Making Serverless Computing Efficient

K. K. Ramakrishnan

University of California, Riverside

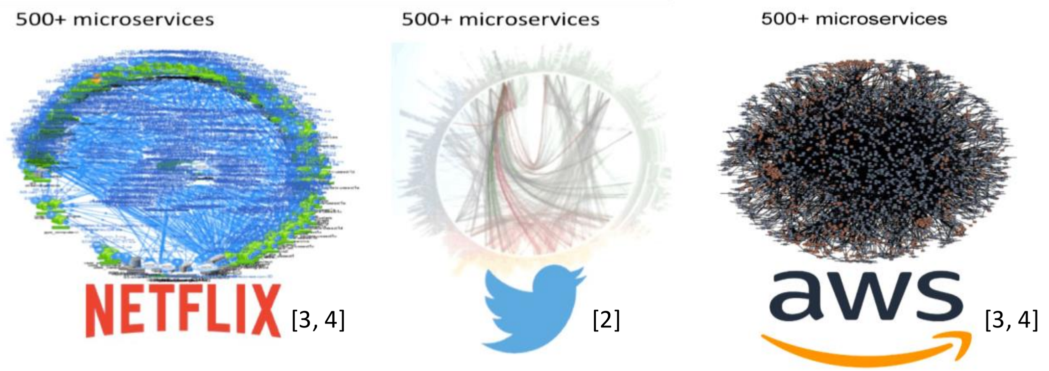
Thanks to:

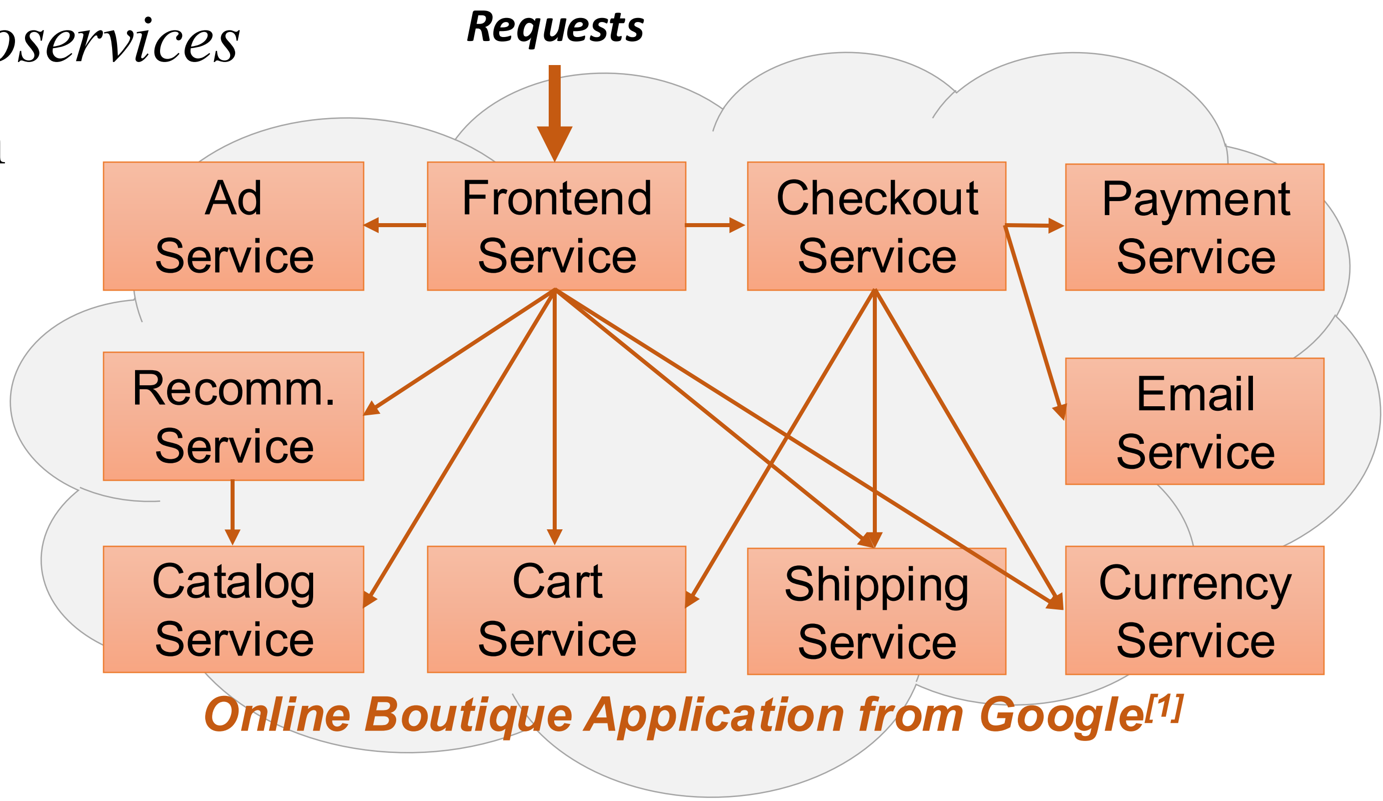
The many people with who I’ve had the good fortune to learn from & collaborate with including: Federico Parola, Shixiong Qi, A. B. Narappa, Fulvio Risso

Networked
Systems Group
UC RIVERSIDE

Building applications by composing "*Microservices*"

- Software development: **cloud-based** applications as a composition of **loosely-coupled** *microservices*
- Benefits: composable software design
  - ***Independently*** deployable
  - ***Easy*** to scale out





Online Boutique Application from Google[1]

This approach has also become popular for software-based network applications

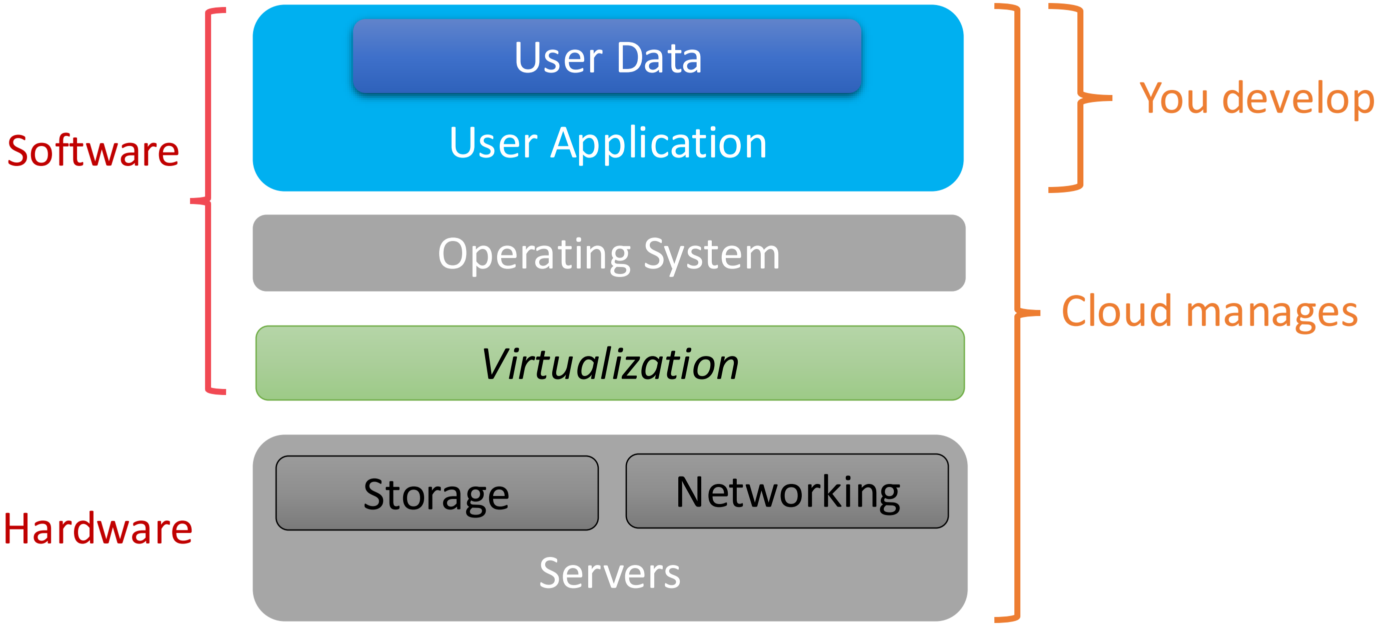
[1] https://github.com/GoogleCloudPlatform/microservices-demo
[2] Decomposing Twitter: Adventures in Service-Oriented Architecture
[3] The Evolution of Microservices. https://www.slideshare.net/ adriancockcroft/evolution-of-microservices-craft-conference.
[4] Adrian Cockroft. Microservices Workshop: Why, what, and how to get there. http://www.slideshare.net/adriancockcroft/ microservices-workshop-craft-conference.

Serverless Computing – Fast-growing Cloud Offering

- Serverless computing is one of the faster-growing offerings in the cloud
- **Serverless: 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**
  - Removes need for traditional always-on server components
  - Reduces user cost and complexity, and greatly improve service scalability and availability
  - Provisioning and managing the infrastructure becomes the cloud providers' job

Understanding Cloud Stacks

- “Serverless” Computing or Function-as-a-Service (FaaS)
  - **“Event-driven” execution**: Applications are triggered based on events, terminated upon event completion
  - **True “Pay-as-you-go” billing**: Pay only for the execution of an application function. No charge when the application is idle

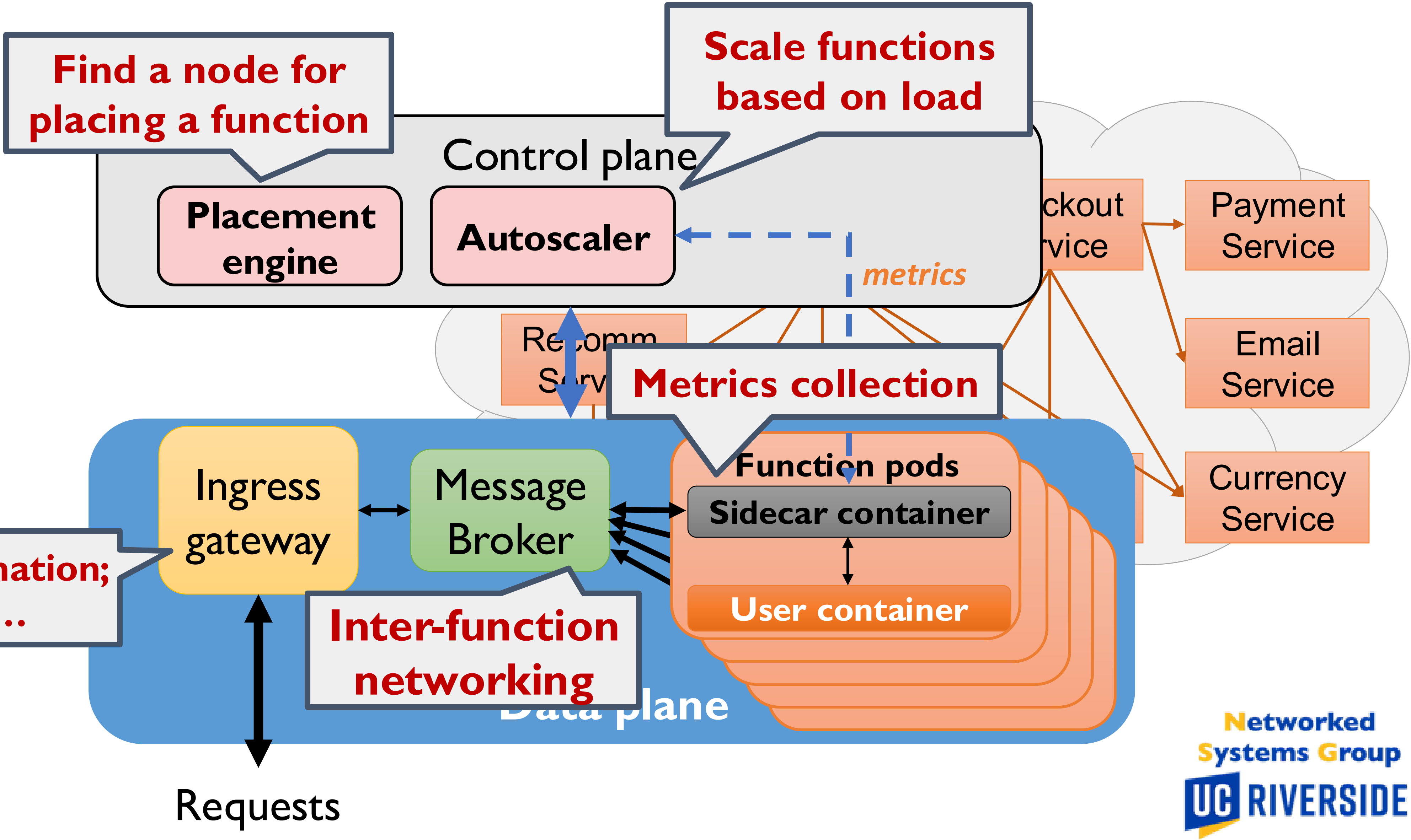


Users can solely focus on the application logic!

Infrastructure Support for a Serverless Cloud

An abstract functional view

Based on Knative[1]



[1] https://knative.dev/docs/

Excessive overhead within the serverless data plane

Overhead Contributor #1: kernel-based networking

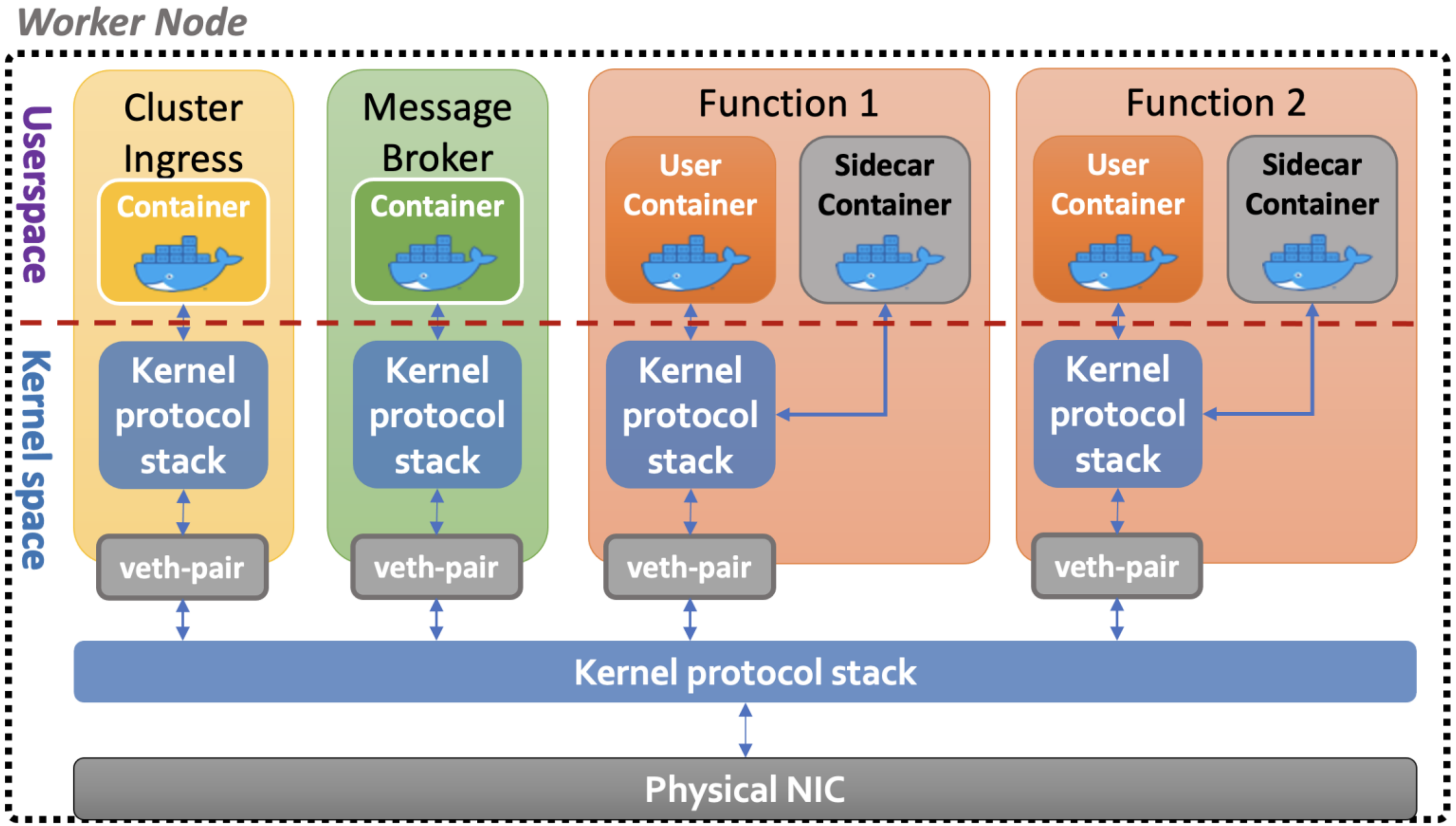
- *Copies, context switch, proto. processing, …*

Overhead Contributor #2: stateful, constantly-running components in the userspace

- *Container-based sidecar*
- *Message broker*
- *Cluster Ingress Gateway*

Performance Loss

Reduced Efficiency

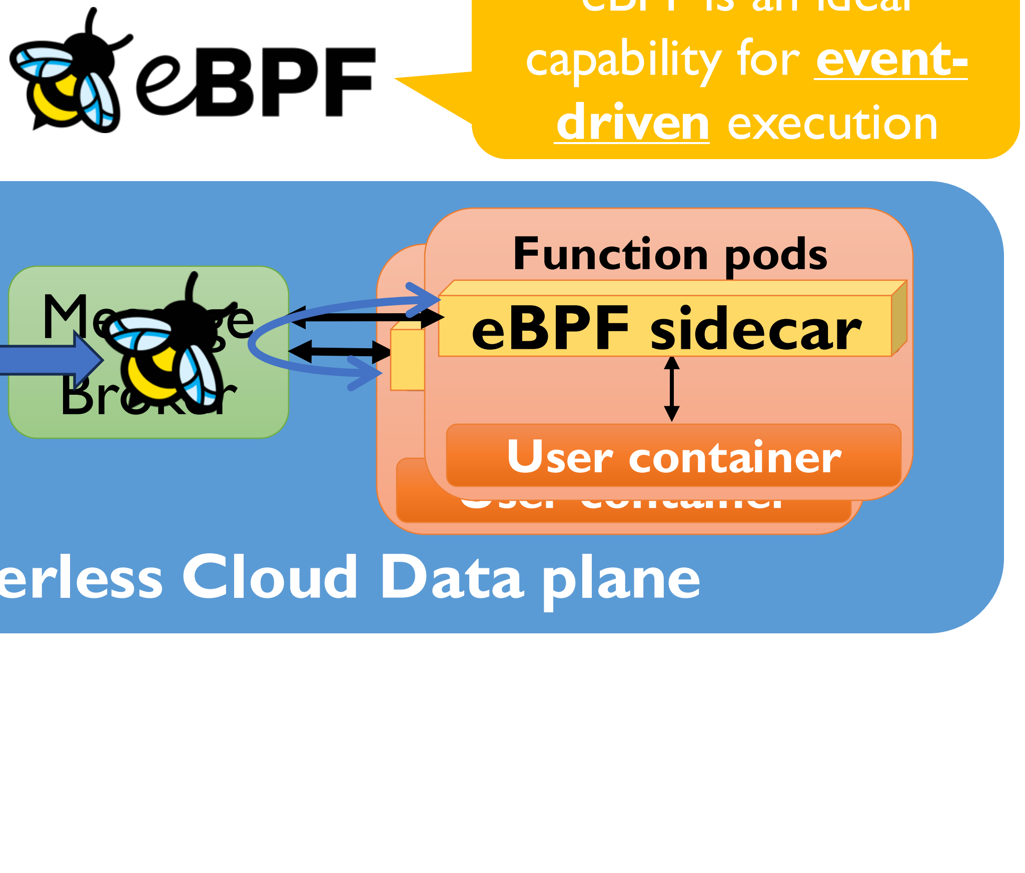


Networked Systems Group
UC RIVERSIDE

Towards Lightweight and High-performance Data Plane

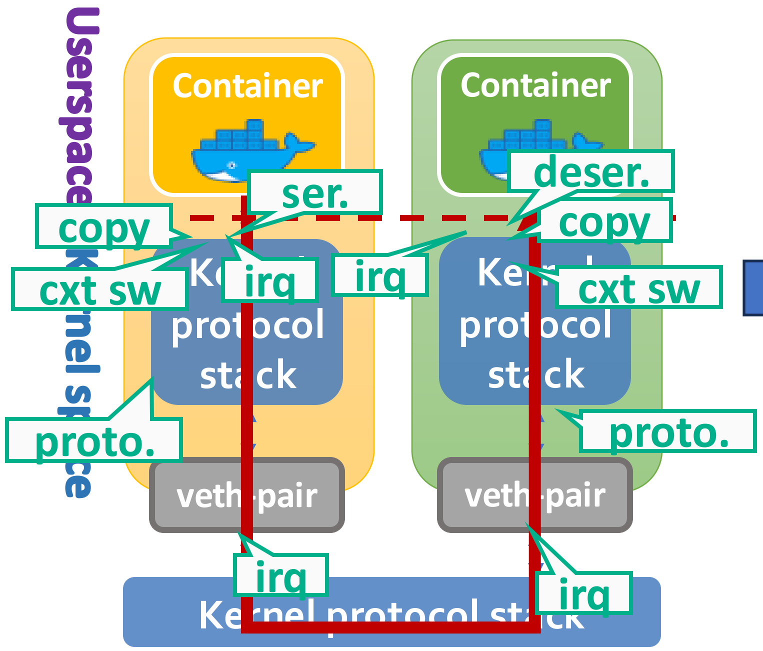
Enhancement: Event-driven Interaction via *extended Berkeley Packet Filter*

eBPF is an ideal capability for **event-driven** execution



Towards Lightweight and High-performance Data Plane

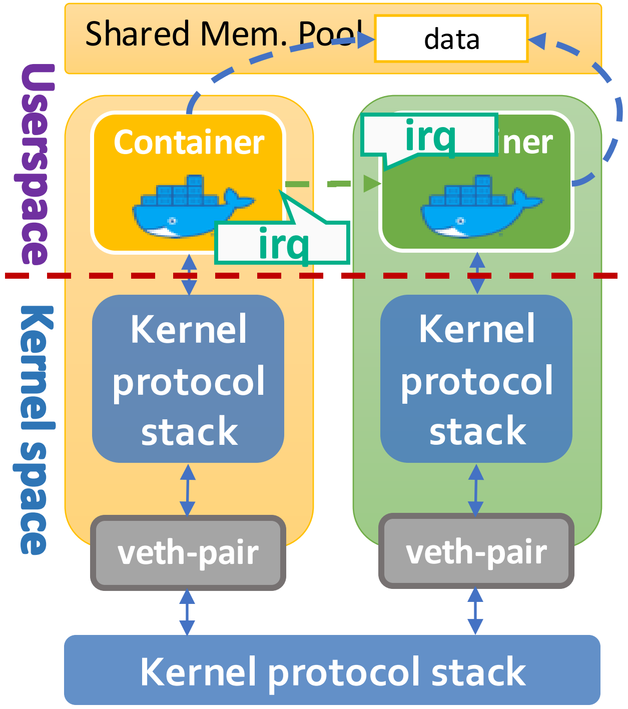
Enhancement: From Kernel-based Networking to Shared Memory Processing



Kernel-based Networking

Bypass the kernel!

Pass by reference!



“Pass-by-reference”
(Shared Mem. Processing)

SPRIGHT: Lightweight Serverless Function Chains

eBPF-based event-driven capability + Shared memory processing

Enhancement #1: eBPF-based sidecar

Replacing individual, constantly-running sidecars

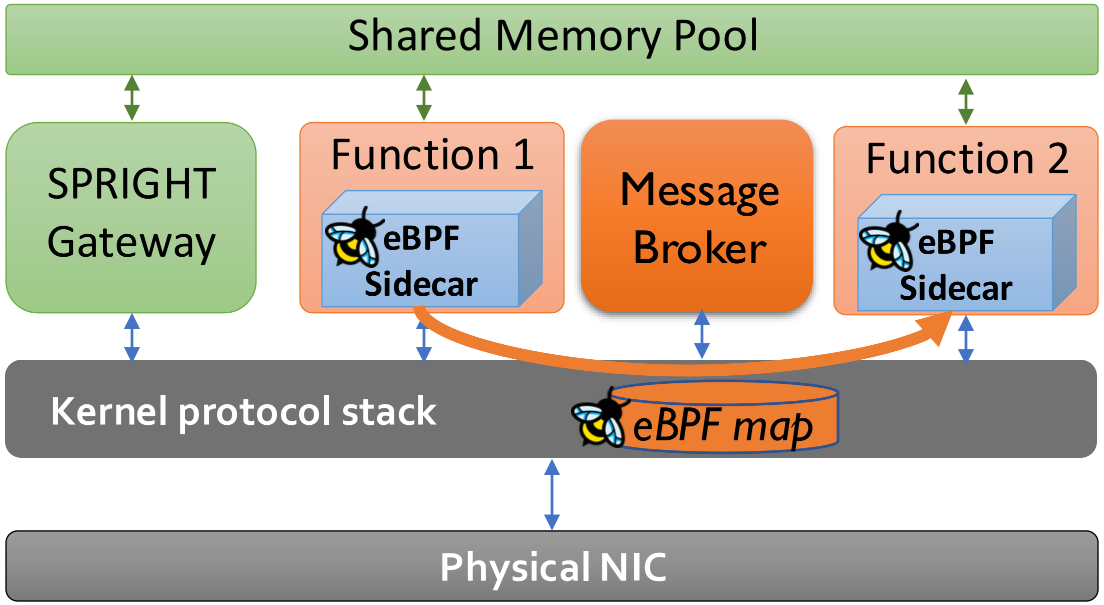
Enhancement #2: Shared memory processing

Reduce data movement overhead

Enhancement #3: Direct Function Routing (DFR)

Simplify inter-function invocations

Evolving to SPRIGHT



Networked Systems Group

UC RIVERSIDE

Overhead auditing: Existing Design vs. SPRIGHT

Existing Design

Data Pipeline No.	External			Within chain				Total
	①	②	total	③	④	⑤	total	
# of copies	1	2	3	4	4	4	12	19
# of ctxt switches	1	2	3	4	4	4	12	19
# of irqs	3	4	7	6	6	6	18	31
# of proto. processing	1	2	3	3	3	3	9	15
# of serialization	0	1	2	2	2	2	6	9
# of deserialization	1	1	1	2	2	2	6	10

SPRIGHT

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	9
# of irqs	3	4	7	2	2	4	13
# 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

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

***eBPF-based event-driven capability and shared memory processing** brings substantial reduction of overheads **within** the serverless function chain*

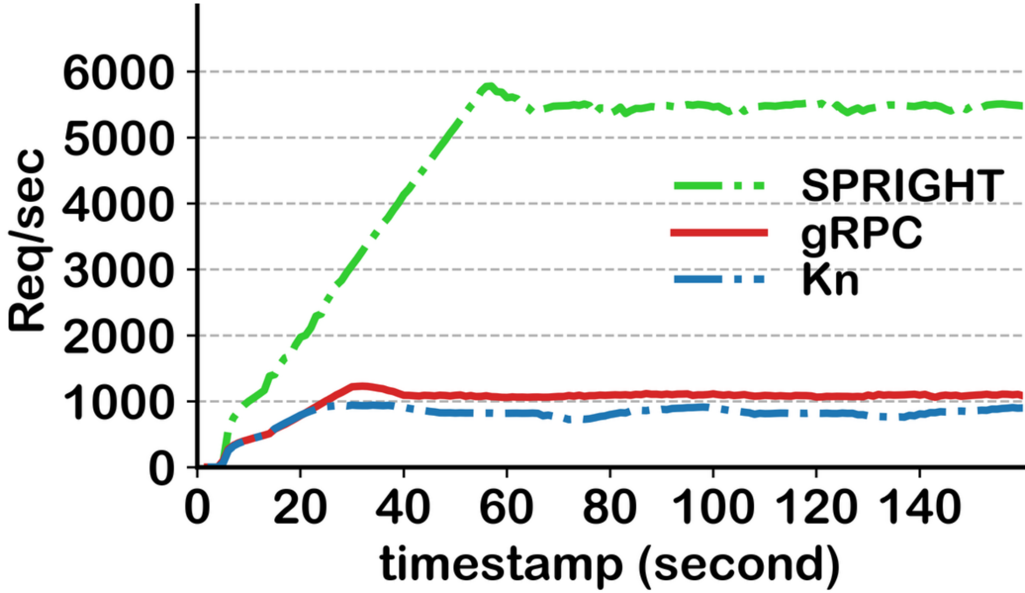
etworked
tems Group
RIVERSIDE

Performance with Online Boutique

SPRIGHT vs. Serverful gRPC mode (no sidecar & DFR) vs. Serverless (Knative)

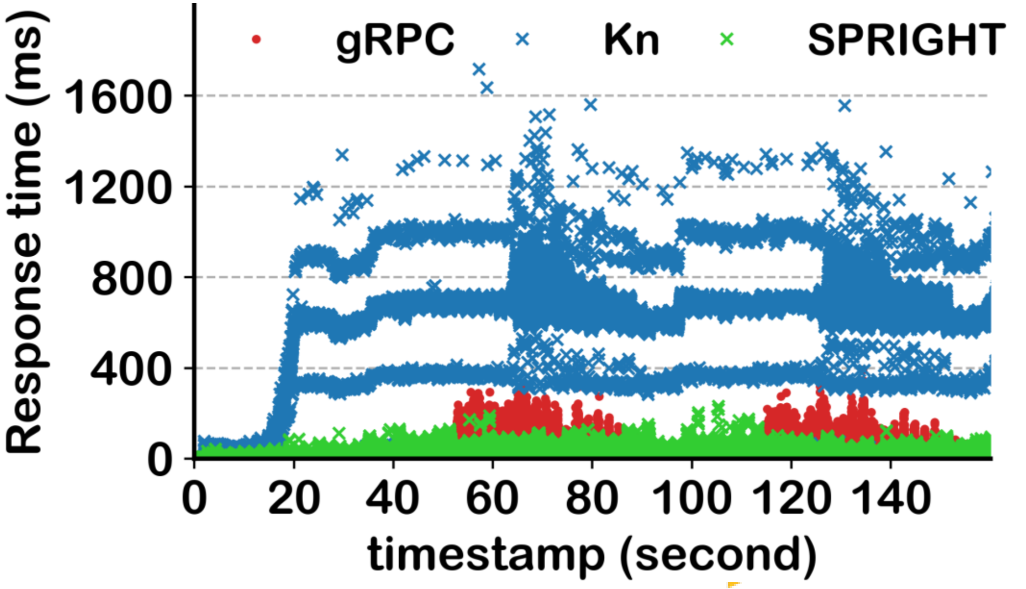
Throughput:

- **SPRIGHT** maintains a stable RPS of ~5500 requests per second
  - ➔ (**5**× more than **Serverful**)
  - ➔ (**6**× more than **Knative**)



Response Latency:

- **SPRIGHT** has very low tail latency (95%ile)
  - **10X** lower than **Serverful**
  - **52X** lower than **Knative**

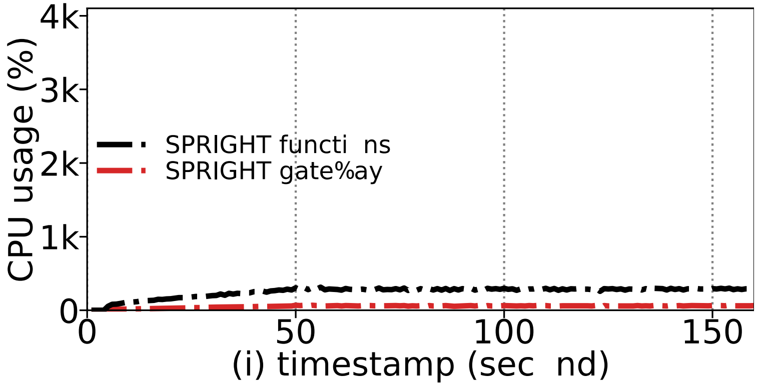


Efficiency with Online Boutique

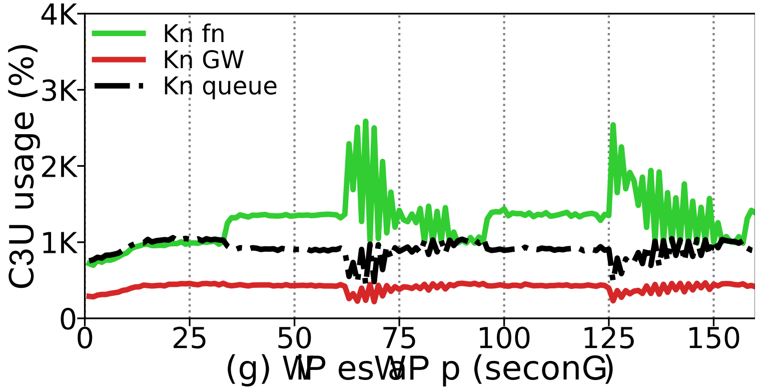
SPRIGHT vs. Serverful gRPC mode (no sidecar & DFR) vs. Serverless (Knative)

CPU efficiency:

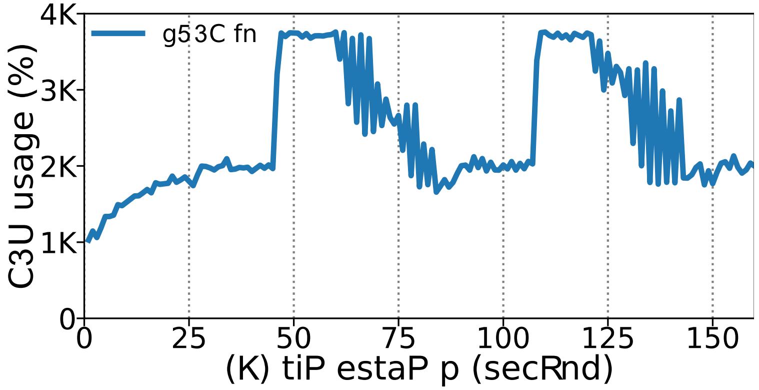
- SPRIGHT consumes in total only **~3 CPU** cores
  - Functions + SPRIGHT Gateway
  - Only **10%** of Knative and gRPC



SPRIGHT



Knative



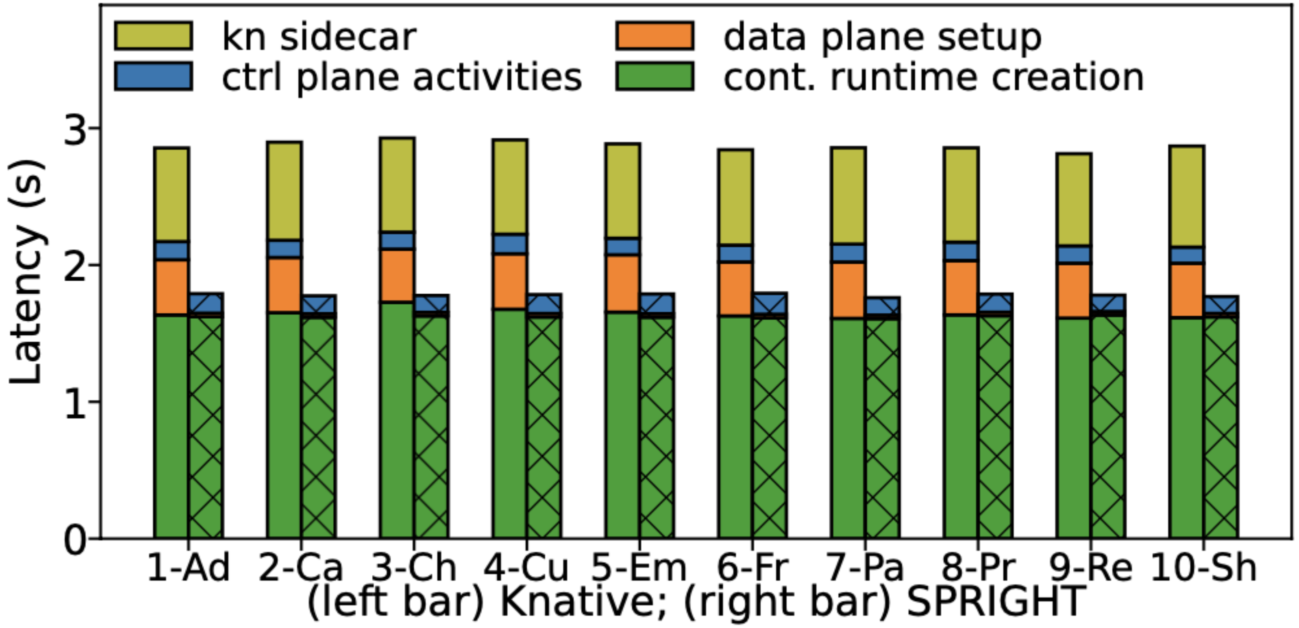
Serverful (gRPC)

Startup latency comparison

Knative (left bars) vs. SPRIGHT (right bars)

- Same control plane and Docker container runtime
- Evaluated with online boutique functions

- **Key observation**:
  - SPRIGHT has negligible latency spent on creating eBPF sidecar and setting up shared memory data plane
  - **<u>But container runtime creation dominates the overall startup latency</u>**



Networked Systems Group

UC RIVERSIDE

The tradeoff between isolation and agility in virtualized runtime

***Problem #1:** Single-address-space unikernel is considered not safe*

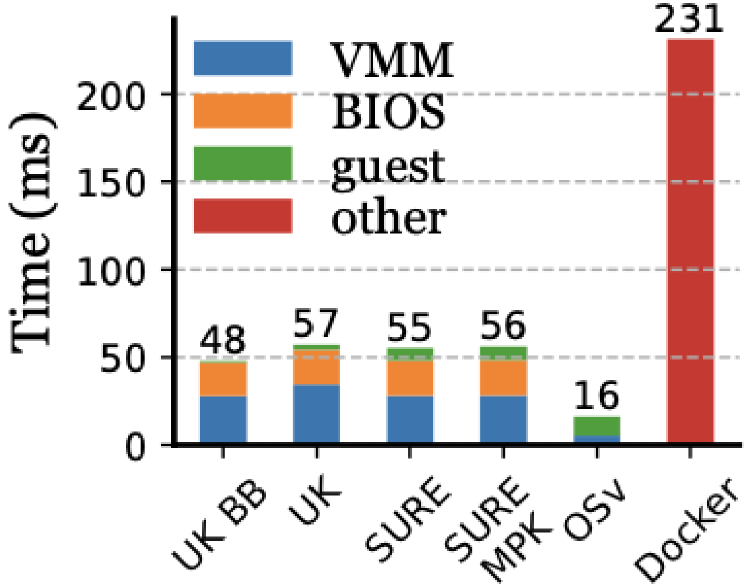
- Isolating serverless functions in open, shared cloud

Virtualized runtime	Isolation	Startup speed
Container	Weak ✗	Not satisfactory ✗
Full-size VM	Strong ✓	Poor ✗
Unikernel	Strong? ✓	Good ✓

- **Unikernel can make serverless functions agile and enable strong isolation**
  - *~4× faster startup compared to Docker containers*
- Unikernel offers **single address space**

➢ Exploration of Unikernels for serverless and microservices
  - ➢ USETL [**APSys'19**], UaaF [**IWQoS'20**], SEUSS [**EuroSys'20**], NanoVMs
  - ➢ MirageOS [**ASPLOS'13**], OSv [**ATC'14**], LightVM [**SOSP'17**], Unikraft [**EuroSys'21**]

Startup Latency



- UK **BB**: **B**are-**B**ones UniKraft
  - UK and SURE use **QEMU**
- OSv: OSv unikernel + **Firecracker**
- Docker: docker container

Sharing Memory space is considered not safe

- **The idea of "memory space sharing" is wonderful**
  - Data plane: zero-copy communication
  - Virtualized runtime: NO user-kernel boundary crossings

- **But, "memory space sharing" is often considered harmful**
  - A potential conduit for data leakage and corruption
  - May be caused by malicious or buggy behavior

- Need to address concerns about "sharing" concerning **two aspects**
  - **Inter-unikernel**: between different functions using shared memory
  - **Intra-unikernel**: between user code and the unikernel LibOS modules

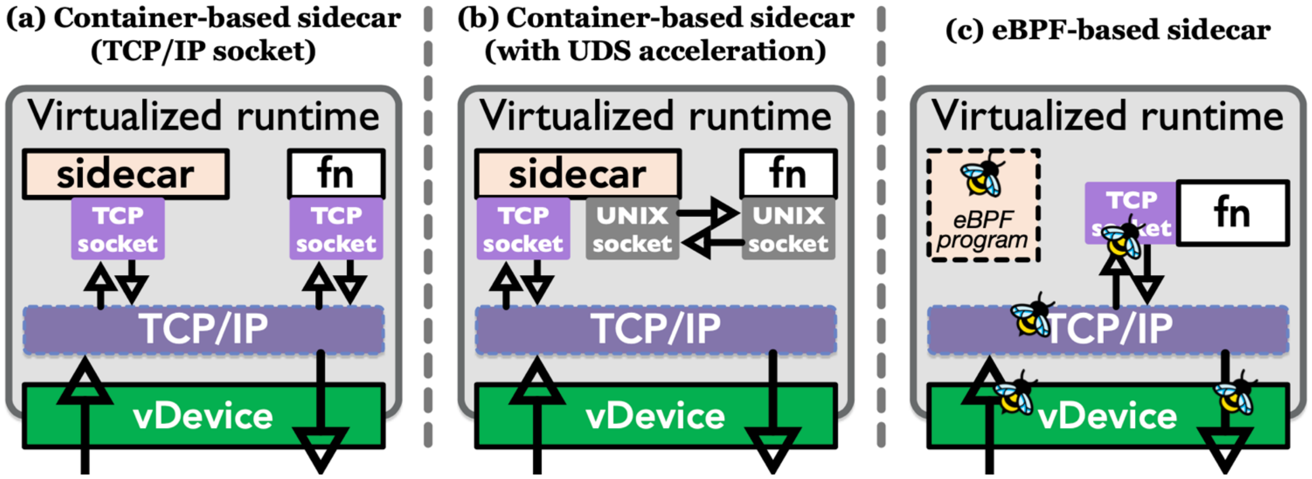
***Problem #1:** Single-address-space unikernel is considered not safe*

***Problem #2:** Shared memory processing is considered not safe*

Networked Systems Group
UC RIVERSIDE

eBPF is not suitable for unikernels

- **eBPF cannot be fully utilized in unikernel environments**
  - lack of certain eBPF hooks
- **eBPF doesn't provide the full (L7) payload visibility**
- **eBPF has a constrained programming model**



Problem #1: ***Single-address-space unikernel is considered not safe***

Problem #2: ***Shared memory processing is considered not safe***

Problem#3: ***Shared memory processing is limited to a single node***

Problem#4: ***eBPF is not suitable for unikernels***

Our solution SURE

Secure Unikernels Make Serverless Computing Rapid and Efficient

Unikernels with protection:

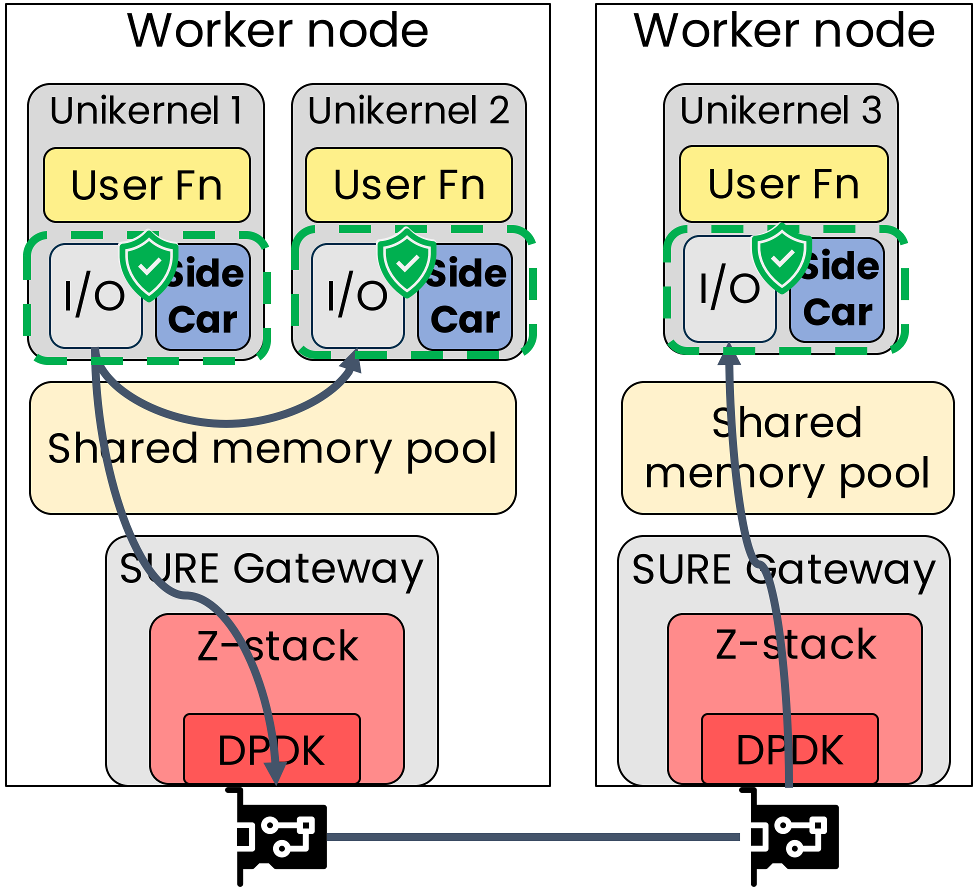
Shared-memory intra-node data plane ✓

Zero-copy inter-node TCP/IP stack (**Z-stack**)

Consolidated proto. processing by SURE Gateway

Library-based sidecar ✓

✓ **MPK-based call gate**



***Design#1:** Secure shared memory while retaining its high performance*

***Design#2:** Enhance intra-unikernel isolation to sandbox user code*

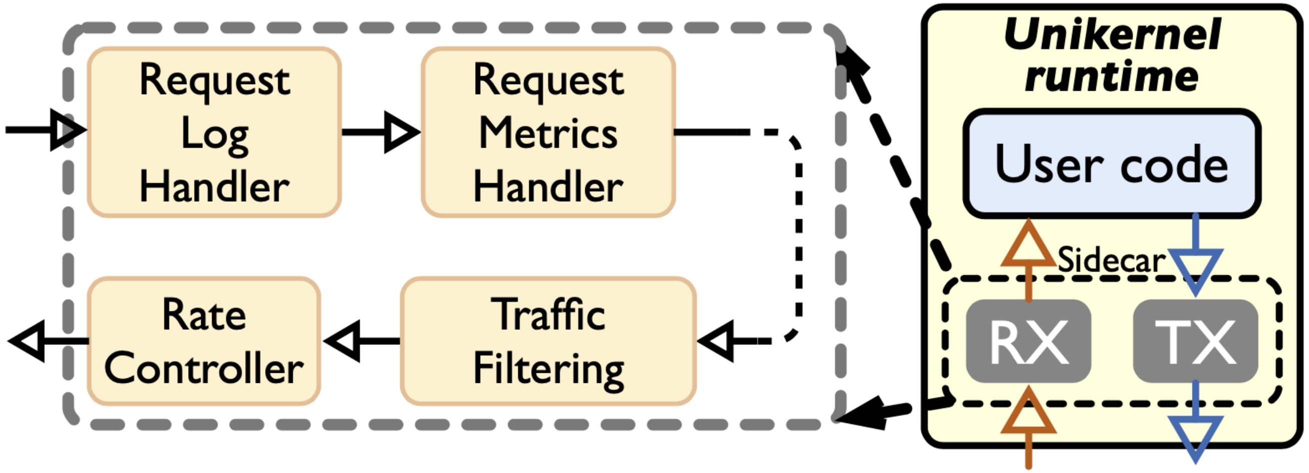
***Design#3:** Extended zero-copy networking to be distributed*

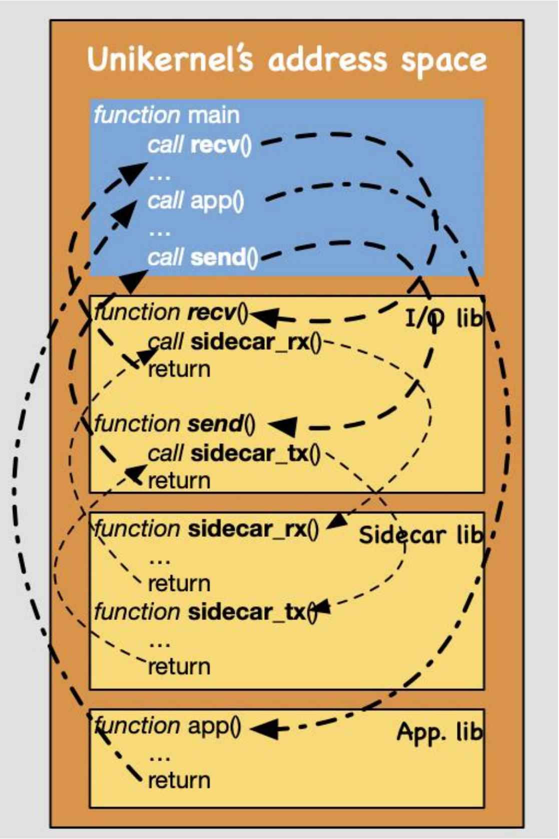
***Design#4:** eBPF-like sidecar with L7 visibility in unikernels*

Library-based SURE Sidecar

Based on the LibOS design of unikernels

- Deploy the sidecar as a **library** linked into the function code within the unikernel
  - The sidecar contains a sequence of handlers that perform certain sidecar functionalities





- The unikernel's **single-address-space** simplifies data exchange between sidecar and user code
  - Invocation is made by procedure call
  - Overcomes shortcomings of an individual userspace sidecar.
- BUT: library-based sidecar must address concerns of sharing the memory space

Microbenchmark Analysis

Improvement with library-based sidecar

Baseline: container-based sidecar (use NGINX)

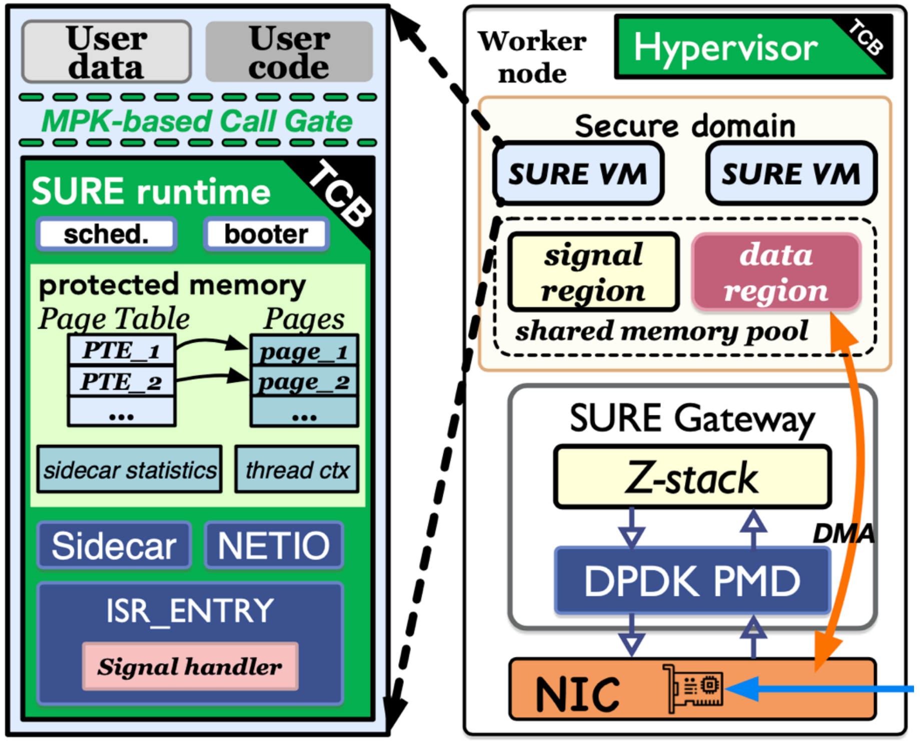
* Same client and server functions as "intra-node shared memory processing" benchmark

- ***Library-based sidecar shows negligible overhead***
  - The CPU cycles consumed by our library-based sidecar are negligible compared to those of a NGINX sidecar (**only 0.9%**)
  - The reduced CPU consumption also results in reduced delay and increased throughput

Msg size	CPU cycles (X 1K)		Added delay (us)		Throughput (Mbytes per sec.)		
	Libsidecar	NGINX	Libsidecar	NGINX	No sidecar	Libsidecar	NGINX
256B	0.50	60.4	0.21	25.2	342	309	12.3
4KB	0.55	59.5	0.23	24.8	3697	3533	185
8KB	0.55	58.2	0.23	24.2	5525	5369	337

Trust model and Threat model in SURE

- **Trusted Computing Base (TCB) in Unikernel:**
  - Hypervisor and associated toolchains
  - Unikernel modules: scheduler, booter, sidecar, network I/O lib …
- **Trust model**:
  - Users trust the serverless infrastructure (SURE), but SURE does not trust users
    - User applications may contain security vulnerabilities, e.g., buggy code
  - Functions within a chain trust each other, functions in different chains may not
- **Threat sources due to the inevitable sharing of the memory space**
  - Vulnerabilities from shared memory processing
  - Intra-unikernel vulnerabilities from a single address space



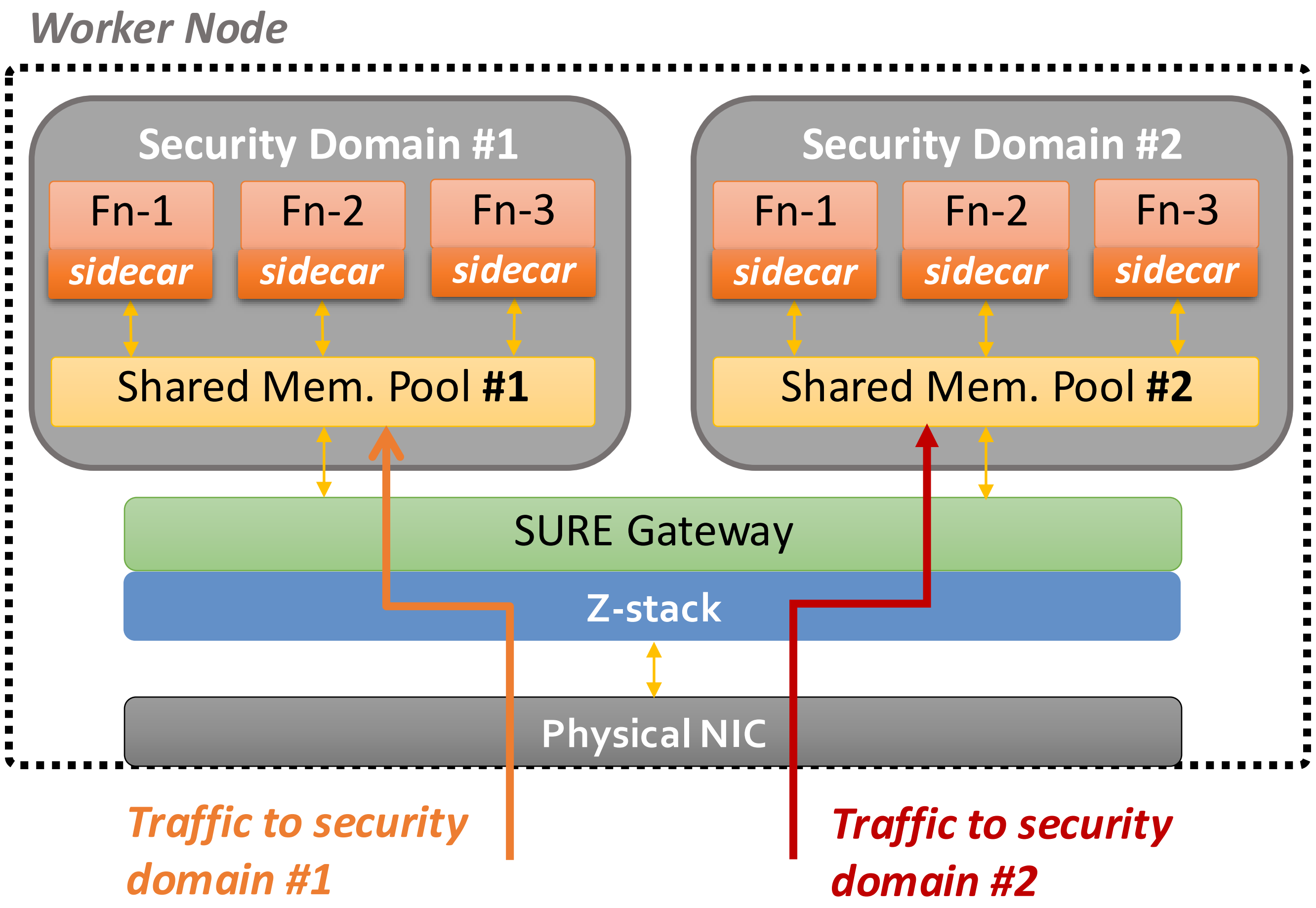
Networked Systems Group
UC RIVERSIDE

Multiple Levels of Isolation in SURE

Overview

- VM-based sandbox (based on QEMU/KVM)
- Group-based security domains with isolated memory pools
- Access control: with SURE gateway and sidecar

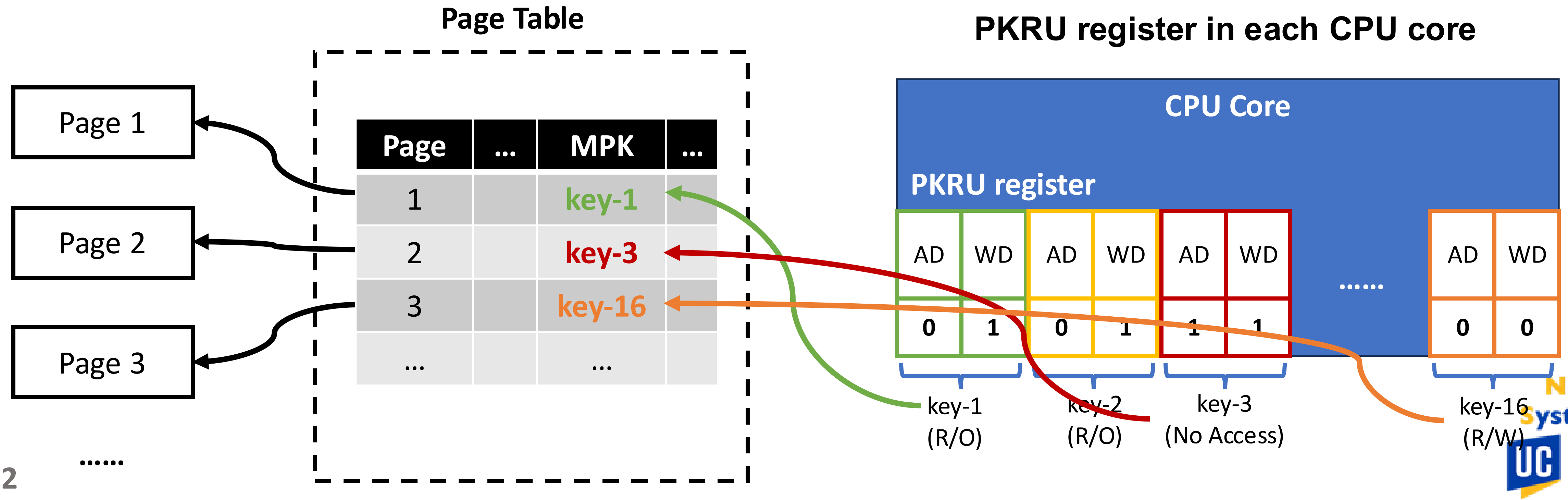
- **MPK-based call gate**
  - Use MPK to enable memory-level isolation in a ***shared*** address space to protect
    1. Shared memory data plane between functions
    2. Sandbox the untrusted user code within the single-address-space Unikernel
  - Relatively small overhead for the reward of robust memory-level isolation



Memory-level isolation in SURE

A Primer on MPK (Memory Protection Key)

- MPK is a hardware-level, intra-process memory isolation feature in Intel’s server CPUs (since 2019)
- **PKRU (Protection Key Register User)**
  - A per-core, 32-bit CPU register defines the access privilege of MPK, described by 2 bits
    - “Access Disable” (**AD**) and “Write Disable” (**WD**)
  - A total of **16** keys available within a SURE function
  - **Read/Write (0, 0), Read-Only (0, 1), or No-Access (1, ×)**



Page Table

PKRU register in each CPU core

Page	…	MPK	…
1		key-1	
2		key-3	
3		key-16	
…		…	

Memory-level isolation in SURE

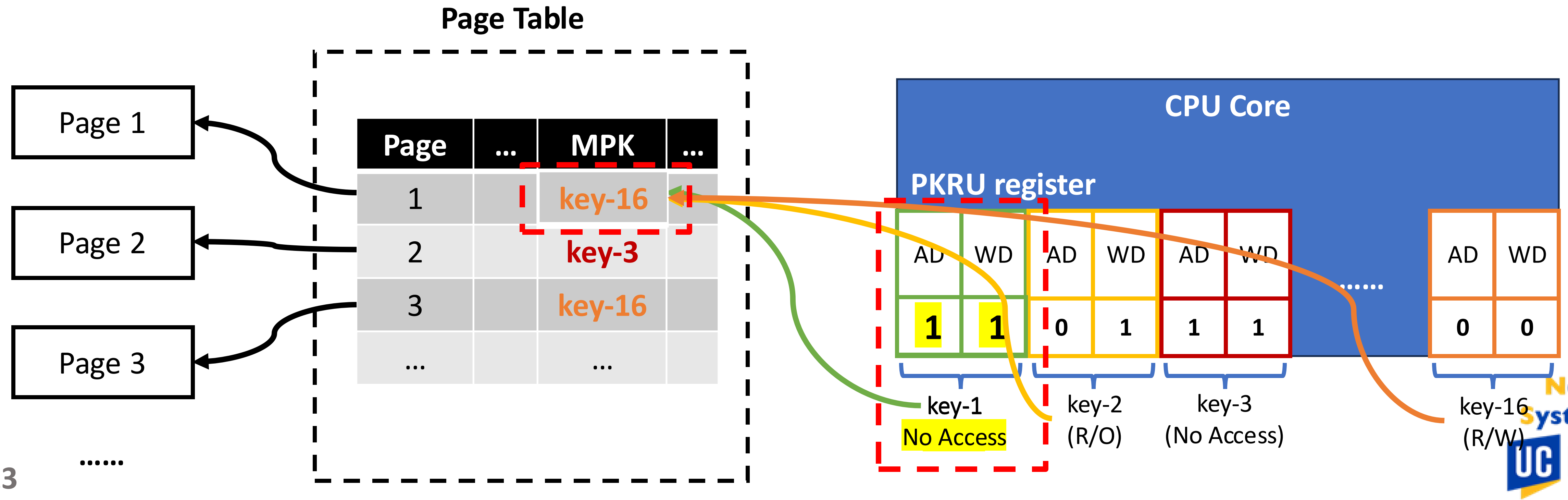
SURE uses *two* approaches to switch the access privilege of a memory page

- **#1 WRPKRU** (Write Data to PKRU)
  - x86 instruction to change the access privilege of the MPK by modifying PKRU
  - *But a SURE function may access more than 16 pages!*
  - *Not feasible to tag each page with a distinct key*
- **Memory related to Unikernel TCB components is managed by WRPKRU**

Coarse-grained but faster

- **#2 "PTE Update"**
  - Update the 4 bits reserved for the MPK key ID in the PTE
  - Then flush the corresponding TLB entry
  - Allow for more ***scalable*** access management
- **Shared memory buffers are managed by "PTE Update"**

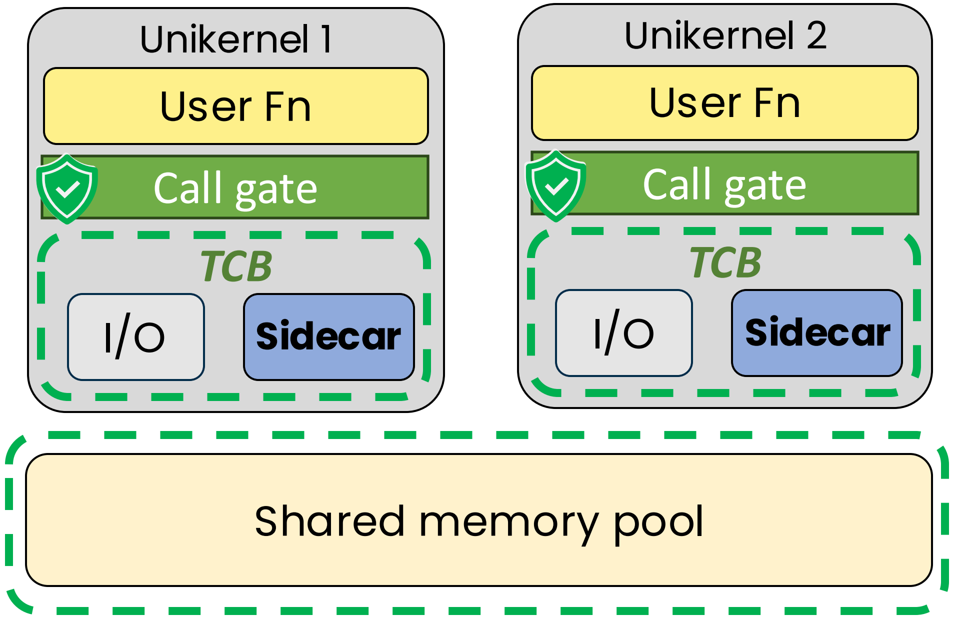
Fine-grained but slower



Secure APIs based on SURE call gates

A “call gate” abstraction for user code to safely interact with protected pages

- **Only call gate can update access privilege**
  - Via WRPKRU or PTE Update
  - Easier to work with ***binary inspection*** to prohibit illegal updates to access privilege
- **Enhanced unikernel TCB (from Unikraft) in SURE**
  - Prevent unwanted update or access to PKRU register and PTEs of protected pages
  - Avoid ***Privilege Escalation*** of MPK in a single address space
  - *Refer to the paper*



Secure APIs based on SURE call gates

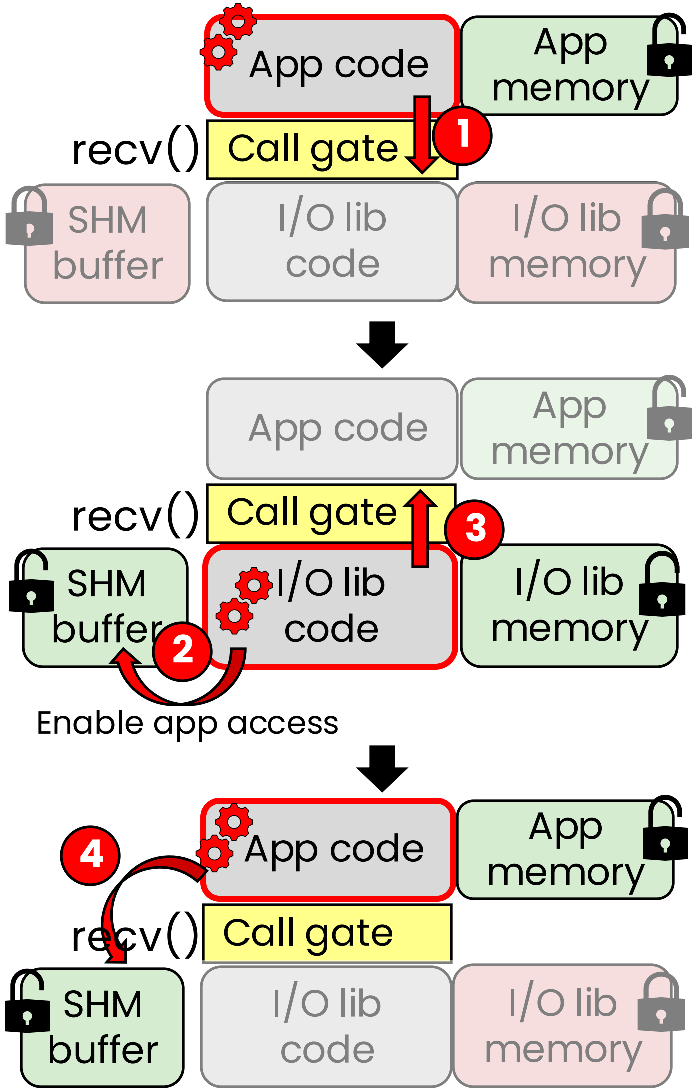
Example: Untrusted user code invokes the privileged API recv()

- Initially, the set of protected pages (i.e., stack memory in TCB or shared memory) is configured to be **inaccessible**

1. User code invokes recv() in NetI/O lib - intercepted by the call gate
2. Call gate makes protected (stack) memory accessible and invokes the recv() API
3. recv() API receives a buffer descriptor and updates the corresponding MPK key to allow user code access to the buffer
4. Call gate returns to user code
5. Call gate disables access to protected stack memory, **while the received buffer remains accessible**

Other privileged APIs in function runtime are guarded in the same way

Memory protection is re-enforced with the send() function



Networked Systems Group

UC RIVERSIDE

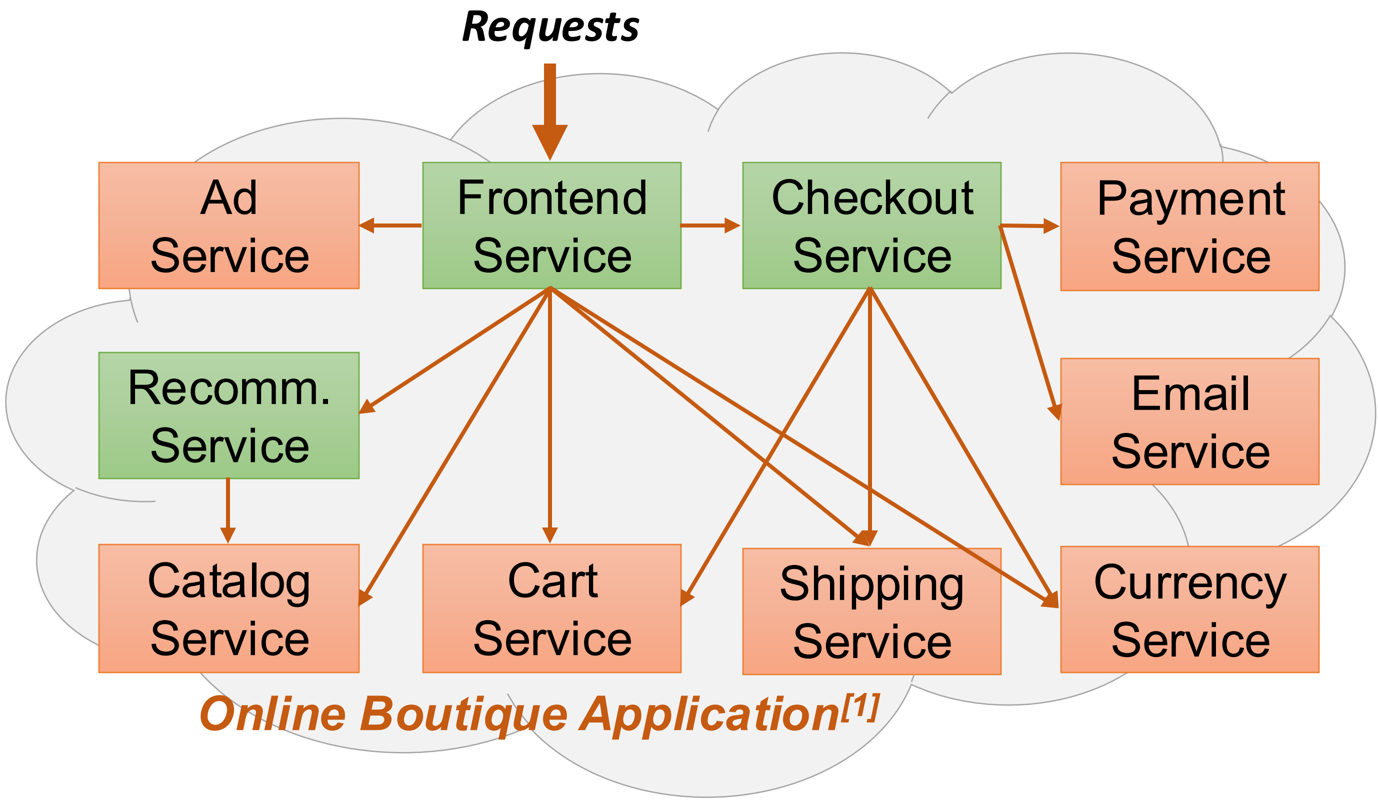
[1] https://github.com/GoogleCloudPlatform/microservices-demo

Realistic Workload Evaluation

Experiment setting

Online Boutique Microservice Chain [1]

- *Intense* web workload with 10 functions
- 6 different function chains



Online Boutique Application[1]

Serverless Alternatives

- Knative
- SPRIGHT [SIGCOMM'22]
- NightCore [ASPLOS'21]

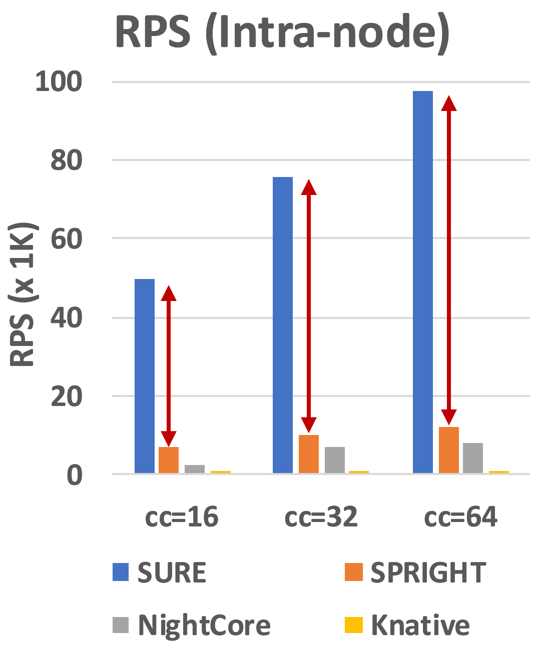
Two distinct deployment settings:

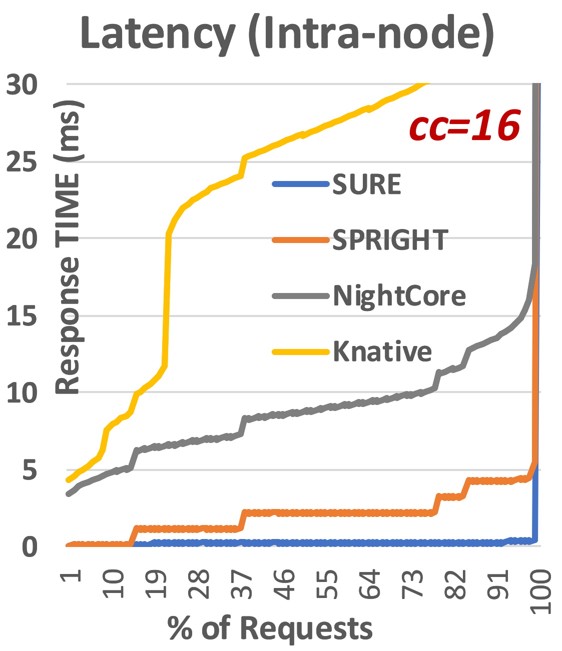
1) Intra-node
2) Inter-node: Orange and Green functions deployed on distinct nodes

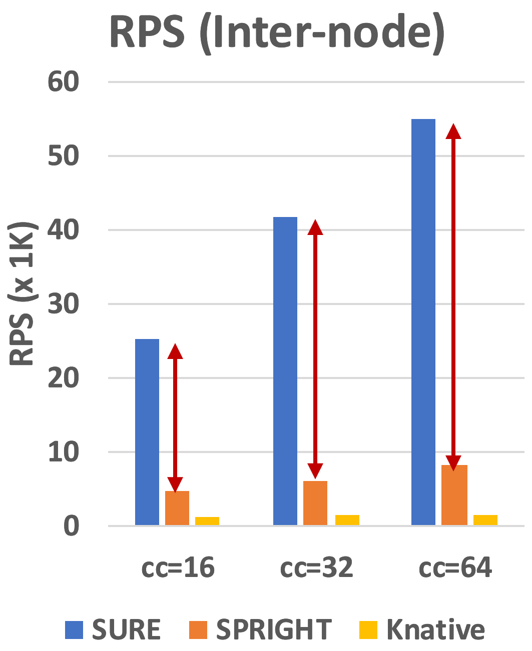
***Testbed**: **Three** sm110p nodes on Cloudlab (with 100 Gbps NIC); Ubuntu 22.04; kernel 5.15*

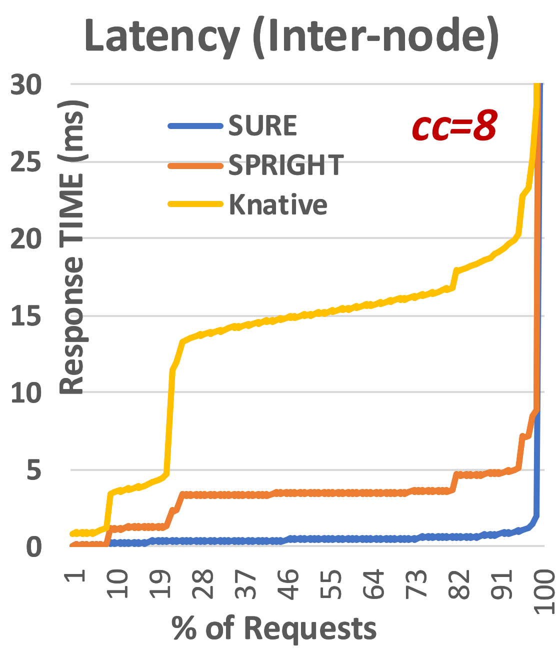
Realistic Workload Evaluation

Requests per second & Tail latency


RPS (Intra-node)


Latency (Intra-node)


RPS (Inter-node)


Latency (Inter-node)

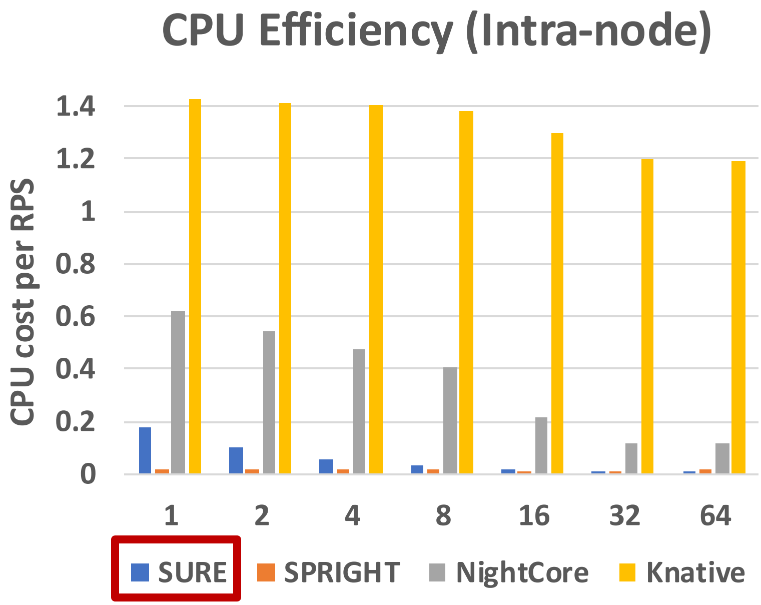
- **SURE is an order of magnitude better than any alternatives we evaluated**
- **Performance improvement attributed to the use of *distributed zero-copy data plane* and lightweight *library-based sidecar***

Realistic Workload Evaluation

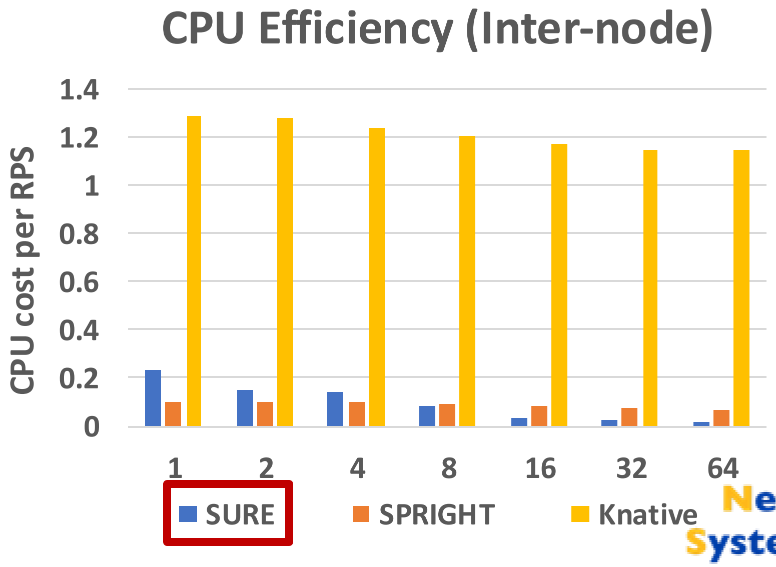
CPU efficiency

- Our metric - “CPU Cost Per RPS” (**CCR**)
  - Defined as $\frac{Average\ CPU\ utilization}{RPS}$
  - ***Lower*** values of **CCR** suggest that each request requires ***fewer*** CPU cycles
    - *A more efficient use of the CPU*

- **SURE** is more efficient than **NightCore** and **Knative**
  - No kernel networking; More lightweight sidecar; etc
- **SURE** is less efficient than **SPRIGHT** at a low concurrency (≤ 16 for intra-node and ≤ 4 for inter-node)
  - Comes from polling cost
- **SURE** is more efficient than **SPRIGHT** under high concurrency levels
  - **SPRIGHT** uses kernel for inter-node traffic, CPU usage grows substantially under high concurrency levels
  - More concurrent processing amortizes the polling cost



Worse efficiency



Better efficiency

Horizontal axis: concurrency

Networked Systems Group

UC RIVERSIDE

Summary

Existing serverless designs involve many long-running, stateful components in userspace

- Container-based sidecar, Message broker
- Amplifying kernel networking overheads
  - Performance loss, significant CPU costs

SPRIGHT enabled truly event-driven, load-proportional serverless computing:

- *eBPF-based stateful processing:* Event-driven, lightweight
- *Shared memory processing:* Streamlined data plane; High performance and resource efficient

SURE is a unikernel-based, lightweight serverless framework

- Unikernel-based runtime brings **4× faster startup** vs. docker containers
- Uses MPK-based call gates to enable **fine-grained memory access management**
  - Mitigate the vulnerabilities of **memory space sharing**
  - While retaining high performance and efficiency
- Offer **zero-copy** inter-function networking and lightweight **library**-based sidecars
  - ***Yield up to 8× RPS improvement compared to SPRIGHT in a distributed environment***
  - ***While being more secure***

Resources

- SPRIGHT (Sigcomm 2022, IEEE/ACM Transactions on Networking 2024)
  - Journal Paper: https://dl.acm.org/doi/abs/10.1109/TNET.2024.3366561
  - Open Source Code: https://github.com/ucr-serverless/spright

- SURE (SoCC 2024)
  - Paper: https://dl.acm.org/doi/10.1145/3698038.3698558
  - Open Source Code: https://github.com/ucr-serverless/sure