Advanced Computer Networks

Network Function Virtualization

Lin Wang

Period 2, Fall 2022

VU VRIJE UNIVERSITEIT AMSTERDAM

Course outline

Essentials

- Introduction (history, principles)
- Networking basics
- Network transport

Data center networking

- Data center networking
- Data center transport
- Software defined networking
- **Network function virtualization**
- Programmable data plane
- Programmable switch architecture

Network innovations

- In-network computing - applications
- In-network computing - hardware
- Network monitoring
- Machine learning for networking

Guest lecture

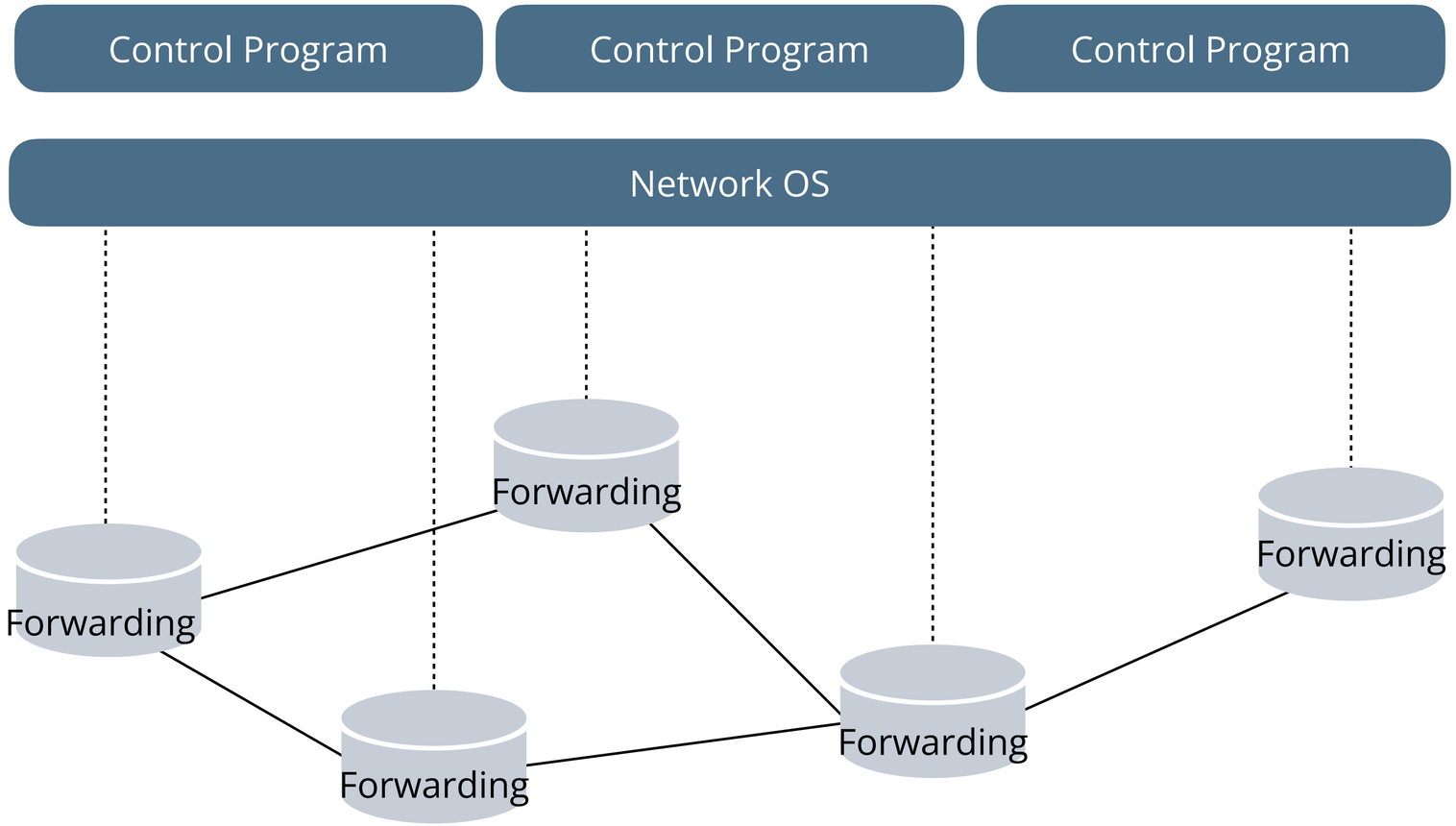
- Fernando Ramos (University of Lisbon)

Learning objectives

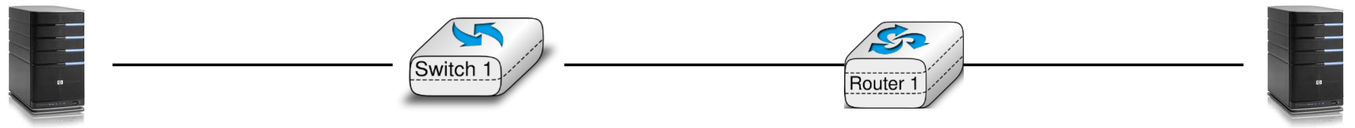
How to **simplify the programming** of software network functions?

How to achieve **high performance** for software network functions?

Network control plane



Network data plane



Application			Application
Transport			Transport
Network		Network	Network
Datalink	Datalink	Datalink	Datalink
Physical	Physical	Physical	Physical

Middleboxes

Essential to network operators

Security

- Firewalls, IDSes, traffic scrubbers, NATs

Traffic shaping

- Rate limiters, load balancers

Performance

- Traffic accelerators, caches, proxies


carrier-grade NAT


ad insertion


IDS


load balancer


session border controller


firewall


transcoder

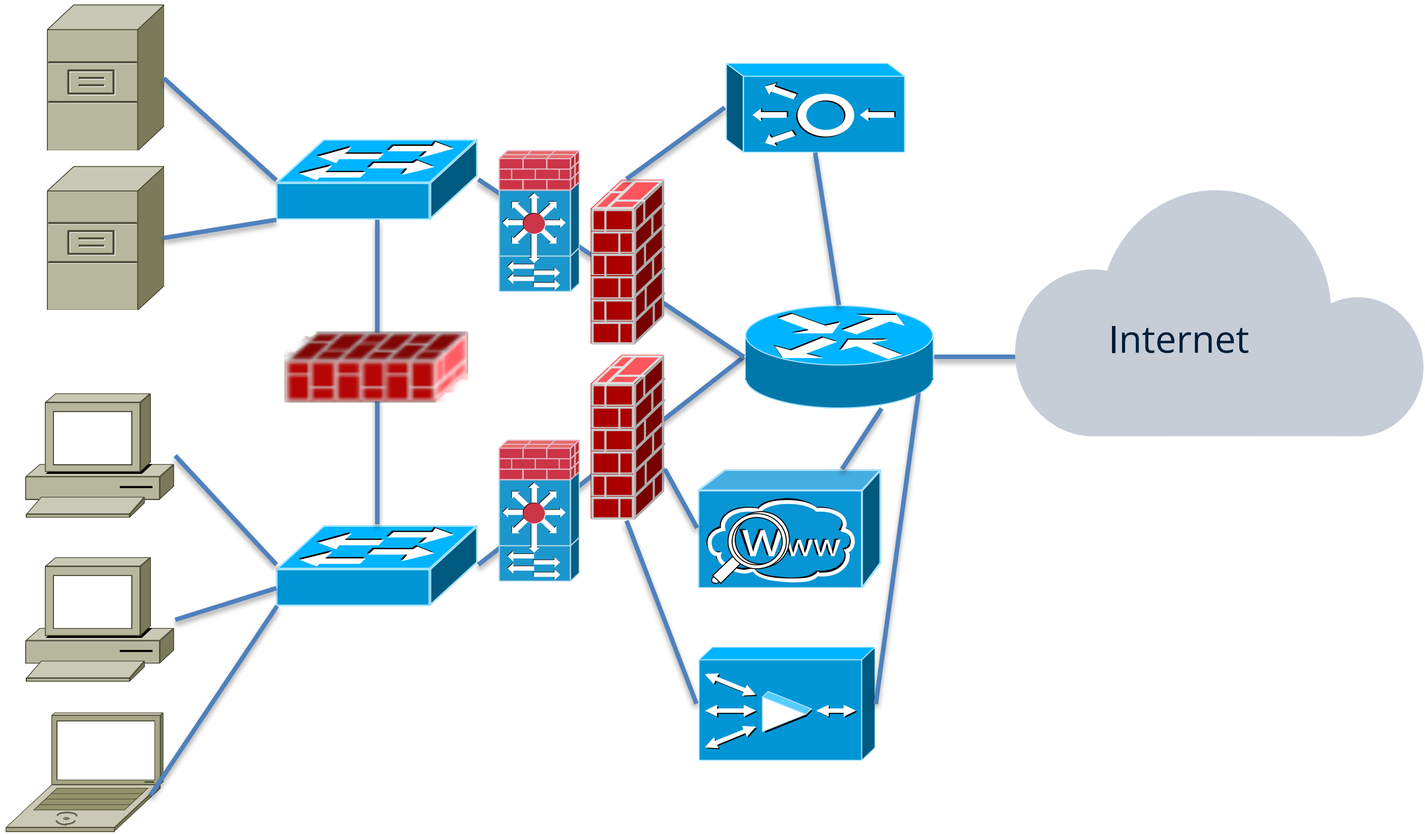

DDoS protection


DPI

QoE monitor


WAN accelerator

The real network is more complex than you think



Challenges with middleboxes

Buying and managing middlexboxes are **expensive** since these are proprietary

Deploying and managing middleboxes are **complex** in terms of wiring the chaining

Introducing new features is **slow**, requiring the purchase of new hardware with software

Not scalable upon demands shifts: devices need to be physically placed or removed

Vendor lock-in: devices are proprietary, raising innovation barriers

Alternative solutions and tradeoffs

Commodity server

High extensibility
Easy management
Low cost
Low performance

Specialized boxes

Low extensibility
Complex management
High cost
High performance

Alternative solutions and tradeoffs

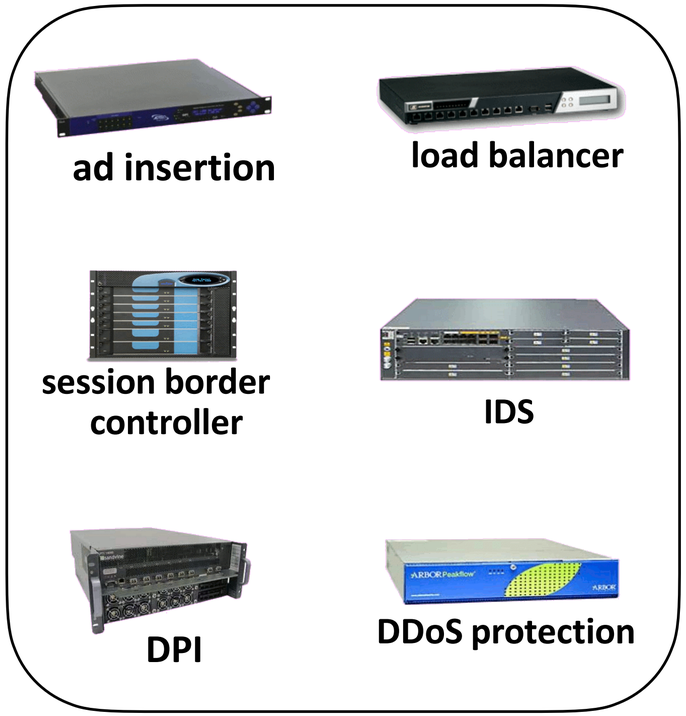
Commodity server

Specialized boxes

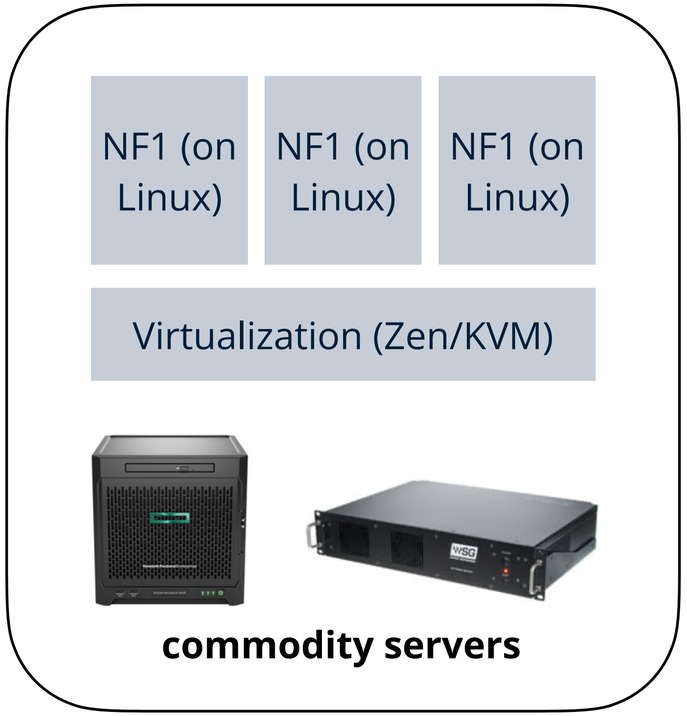
High extensibility
Easy management
Low cost
Low performance

Low extensibility
Complex management
High cost
High performance

Network function virtualization (NFV)



Current middlebox approach



Network function virtualization

Benefits with NFV

Hardware **sharing** across multiple tenants/network functions

Cost saving via reduced hardware/power costs through dynamic scaling

More **flexible management** using software

Safe to try **new features** on an operational network/platform

How to simplify the programming of software network functions?

Network functions are complex to implement

An IP router involves at least the following components

Packet classification

ARP handling

IP header operations

IP fragmentation

IP address lookup

ICMP processing

Do you still remember the fundamental principle for managing complexity?

Managing complexity with modularity

Machine languages: no abstractions

- Hard to deal with low-level details
- Mastering complexity is crucial

High-level languages: operating systems and other abstractions

- File systems, virtual memory, abstract data types...

Modern languages: even more abstractions

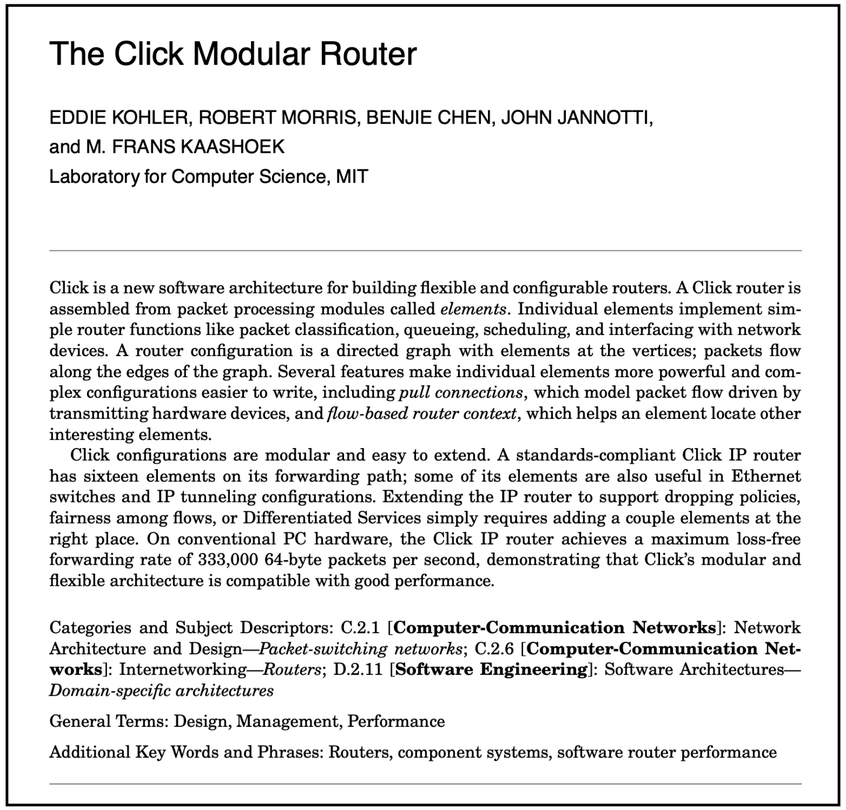
- Object oriented, garbage collection...

"Modularity based on abstractions is the way things get done!"

Barbara Liskov
(MIT, ACM Turing Award 2008, pioneer in programming languages, operating systems, distributed computing)

Click

A modular architecture for programming network functions



The Click Modular Router

EDDIE KOHLER, ROBERT MORRIS, BENJIE CHEN, JOHN JANNOTTI,
and M. FRANS KAASHOEK
Laboratory for Computer Science, MIT

Click is a new software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules called *elements*. Individual elements implement simple router functions like packet classification, queueing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Several features make individual elements more powerful and complex configurations easier to write, including *pull connections*, which model packet flow driven by transmitting hardware devices, and *flow-based router context*, which helps an element locate other interesting elements.

Click configurations are modular and easy to extend. A standards-compliant Click IP router has sixteen elements on its forwarding path; some of its elements are also useful in Ethernet switches and IP tunneling configurations. Extending the IP router to support dropping policies, fairness among flows, or Differentiated Services simply requires adding a couple elements at the right place. On conventional PC hardware, the Click IP router achieves a maximum loss-free forwarding rate of 333,000 64-byte packets per second, demonstrating that Click's modular and flexible architecture is compatible with good performance.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

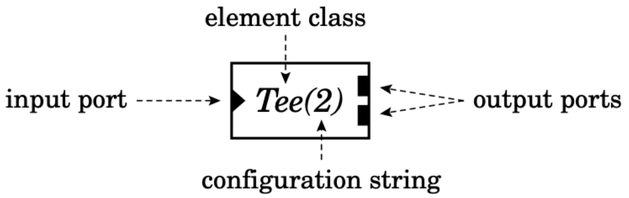
General Terms: Design, Management, Performance

Additional Key Words and Phrases: Routers, component systems, software router performance

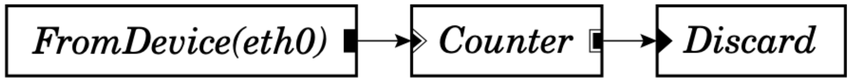
ACM SOSP 1999

Basic concepts

Element (a unit of packet processing, small in the amount of computation)

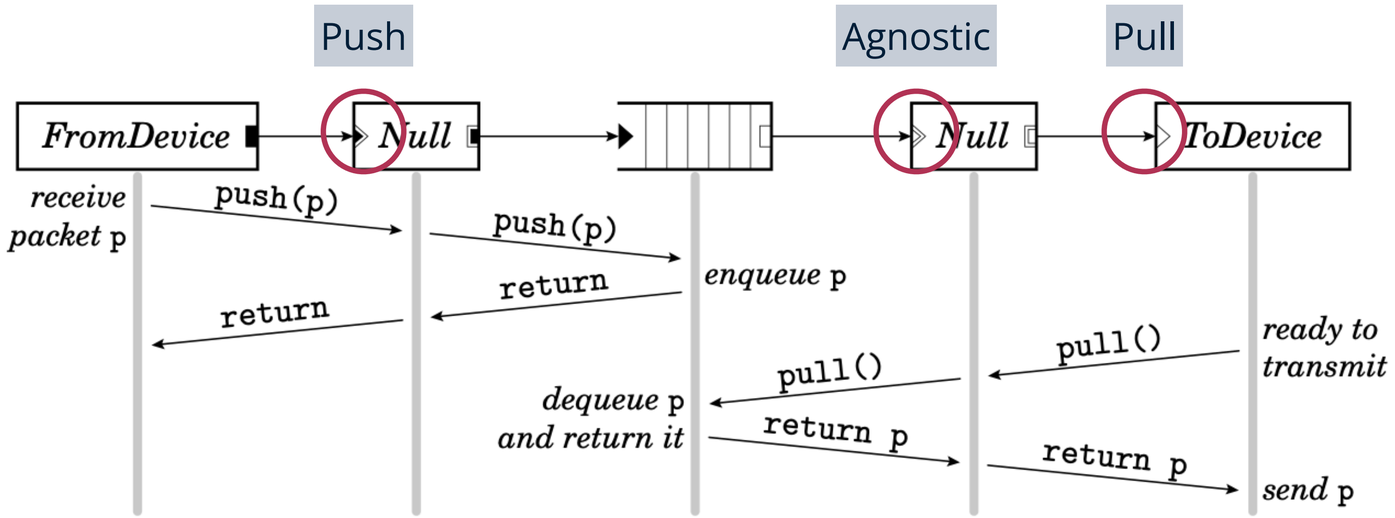


Connection (possible path for packet transfer) and **configuration**



Connection

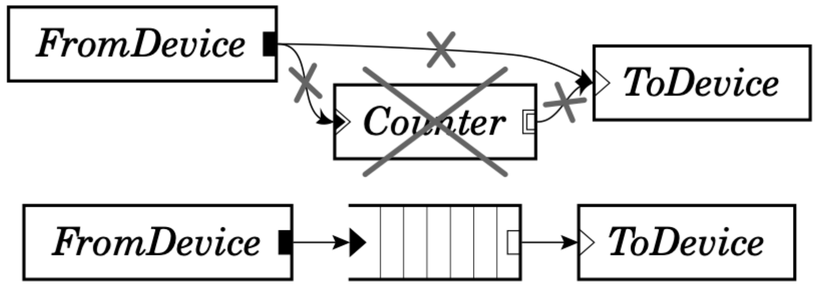
Three types of ports: push, pull, and agnostic



Push connections push packets to the downstream element (appropriate for handling unsolicited packet arrivals)
pull connections pull packets from upstream elements (appropriate for controlling the timing of packet processing)
Agnostic ports are decided when they are connected to the other side.

Connection

Push/pull connection constraints



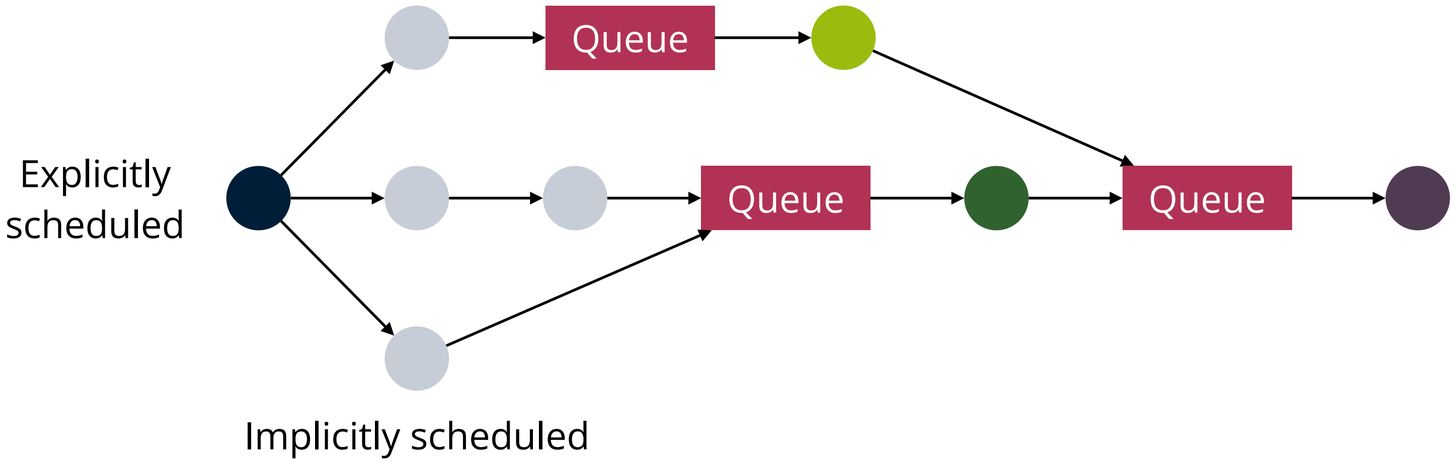
Rule 1: A connection must have the same type on the two ends

Rule 2: A port must have only one connection

Rule 3: An element must not have different types on its input/output ports if the element processes packets immediately (counter example: Queue element)

Click CPU scheduling

Single-thread scheduling, scheduling unit: an element

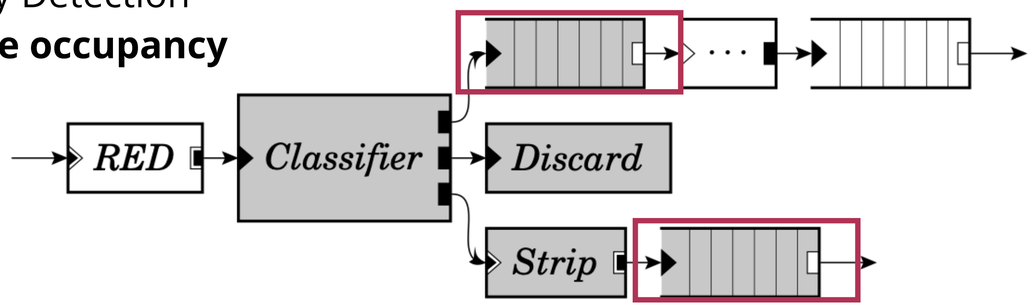


Method sharing across elements

Use **flow-based router context** to discover an element B for using B's method interface

Random Early Detection
based on **queue occupancy**

Queues found by the RED, whose queue_len()
method may be used by RED

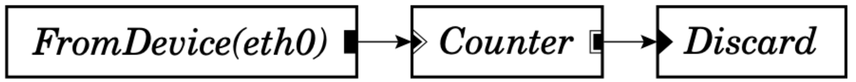


During configuration initialization, the system issues questions like:
"If I were to emit a packet and it stops at the first queue, what are
the elements the packet might traverse?"

Solved by a data-flow algorithm during the initialization

Click language

The interface for programmers to specify elements, connections, and configurations




```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// ... and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

The language is simple and wholly declarative: it specifies what elements to create and how they should be connected, not how to process packets procedurally.

Click language

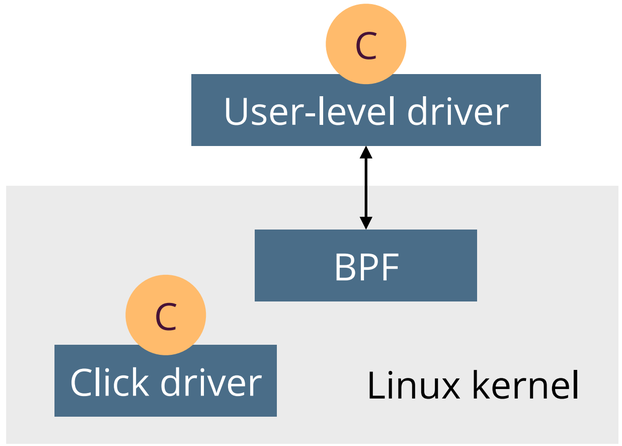
Implementing a Click element

```
class NullElement: public Element { public:
  NullElement()                          { add_input(); add_output(); }
  const char *class_name() const         { return "Null"; }
  NullElement *clone() const             { return new NullElement; }
  const char *processing() const         { return AGNOSTIC; }
  void push(int port, Packet *p)         { output(0).push(p); }
  Packet *pull(int port)                 { return input(0).pull(); }
};
```

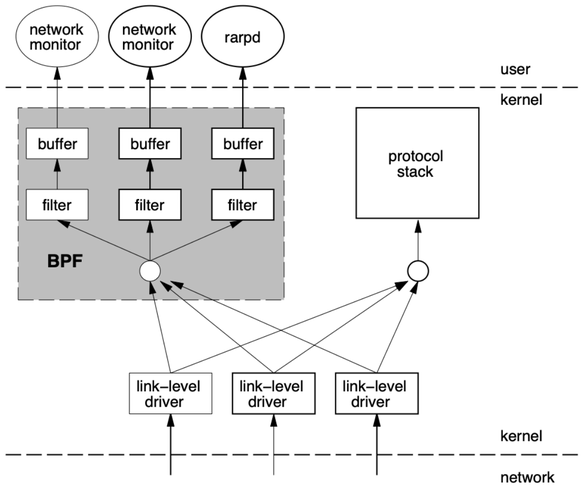
The complete implementation of a do-nothing element: Null passes packets from its single input to its single output unchanged.

Click implementation

How does a Click router get installed on Linux?



Two Click drivers implemented



Installing a new configuration typically destroys an old configuration (e.g., packets in queues are dropped).

Each element can install a number of **handlers** which are access points for user-interactive configurations. They appear to users a files in Linux's **/proc** file system. (Think about the control plane.)

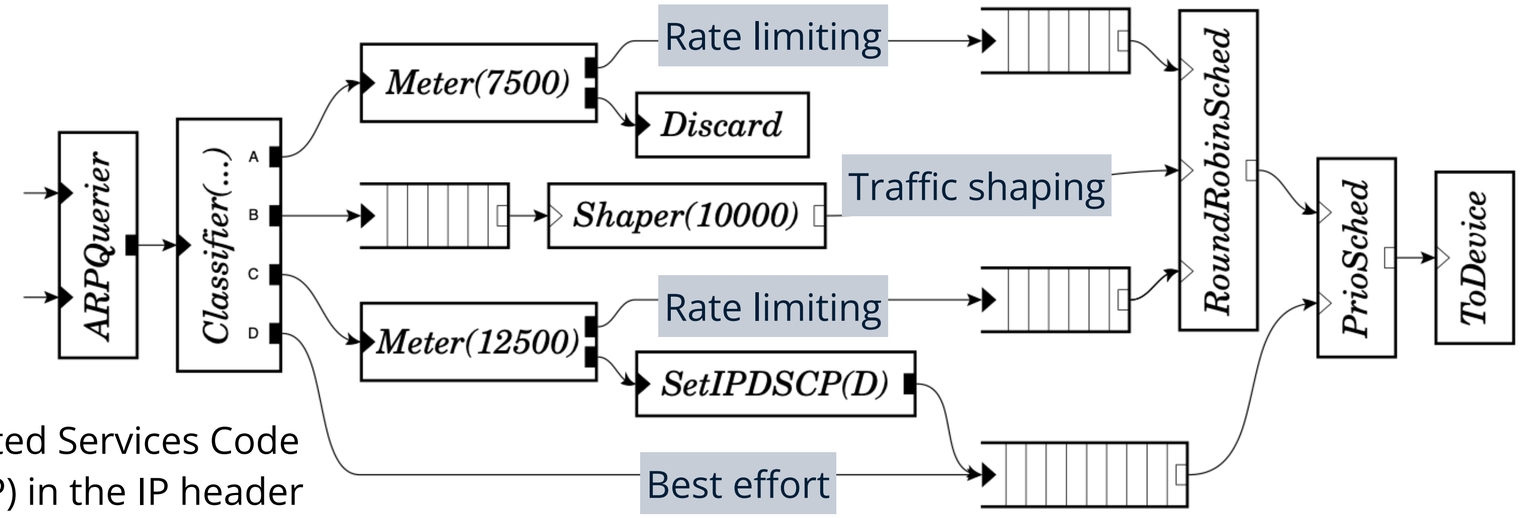
A user can write a new configuration file and install it with a **hot-swapping** option that allows the old configuration to run before the new configuration is in place.

https://github.com/kohler/click

https://ebpf.io

Click example

Traffic conditioning



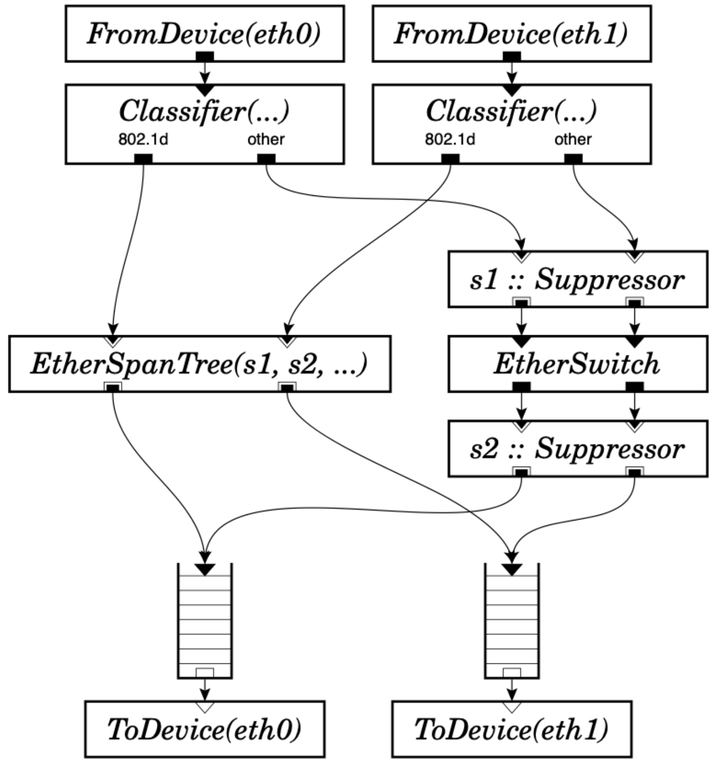
Differentiated Services Code Point (DSCP) in the IP header

A diffserv packet conditioning block

Click example

Ethernet switch

Calculating the spanning tree

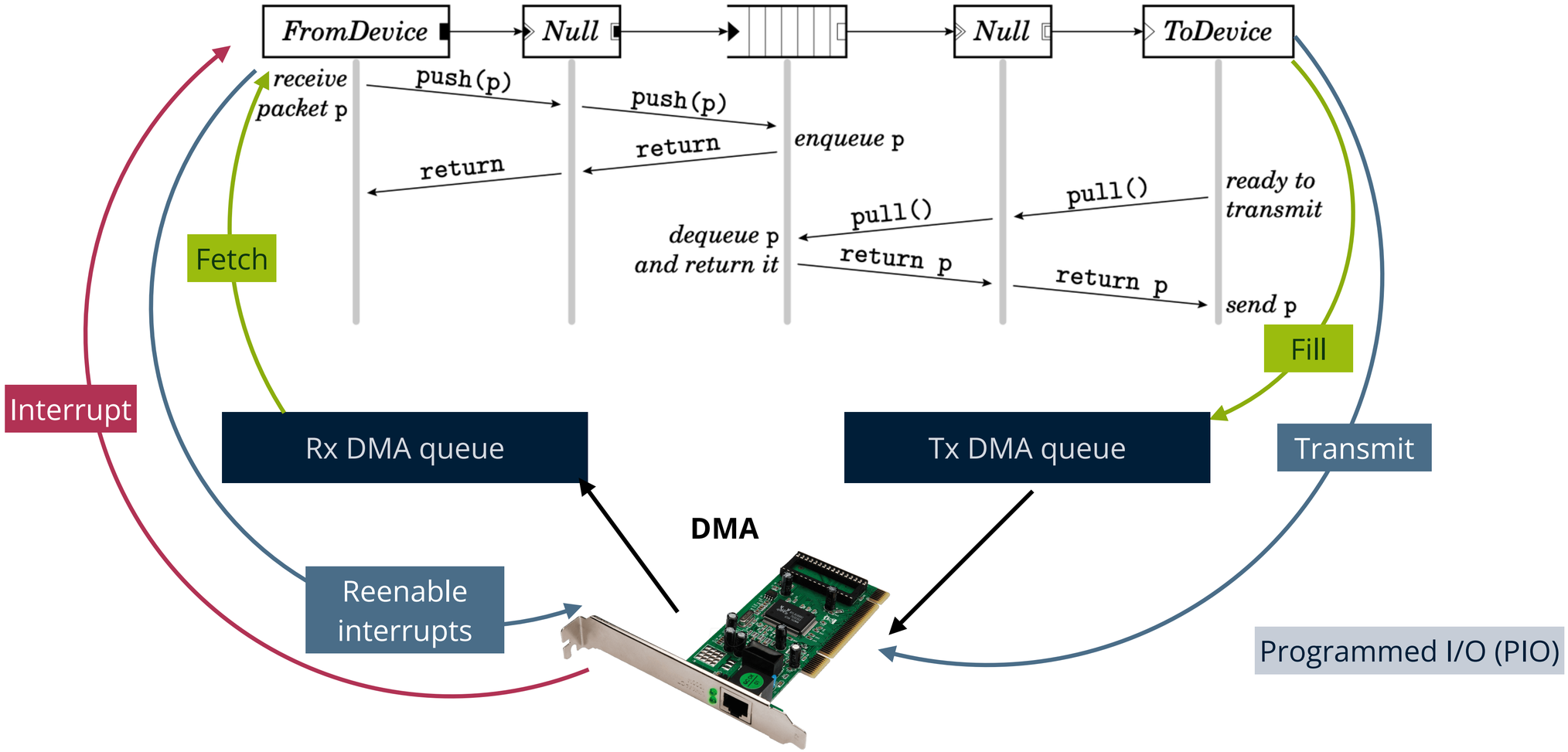


Enforcing the spanning tree

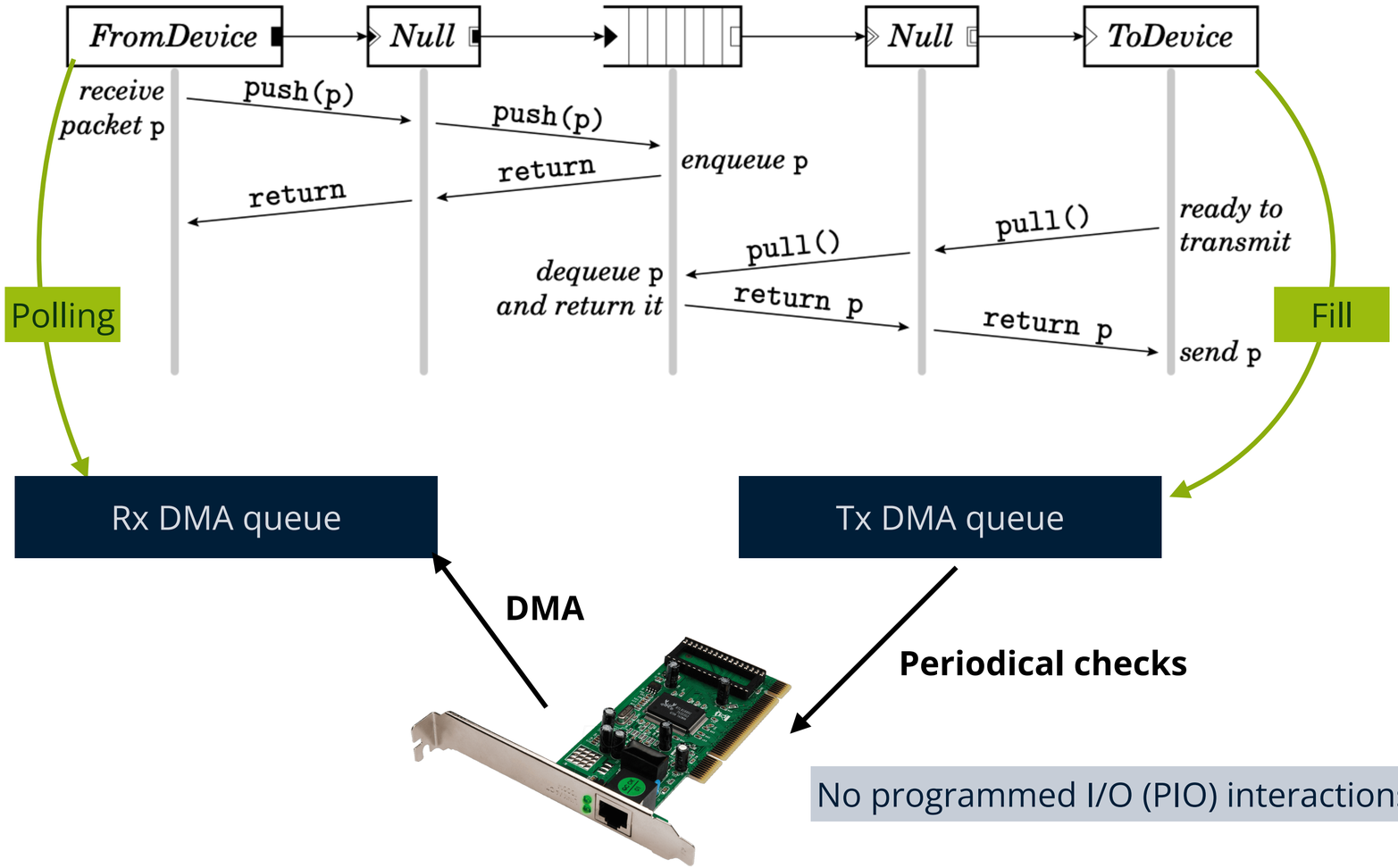
The suppressors cannot be found by the EtherSpanTree element with flow-based router context, so they must be manually specified by users.

Click performance issues

Interrupts and programmed I/Os are expensive



Click performance optimization: polling instead of interrupts



No programmed I/O (PIO) interactions with the NIC

Click performance optimizations

Fast Userspace Packet Processing

Tom Barbette
University of Liege
Belgium
tom.barbette@ulg.ac.be

Cyril Soldani
University of Liege
Belgium
cyril.soldani@ulg.ac.be

Laurent Mathy
University of Liege
Belgium
laurent.mathy@ulg.ac.be

ABSTRACT

In recent years, we have witnessed the emergence of high speed packet I/O frameworks, bringing unprecedented network performance to userspace. Using the Click modular router, we first review and quantitatively compare several such packet I/O frameworks, showing their superiority to kernel-based forwarding.

We then reconsider the issue of software packet processing, in the context of modern commodity hardware with hardware multi-queues, multi-core processors and non-uniform memory access. Through a combination of existing techniques and improvements of our own, we derive modern general principles for the design of software packet processors.

Our implementation of a fast packet processor framework, integrating a faster Click with both Netmap and DPDK, exhibits up-to about 2.3x speed-up compared to other software implementations, when used as an IP router.

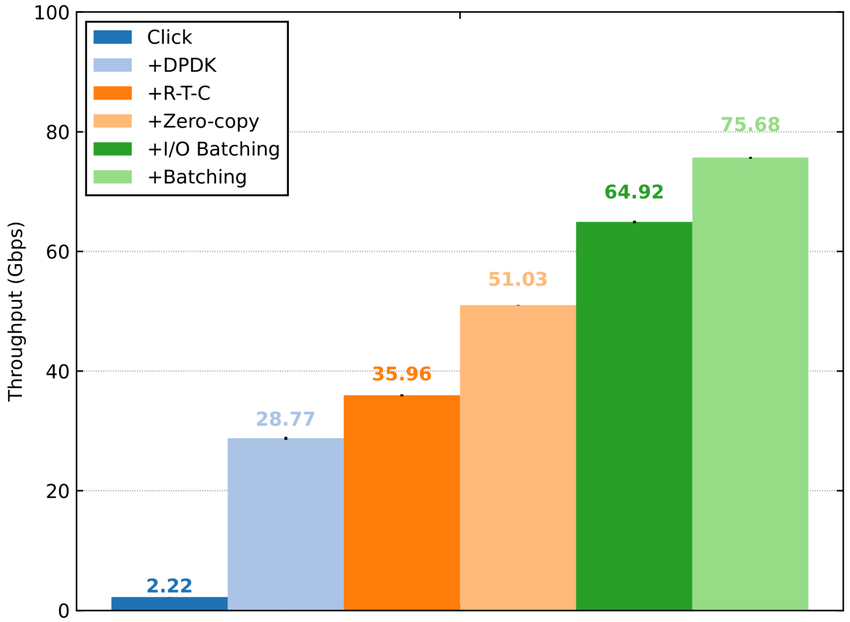
think its abstraction may lend itself well to network stack specialization (even if it's mostly router-oriented for now).

Multiple authors proposed enhancements to the Click Modular Router. RouteBricks [6] focuses on exploiting parallelism and was one of the first to use the multi-queue support of recent NICs for that purpose. However, it only supports the in-kernel version of Click. DoubleClick [11] focuses on batching to improve overall performances of Click with PacketShader I/O [8]. SNAP [23] also proposed a general framework to build GPU-accelerated software network applications around Click. Their approach is not limited to linear paths and is complementary to the others, all providing mostly batching and multi-queuing. All these works provide useful tips and improvements for enhancing Click, and more generally building an application on top of a "raw packets" interface.

The first part of our contribution is the critical analysis of those enhancements, and discuss how they interact with

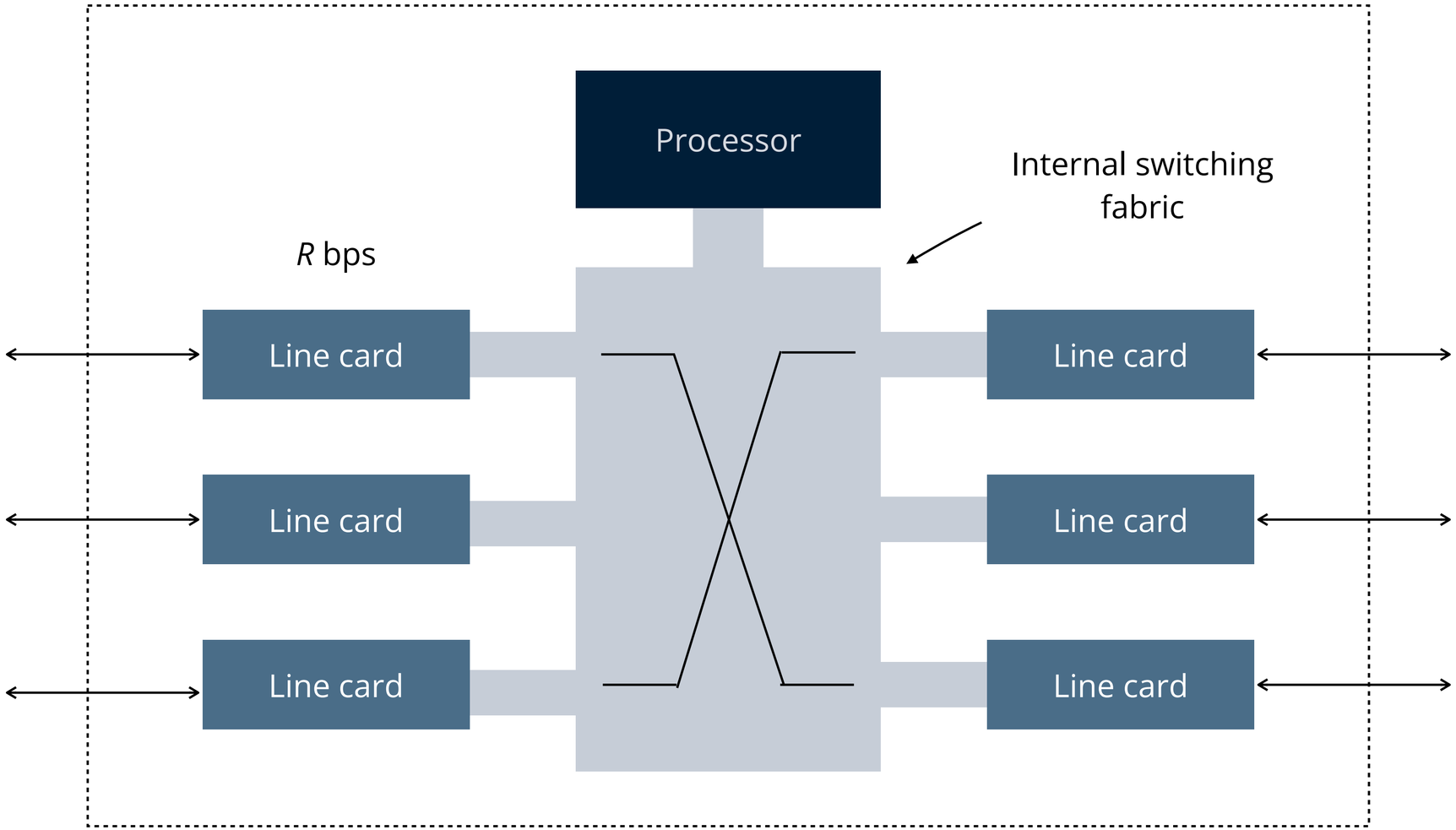
ANCS 2015

Full Push execution model, multi-queueing, compute batching

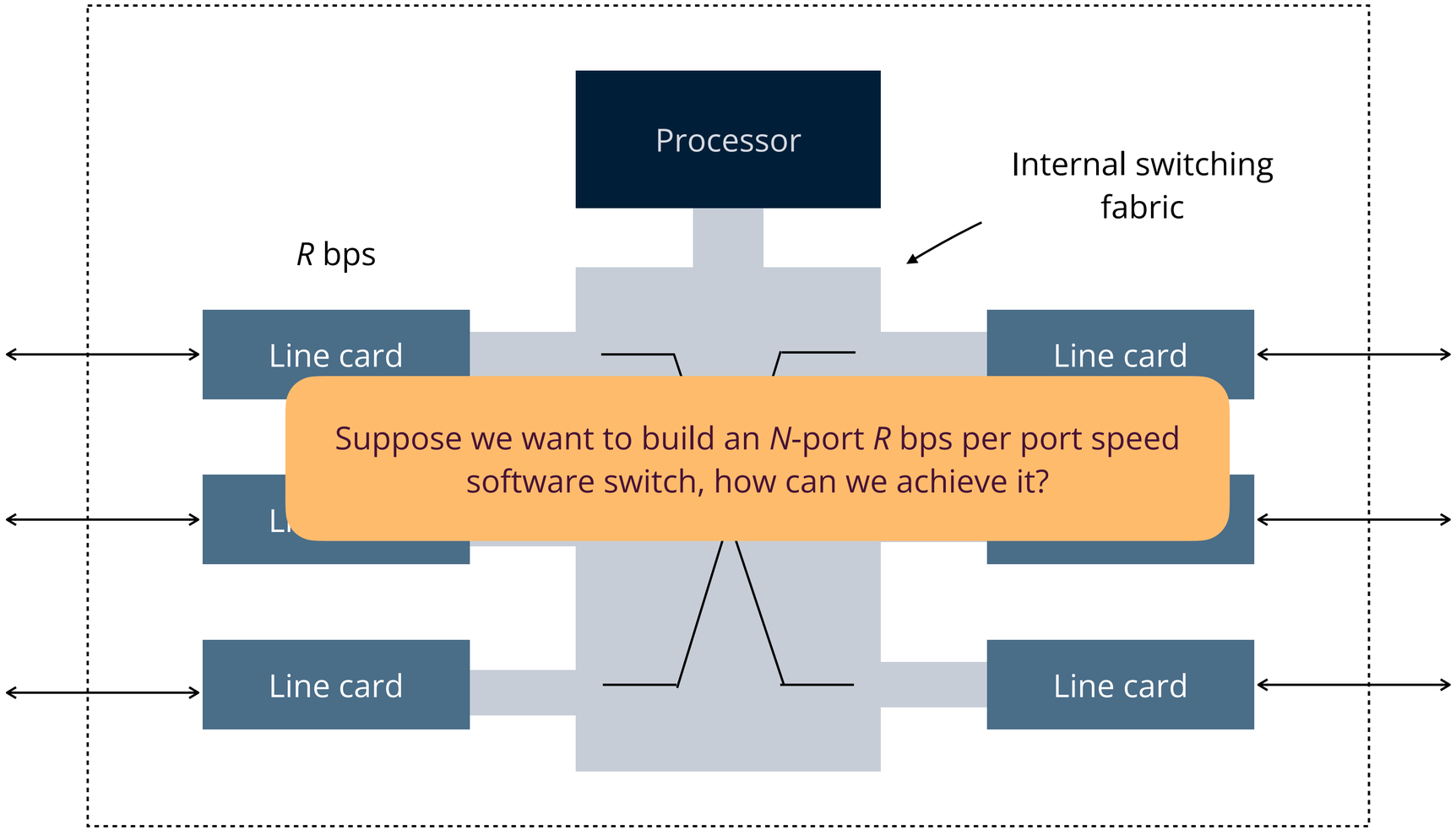


How to achieve high performance with virtualized network functions in software?

Traditional router architecture



Traditional router architecture



The case with a single server

R = 1, 1.25, 10 Gbps

N =10s - 1000s

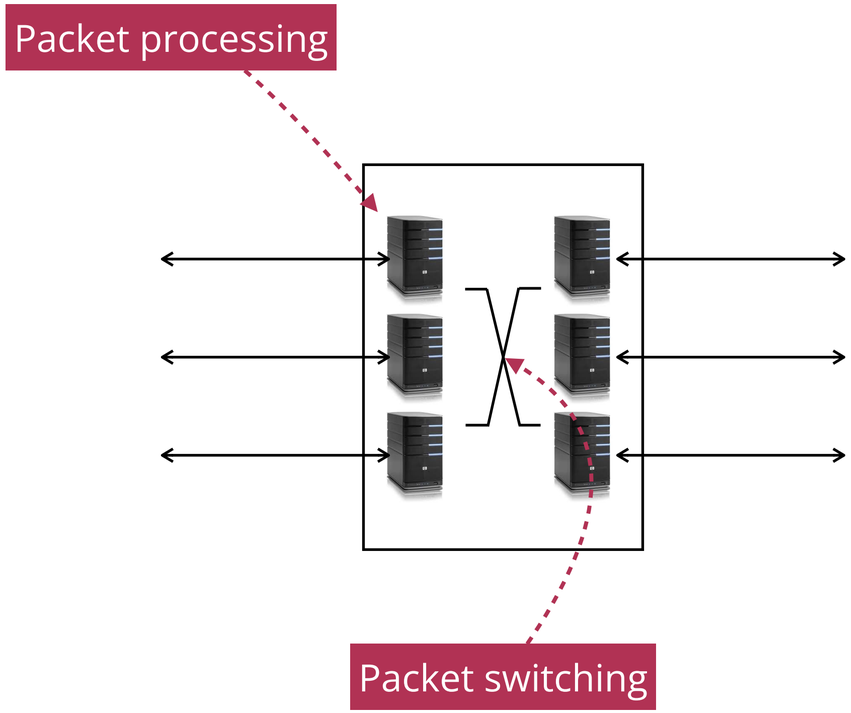
$N \times R$ bps

(required: up to few Tbps, achievable: 1-5Gbps)

What are possible approaches for scaling up?

Scaling up with parallelism

Two basic functionalities: packet processing (classification, lookup), and packet switching

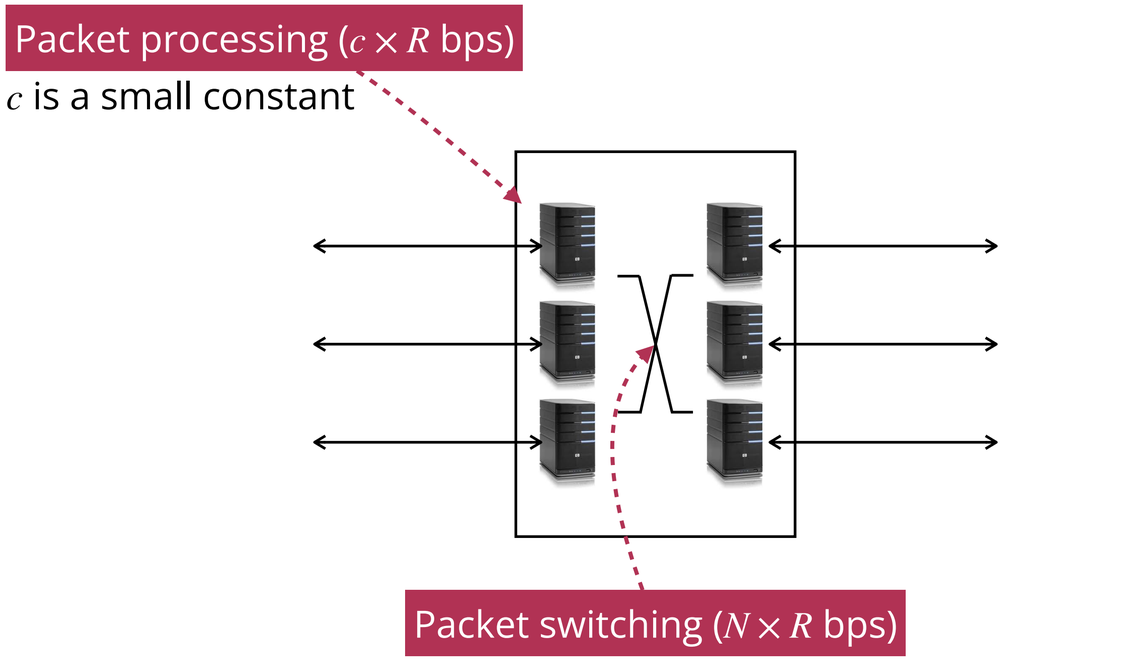


Scaling up with parallelism

Two basic functionalities: packet processing (classification, lookup), and packet switching

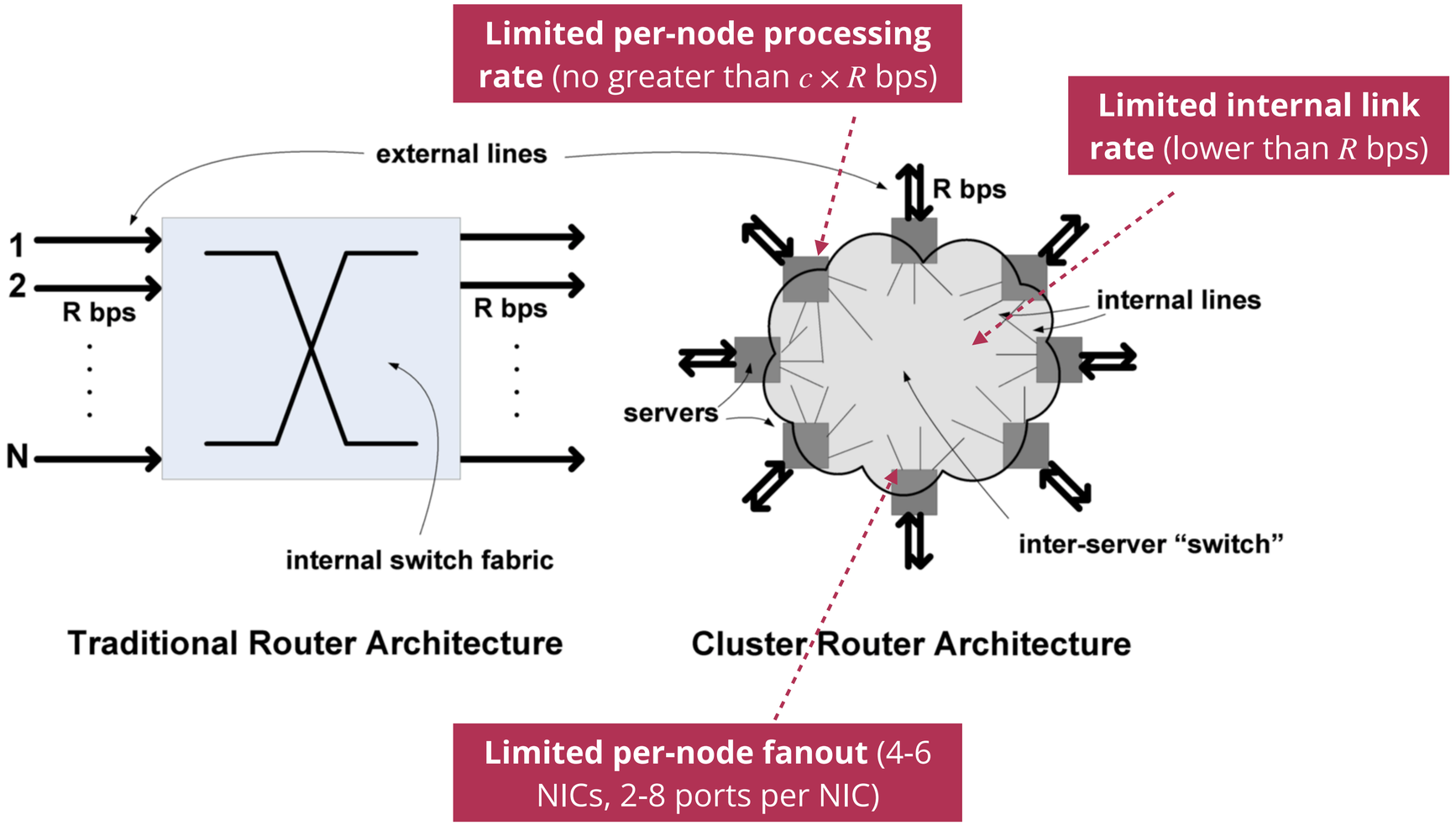
R = 1, 1.25, 10 Gbps

N =10s - 1000s



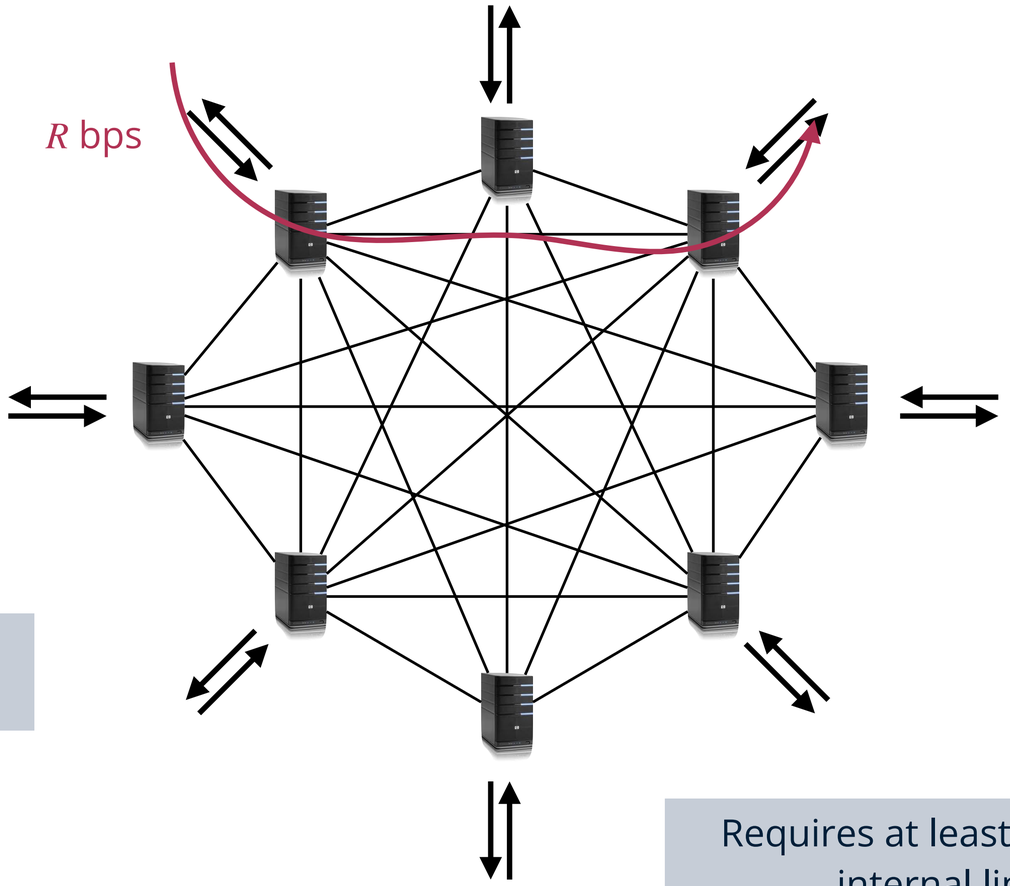
Require distributed solutions to scale

A software router built from a cluster of servers



Routing with a single path

Forward the packet directly from input server to output server

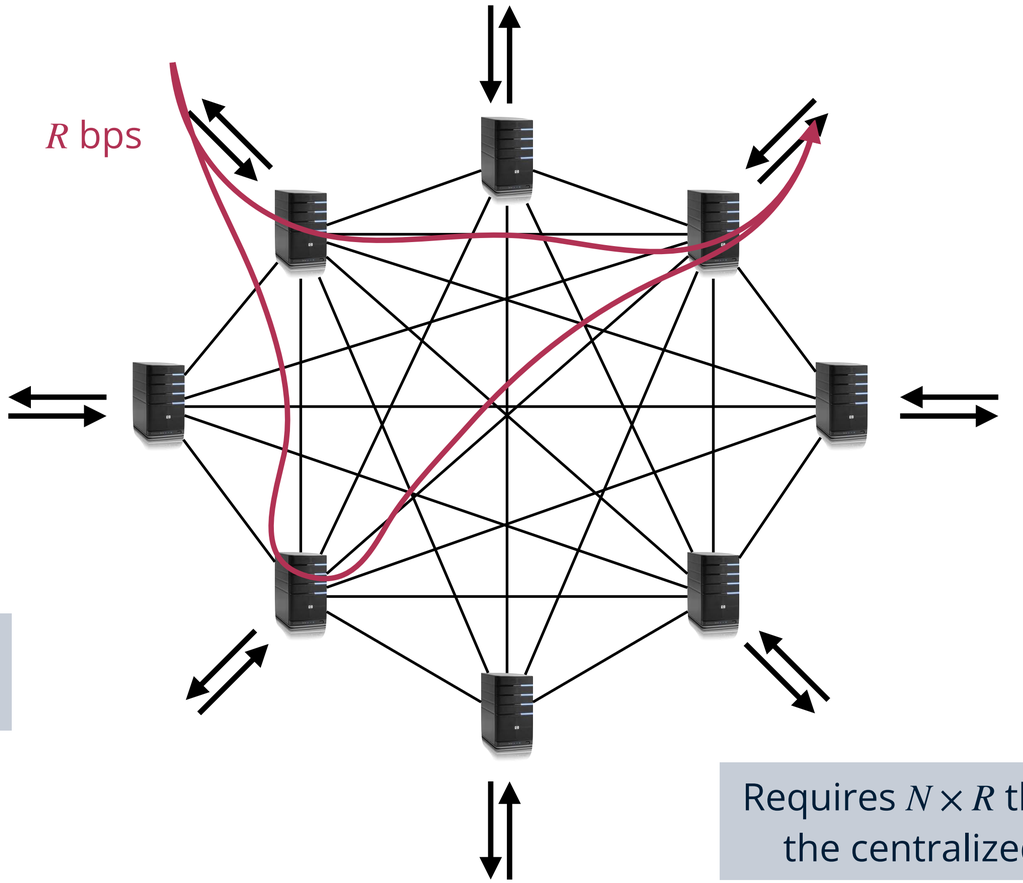


Simple, no packet reordering

Requires at least R bps for internal links

Routing with dynamic single paths

Dynamically allocate a path for a source-destination pair

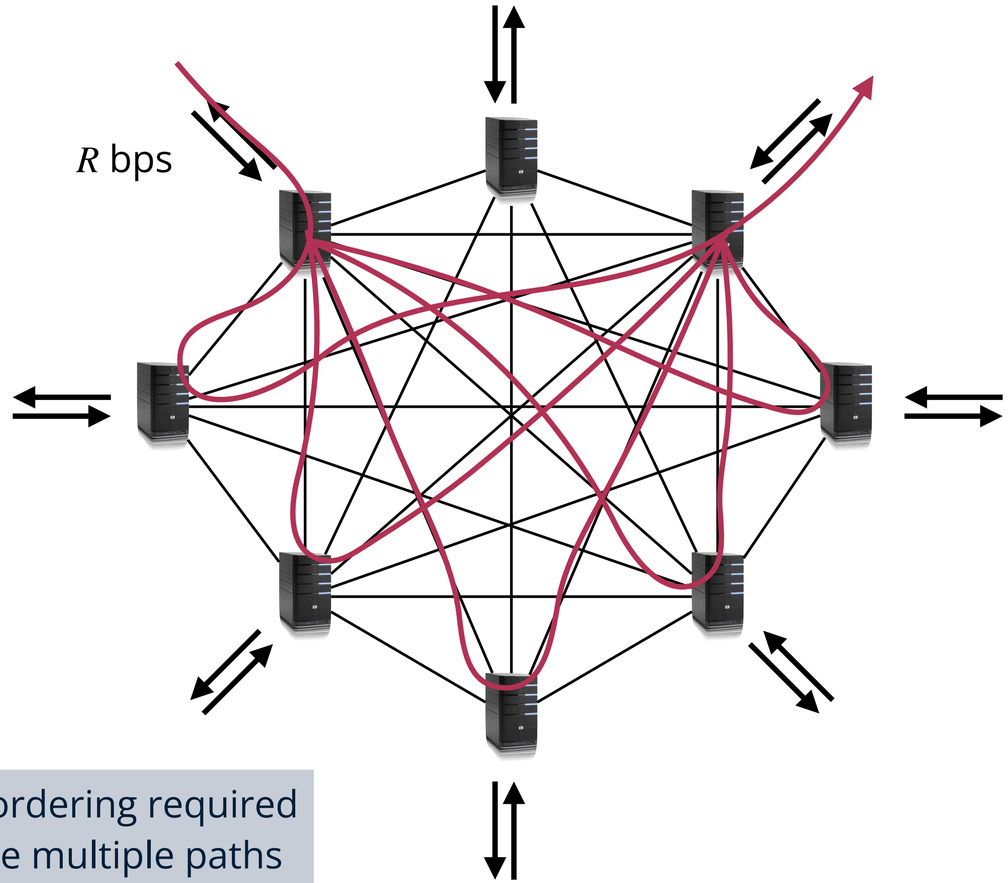


Simple, no packet reordering

Requires $N \times R$ throughput for the centralized scheduler

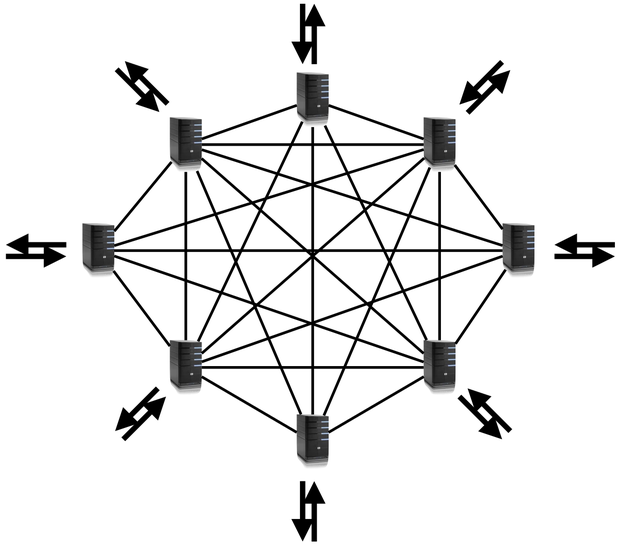
Routing with perfect load balancing

Traffic between a given input and output port is spread across multiple paths

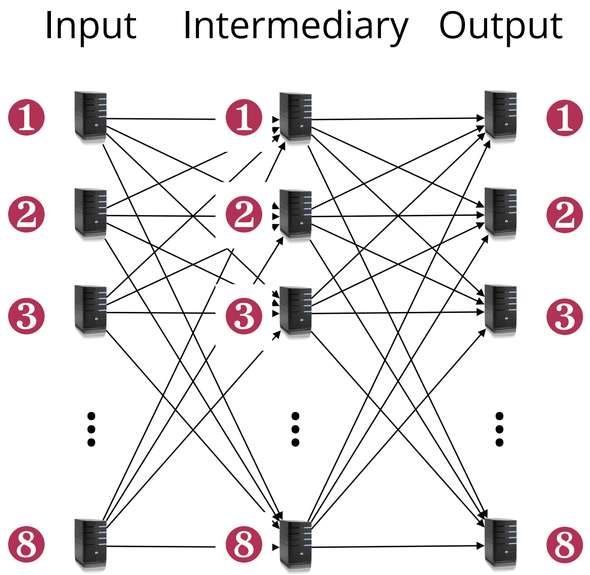


Packet reordering required due to the multiple paths

Valiant load balancing (VLB)



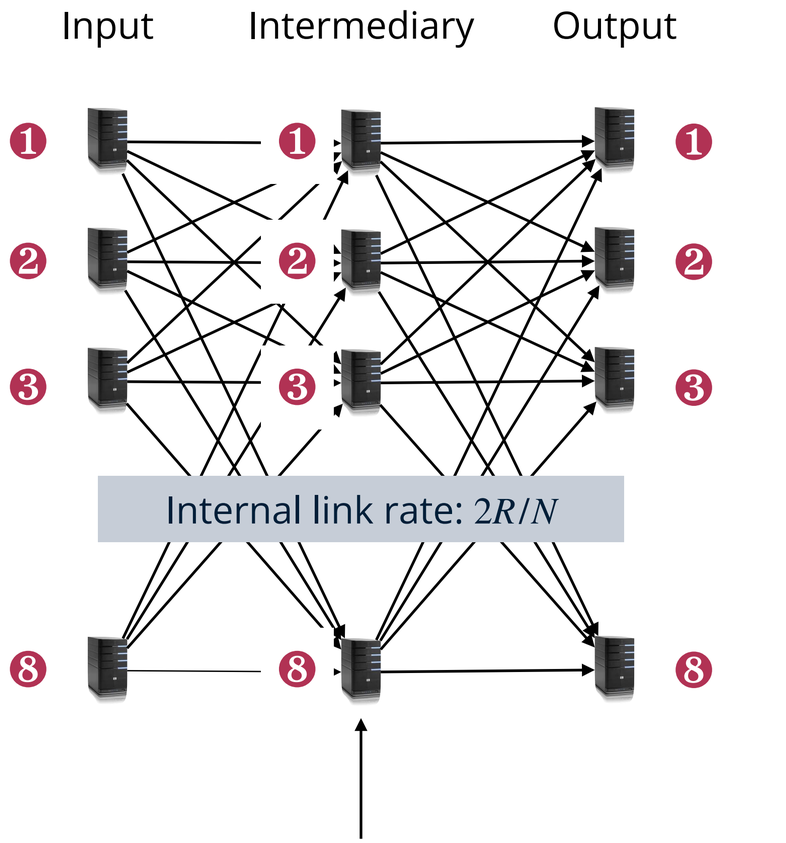
Full-mesh network



2-stage routing

Leslie Valiant, Gordon J. Brebner, Universal schemes for parallel communication, STOC 1981.

Distributed routing in VLB

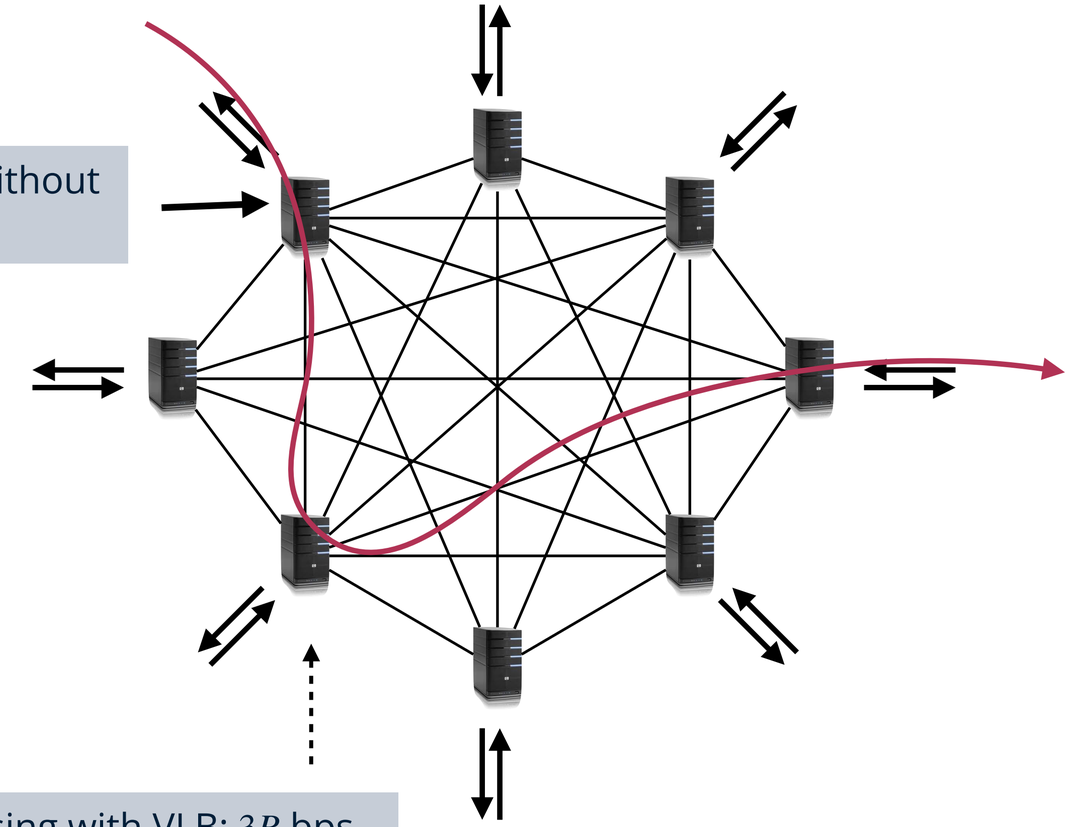


Uniform traffic sampling → 100% throughput

Server performance requirement

What is the required packet processing rate of each server?

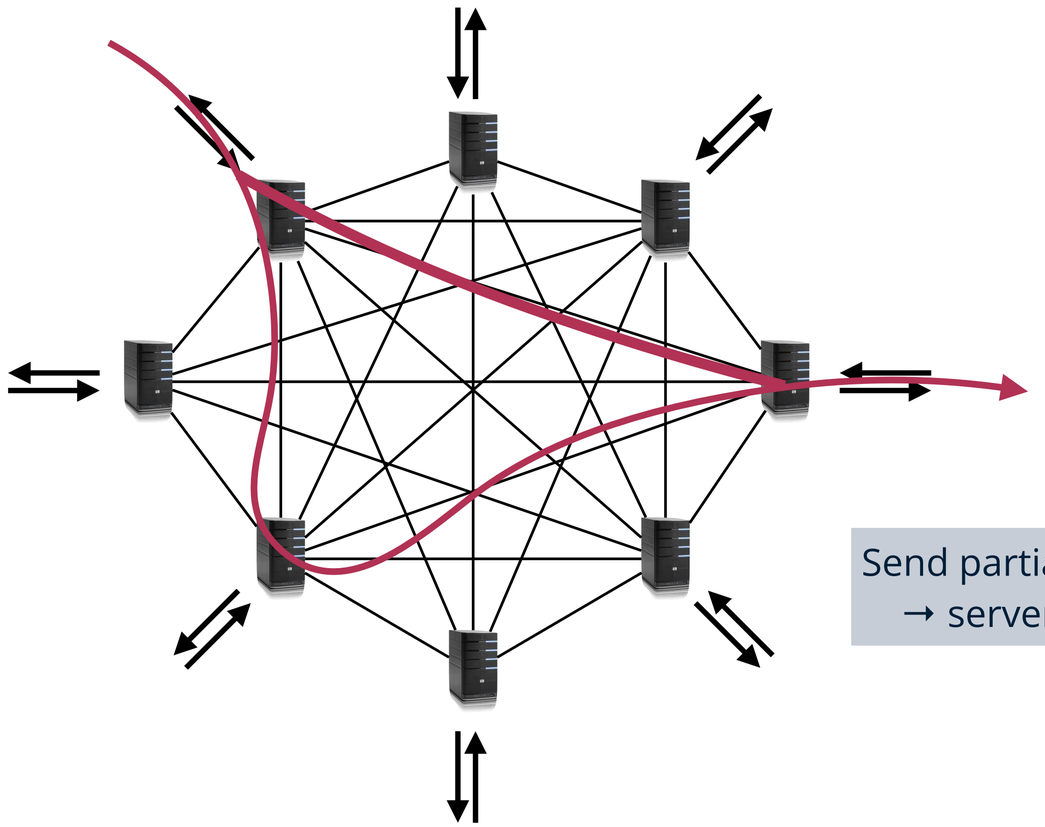
Packet processing without VLB: $2R$ bps



Packet processing with VLB: $3R$ bps

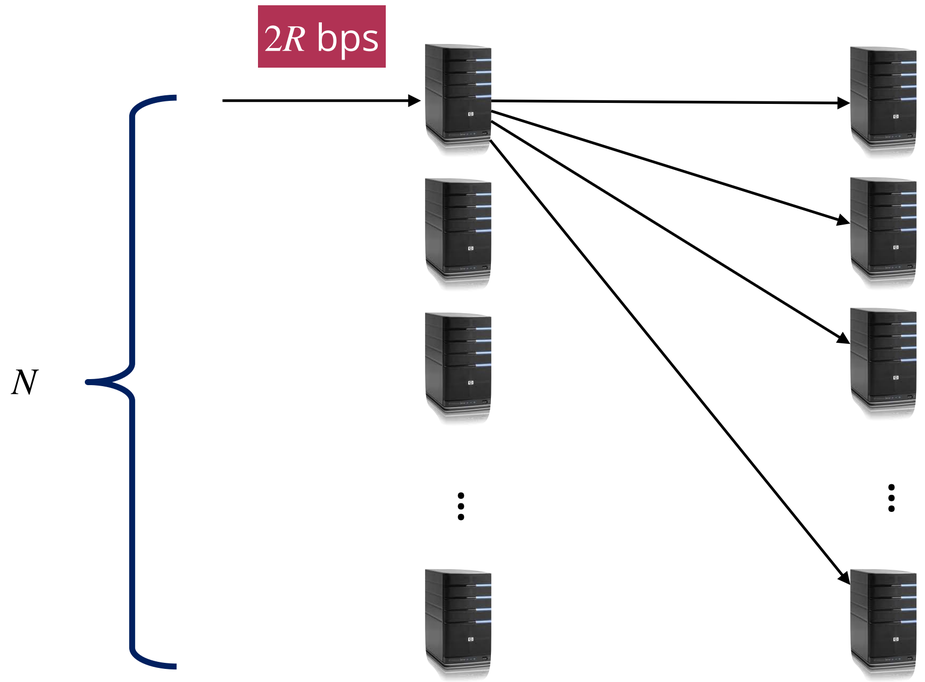
Reducing the server performance requirement with Direct VLB

If the traffic at the input is already uniformly distributed, no need to spread the traffic



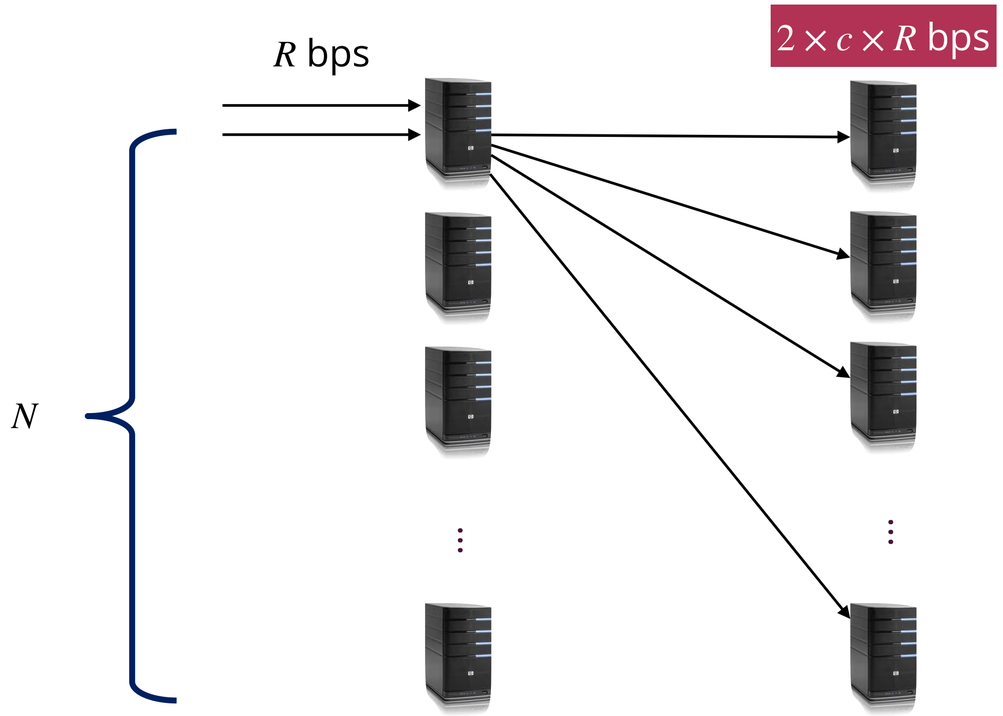
Send partial traffic directly to the destination
→ server processing rate: $c \times R$ ($c \in [2,3]$)

Scaling up with more ports or higher throughput



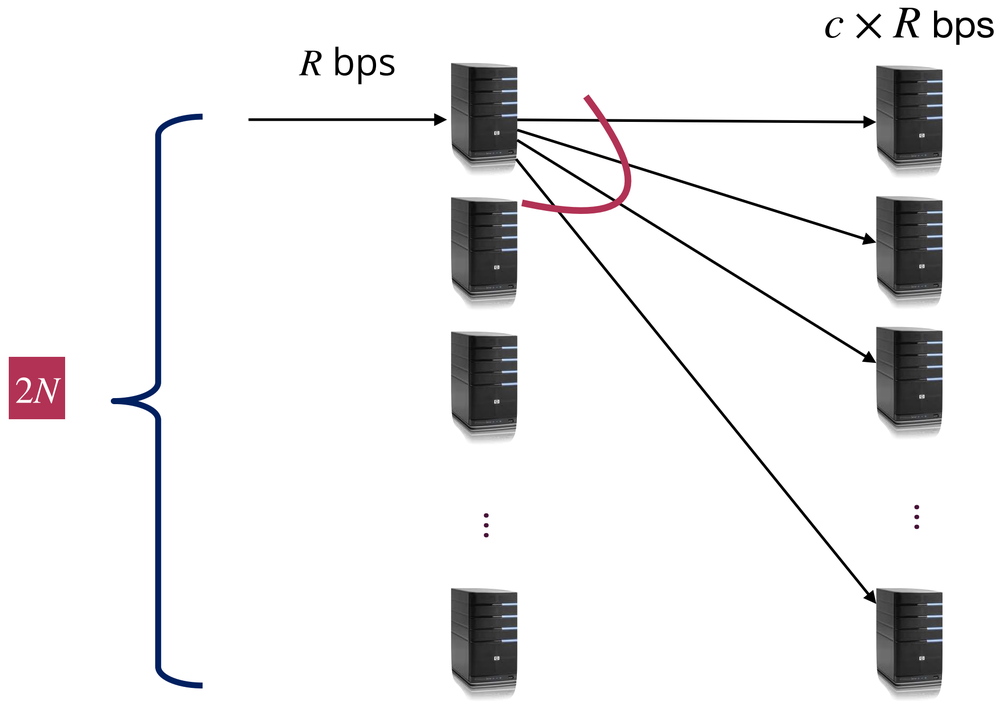
Approach #1: Increase **server capacity**

Scaling up with more ports or higher throughput



Approach #2: **Doubling the input/output port number** on a server requires the server to be able to handle traffic at 2cR bps rate.

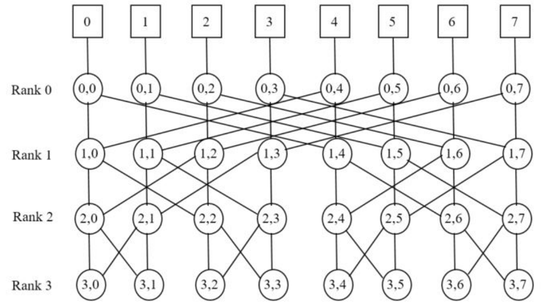
Scaling up with more ports or higher throughput

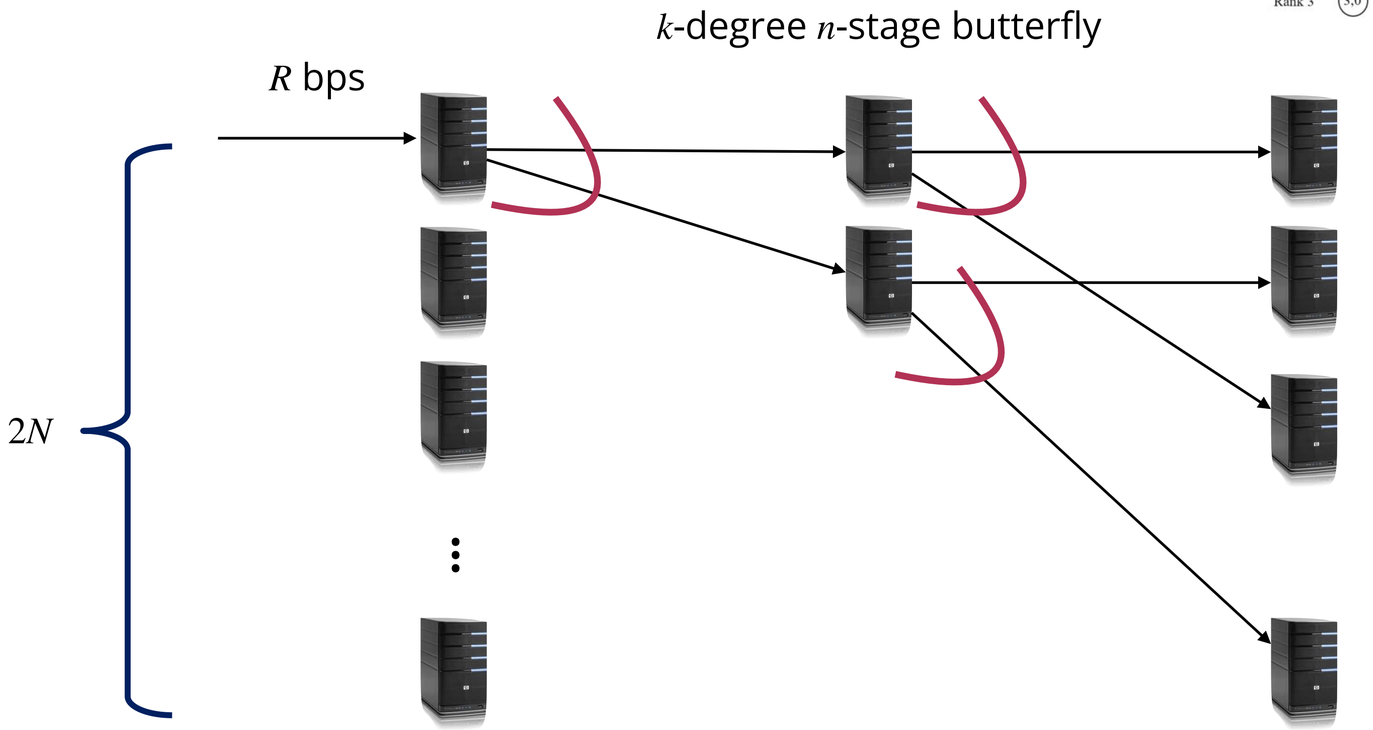


Approach #3: **Doubling the number of servers** would double the number of ports, but it requires higher server fan-outs which are also limited.

Introducing intermediate nodes

Use butterfly network topologies to overcome the fan-out limit issue



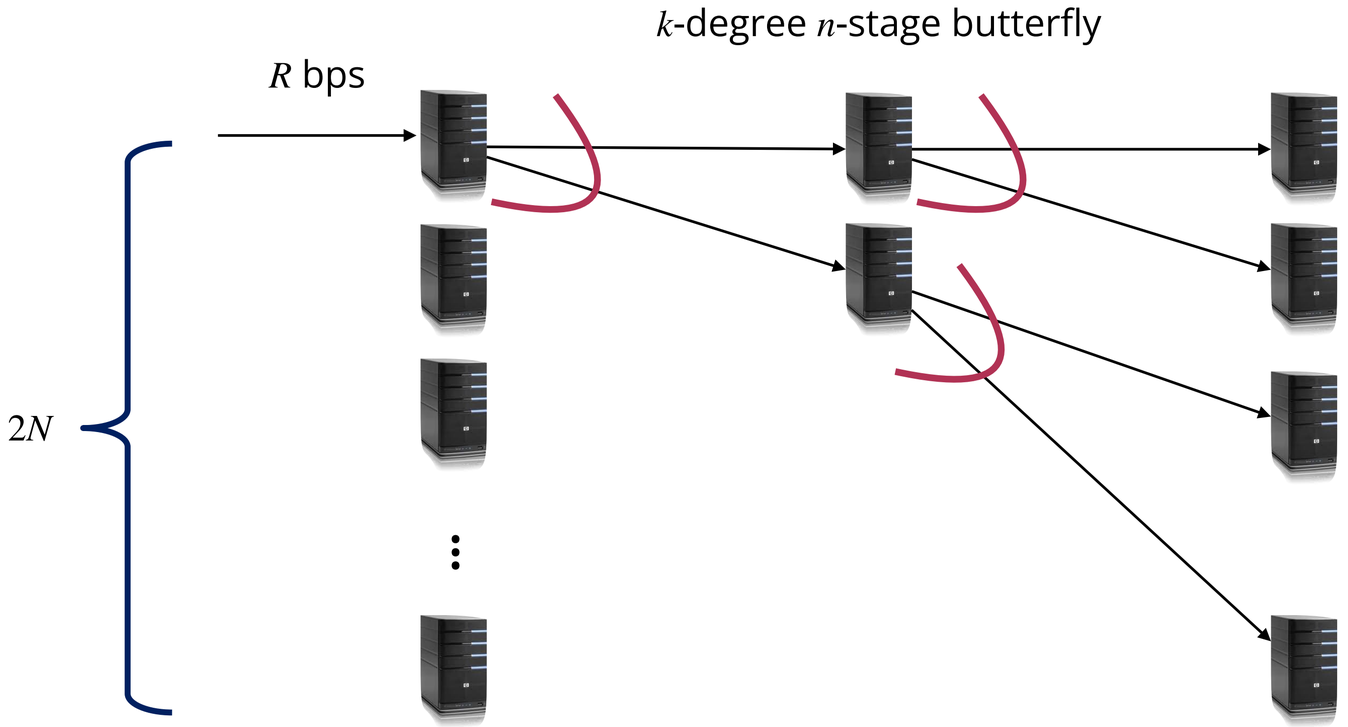


What problems can this approach bring?

Butterfly network: https://en.wikipedia.org/wiki/Butterfly_network

Introducing intermediate nodes

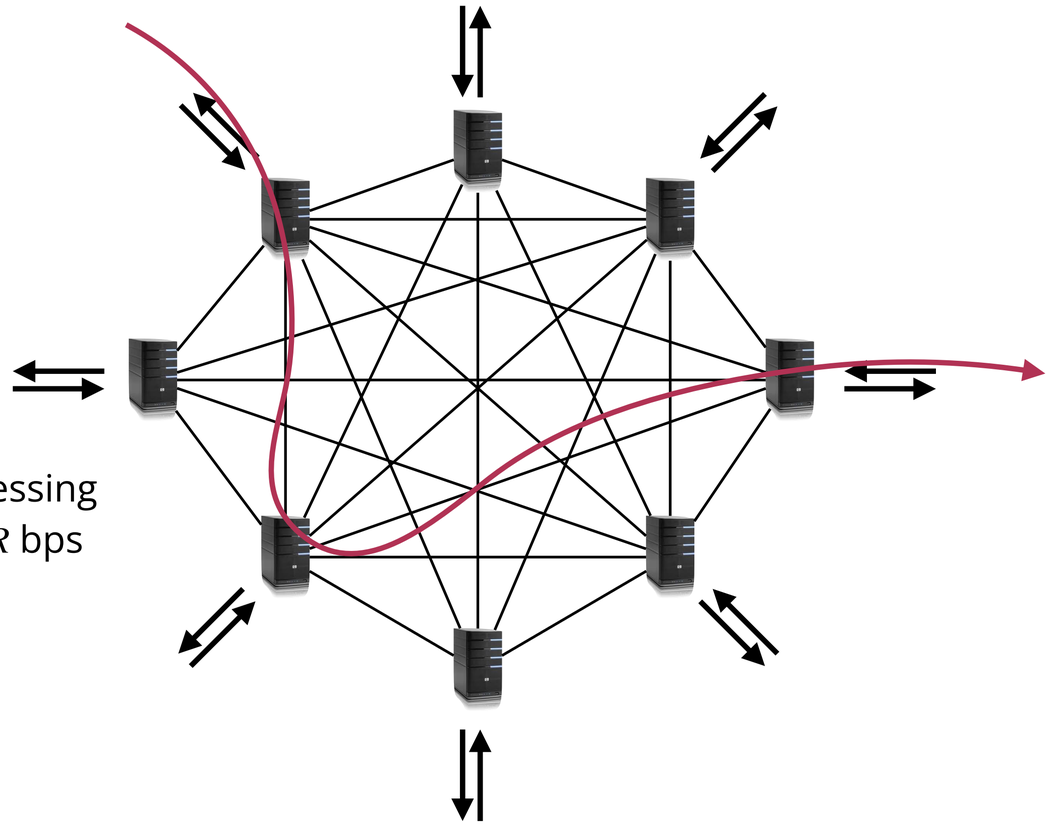
Use butterfly network topologies to overcome the fan-out limit issue



Higher **per-packet latency** since packets need to travel more servers.

Butterfly network: https://en.wikipedia.org/wiki/Butterfly_network

Recall server processing rate



Packet processing with VLB: $3R$ bps

Achieving such a rate on a server is non-trivial!

Server-side optimization summary

State of the art and emerging hardware

- NUMA architecture, multi-queue NICs

Modified NIC driver

- Batching (poll multiple packets from the NIC instead of one at a time)

Careful queue-to-core allocation

- One core per queue, per packet

RouteBricks: Exploiting Parallelism To Scale Software Routers

Mihai Dobrescu[1] and Norbert Egi[2*], Katerina Argyraki[1], Byung-Gon Chun[3], Kevin Fall[3], Gianluca Iannaccone[3], Allan Knies[3], Maziar Manesh[3], Sylvia Ratnasamy[3]

[1] EPFL
Lausanne, Switzerland

[2] Lancaster University
Lancaster, UK

[3] Intel Research Labs
Berkeley, CA

ABSTRACT

We revisit the problem of scaling software routers, motivated by recent advances in server technology that enable high-speed parallel processing—a feature router workloads appear ideally suited to exploit. We propose a software router architecture that parallelizes router functionality both across multiple servers and across multiple cores within a single server. By carefully exploiting parallelism at every opportunity, we demonstrate a 35Gbps parallel router prototype; this router capacity can be linearly scaled through the use of additional servers. Our prototype router is fully programmable using the familiar Click/Linux environment and is built entirely from off-the-shelf, general-purpose server hardware.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking; C.4 [**Performance of Systems**]; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design; D.4.8 [**Operating Systems**]: Performance

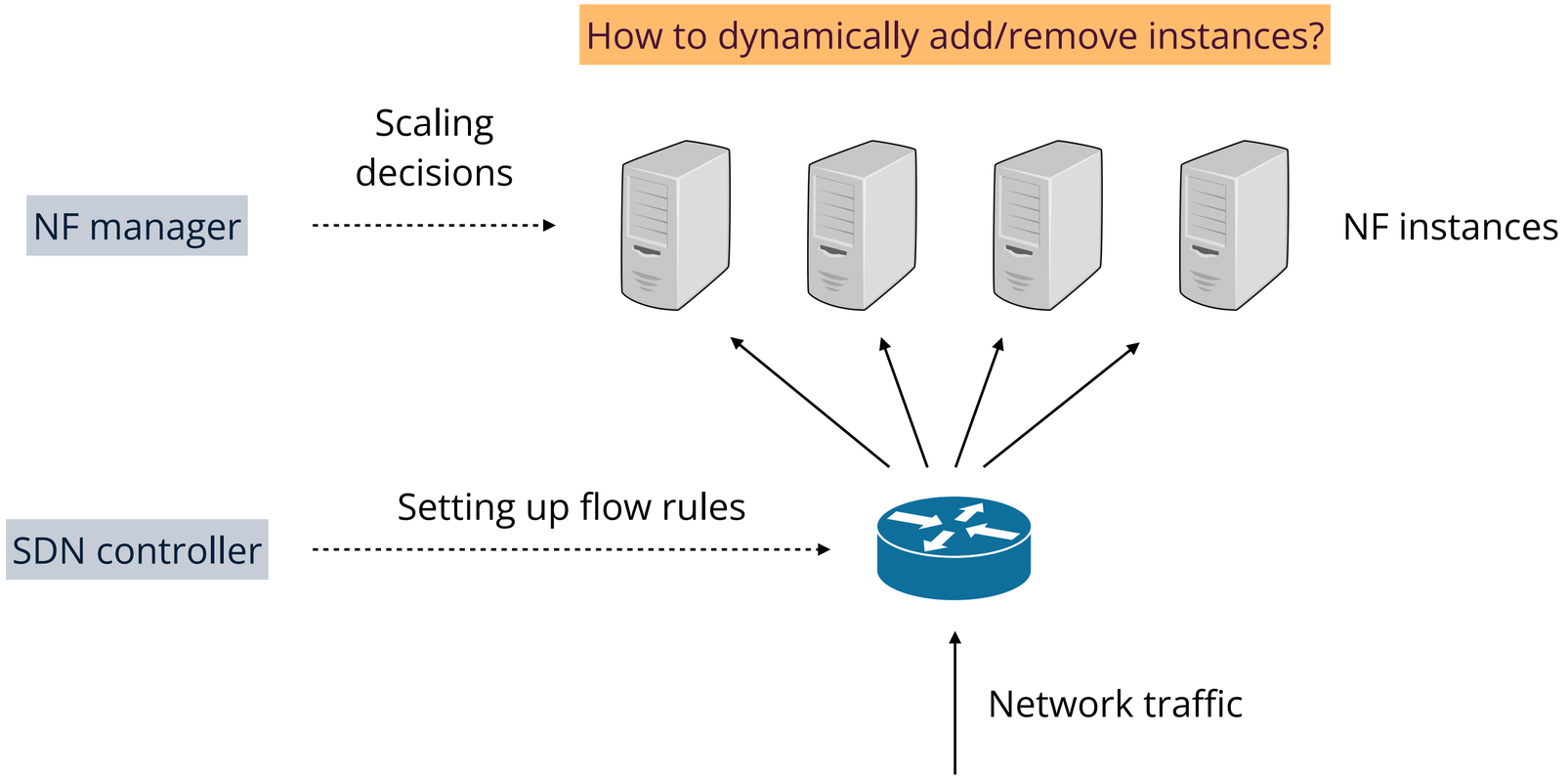
have typically incorporated new functionality by deploying special-purpose network "appliances" or middleboxes [1, 8, 13–15]. However, as the cost of deploying, powering, and managing this assortment of boxes grows, the vision of a consolidated solution in the form of an extensible packet-processing "router" has grown more attractive. And indeed, both industry and research have recently taken steps to enable such extensibility [9, 17, 18, 40, 41].

The difficulty is that the necessary extensions often involve modification to the per-packet processing on a router's high-speed data plane. This is true, for example, of application acceleration [13], measurement and logging [8], encryption [1], filtering and intrusion detection [14], as well as a variety of more forward-looking research proposals [21, 36, 39]. In current networking equipment, however, high performance and programmability are often competing goals—if not mutually exclusive. On the one hand, high-end routers, because they rely on specialized and closed hardware and software, are notoriously difficult to extend, program, or otherwise experiment with. On the other, "software routers" perform packet-processing in software running on general-purpose platforms; these

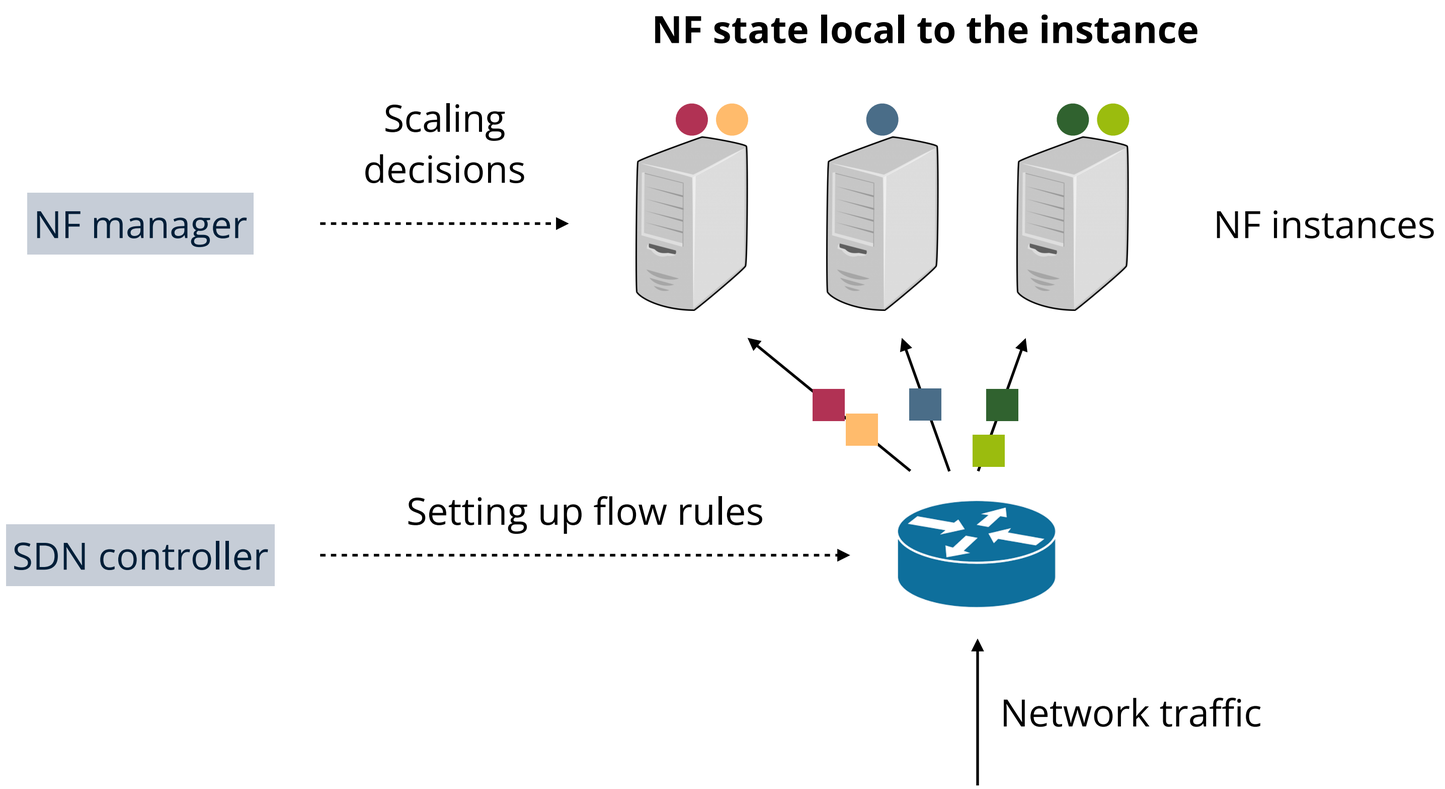
SOSP 2009

How to achieve elastic scaling?

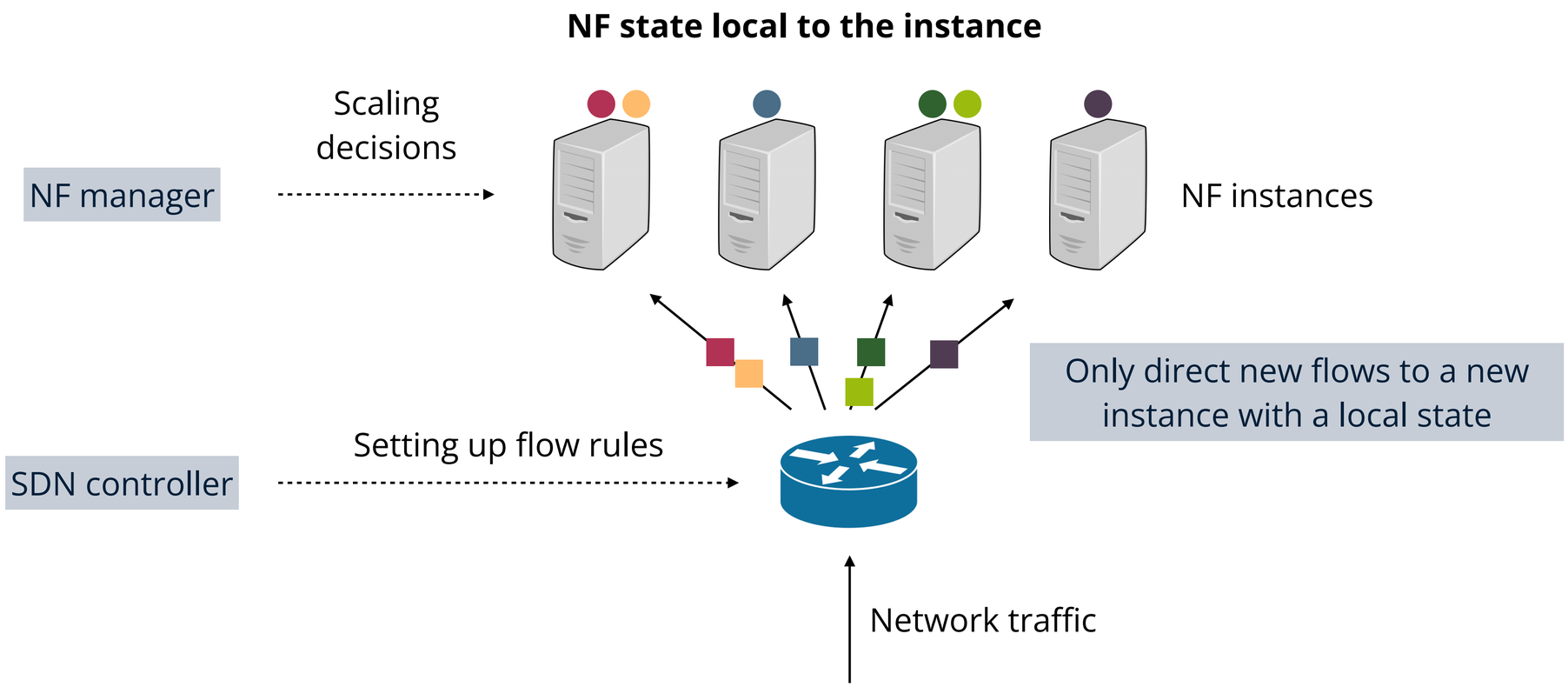
Elastic scaling of virtualized network functions (NFs)



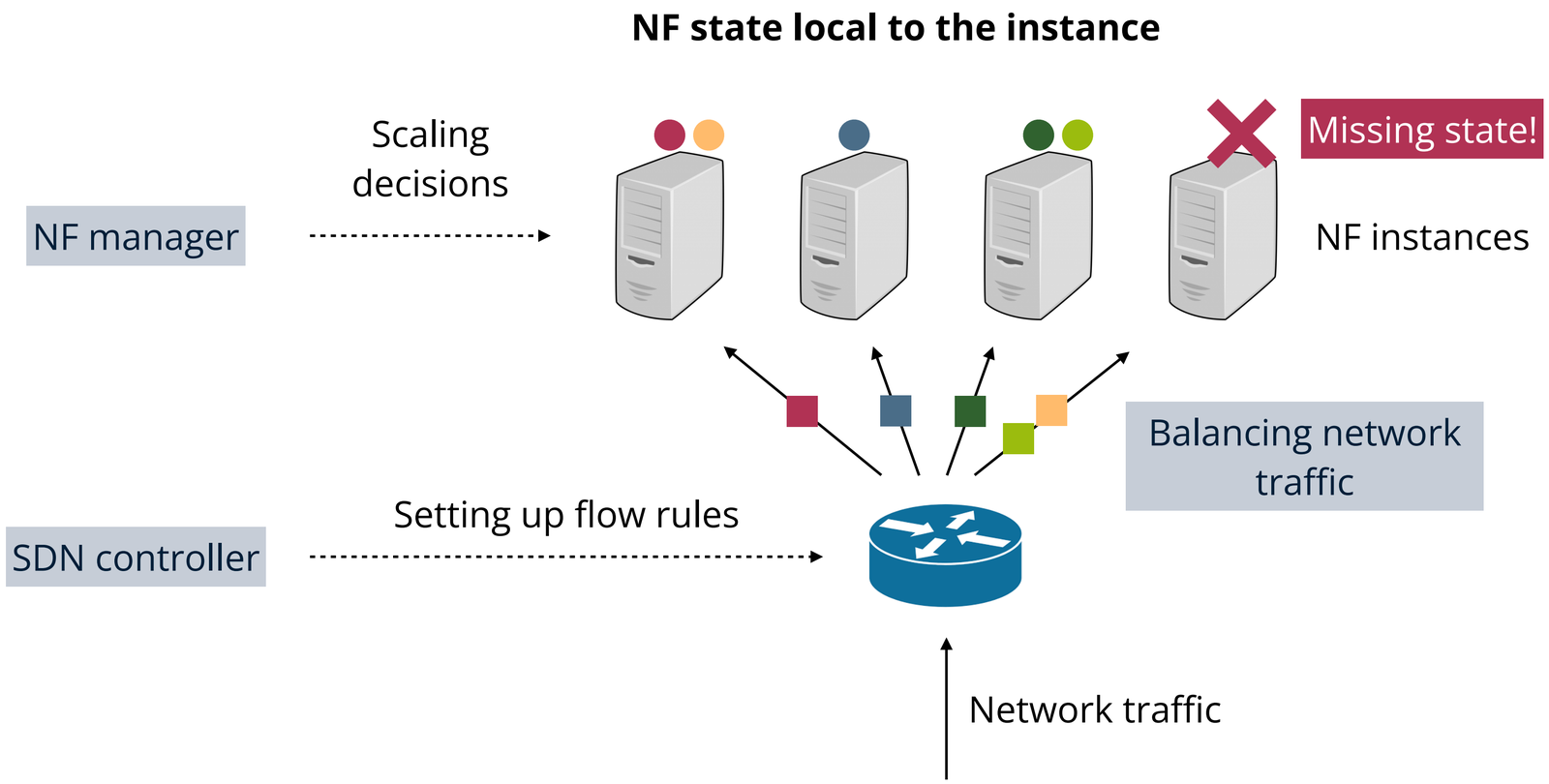
Elastic scaling for stateless NFs



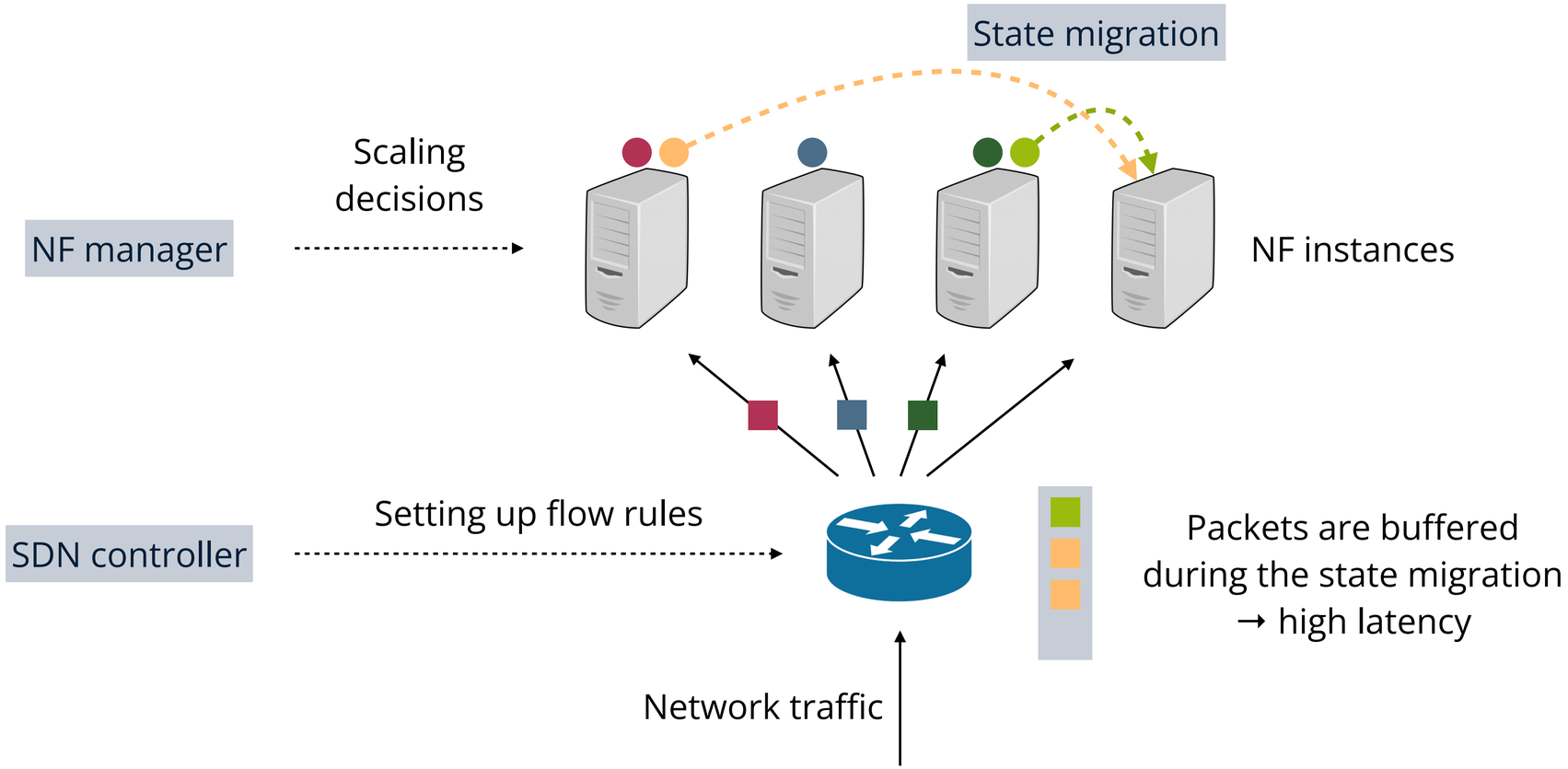
Elastic scaling for stateless NFs: rescaling



Elastic scaling for stateless NFs: rescaling

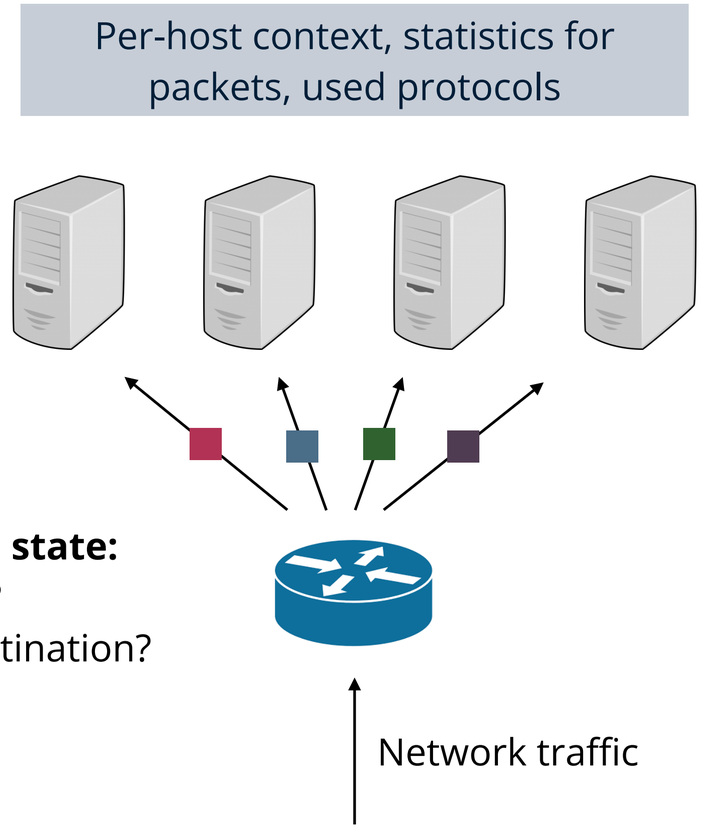


Elastic scaling for stateless NFs: rescaling



Non-partitionable NF states

Example of traffic monitoring

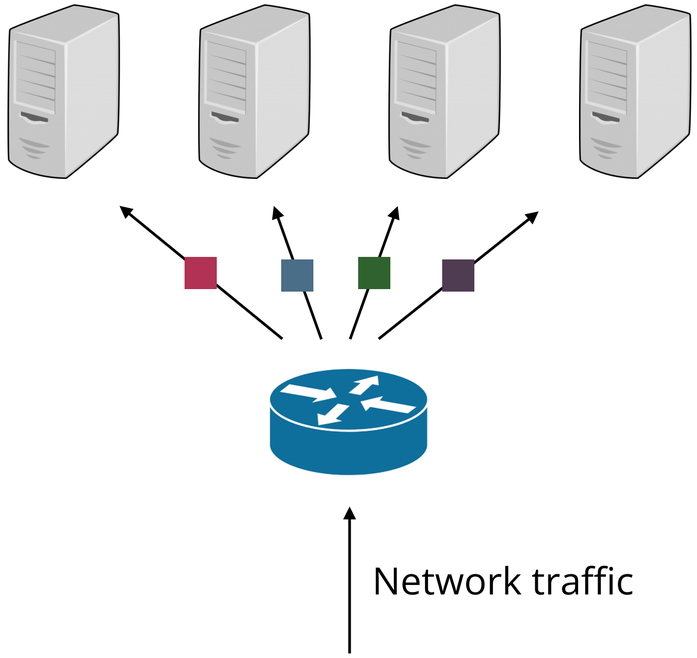


Queries that cannot be answered by a local state:
What is the total number of flows from a host?
How many packets are sent to a particular destination?
What are the protocols used in the traffic?

Non-partitionable NF states

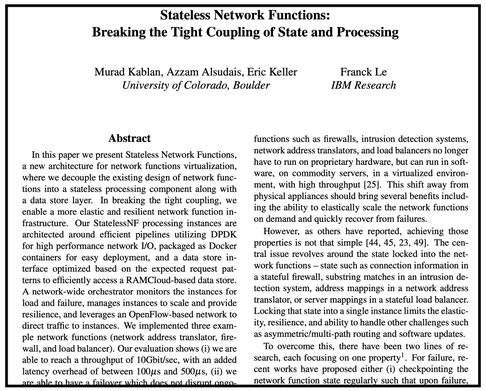
Example of traffic monitoring

Per-host context, statistics for packets, used protocols



How to share the state across NF instances?

Remote state sharing



Stateless Network Functions:
Breaking the Tight Coupling of State and Processing

Murad Kablan, Azzam Alsudais, Eric Keller — *University of Colorado, Boulder*
Franck Le — *IBM Research*

Abstract

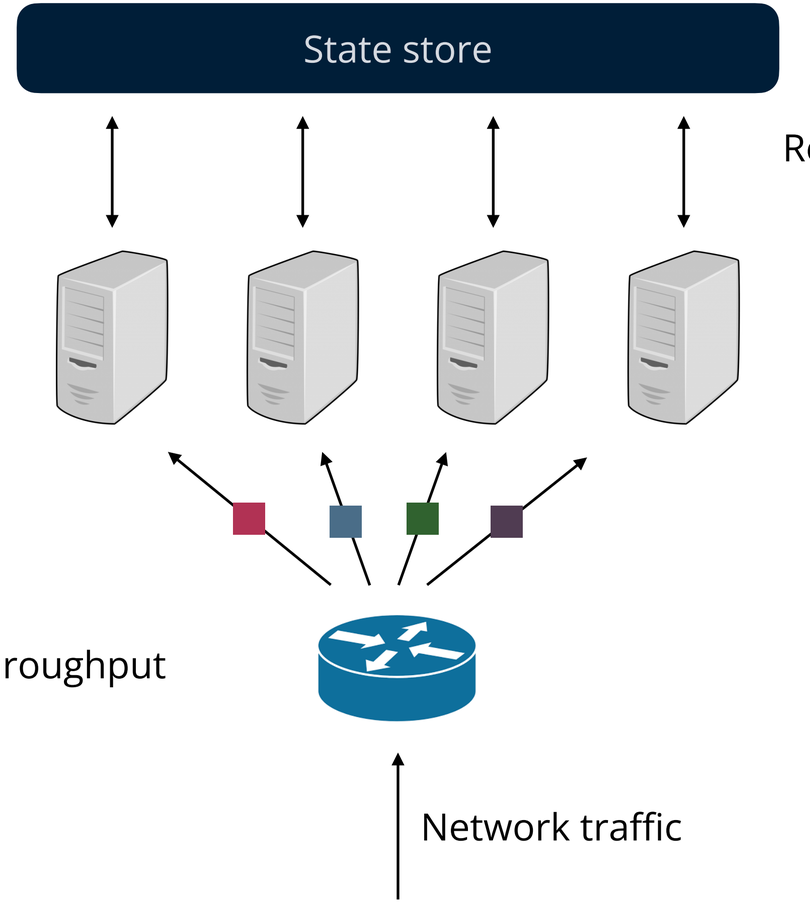
In this paper we present Stateless Network Functions, a new architecture for network functions virtualization, where we decouple the existing design of network functions into a stateless processing component along with a data store layer. In breaking the tight coupling, we enable a more elastic and resilient network function infrastructure. Our StatelessNF processing instances are architected around efficient pipelines utilizing DPDK for high performance network I/O, packaged as Docker containers for easy deployment, and a data store interface optimized based on the expected request patterns to efficiently access a RAMCloud-based data store. A network-wide orchestrator monitors the instances for load and failure, manages instances to scale and provide resilience, and leverages an OpenFlow-based network to direct traffic to instances. We implemented three example network functions (network address translator, firewall, and load balancer). Our evaluation shows (i) we are able to reach a throughput of 10Gbit/sec, with an added latency overhead of between 100µs and 500µs, (ii) we are able to have a failover which does not disrupt ongoing

functions such as firewalls, intrusion detection systems, network address translators, and load balancers no longer have to run on proprietary hardware, but can run in software, on commodity servers, in a virtualized environment, with high throughput [25]. This shift away from physical appliances should bring several benefits including the ability to elastically scale the network functions on demand and quickly recover from failures.

However, as others have reported, achieving those properties is not that simple [44, 45, 23, 49]. The central issue revolves around the state locked into the network functions – state such as connection information in a stateful firewall, substring matches in an intrusion detection system, address mappings in a network address translator, or server mappings in a stateful load balancer. Locking that state into a single instance limits the elasticity, resilience, and ability to handle other challenges such as asymmetric/multi-path routing and software updates.

To overcome this, there have been two lines of research, each focusing on one property[1]. For failure, recent works have proposed either (i) checkpointing the network function state regularly such that upon failure,

NSDI 2017

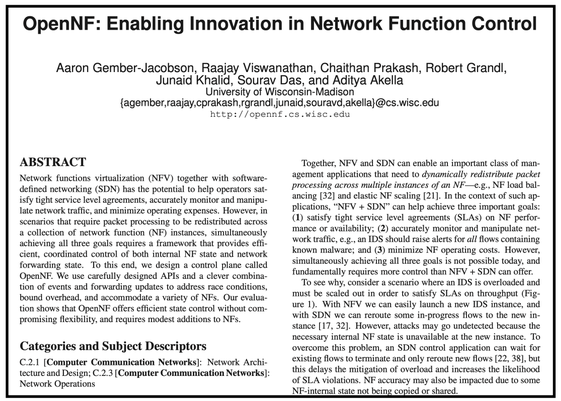


Remote access to the state store shared by all NF instances

Pros: simple, easy to implement
Cons: high per-packet latency, low throughput

Local + remote access

All states are accessed locally, but synchronized before accesses



OpenNF: Enabling Innovation in Network Function Control

Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl,
Junaid Khalid, Sourav Das, and Aditya Akella
University of Wisconsin-Madison
{agember,raajay,cprakash,rgrandl,junaid,souravd,akella}@cs.wisc.edu
http://opennf.cs.wisc.edu

ABSTRACT

Network functions virtualization (NFV) together with software-defined networking (SDN) has the potential to help operators satisfy tight service level agreements, accurately monitor and manipulate network traffic, and minimize operating expenses. However, in scenarios that require packet processing to be redistributed across a collection of network function (NF) instances, simultaneously achieving all three goals requires a framework that provides efficient, coordinated control of both internal NF state and network forwarding state. To this end, we design a control plane called OpenNF. We use carefully designed APIs and a clever combination of events and forwarding updates to address race conditions, bound overhead, and accommodate a variety of NFs. Our evaluation shows that OpenNF offers efficient state control without compromising flexibility, and requires modest additions to NFs.

Categories and Subject Descriptors

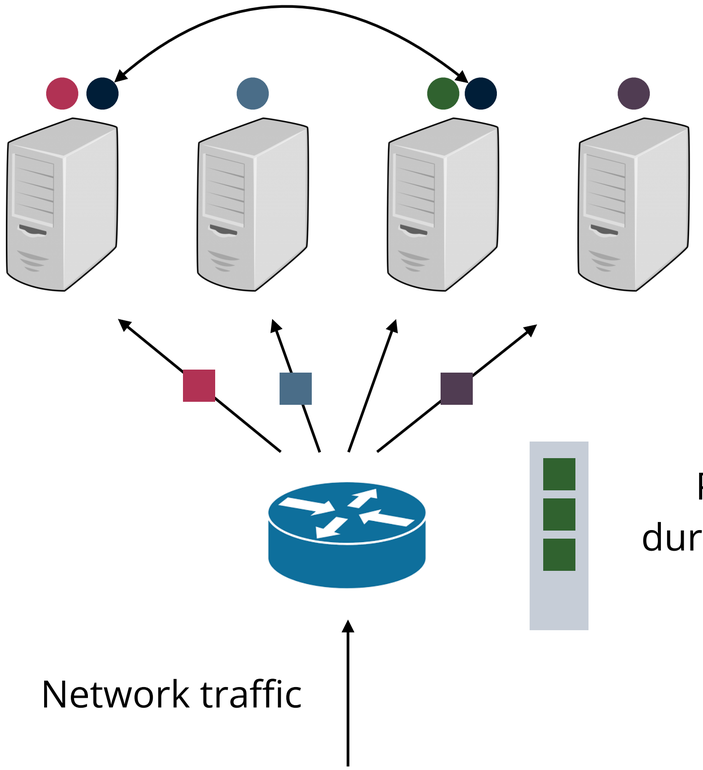
C.2.1 [**Computer Communication Networks**]: Network Architecture and Design; C.2.3 [**Computer Communication Networks**]: Network Operations

Together, NFV and SDN can enable an important class of management applications that need to *dynamically redistribute packet processing across multiple instances of an NF*—e.g., NF load balancing [32] and elastic NF scaling [21]. In the context of such applications, "NFV + SDN" can help achieve three important goals: (**1**) satisfy tight service level agreements (SLAs) on NF performance or availability; (**2**) accurately monitor and manipulate network traffic, e.g., an IDS should raise alerts for *all* flows containing known malware; and (**3**) minimize NF operating costs. However, simultaneously achieving all three goals is not possible today, and fundamentally requires more control than NFV + SDN can offer.

To see why, consider a scenario where an IDS is overloaded and must be scaled out in order to satisfy SLAs on throughput (Figure 1). With NFV we can easily launch a new IDS instance, and with SDN we can reroute some in-progress flows to the new instance [17, 32]. However, attacks may go undetected because the necessary internal NF state is unavailable at the new instance. To overcome this problem, an SDN control application can wait for existing flows to terminate and only reroute new flows [22, 38], but this delays the mitigation of overload and increases the likelihood of SLA violations. NF accuracy may also be impacted due to some NF-internal state not being copied or shared.

SIGCOMM 2014

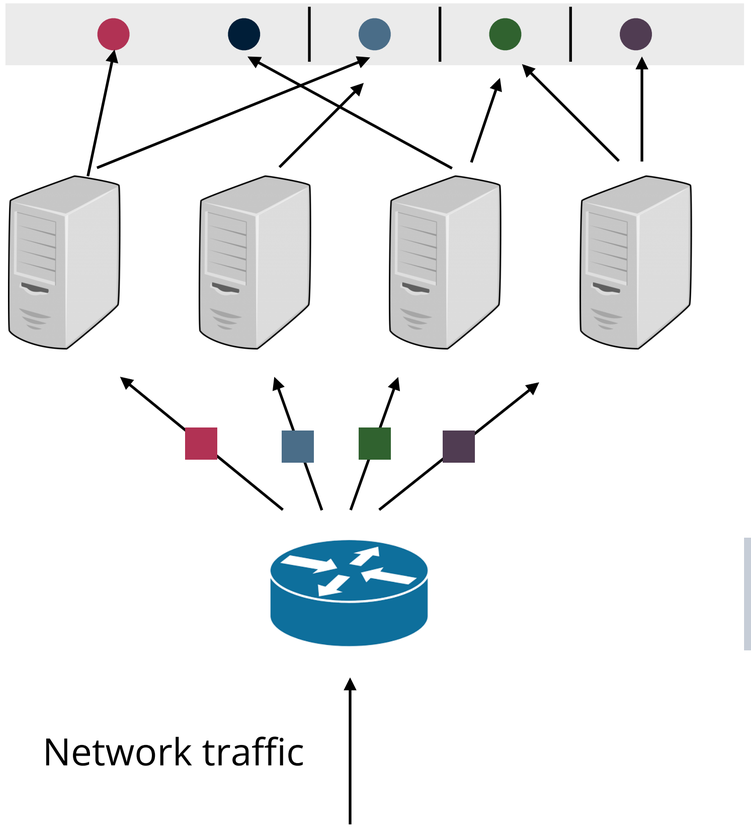
Stop + Synchronize + Resume



Packets are buffered during the state migration/ synchronization

Network traffic

Distributed shared state space



Any NF instances can access any state

Minimal performance overhead
No system-wide pausing during scaling events

Object for NF state abstraction

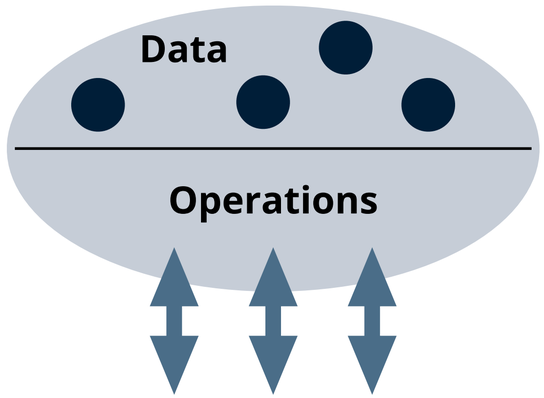
Object encapsulation enables easy state management

Integrity protection of state

- Single writer vs. multiple writer

Optimization per object

- Performance vs. consistency: different sweet spot per object



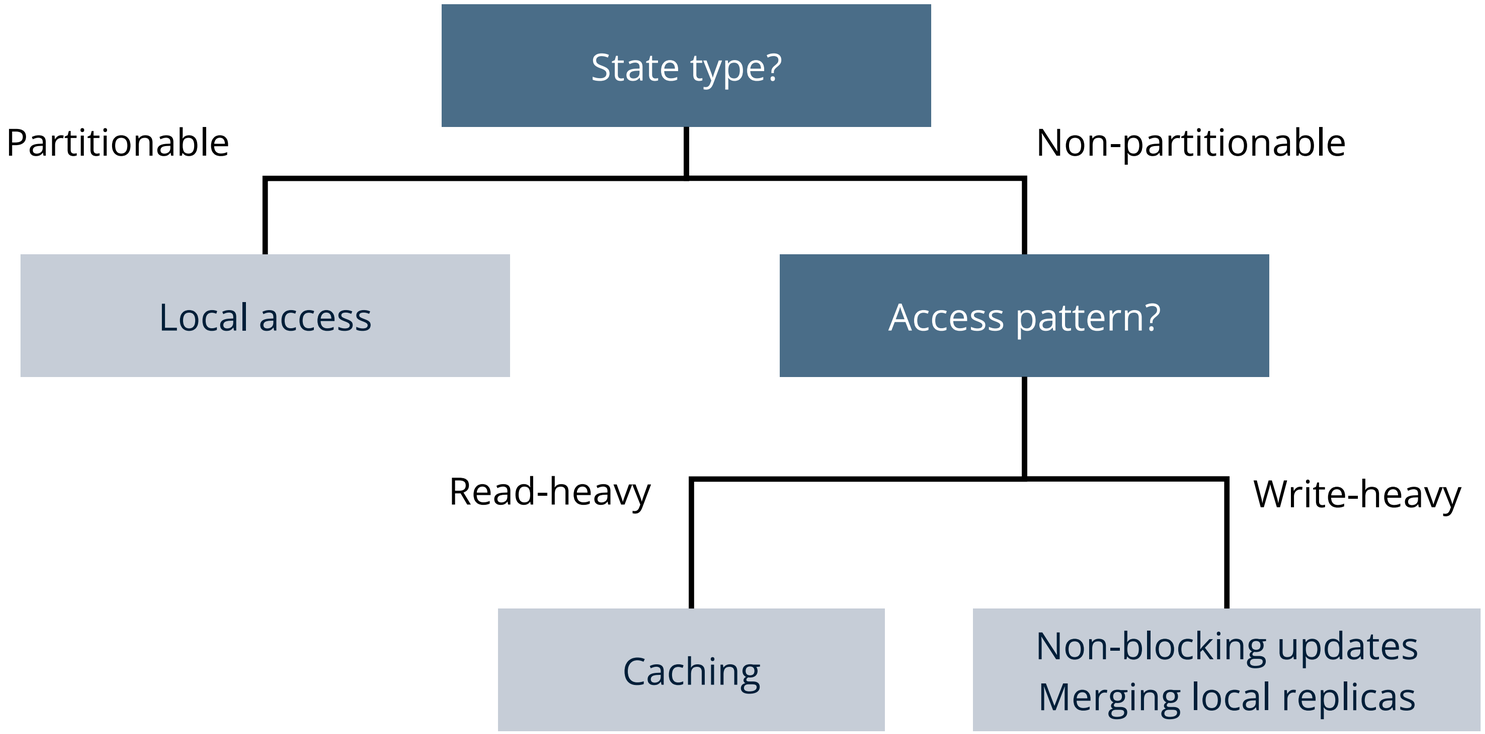

```
class Counter : public MultiWriter {
  private:
    uint32_t counter;
  public:
    uint32_t int_and_get();
    void inc(uint32_t x) untethered;
    uint32_t get() const stale;
};
```

updating from multiple instances

non-blocking updates

return from cache

NF state optimization strategies



Elastic Scaling of Stateful Network Functions

Shinae Woo[⋆†], *Justine Sherry*[‡], *Sangjin Han*[⋆], *Sue Moon*[†], *Sylvia Ratnasamy*[⋆], *and Scott Shenker*[⋆§]

[⋆]*University of California, Berkeley* [†]*KAIST* [‡]*CMU* [§]*ICSI*

Abstract

Elastic scaling is a central promise of NFV but has been hard to realize in practice. The difficulty arises because most Network Functions (NFs) are *stateful* and this state need to be *shared* across NF instances. Implementing state sharing while meeting the throughput and latency requirements placed on NFs is challenging and, to date, no solution exists that meets NFV's performance goals for the full spectrum of NFs.

S6 is a new framework that supports elastic scaling of NFs without compromising performance. Its design builds on the insight that a distributed shared state abstraction is well-suited to the NFV context. We organize state as a distributed shared object (DSO) space and extend the DSO concept with techniques designed to meet the need for elasticity and high-performance in NFV workloads. S6 simplifies development: NF writers program with no awareness of how state is distributed and shared. Instead, S6 transparently migrates state and handles accesses to shared state. In our evaluation, compared to recent solutions for dynamic scaling of NFs, S6 improves performance by 100x during scaling events [25], and by 2-5x under normal operation [27].

In addition, elastic scaling must ensure *affinity* between packets and their state (*i.e.*, that a packet is directed to the NF instance that holds the state necessary to process that packet), and such affinity must be correctly enforced even in the face of state migrations. A final complication is that some types of state are not partitionable, but *shared* across instances (see §2 for examples). In such cases, elastic scaling must support access to shared state in a manner that ensures the consistency requirements of that state are met, and with minimal disruption to NF throughput and latency.

The core of any elastic scaling solution is how state is organized and abstracted to NF applications. Recent work has explored different options in this regard. Some [33] assume that all state is local, but neither shared or migrated – we call this the *local-only* approach. Others [25,37] support a richer model in which state is exposed to NF developers as either local or remote, and developers can migrate state from remote to local storage, or explicitly access remote state – we call this the *local+remote* approach. Still others [27] assume that *all* state is remote, stored in a centralized store – we call this the *remote-only* approach.

The above were pioneering efforts in exploring the de-

NSDI 2018

Summary

Easy programming and high performance

Next time: programmable data plane