NSDI 2018

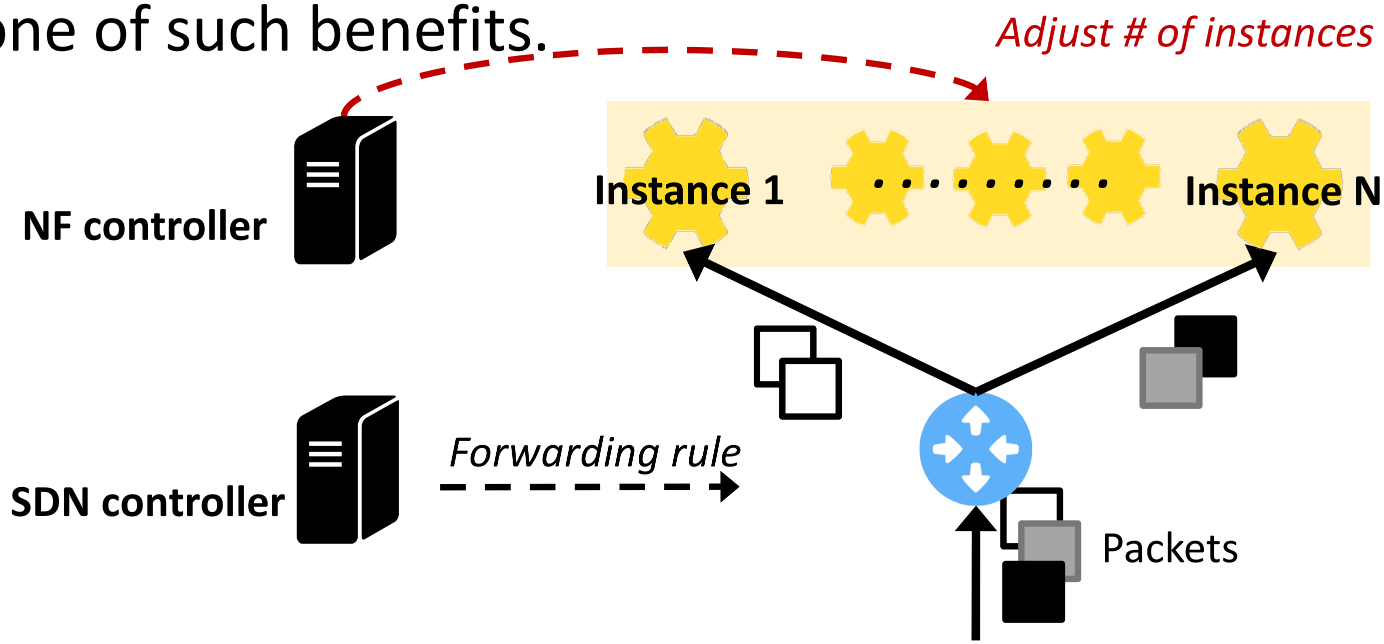
Elastic Scaling of Stateful Network Functions

Shinae Woo[*+], Justine Sherry[*], Sangjin Han[*],
Sue Moon[+], Sylvia Ratnasamy[*], Scott Shenker[*]

[+] KAIST, [*] UC Berkeley

Elastic Scaling of NFs

- NFV promises the benefit of virtualization; Elastic scaling is one of such benefits.



- Elastic scaling: Adjusting the number of NF instances in response to varying load.
- In practice, realizing elastic scaling comes at a significant cost of *correctness* and *performance*.

Requirements of Elastic Scaling

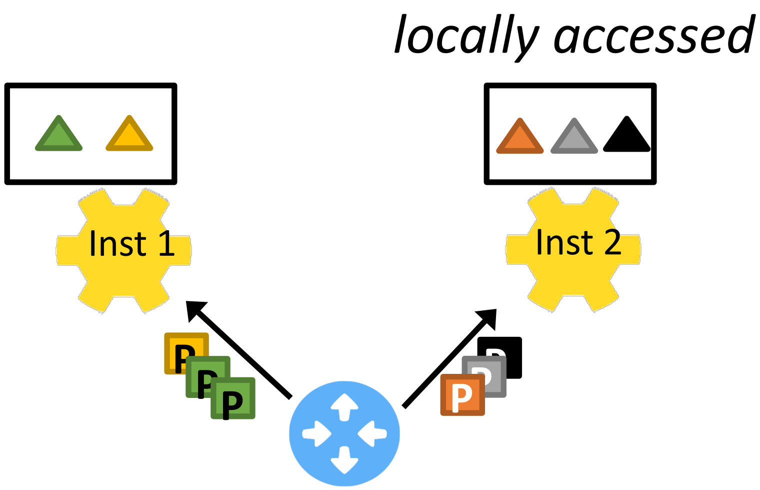
- Correct NF operations
  - Multiple instances work like a single instance, no matter how many and where they are.
- High performance
  - High throughput (10s – 100s of Mpps)
  - Low latency (sub-millisecond)
- Scaling events should not compromise above.

***Stateful* NFs make elastic scaling challenging.**

Background: NF State Types

Can state be distributed in a way that no remote access is necessary?

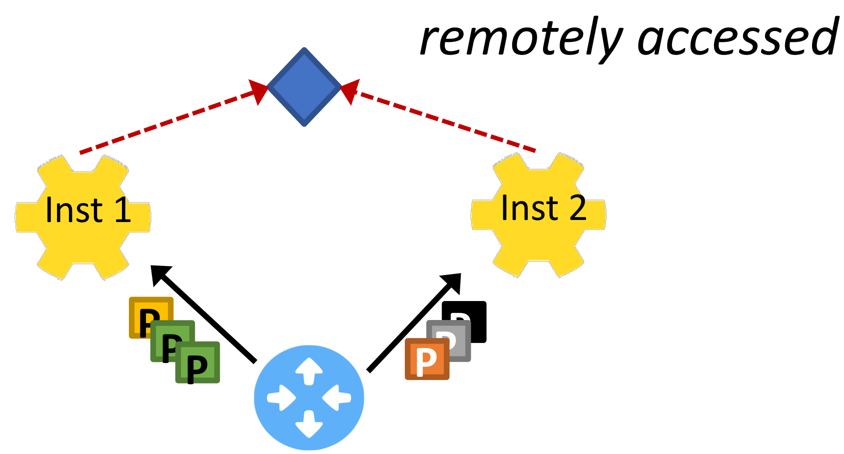
YES: Partitionable



- TCP connection state
- Per-flow statistics

State locality changes when scaling

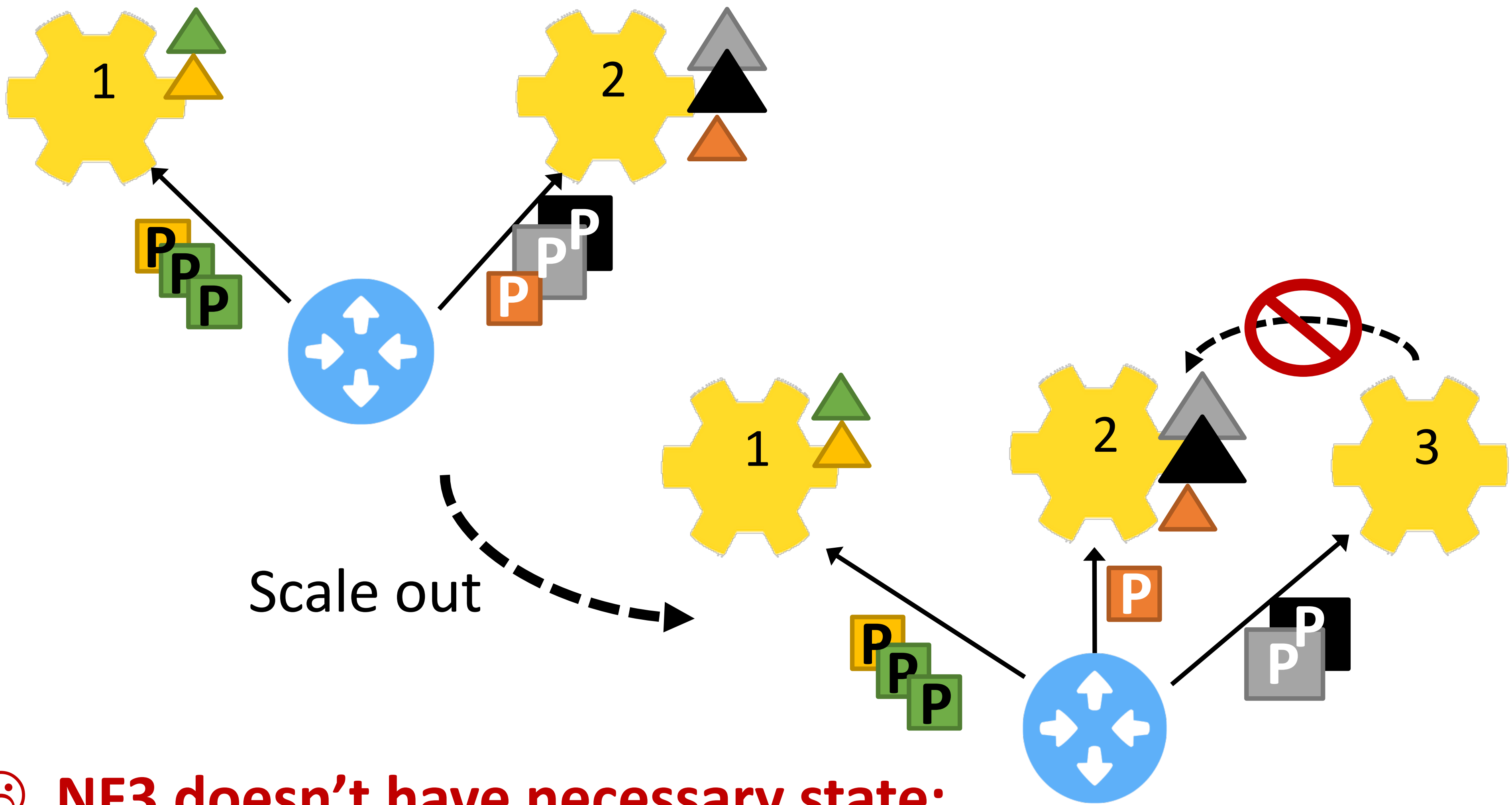
NO: Non-partitionable



- Attack detection status such as port scanner and password guesser

Remote access cost is expensive

Partitionable State: Scaling Breaks Correctness



☹ **NF3 doesn't have necessary state: sharing/migration is a must**

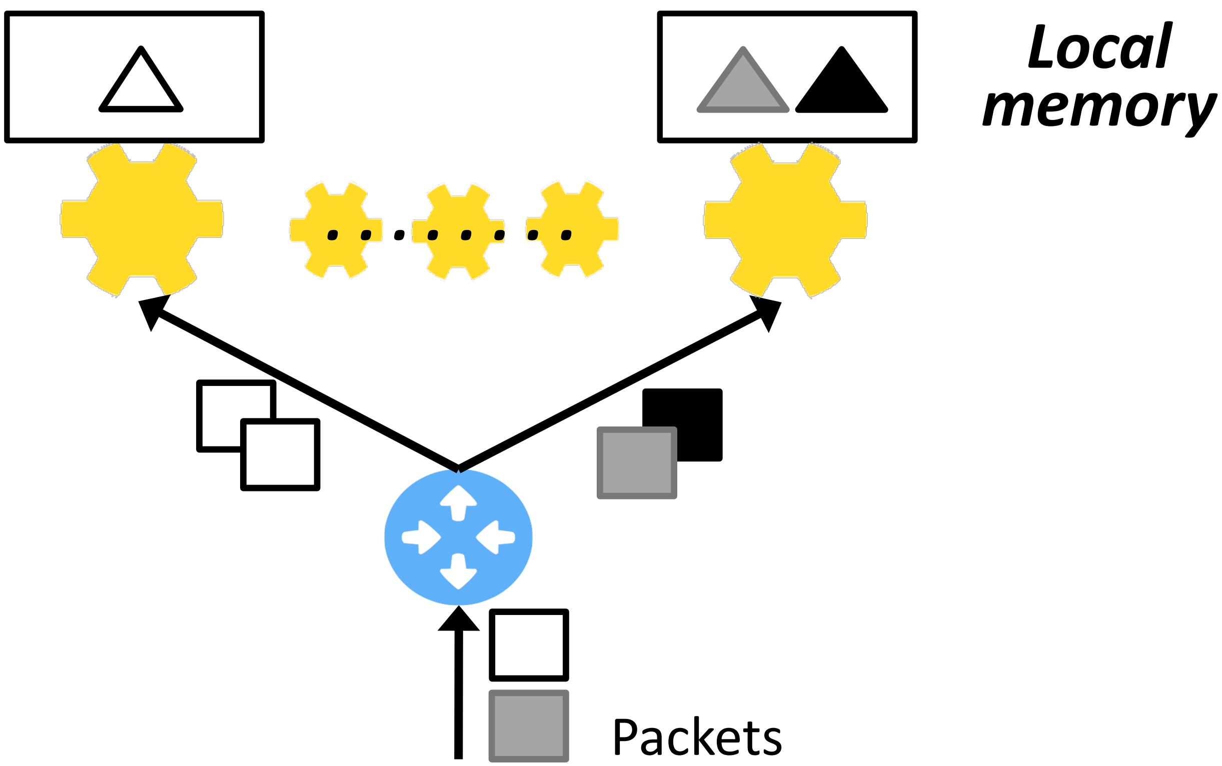
Prior NF state management models

(or, why managing **NF state** is so challenging?)

Traditional Model: Local-only

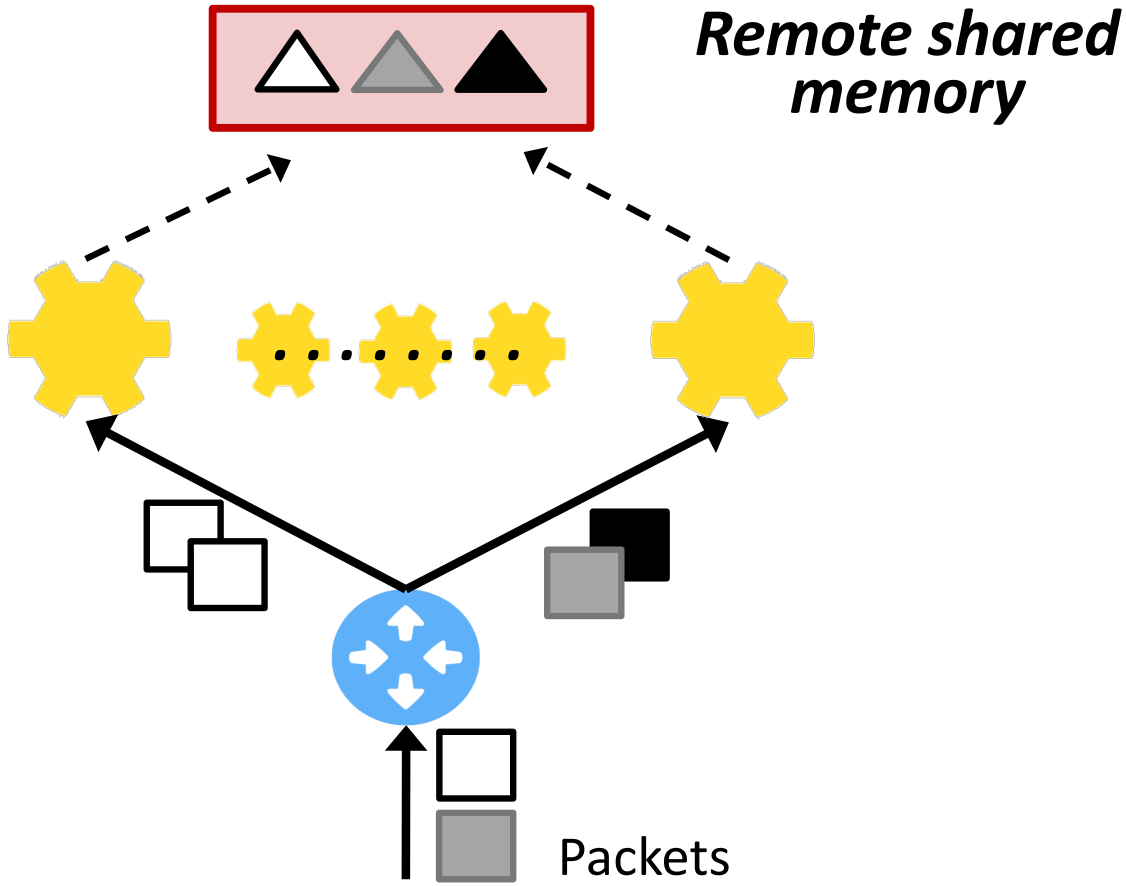
- NF states are in local memory

☹ No sharing support ☹ Incorrect behavior when scale-out

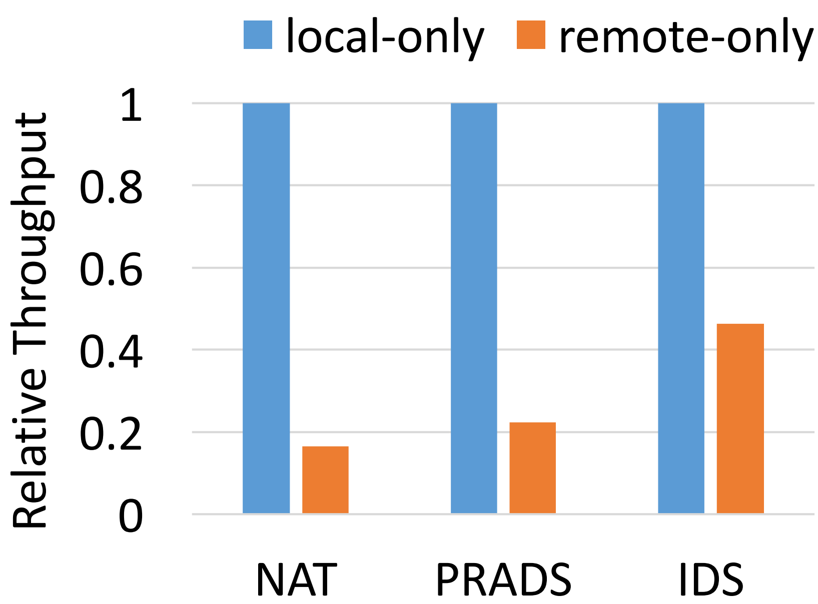


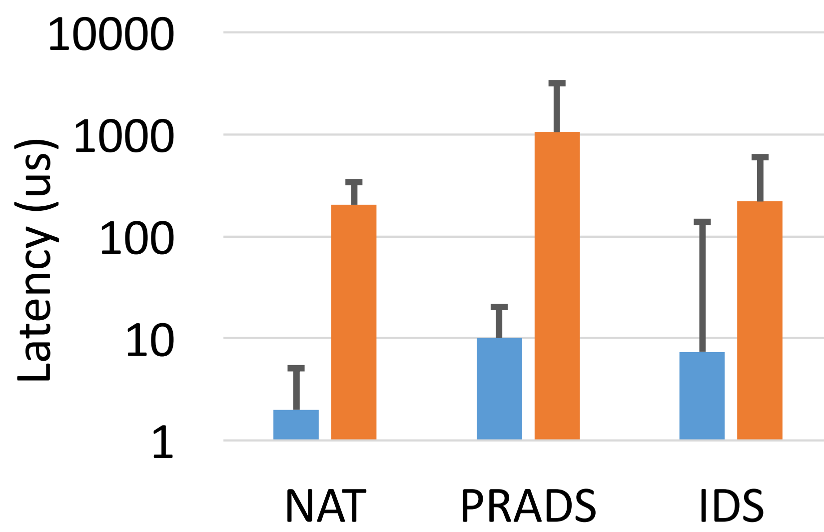
Remote-Only Model

- All state management is offloaded to remote storage



Remote-Only Sacrifices Performance





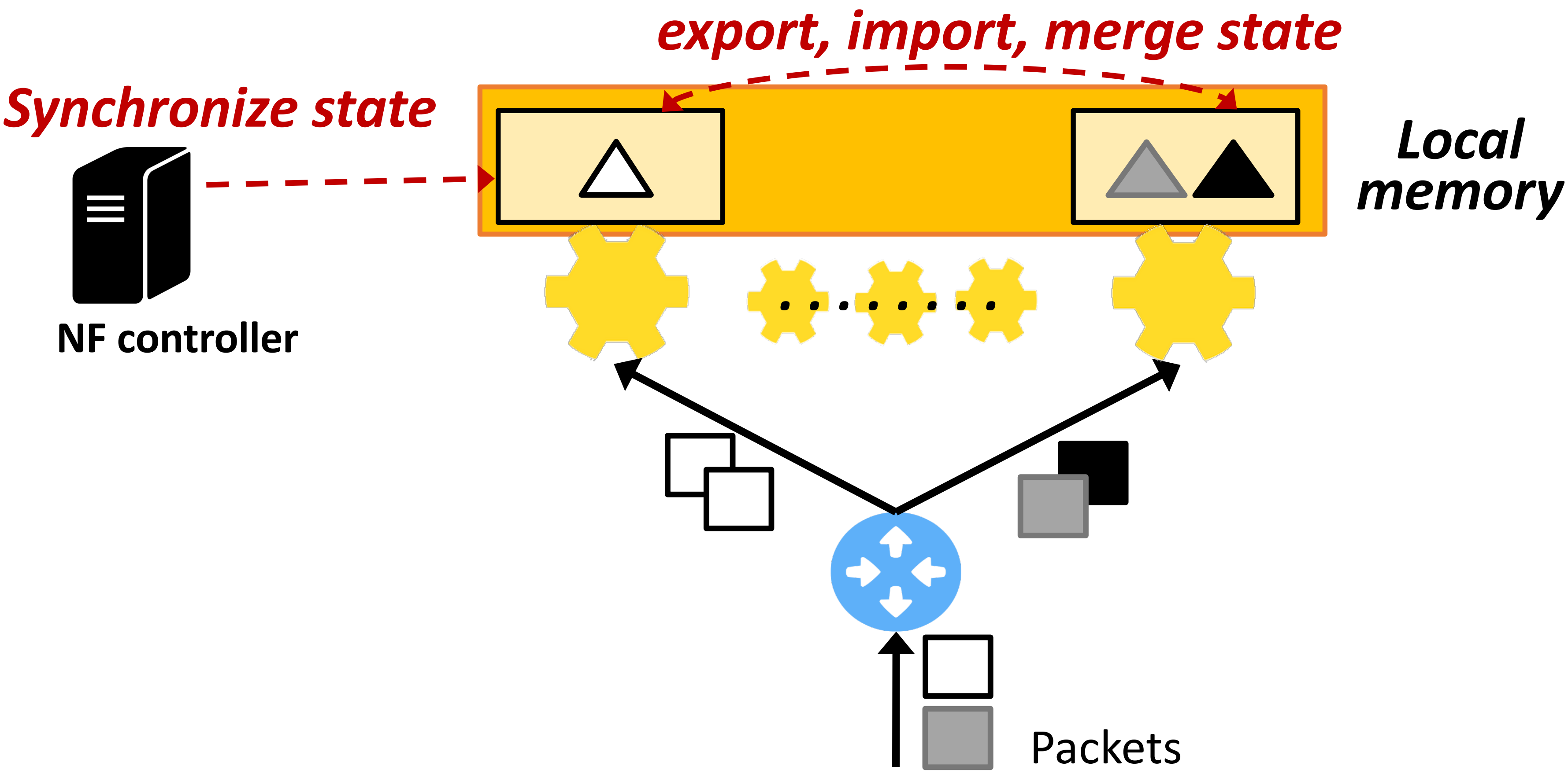
☹ **Losing throughput**

☹ **Inflating packet latency**

* For remote-only, we follow the algorithm described in *"Stateless Network Functions: Breaking the Tight Coupling of State and Processing", NSDI 2017*

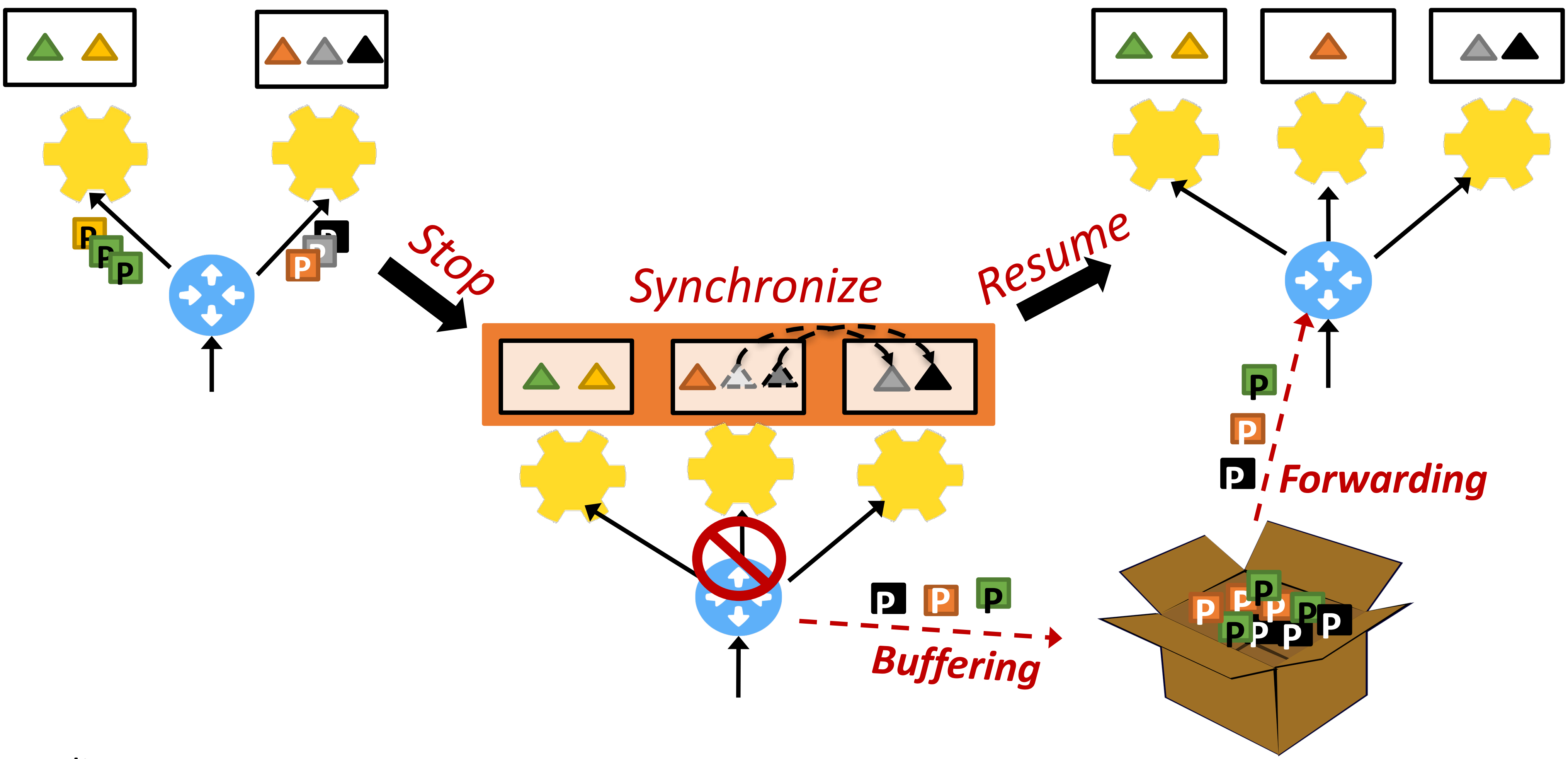
Local+Remote Model

- All state access is local
- Out-of-band control for state synchronization



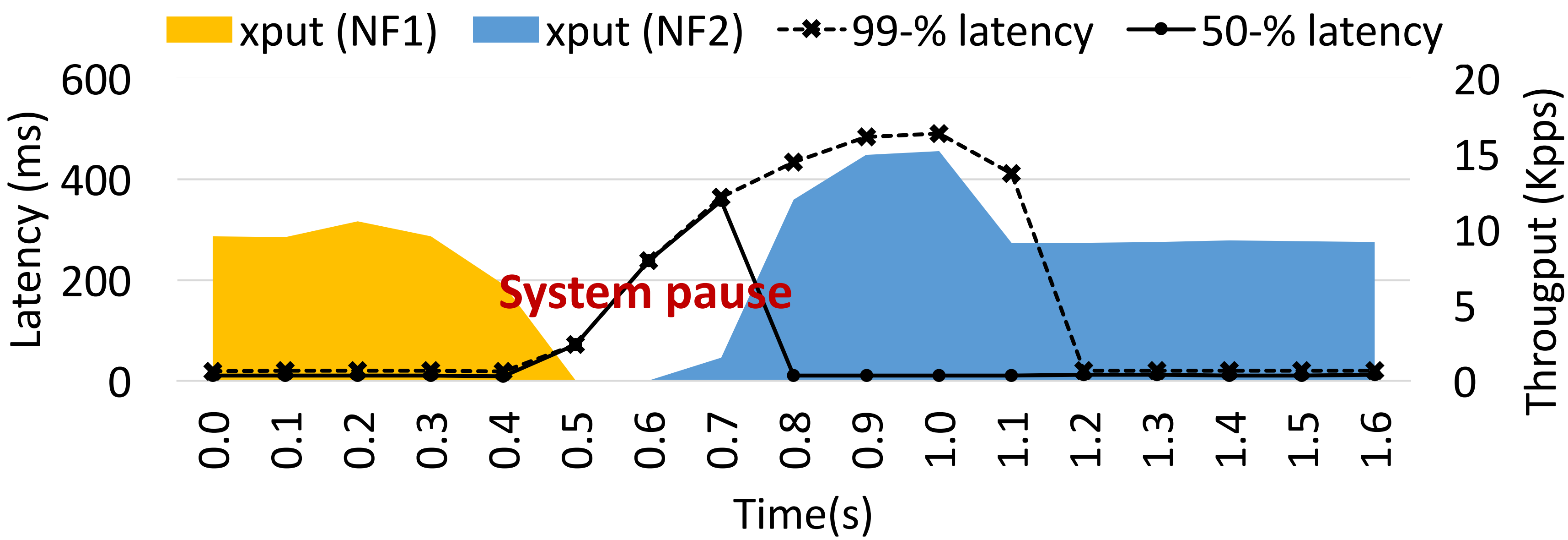
Stop-Synchronize-Resume: NO GOOD

- Centralized controller keeps state locality and consistency+
  - Proactively prepare state before it is accessed



+ SplitMerge[NSDI 2013], OpenNF[SIGCOMM 2015]

Local+Remote Trades Performance for Correctness



OpenNF*, PRADS (monitoring)

10kpps, 1500 flows context migration from NF1 to NF2

☹ 100s of ms median latencies

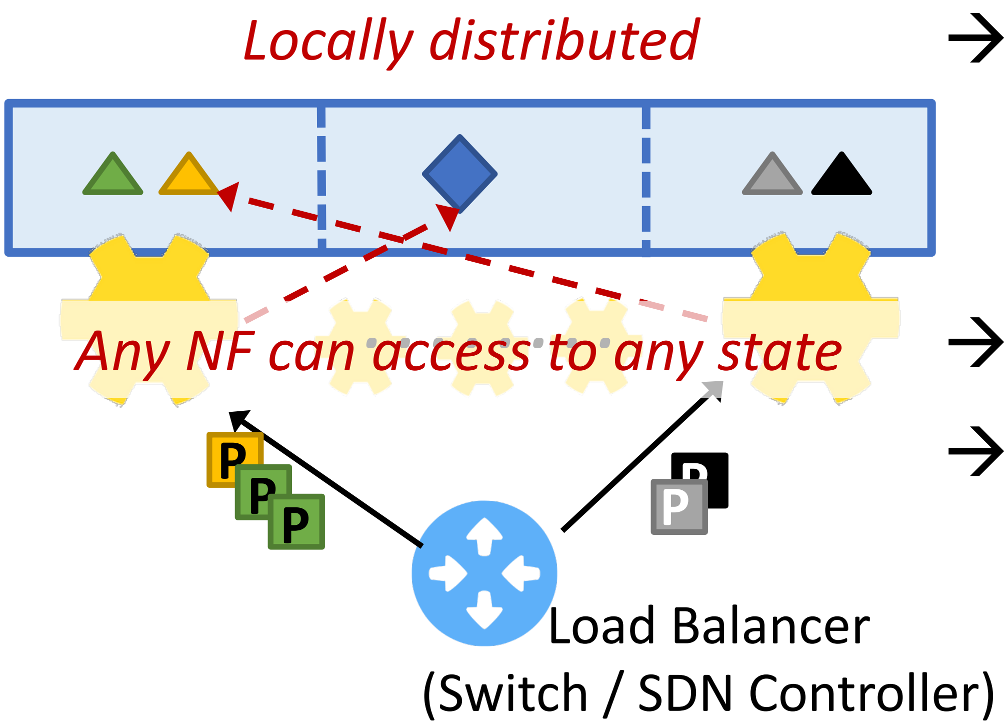
* *"OpenNF: Enabling Innovation in Network Function Control", SIGCOMM 2014*

Summary on State Management Model

	Normal Operation (Without scaling-out)	Scaling-out
Local-only	☹ No scaling	
Remote-only	☹ Low performance	☺ No disruption
Local + Remote	☺ Little overhead	☹ System-wide pause
Distributed Shared Space	**☺ Little overhead**	**☺ Minimal disruption**

S6: A Framework to Build Scalable NFs

Distributed Shared Space



→ Minimal performance overhead

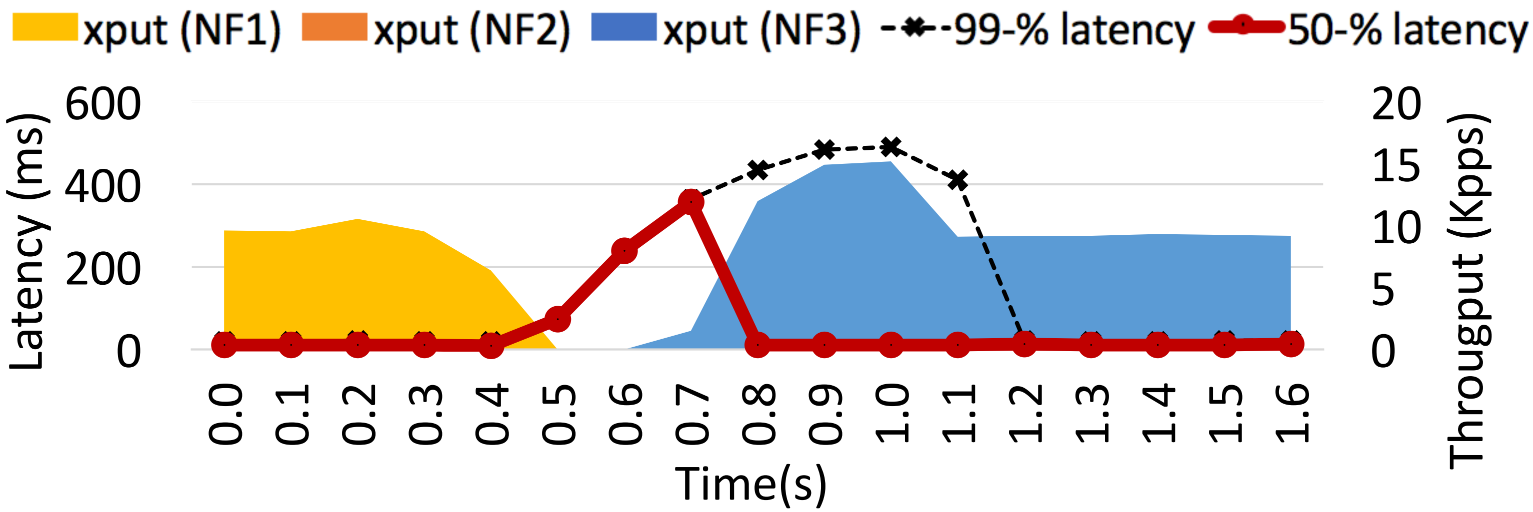
→ State sharing

→ No system-wide pausing during scaling events

S6 Scales Elastically and Gracefully

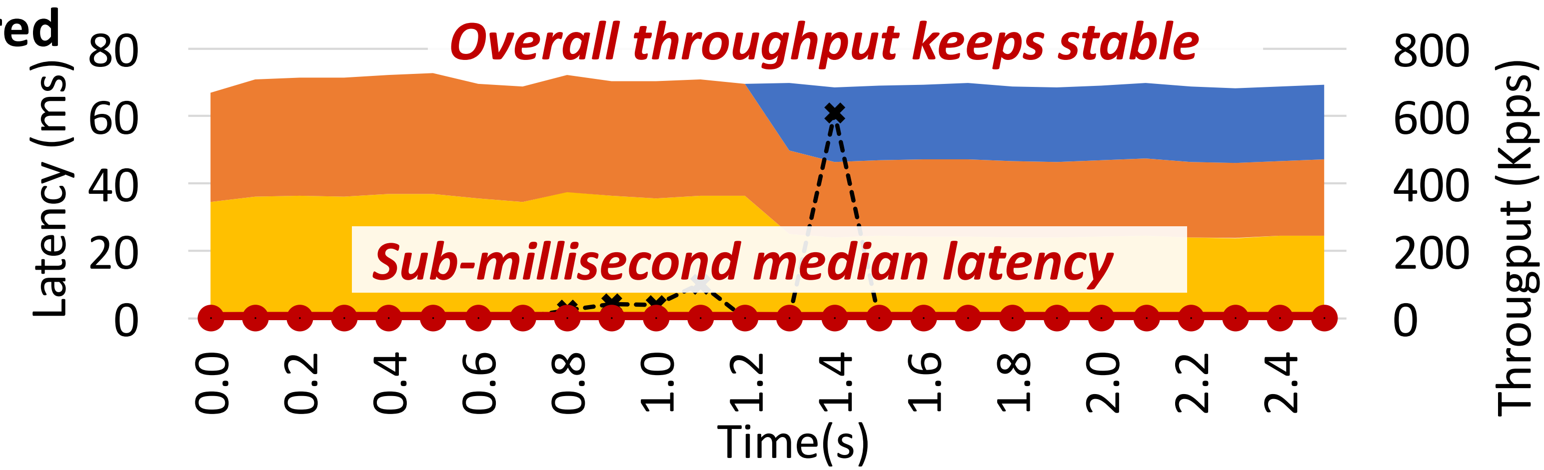
local+remote (OpenNF*)

10kpps, 1.5k flows



Distributed Shared (S6)

700kpps, 8k flows



Even with more extreme scenarios,

1000x ***higher workload (Mpps),*** ***1000x*** ***lower median latency***

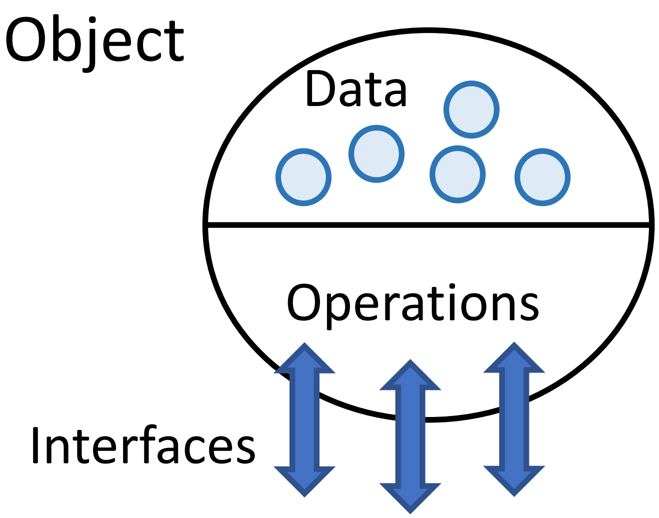
* *"OpenNF: Enabling Innovation in Network Function Control", SIGCOMM 2014*

S6: A Framework to Build Scalable NFs

1. **NF State Abstraction**
2. Elastic Scaling
3. S6 Programming models
4. Optimizations for minimizing remote access costs

Object for NF State Abstraction

Object encapsulation enables easy state management



✓ Integrity protection of state

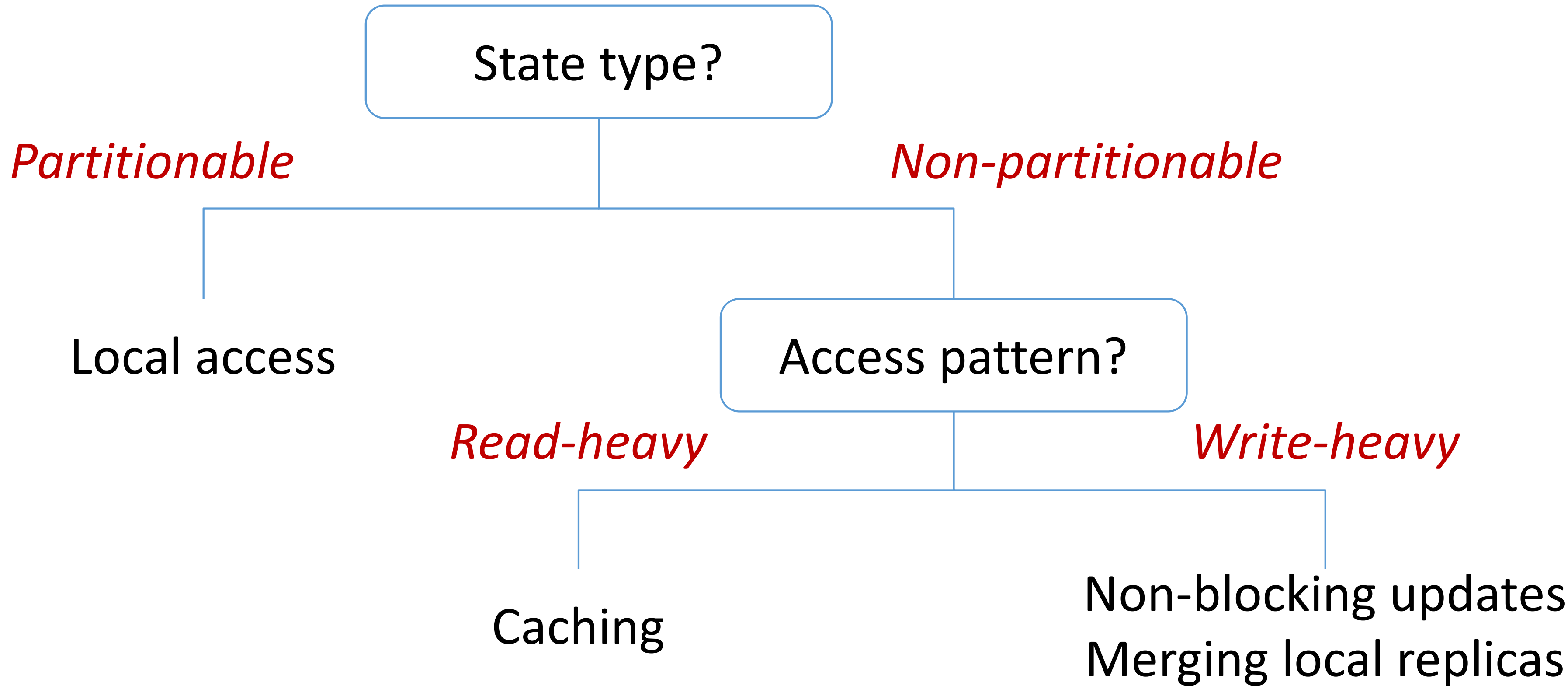
- Single writer vs. Multiple writer

✓ Optimization per object

- Performance vs. consistency:
  Different sweet spot per object

Optimization Strategies for NF State

Most NF state variables are covered by these strategies*



**From our survey on 8 popular network functions*

Examples of Optimization for NF state

function shipping for updating from multiple instances
c.f., SingleWriter

```
class Counter : public MultiWriter {
    private:
        uint32_t counter;

    public:
        uint32_t int_and_get();
        void inc(uint32_t x) untether;
        uint32_t get() const stale;
};
```

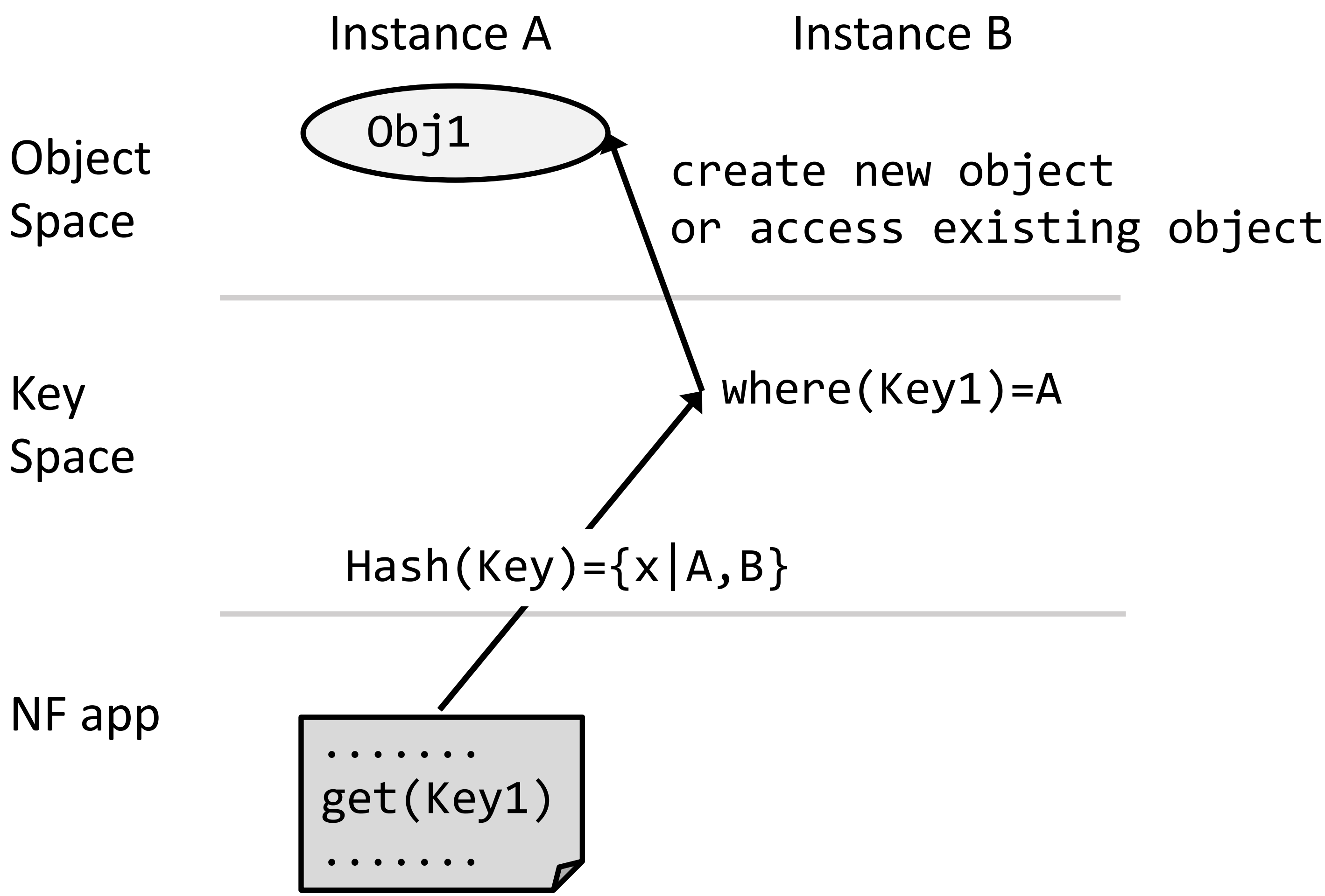
non-blocking update (← `untether`)

return from cache (← `stale`)

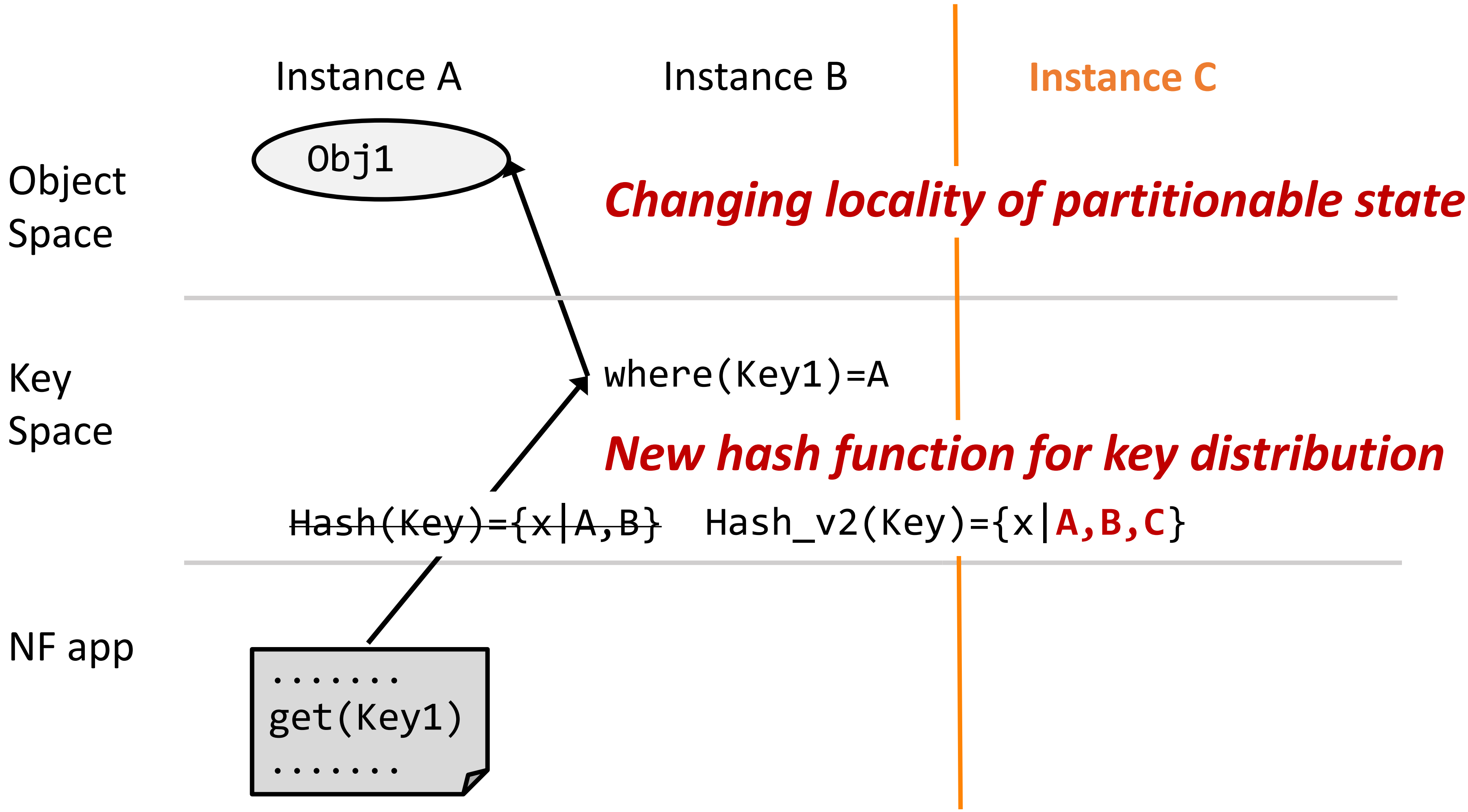
S6: A Framework to Build Scalable NFs

1. NF State Abstraction
2. **Elastic Scaling**
3. S6 Programming models
4. Optimizations for minimizing remote access costs

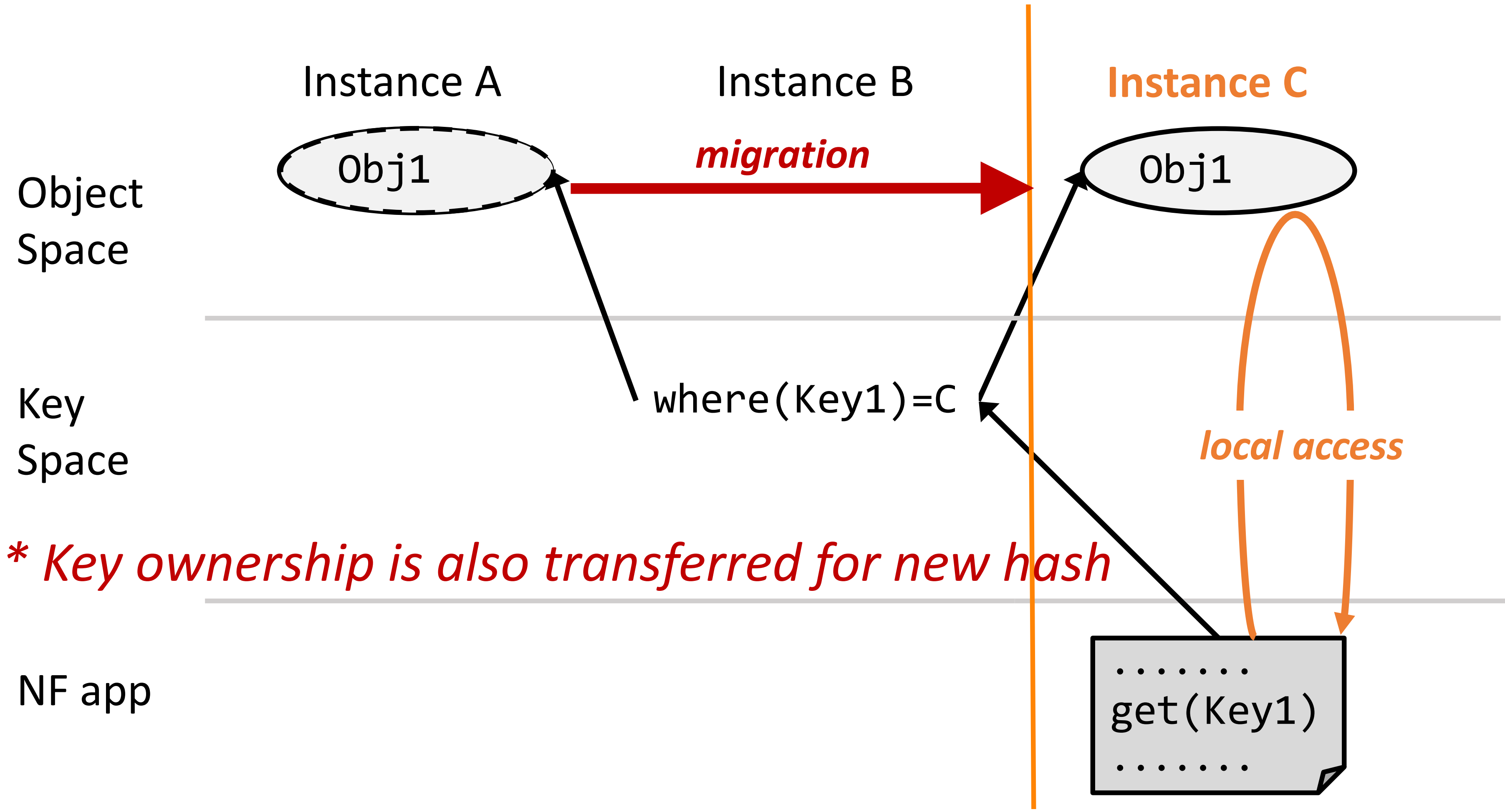
S6 Shared Object Space Architecture



Elastic Scaling Requires Space Reorganizing



State Migration for Locality

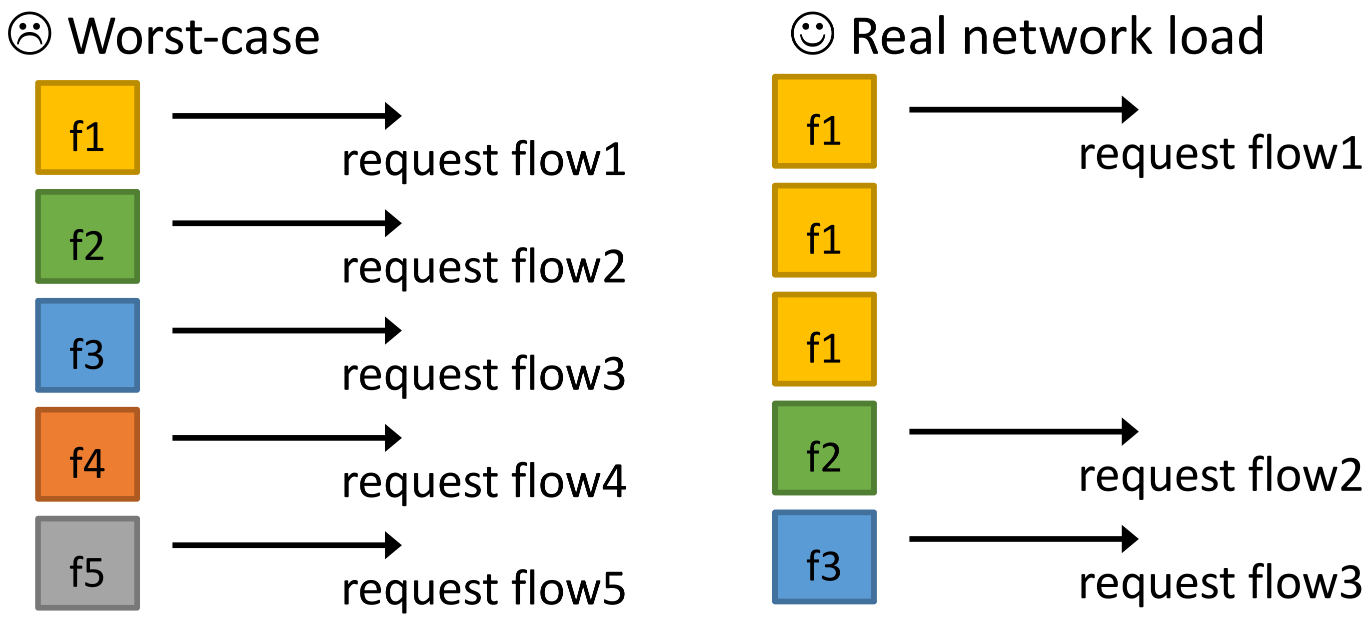


** Key ownership is also transferred for new hash*

When scaling-out, does bursty state migration degrade performance?

State Migration Happens Gradually Behind

- Flow state doesn’t need to be migrated at once
  - Packets in the same flow come in bursts
  - Long inter-arrival time between packet chunks in the same flow



- Micro-threading: Keep processing even with unavoidable blocking remote access

S6: A Framework to Build Scalable NFs

1. NF State Abstraction
2. Elastic Scaling
3. **S6 Programming models**
4. **Optimizations for minimizing remote access costs**

More details in the paper

Implementation

- S6 Compiler
  - Compiles S6 C++ extension into plain C++ code
  - Generates S6 object wrappers (stub, skeleton)
  - Uses clang 3.6 library
- S6 Runtime
  - Built in 12K lines of C++ code
  - Uses boost co-routine for micro-threads
- Applications
  - PRADS: a Passive Real-time Asset Detection System
  - Snort: Intrusion Detection System
  - NAT

Applications

- PRADS
  - a Passive Real-time Asset Detection System
  - allows to access real-time network monitoring results
    - protocols, services, and devices
- Snort
  - Intrusion Detection System
  - We port logic to detect malicious packets

State	Size (B)	Update	Access Frequency
Flow	160	Exclusive	Per-packet RW
Statistics	208	Concurrent	Per-packet RW
Asset	112 + 64n	Concurrent	Rarely R Per-packet W
Config	1.16Mi	Exclusive	Per-packet R Rarely W
Flow hashtable	40n	Concurrent	Per-packet RW
Asset hashtable	32n	Concurrent	Per-packet RW

State	Size (B)	Update	Access Frequency
Flow	160~32Ki	Exclusive	Per-packet RW
Whitelist	12 + 28n	Exclusive	Per-packet RW
Malicious	12 + 28n	Concurrent	Per-packet RW
Config	1.43 Mi	Exclusive	Per-packet R Rarely W
Maclicious hashtable	32n	Concurrent	Per-packet RW
Whitelisth ashtable	32n	Concurrent	Per-packet RW

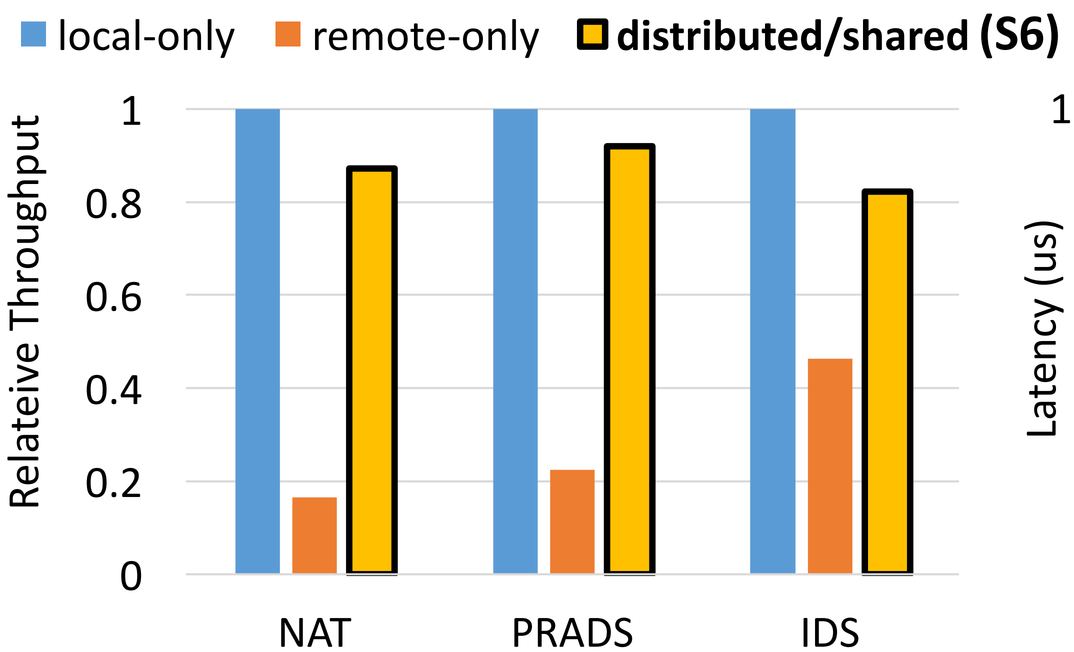
Evaluation

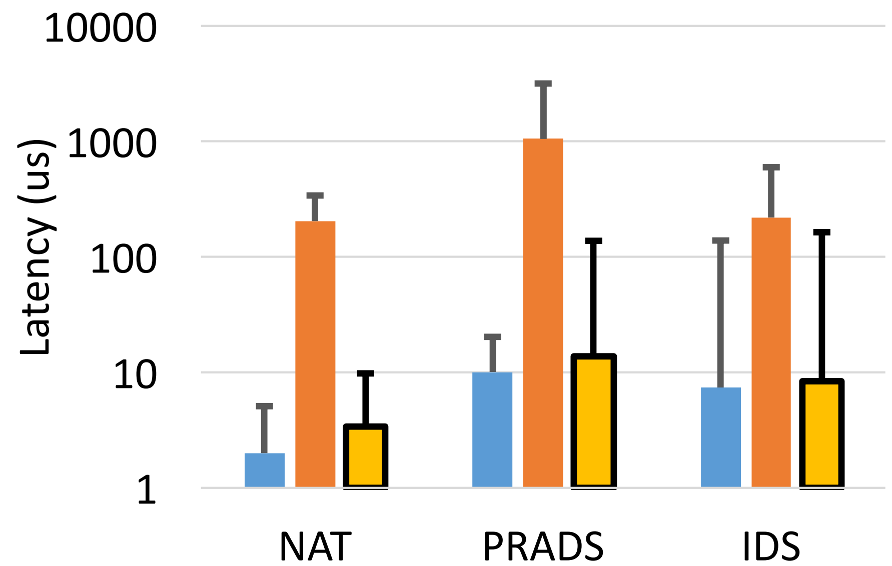
- Scaling experiments
  - Use Amazon EC2 instance as NF instances (Docket container)
  - C4.xlarge, 4 cores @ 2.90 GHz

- Workloads: Synthetic TCP traffic
  - Empirical flow distribution in size and arrival rate

S6 Performance During Normal Phase

Keys are evenly distributed through 2 instances
→ Half of the first state accesses are remote



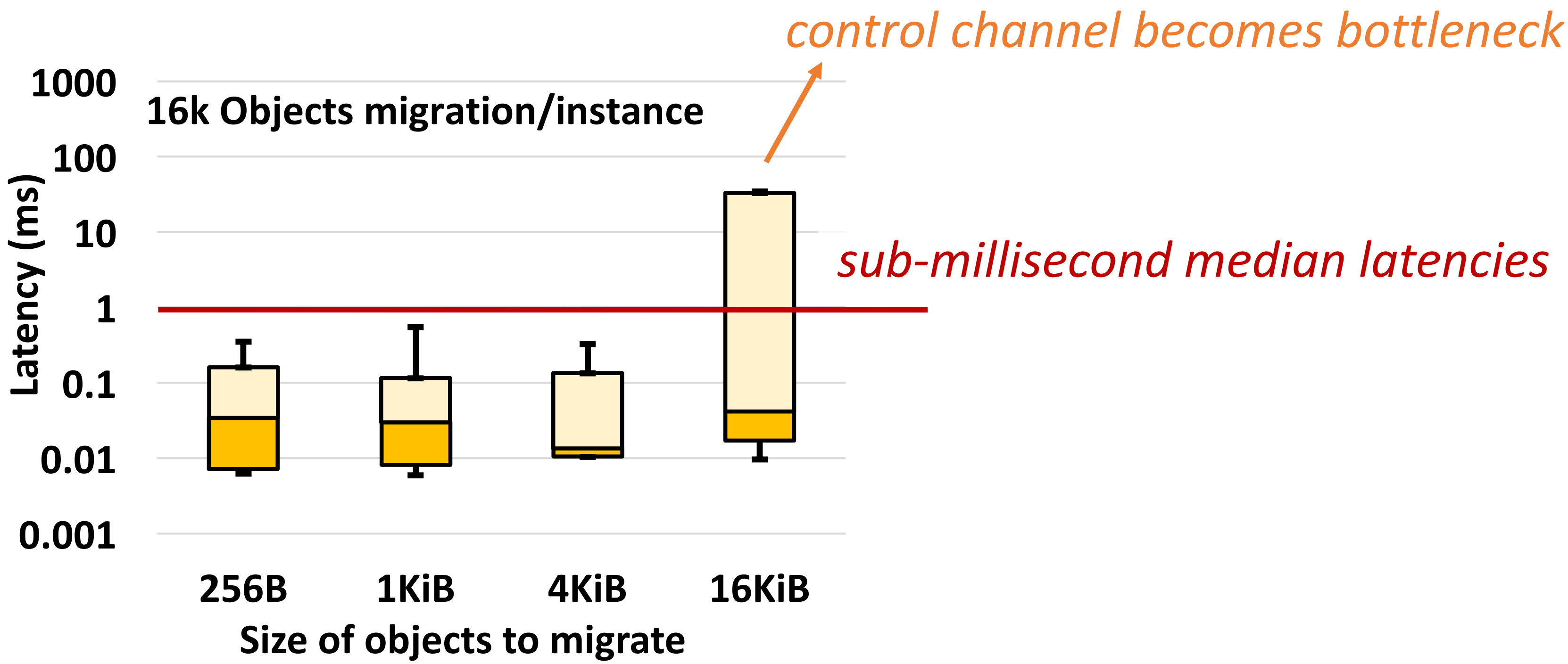


☺ S6 preserves 80 ~ 95% throughput from local-only

☺ S6 keeps similar median latency from local-only

Space Reorganization Overhead during Scale-out

- Latency distribution of scale-out
  - Scale-out from 1 to 2 instances (1Mpps → 0.5Mpps * 2)



S6 shows minimal performance overhead when scaling-out

Conclusion

S6: A framework to build scalable NFs

- Allows NF state to be *shared/distributed/migrated* across instances
- Achieves high performance with:
  - State abstractions specifying state requirements
  - When scaling, gradual object migration and space reorganization

- Has minimal performance impact during *normal operations* as well as *scaling event*

- https://github.com/NetSys/S6