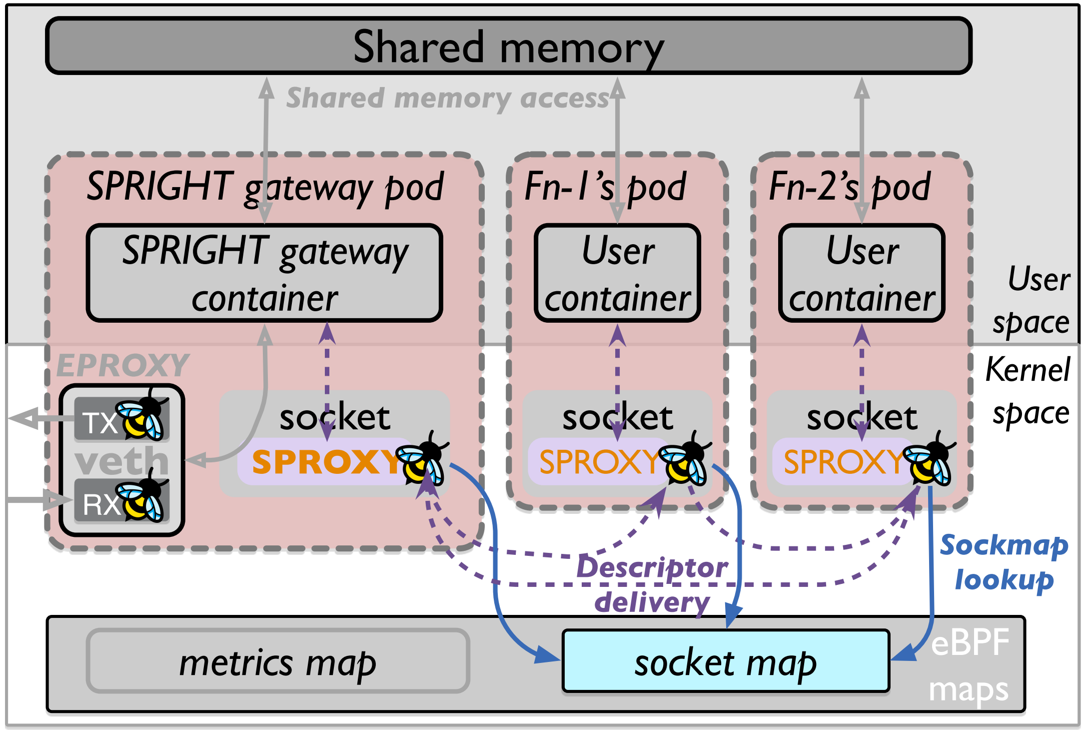


Networked Systems Group
UC RIVERSIDE

SPRIGHT: Lightweight Serverless Function Chains

Optimization#3: Direct Function Routing

- Having the broker/front-end perform invocations between functions is unnecessary
  - Routing overhead
- DFR optimizes invocations within a function chain
  - The upstream function in the chain directly invokes the downstream function: bypass the gateway
  - DFR rules in eBPF's socket map
- DFR can reduce end-to-end latency of the function chain and improve the scalability
  - Eliminate an extra hop on the datapath
  - Benefit increases as the chain scales



Networked Systems Group
UC RIVERSIDE

Overhead auditing: Knative vs. SPRIGHT

Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4	4	4	12	15
# of ctxt switches	1	2	3	4	4	4	12	15
# of irqs	3	4	7	6	6	6	18	25
# of proto. processing	1	2	3	3	3	3	9	12
# of serialization	0	1	2	2	2	2	6	8
# of deserialization	1	1	1	2	2	2	6	7

Data Pipeline No.	External			Within chain			Total
	①	②	total	③	④	total	
# of copies	1	2	3	0	0	0	3
# of ctxt switches	1	2	3	2	2	4	7
# of irqs	3	4	7	2	2	4	11
# of proto. processing	1	2	3	0	0	0	3
# of serialization	0	1	2	0	0	0	2
# of deserialization	1	1	1	0	0	0	1

- 0 data copies, 0 protocol processing, 0 serialization/deserialization overheads **within the chain**

*The **event-based shared memory processing** brings substantial reduction of overheads for communication **within** the serverless function chain*

[1] https://github.com/GoogleCloudPlatform/microservices-demo
[2] http://www.merl.com/wmd
[3] http://cnrpark.it/

Evaluation: across multiple serverless workloads

1. Online Boutique from Google [1]

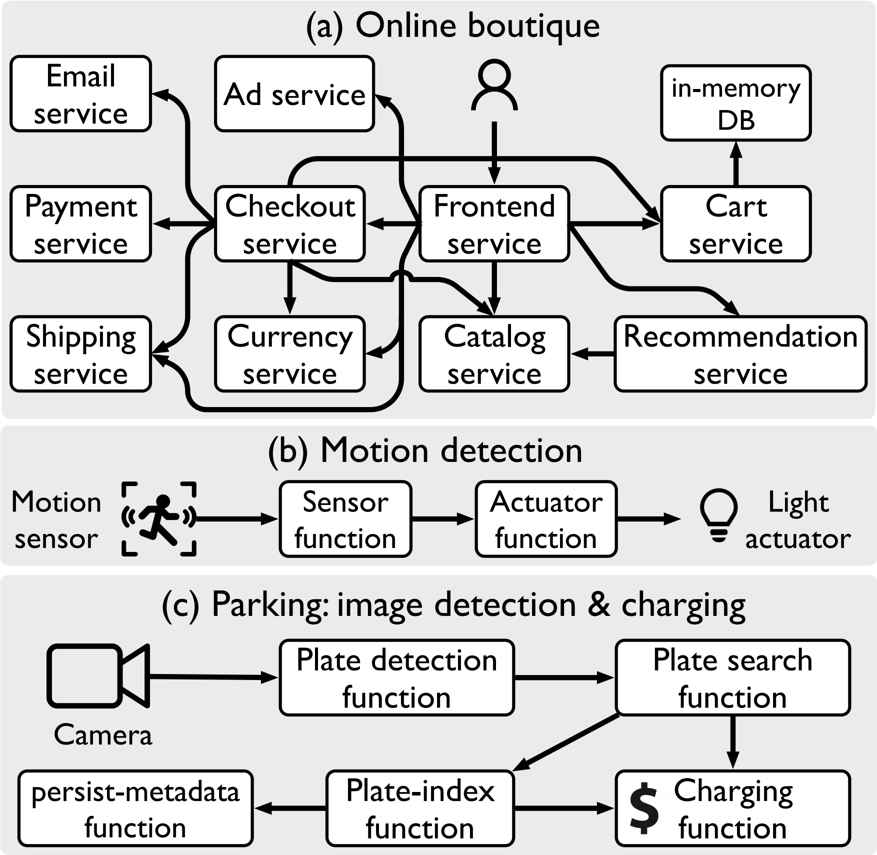
- *Intense* web traffic
- 10 functions
- *6 different sequences of function chains*

2. Motion detection [2]

- *Intermittent* IoT traffic (a burst every few seconds)
- 2 simple functions

3. Parking: image detection & charging [3]

- *Intermittent & Periodic* IoT traffic (once every 240 seconds)
- 5 functions with *heterogeneous* CPU service time for each (from 1ms to 435ms)



Networked Systems Group
UC RIVERSIDE

Performance with *Online Boutique*

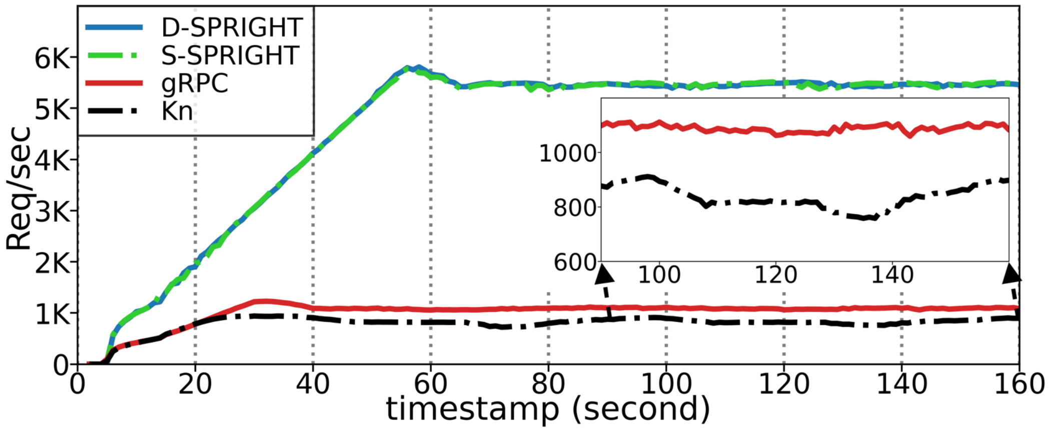
S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

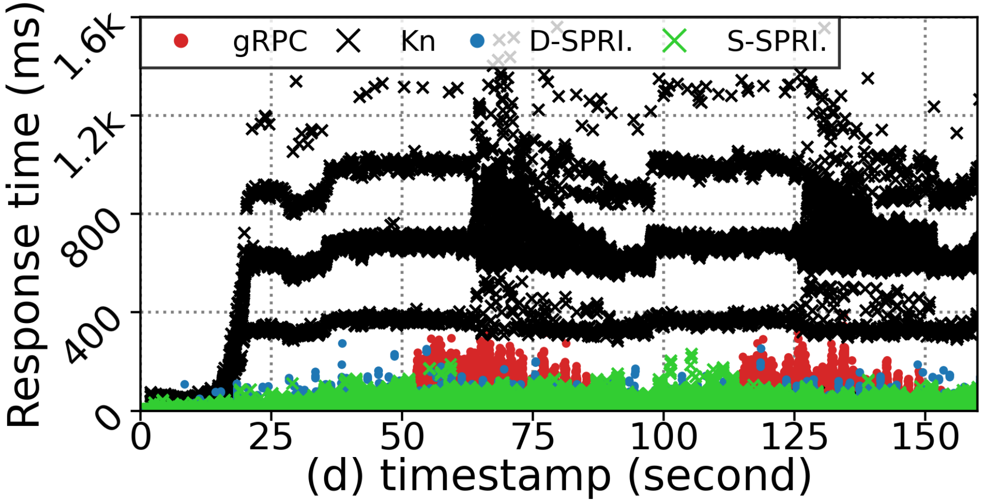
Throughput:

- Both D- and S-SPRIGHT maintain a stable RPS of ~5500 req/sec ➔ (**6×** more than Knative)
- Without sidecar and front-end proxy, gRPC is slightly better than Knative, but its throughput is much lower than SPRIGHT (5x lower)

Latency:

- Knative shows clear overload behavior: high tail latency
- SPRIGHT's shared memory processing reduces communication overhead within function chains, achieving lower latency than Knative and gRPC, even at much higher traffic load





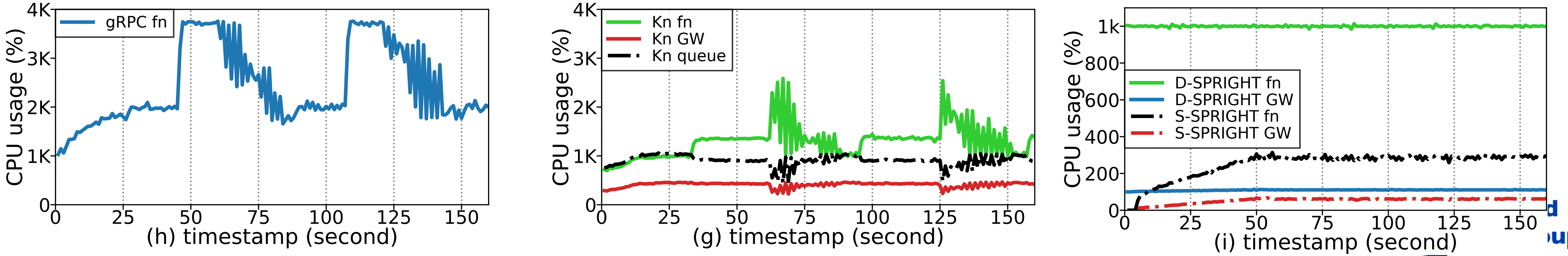
***Note**: for simplicity, we only show the latency results of a representative function chain in online boutique. More results can be found in the paper.*

Performance with *Online Boutique*

S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

Resource efficiency:

- entire Knative setup (including the gateway and queue proxies, which are constantly running): **~26 CPU** cores (out of 40)
- Entire gRPC setup (only functions, no gateway and sidecars): **~36 CPU** cores (out of 40)
- D-SPRIGHT consumes **11 cores** (one core for Gateway, 10 cores for functions)
- S-SPRIGHT consumes in total only **~3 CPU** cores

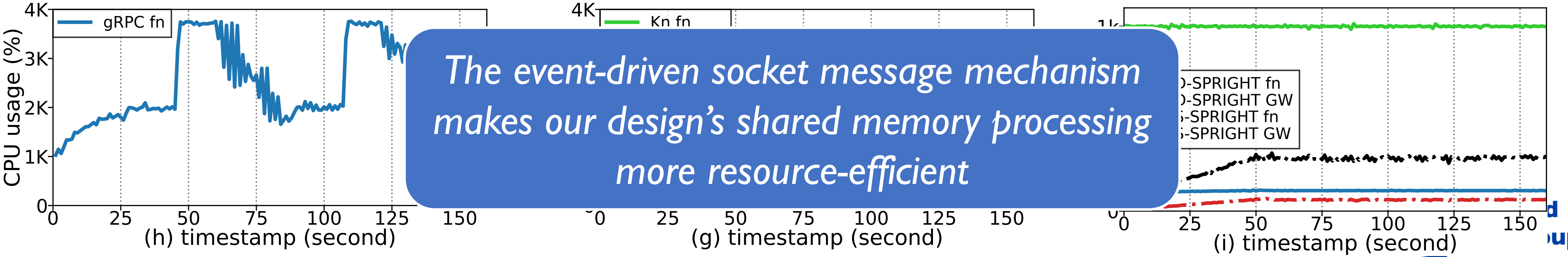


Performance with *Online Boutique*

S-SPRIGHT vs. D-SPRIGHT vs. Knative mode vs. gRPC mode (no sidecar)

Resource efficiency:

- entire Knative setup (including the gateway and queue proxies, which are constantly running): **~26 CPU** cores (out of 40)
- Entire gRPC setup (only functions, no gateway and sidecars): **~36 CPU** cores (out of 40)
- D-SPRIGHT consumes **11 cores** (one core for Gateway, 10 cores for functions)
- S-SPRIGHT consumes in total only **~3 CPU** cores



The event-driven socket message mechanism makes our design's shared memory processing more resource-efficient

UC RIVERSIDE

[1] https://github.com/GoogleCloudPlatform/microservices-demo
[2] http://www.merl.com/wmd
[3] http://cnrpark.it/

Evaluation: across multiple serverless workloads

1. Online Boutique from Google [1]

- *Intense* web traffic
- 10 functions
- *6 different sequences of function chains*

2. Motion detection [2]

- *Intermittent* IoT traffic (a burst every few seconds)
- 2 simple functions

3. Parking: image detection & charging [3]

- *Intermittent & Periodic* IoT traffic (once every 240 seconds)
- 5 functions with *heterogeneous* CPU service time for each (from 1ms to 435ms)

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

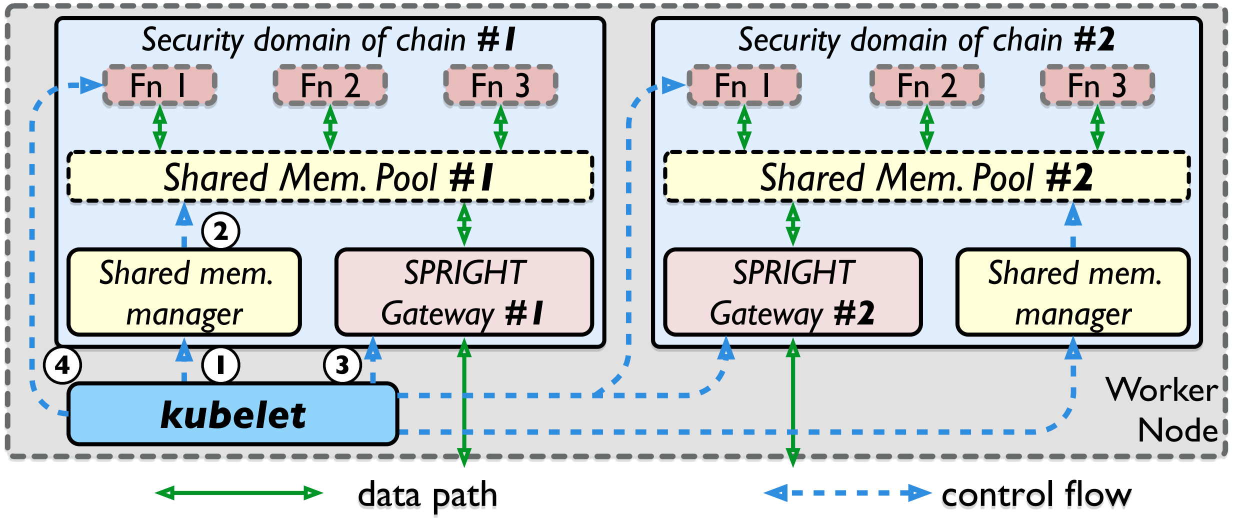
Our design's event-driven features make it more efficient even if we keep functions warm compared to 'pre-warming' Knative functions

Networked Systems Group
UC RIVERSIDE

Shared memory considered harmful?

Our Solution: Security domain

- **Trust model**: functions within a chain trust each other, functions in different chains may not
- SPRIGHT constructs a security domain for each function chain with:
  - a private shared memory pool for each chain
    - DPDK's multi-process support
    - Use different **shared data file prefix** to separate memory pool
  - inter-function packet descriptor filtering with the **SPROXY**
    - Use **SPROXY** to construct packet descriptor filtering between functions
    - Restrict unauthorized access to the shared memory



Networked Systems Group
UC RIVERSIDE

Conclusion

SPRIGHT: event-driven + shared memory processing = load-proportional, high-performance

*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*

- *When serving an intense web workload (online boutique):*
  - **6x** throughput improvement, **70x** tail latency reduction and **30x** CPU usage savings over Knative
- *When serving intermittent IoT workload (motion detection; parking):*
  - Better than Knative even with 'pre-warmed' functions; side-stepping the 'cold-start'

Artifacts Evaluated Functional V1.1 · Artifacts Available V1.1 · Results Reproduced V1.1

Find SPRIGHT at: https://github.com/ucr-serverless/spright

Networked Systems Group
UC RIVERSIDE