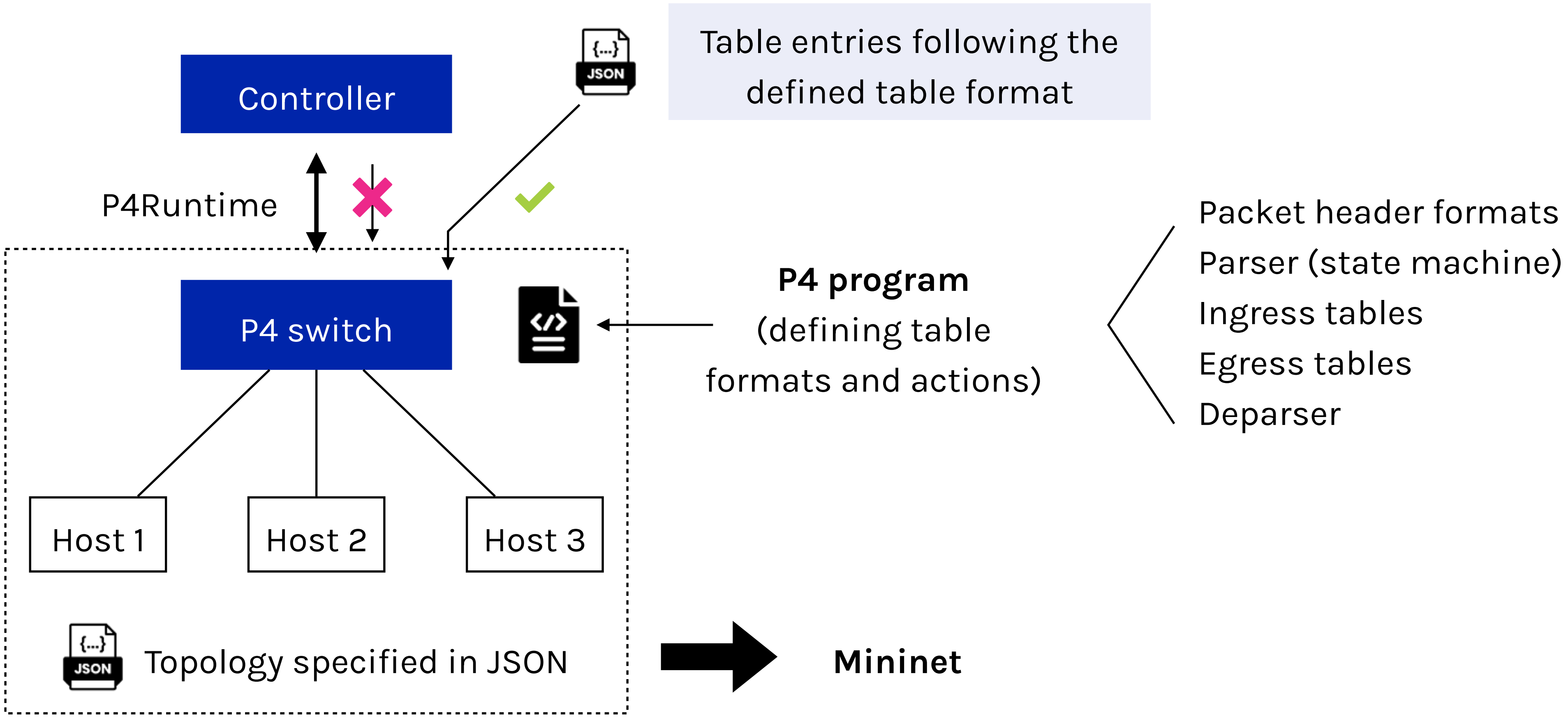
PADERBORN UNIVERSITY

Advanced Networked Systems SS24

In-Network Computing

Prof. Lin Wang, Ph.D.
Computer Networks Group
Paderborn University
https://cs.uni-paderborn.de/cn

Lab4 setup



Controller
P4Runtime
JSON
Table entries following the defined table format
P4 switch
P4 program
(defining table formats and actions)
Packet header formats
Parser (state machine)
Ingress tables
Egress tables
Deparser
Host 1
Host 2
Host 3
JSON
Topology specified in JSON
Mininet

Lab4 examples

Header

```
typedef bit<48> macAddr_t;
header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

struct headers {
    ethernet_t    ethernet;
}
```

Parser

```
state parse_ethernet {
    packet.extract(hdr.ethernet);
    transition select(hdr.ethernet.etherType) {
        TYPE_ARP: parse_arp;
        TYPE_IPV4: parse_ipv4;
        default: accept;
    }
}
```

Lab4 examples

Table definition

```
table ipv4_lpm {
    key = {
        hdr.network.ipv4.dstAddr: exact;
    }
    actions = {
        ipv4_forward;
        drop;
        NoAction;
    }
    size = 1024;
    default_action = drop();
}
```

Table control flow

```
action ipv4_forward(
        macAddr_t dstAddr,
        egressSpec_t port) {...}

if (hdr.network.ipv4.isValid()) {
    ipv4_lpm.apply();
}
```

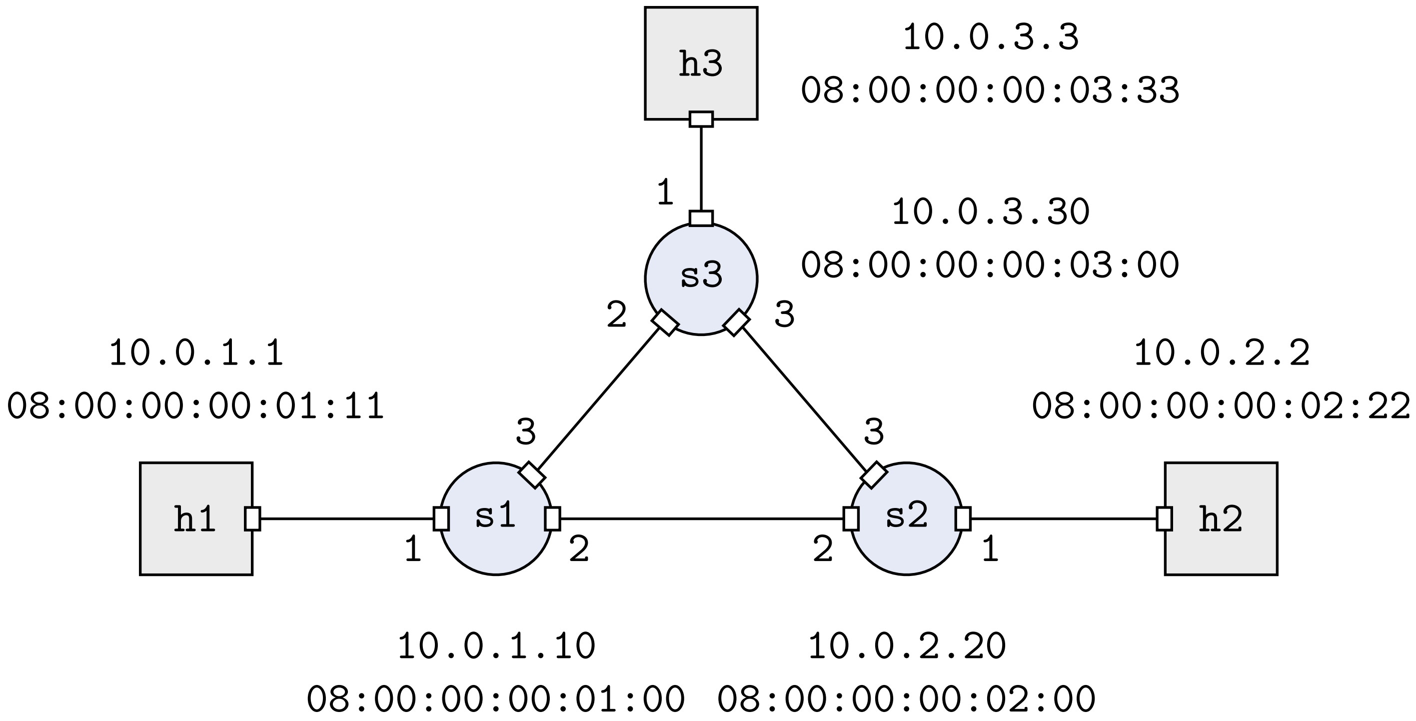
Lab4 examples

Table rules in the format of JSON

```
"table_entries": [
    {
      "table": "MyIngress.ipv4_lpm",
      "match": {
        "hdr.network.ipv4.dstAddr": "10.0.1.1"
      },
      "action_name": "MyIngress.ipv4_forward",
      "action_params": {
        "dstAddr": "08:00:00:00:01:11",
        "port": 1
      }
    }
]
```

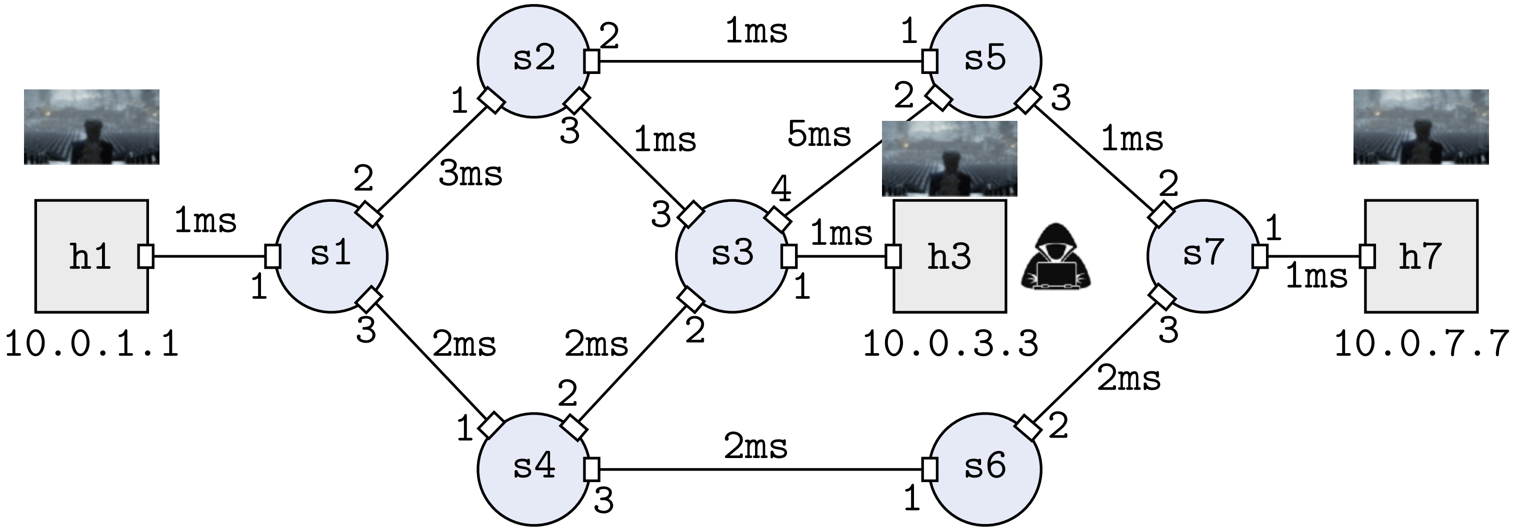
Check P4 tutorials: https://github.com/p4lang/tutorials

Lab4: longest path routing



Routing + ARP resolving

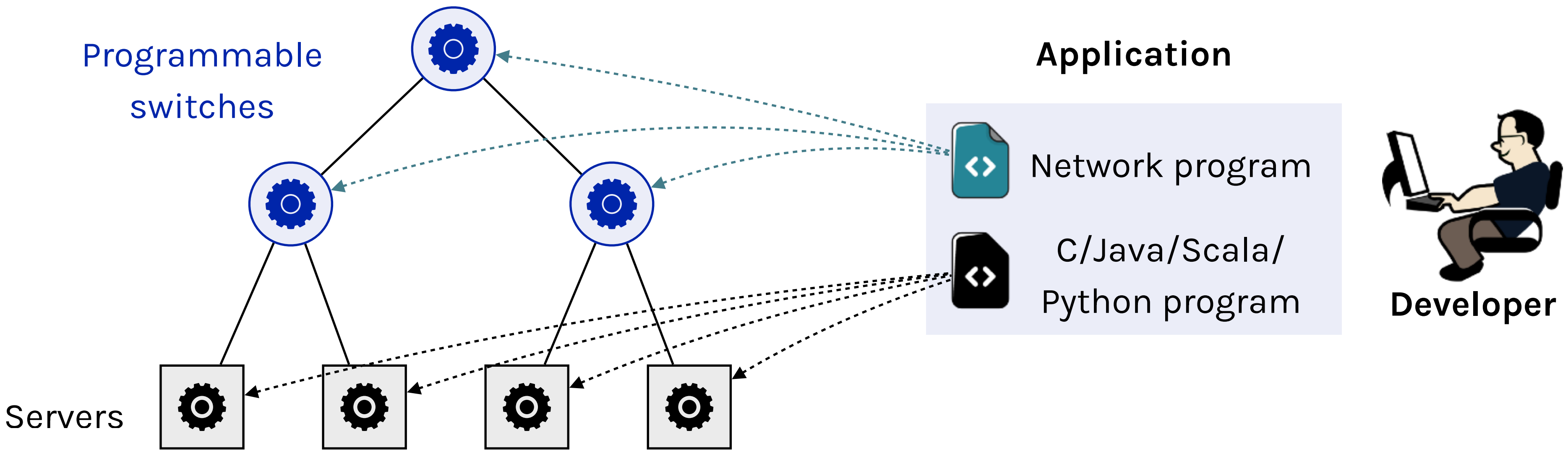
Lab4: video interception



Routing + ARP resolving + traffic replication + small fixes

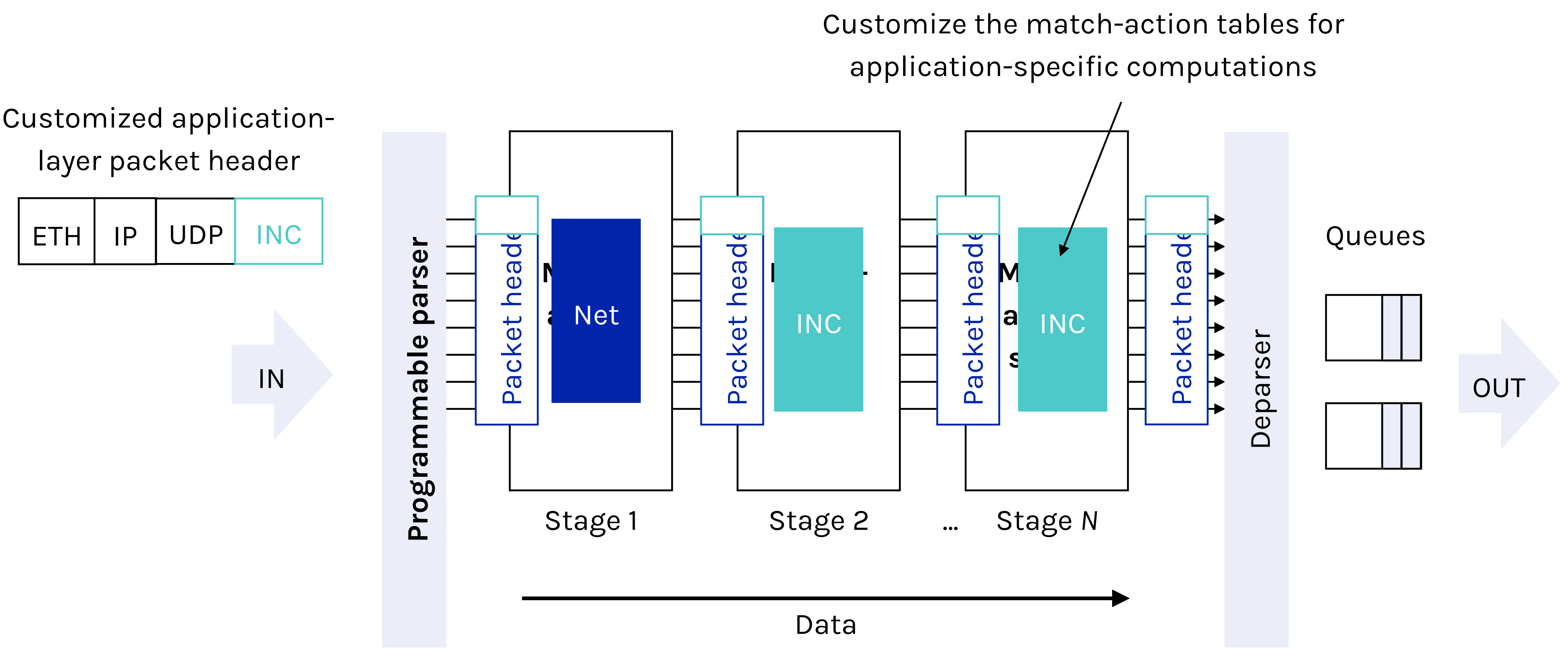
Questions?

In-network computing



In-network computing: performing application-specific computations “in the network” on the path between data sources and sinks, leveraging modern programmable switches

In-network computing paradigm



Learning objectives

How to implement an **in-network caching** service?

How to implement an **in-network coordination** service?

How to use in-network computing for **accelerating distributed machine learning**?

How to implement an in-network caching service?

Key-value storage

Store, retrieve, manage key-value objects

- Critical building block for large-scale cloud services
- Need to meet aggressive latency and throughput objectives efficiently

...

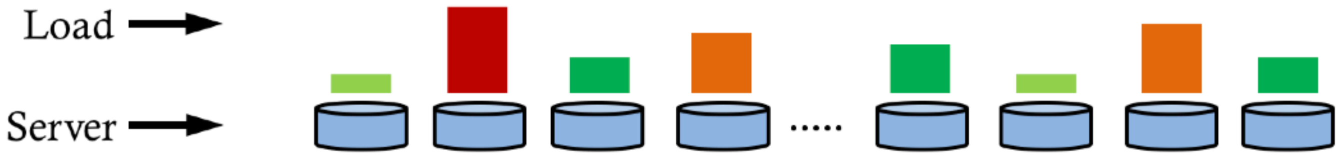
Target workloads

- Small objects
- Read intensive
- **Highly skewed and dynamic key popularity**

Challenge

Highly skewed and rapidly changing workloads

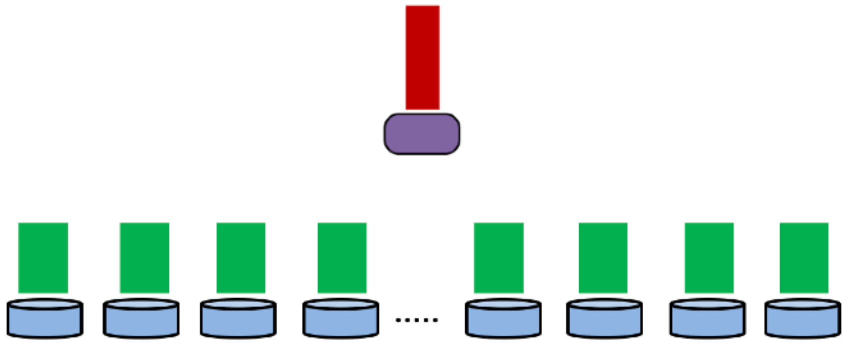
Requirement: high throughput, low (tail) latency



How to provide effective dynamic load balancing?

Opportunity

Fast, small cache can ensure load balancing



Cache $O(N \log N)$ **hottest items**

- E.g., 10,000 hot objects

N**: number of servers**

- E.g., 100 backend servers with 100 billion items

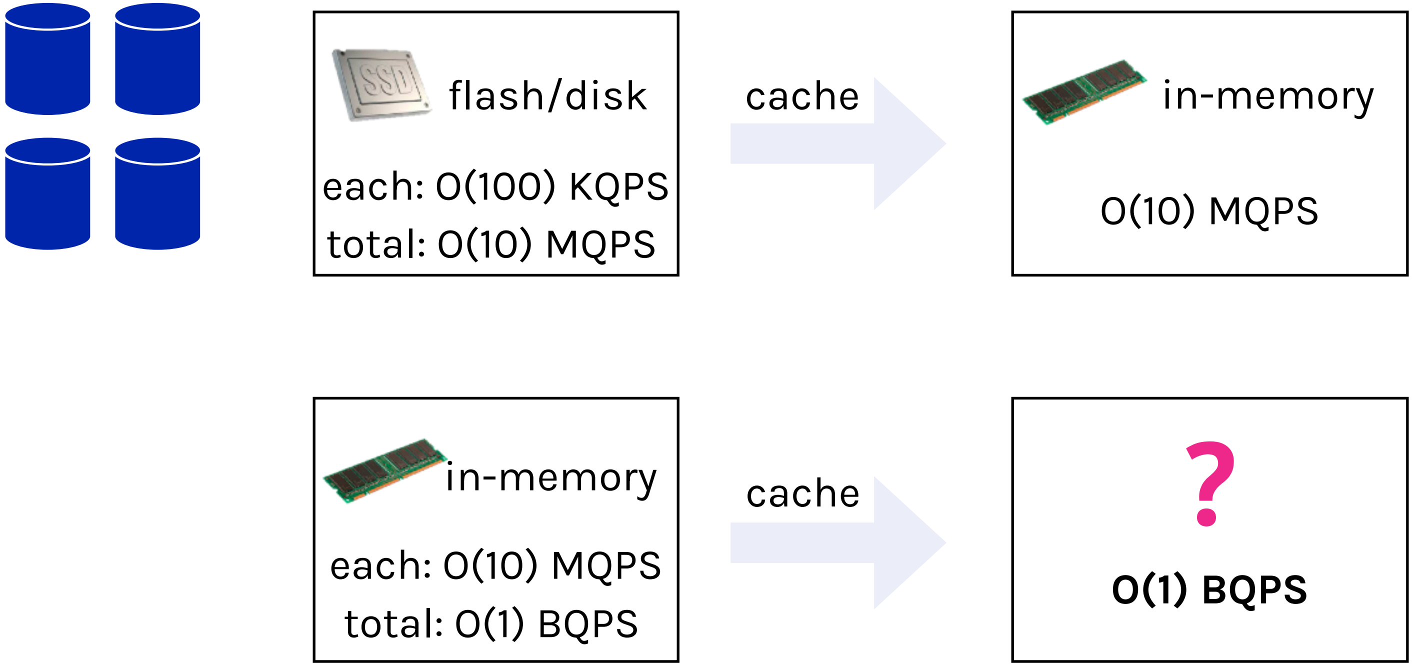
Requirement: cache throughput >= backend aggregate throughput

- Cache not being the performance bottleneck of the system

Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. ACM SoCC, 2011.

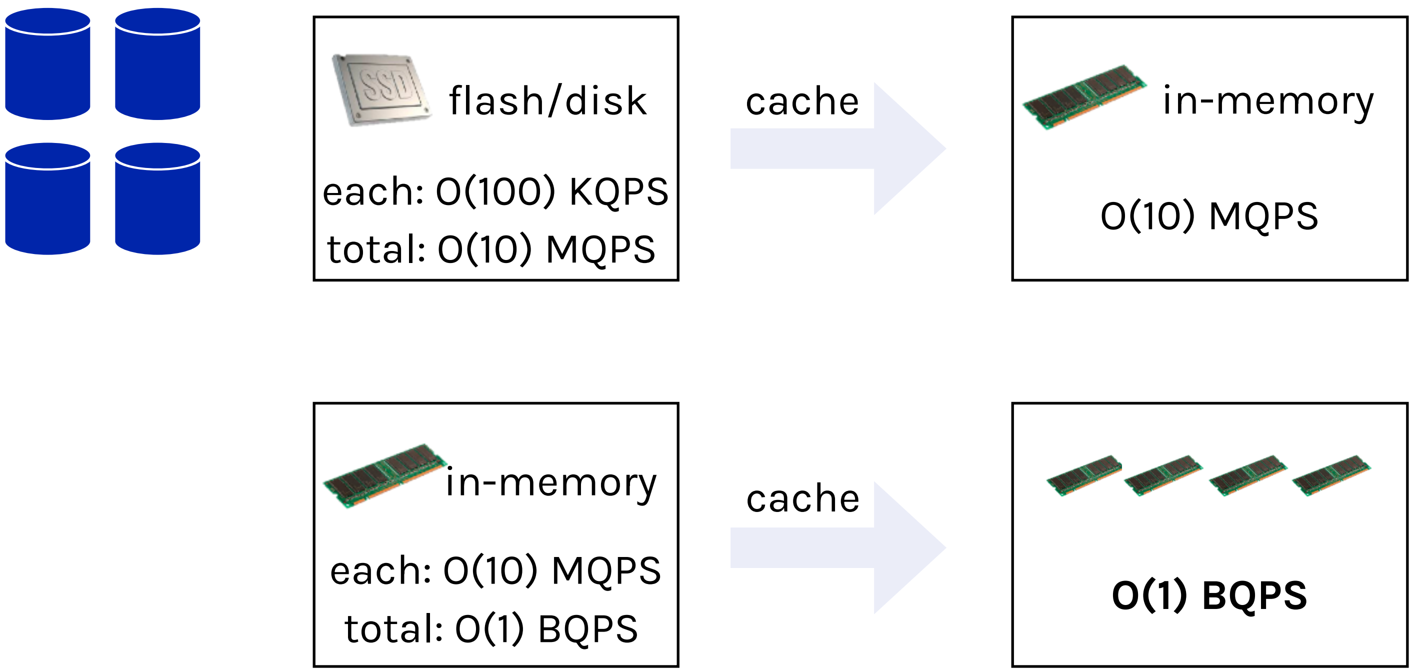
How to build the cache?

Cache needs to provide the aggregate throughput of the storage layer



How to build the cache?

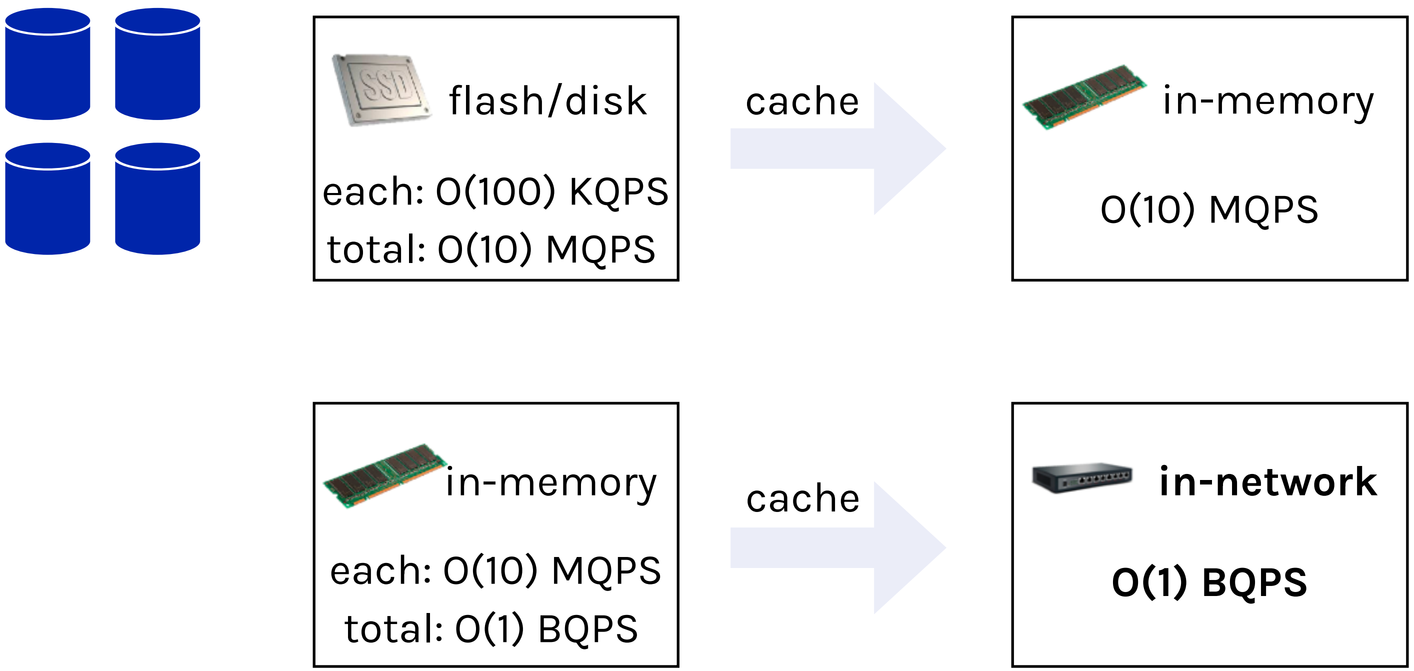
Cache needs to provide the aggregate throughput of the storage layer



Multiple cache servers: (1) high cost, (2) high overhead to ensure cache coherence

How to build the cache?

Cache needs to provide the aggregate throughput of the storage layer



Limited on-chip memory?

But we only need to cache $O(N \log N)$ small items!

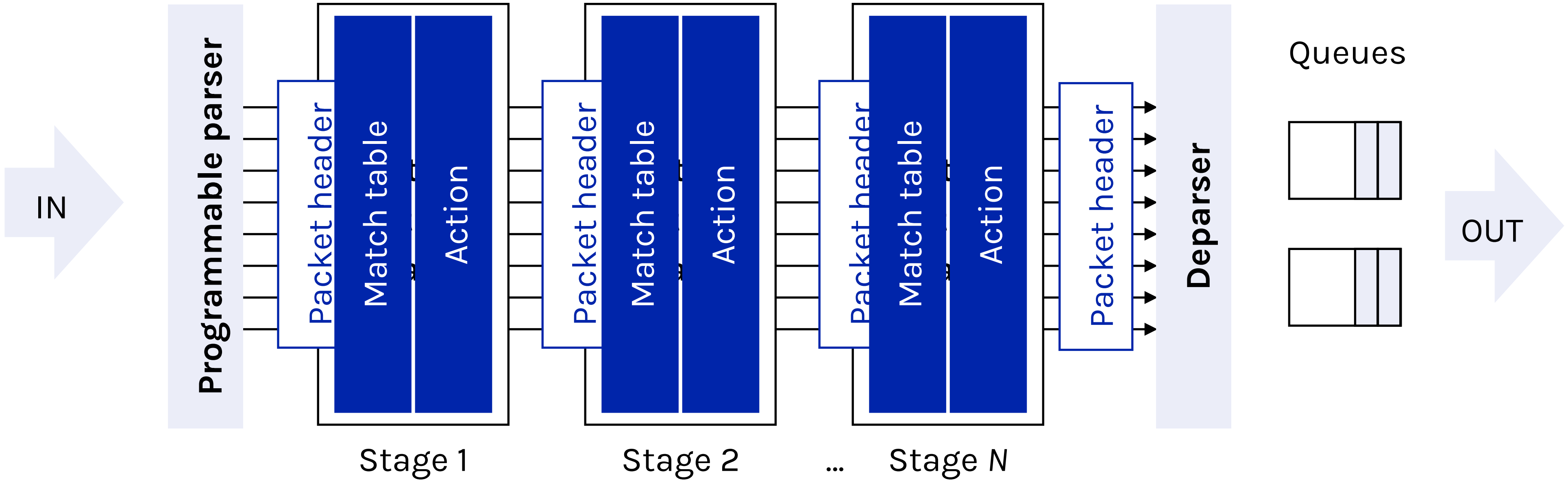
Recall RMT

Programmable parser

- Extracts packet header fields and converts packet data into metadata

Programmable match-action pipeline

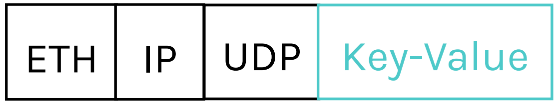
- Operate on packet header vector and metadata, and update memory states



Use RMT for key-value store

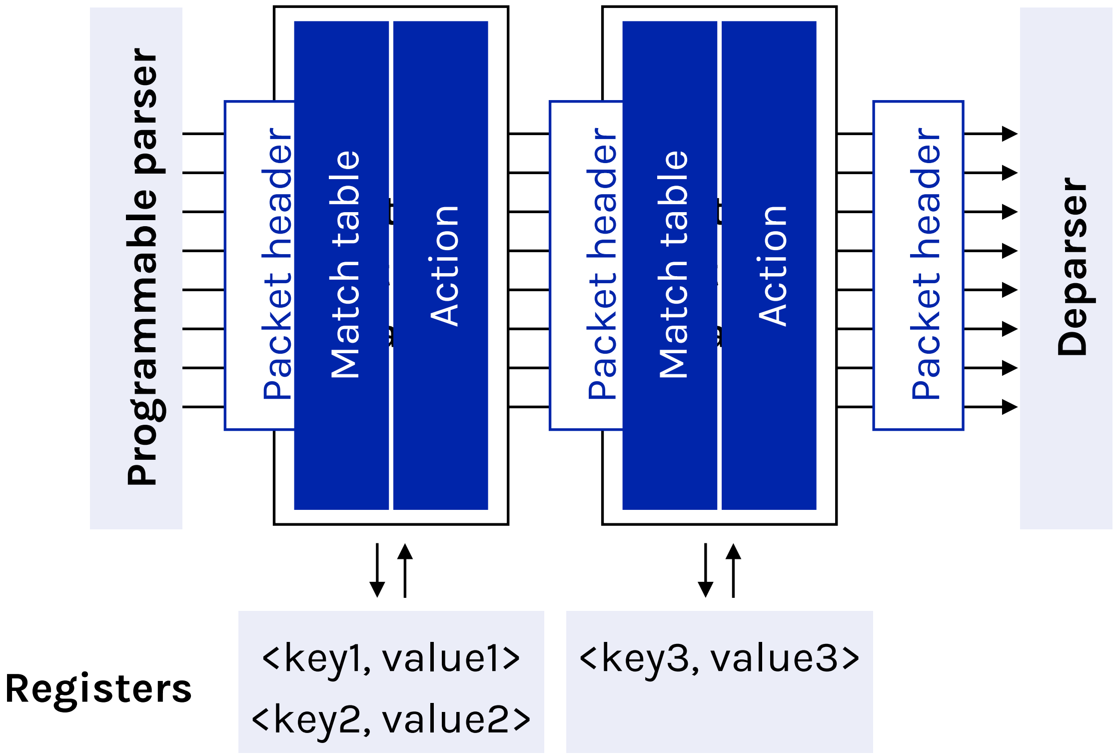
Programmable parser

- Parse custom **key-value fields** in the packet header



Programmable match-action pipeline

- **Read** and **update** key-value data: read(key1), write (key6, value6)
- Provide query **statistics** for cache updates



Cached entries will be updated based on query statistics (e.g., frequency).

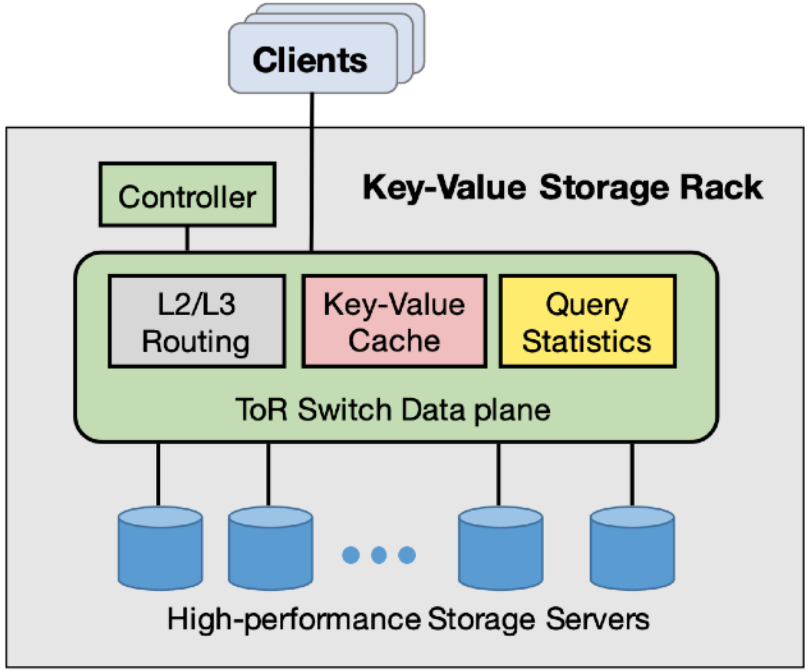
NetCache rack-scale architecture

Switch data plane (fast)

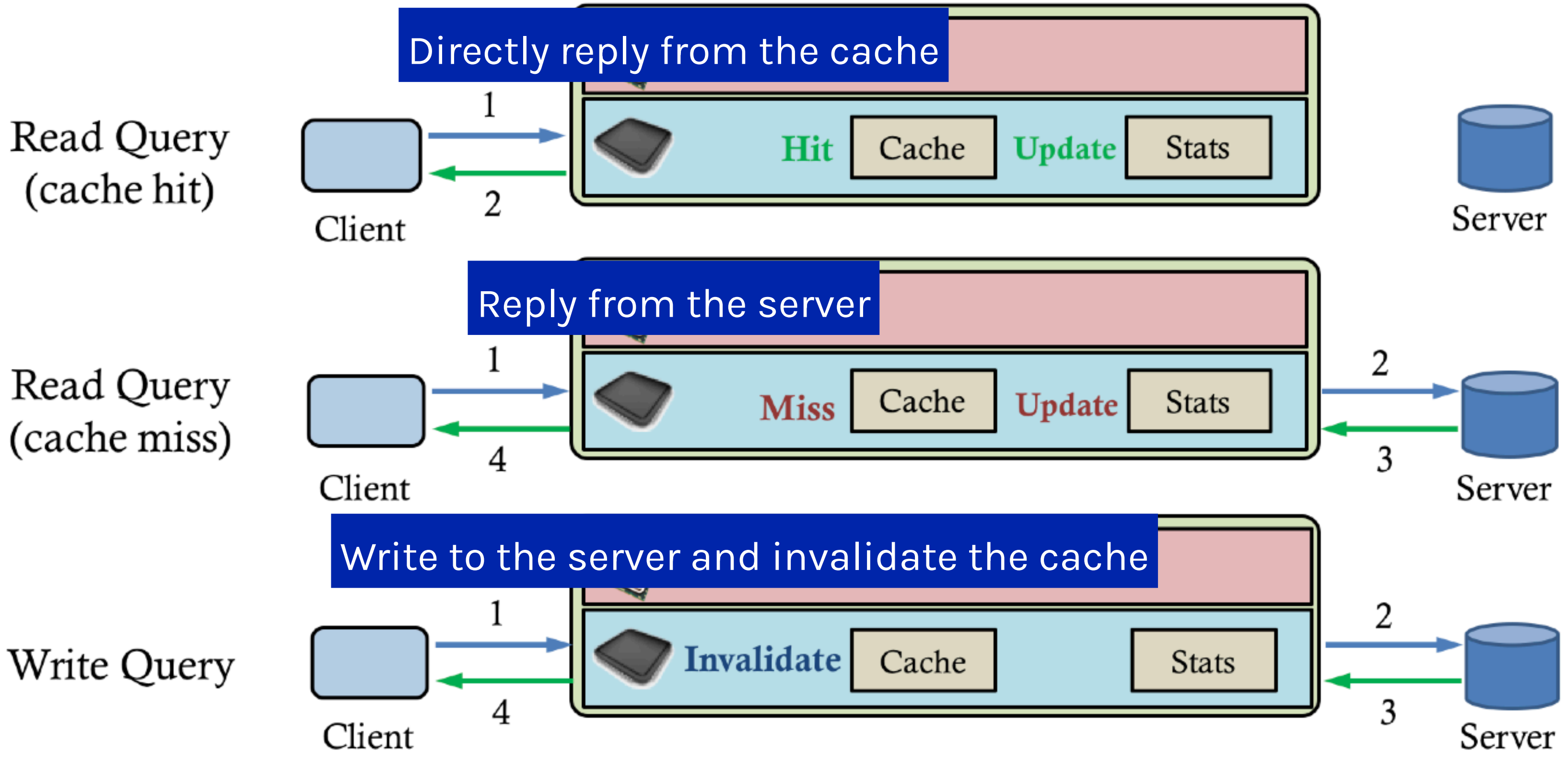
- **Key-value** store to serve queries for cached keys
- **Query statistics** to enable efficient cache updates

Switch control plane (slow)

- **Cache updates:** insert hot items into the cache and evict less popular items
- **Memory management:** memory allocation for on-chip key-value store



Data plane query handling



In-Network Caching

Key-value caching in network ASIC at line rate?!

How to **identify** application-level **packet fields**?

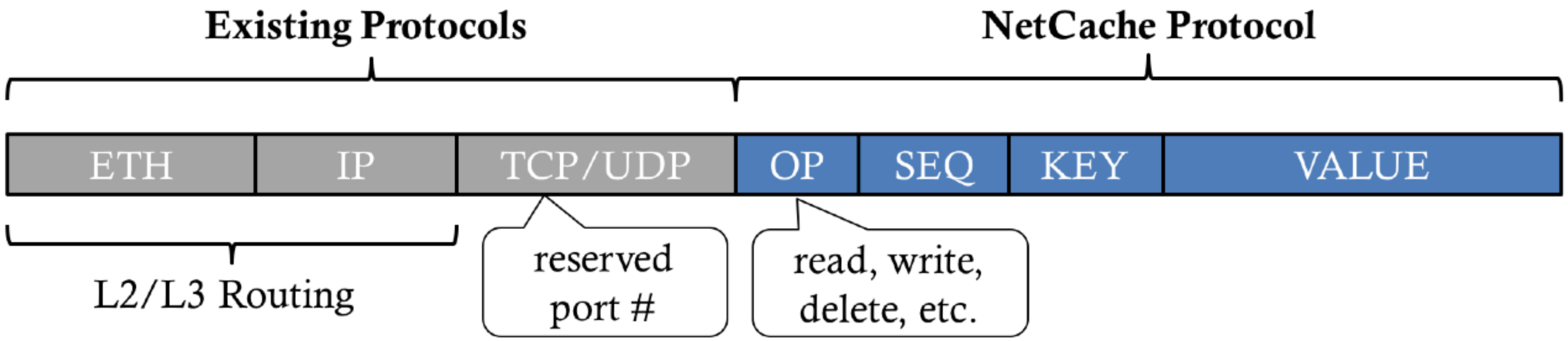
How to store and serve **variable-length data** on switches?

How to efficiently keep the cache **up-to-date**?

NetCache packet format

Application-layer protocol: compatible with existing L2-L4 layers

Only the top-of-rack switch needs to parse NetCache fields



Only the top-of-rack switch needs to parse NetCache fields for NetCache traffic

In-Network Caching

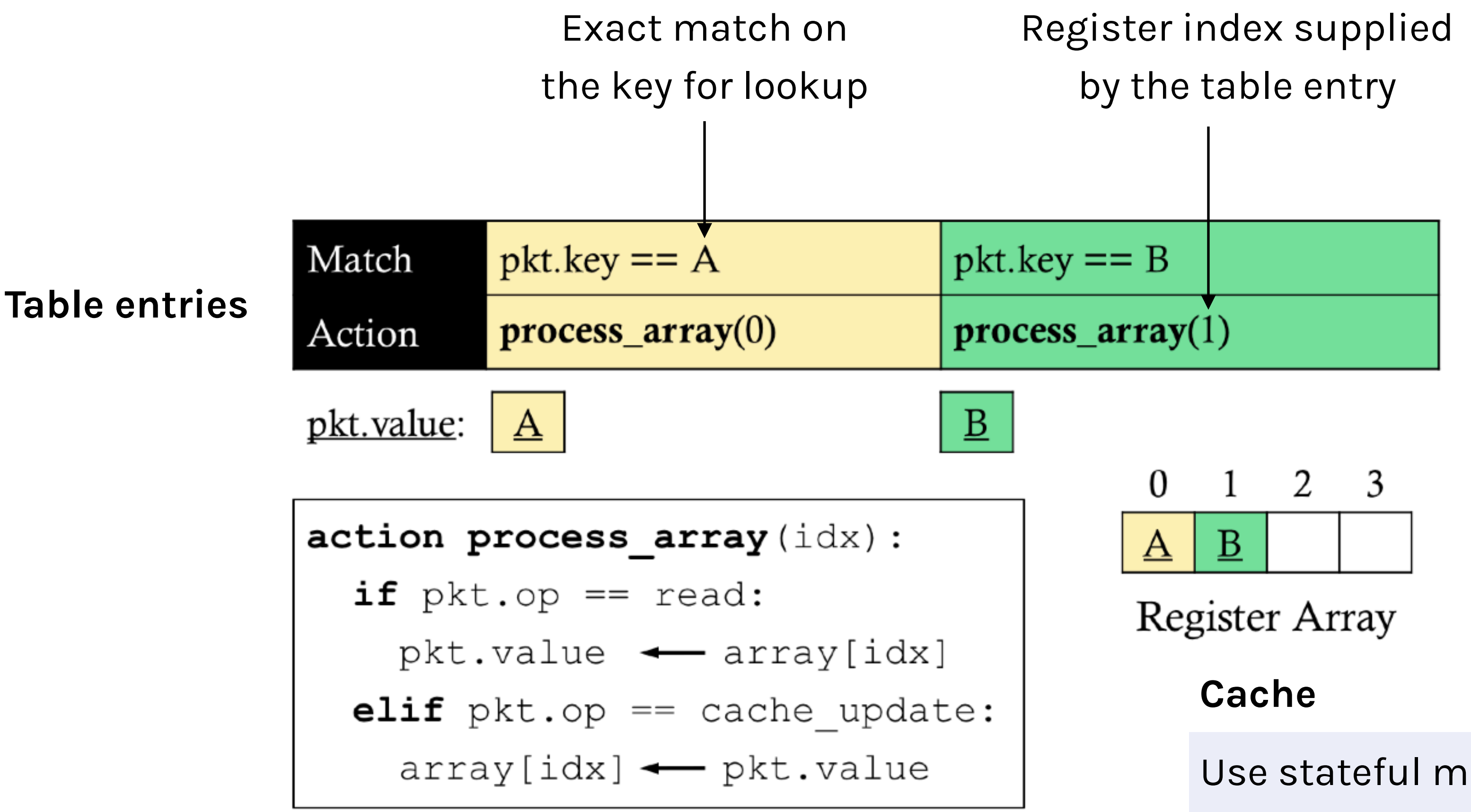
Key-value caching in network ASIC at line rate?!

How to **identify** application-level **packet fields**?

How to store and serve **variable-length data** on switches?

How to efficiently keep the cache **up-to-date**?

Use register array



Cache

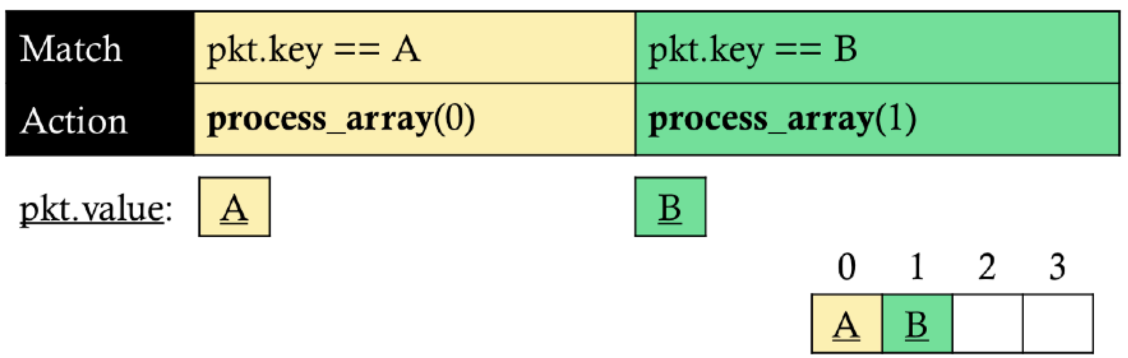
Use stateful memory to store key-value items

Challenges with variable length

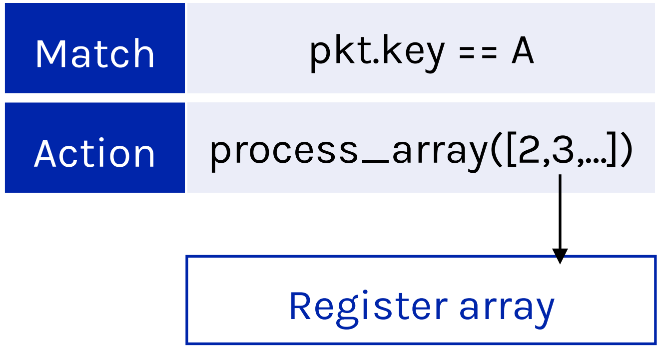
No loop or string in P4 due to strict timing requirements

Need to optimize hardware resources consumption

- Number of entries in the match-action tables
- Size of action data given by a table match
- Size of intermediate metadata across tables (in case of using multiple tables)



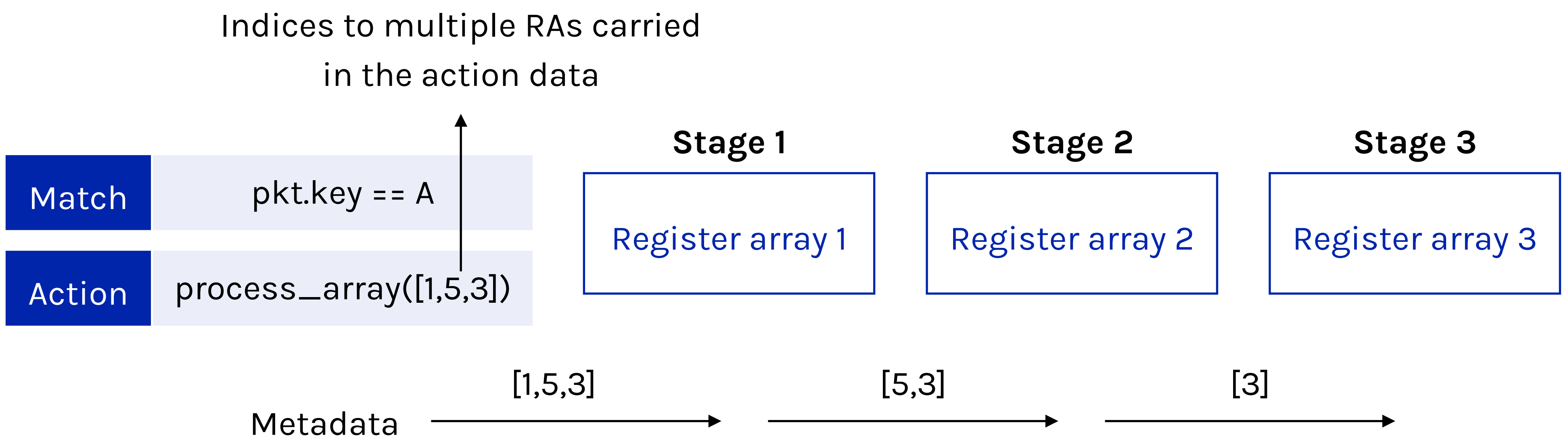
Use one register array



Use action data to hold the indices

Problem: not feasible due to limited number of lookups in one register array (read only one index allowed), high action data

Use multiple register arrays (RAs)



Problem: high action data and metadata

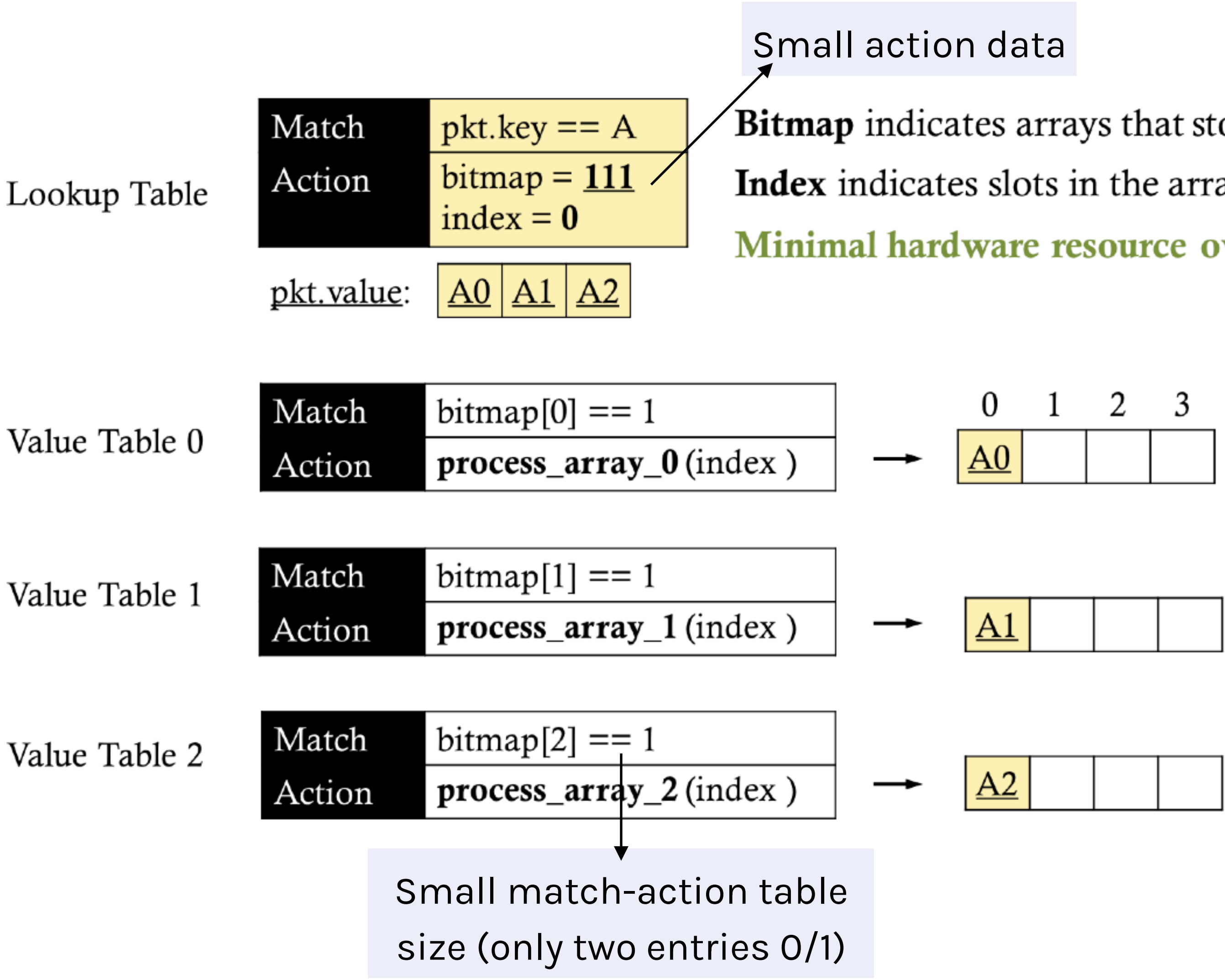
Use multiple register arrays (RAs)

Use multiple tables to provide indices for the RAs

	Stage 1	Stage 2	Stage 3
Match	pkt.key == A	pkt.key == A	pkt.key == A
Action	process_array1(1)	process_array2(5)	process_array3(3)
	Register array 1	Register array 2	Register array 3

Problem: too many match action table entries

NetCache: two-level lookup

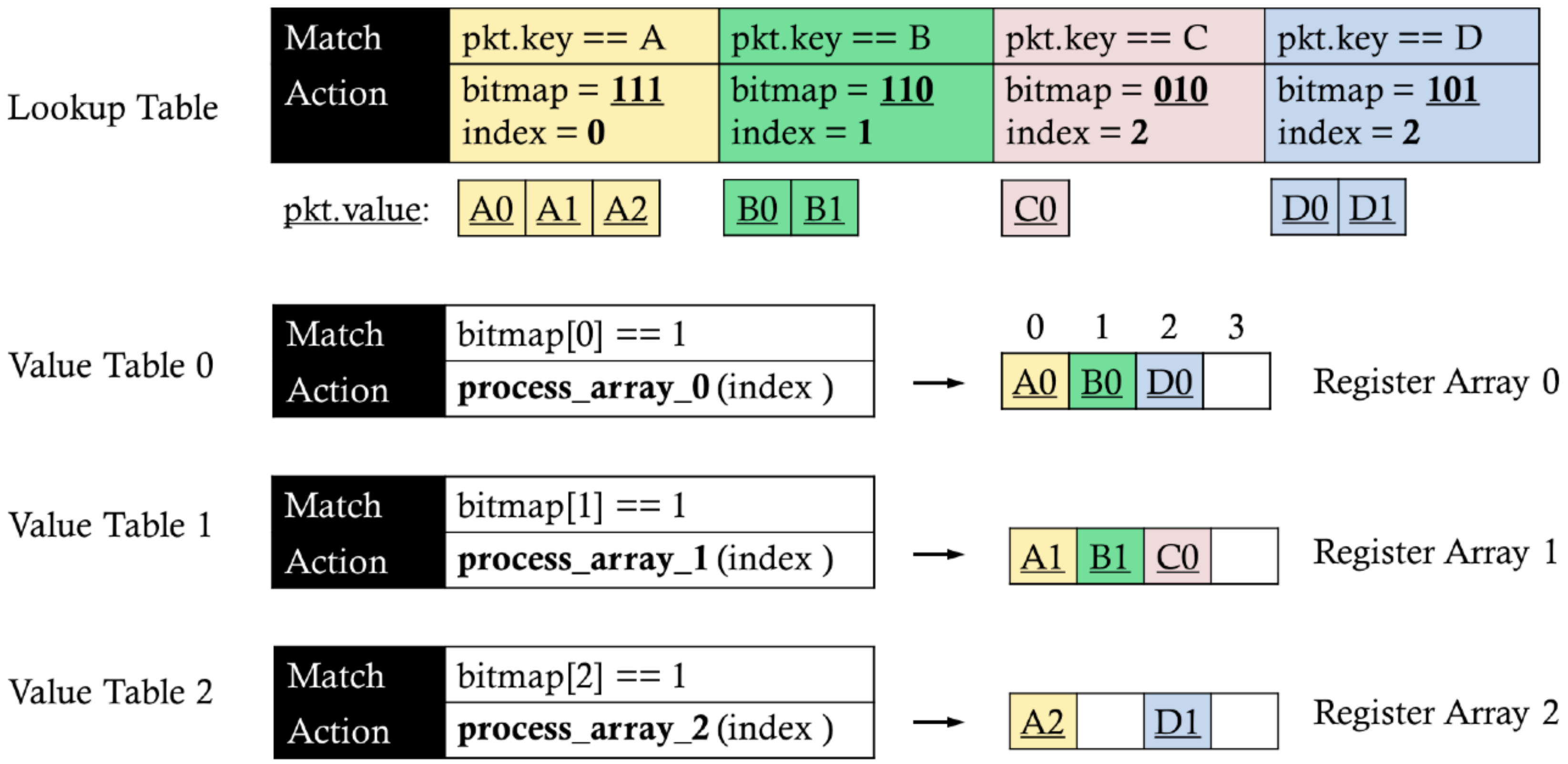


Bitmap indicates arrays that store the key's value

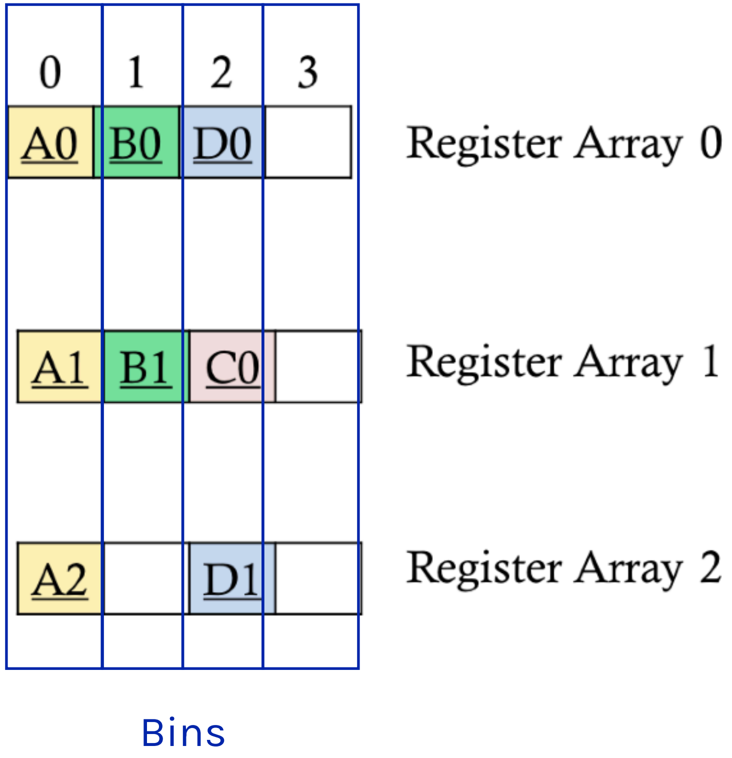
Index indicates slots in the arrays to get the value

Minimal hardware resource overhead

Combine outputs from multiple arrays



Memory management



Solve a bin-packing problem: use First-Fit heuristics

In-Network Caching

Key-value caching in network ASIC at line rate?!

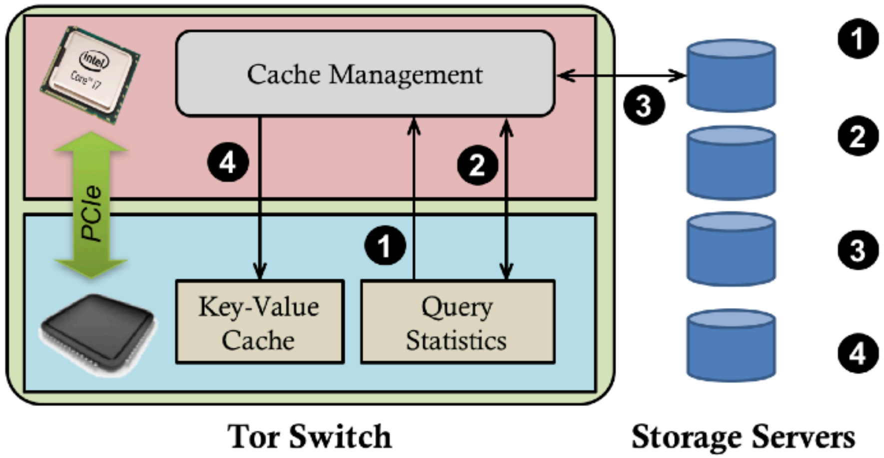
How to **identify** application-level **packet fields**?

How to store and serve **variable-length data** on switches?

How to efficiently keep the cache **up-to-date**?

Cache insertion and eviction

Goal: react quickly and effectively to workload changes with minimal updates



1. Data plane reports hot keys
2. Control plane compares loads of new hot and sampled cached keys
3. Control plane fetches values for keys to be inserted to the cache
4. Control plane inserts and evicts keys

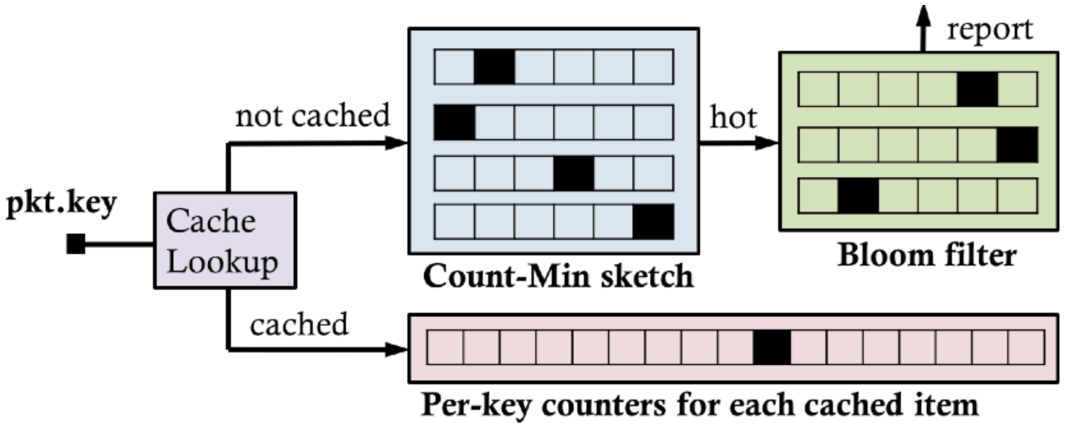
Challenge: cache the hottest $O(N \log N)$ items with limited insertion rate

Query statistics

Cached key (small in size): per-key counter array

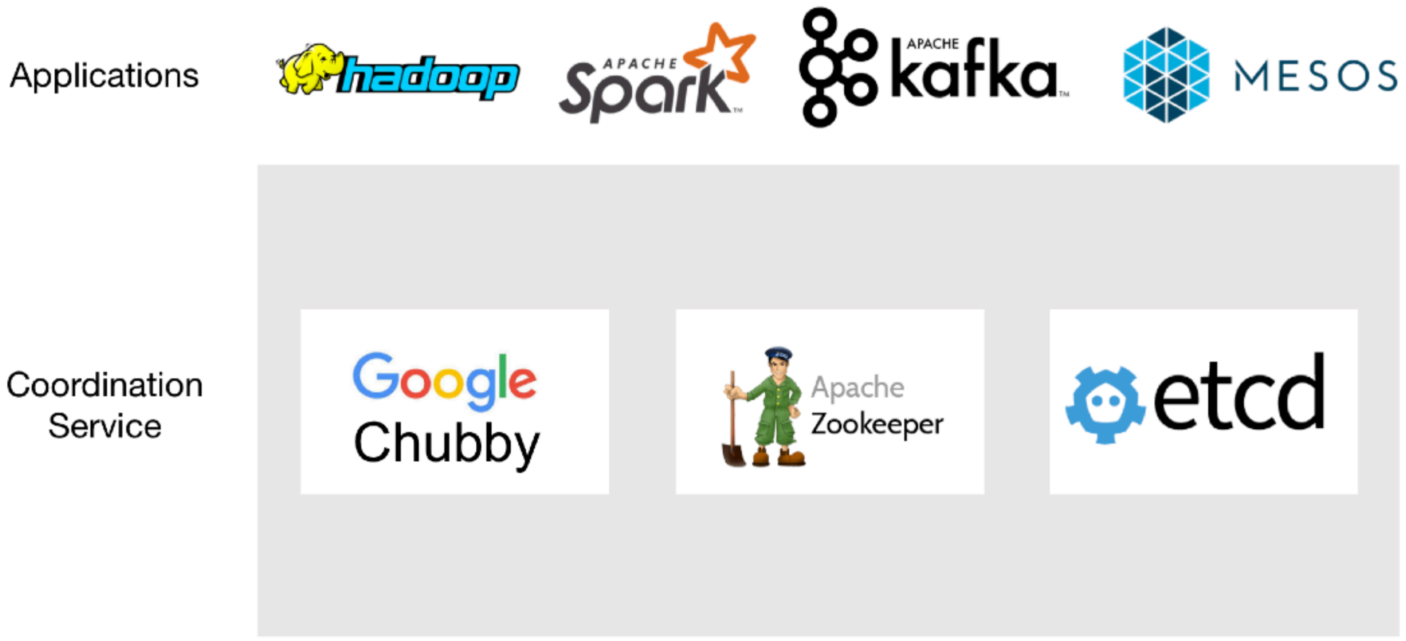
Uncached key (large in size):

- Count-min sketch (an approximate data structure): report new hot keys
- Bloom filter: remove duplicated hot key reports

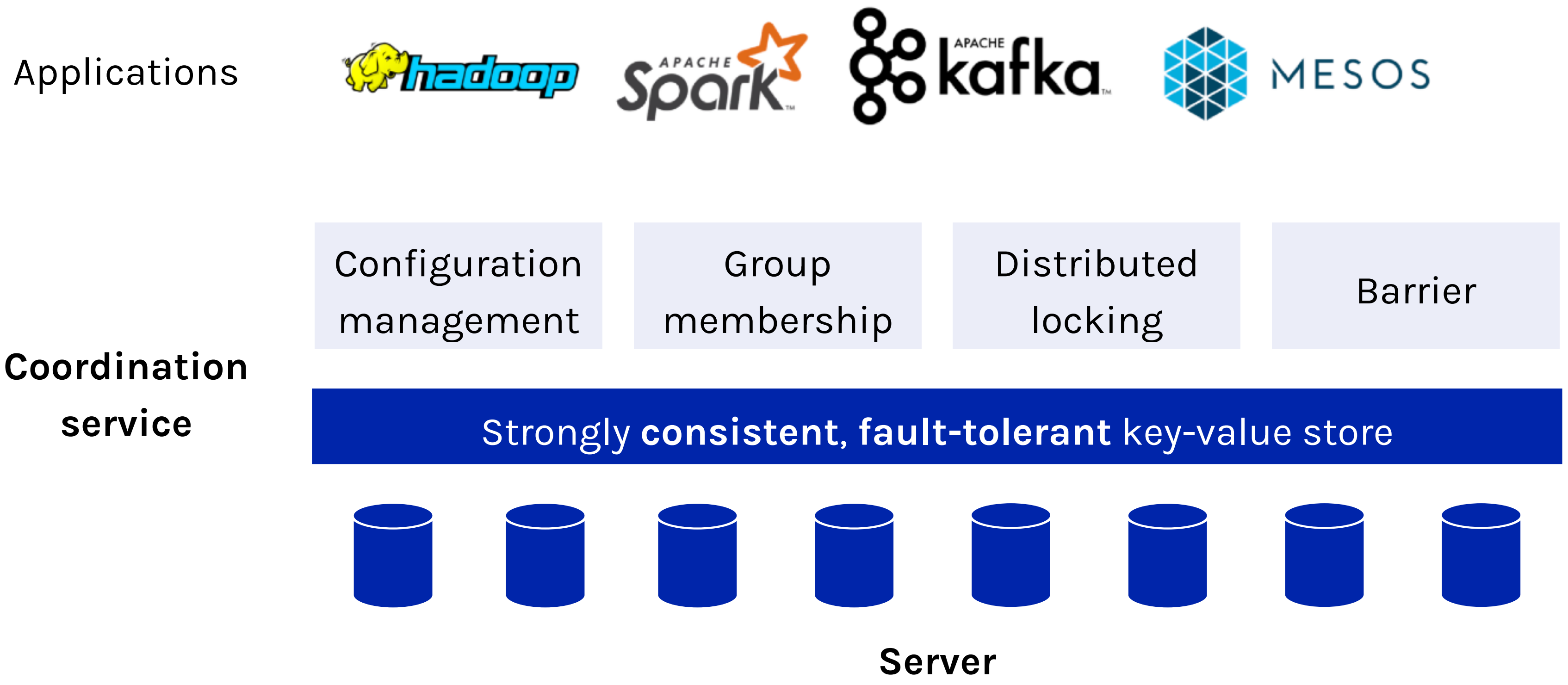


How to implement an in-network coordination service?

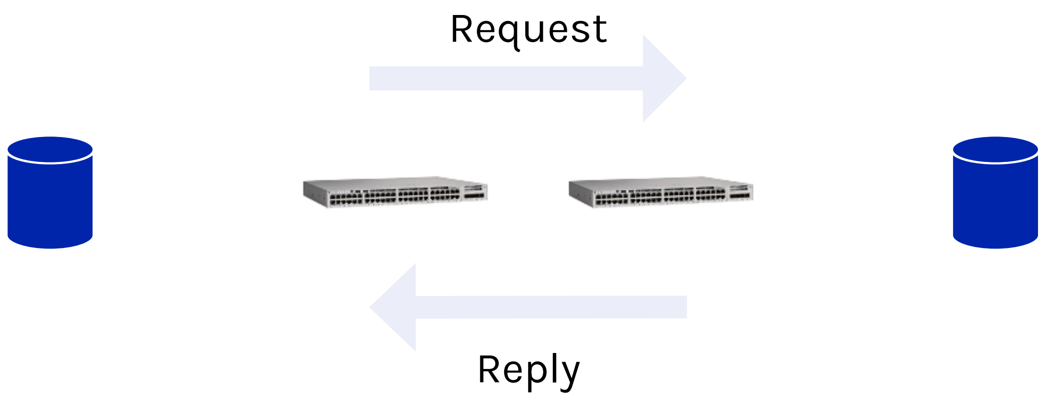
Coordination service



Coordination service



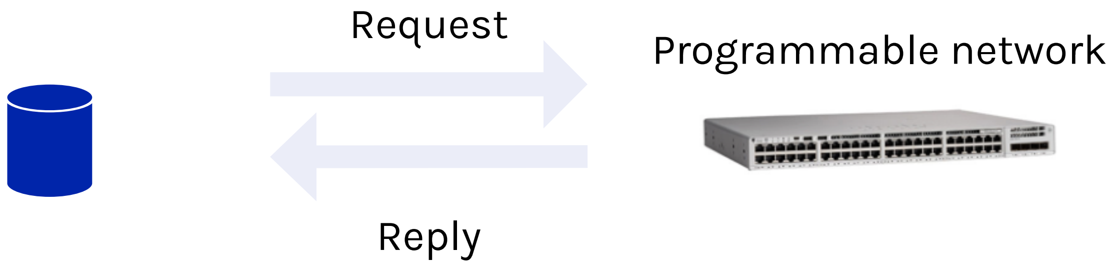
Workflow of coordination service



Throughput: limited by the server NIC throughput

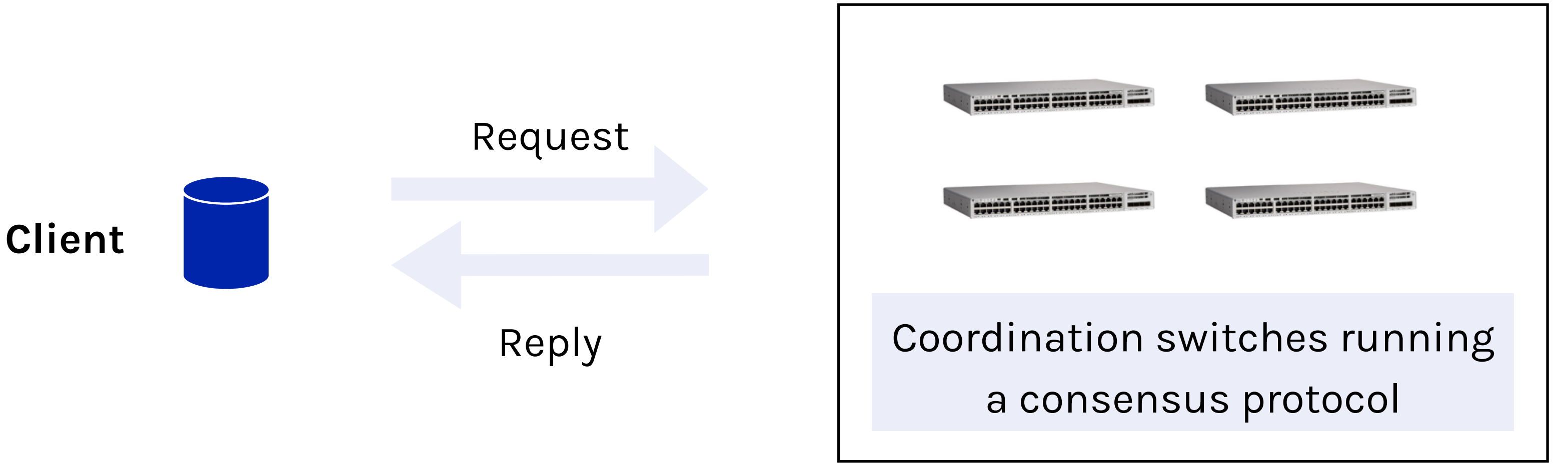
Latency: at least one RTT, typically a few RTTs

In-network coordination



	Server	Switch
Example	[NetBricks, OSDI'16]	**Barefoot Tofino**
Packets per second	30 million	**A few billion**
Bandwidth	10-100 Gbps	**6.5 Tbps**
Processing delay	10-100 us	**< 1 us**

In-network coordination



Throughout: switch throughput

Latency: sub-RTT

NetChain design goals

High throughput

Low latency

Consistency

Fault tolerance

Already satisfied with the high-performance switches

How to?

NetChain design goals

High throughput

Low latency

Consistency

Fault tolerance

Already satisfied with the high-performance switches

Vertical Paxos

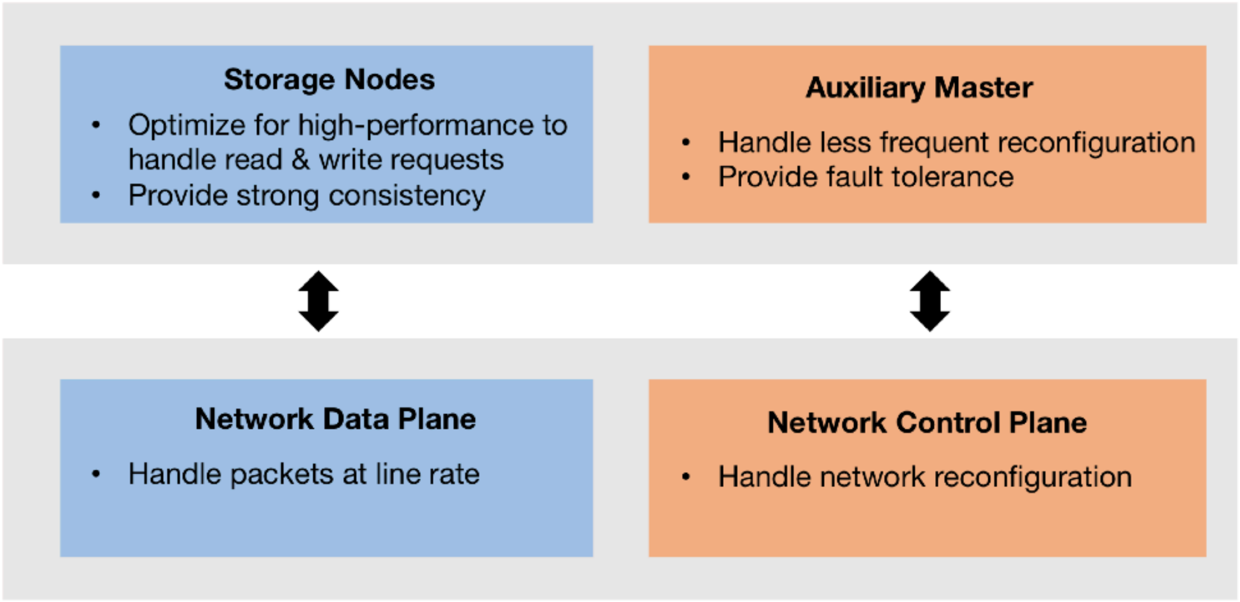
NetChain division of labor

Steady state protocol (primary-backup)

Reconfiguration protocol (fault-tolerance)

Vertical Paxos

Programmable network

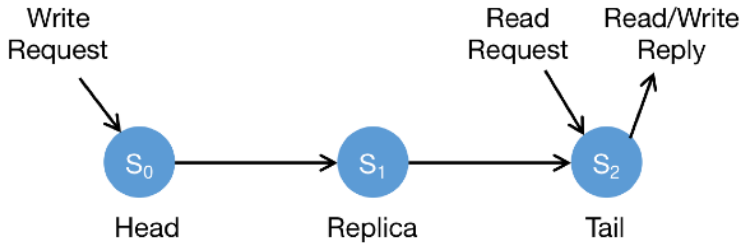


Chain replication for the steady state protocol

Nodes are organized in a chain structure

Handle operations:

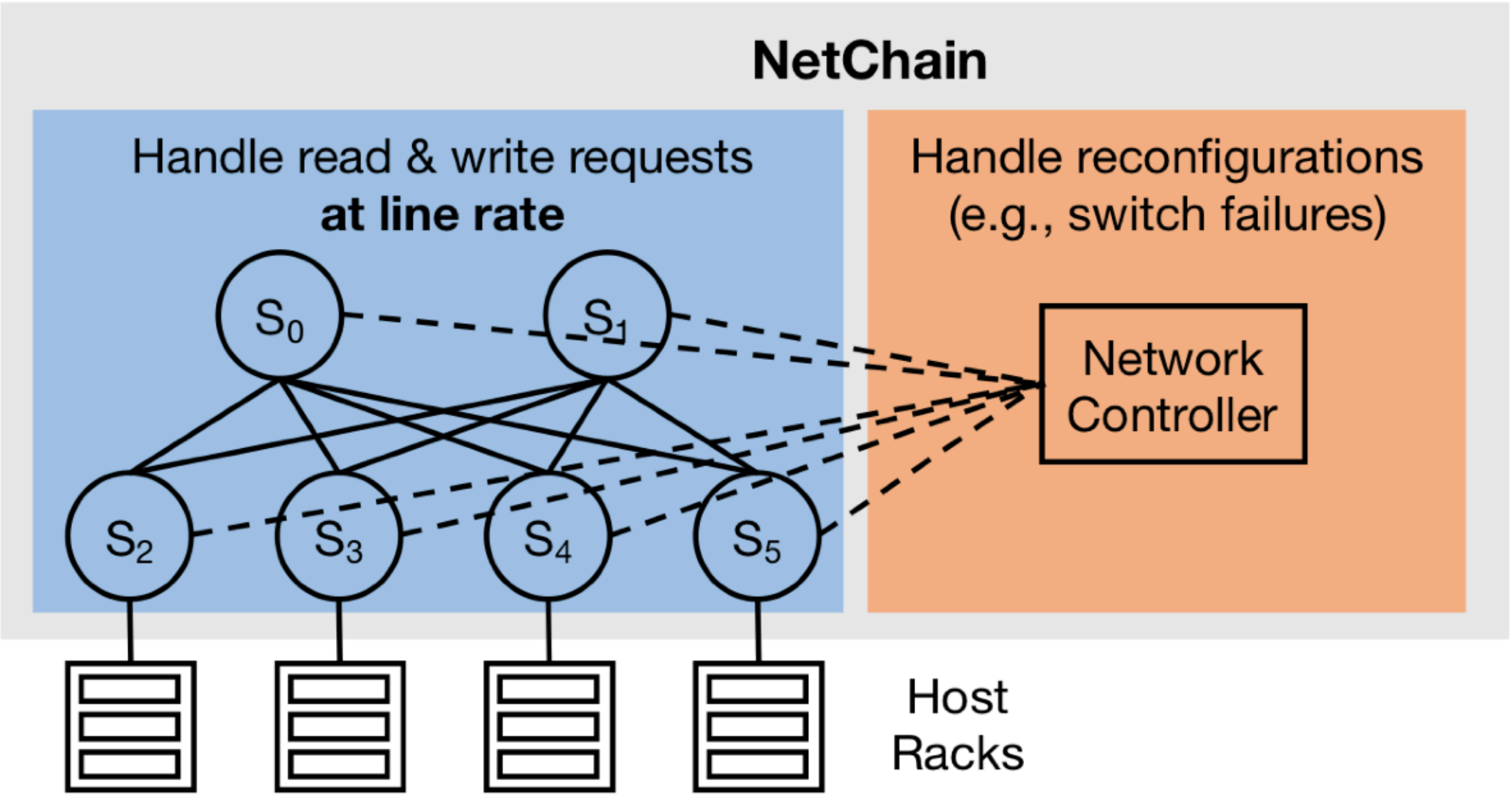
- Read from the tail
- Write from head to tail



Provide strong consistency and fault tolerance

- Tolerate f failures with $f + 1$ nodes
- Fault tolerance based on the reconfiguration protocol in Vertical Paxos

NetChain overview

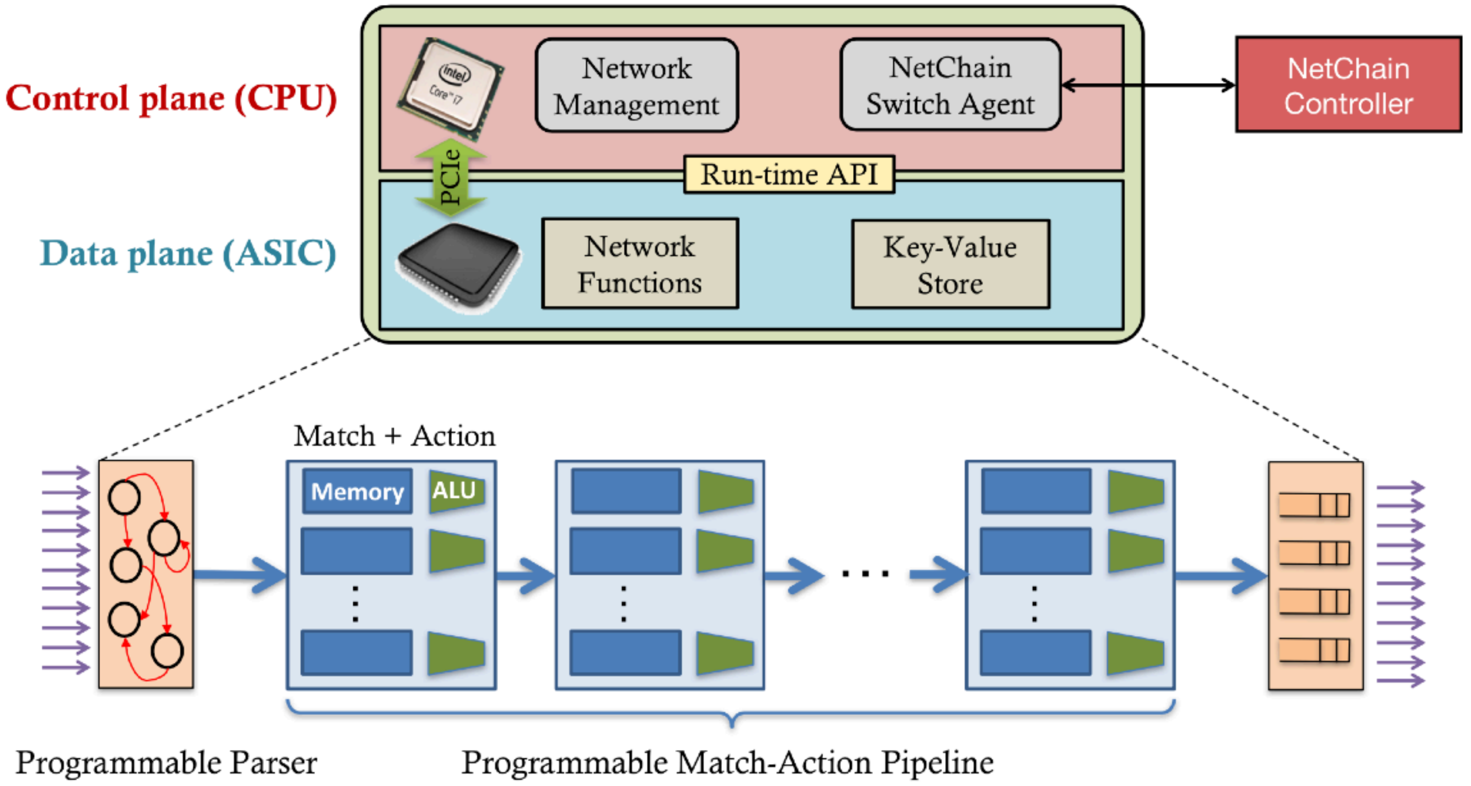


NetChain challenges

Data plane			Control plane
How to store and serve key-value items?	How to route queries according to the chain structure?	How to handle out of order delivery in the network?	How to handle switch failures?

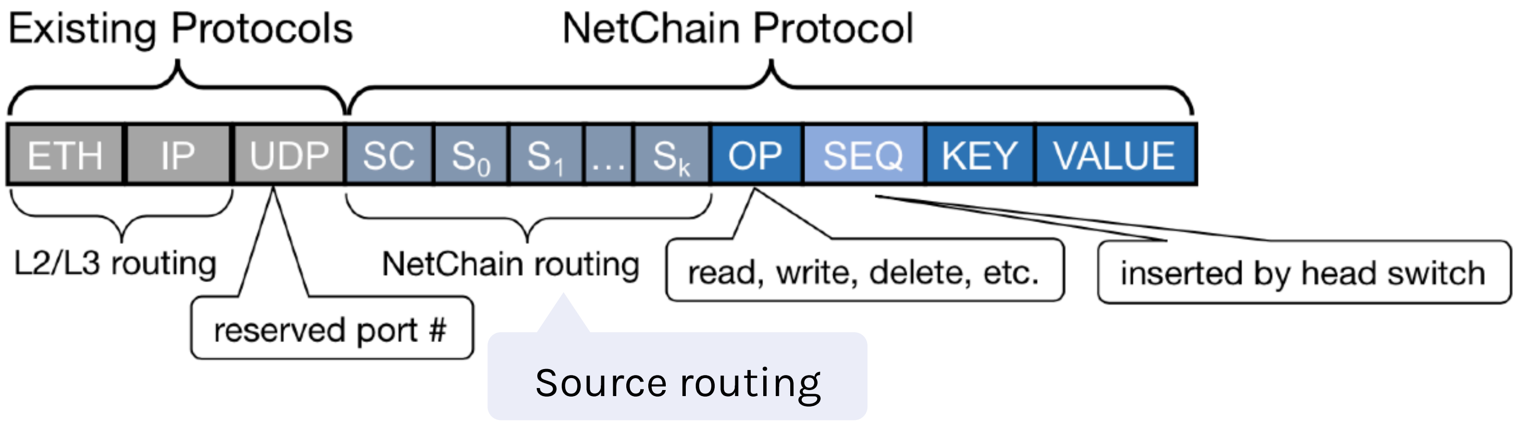
NetChain switch design

Similar to NetCache, except the coordination components



NetChain packet format

Application-layer protocol: compatible with existing L2-L4 layers, invoked with a reserved UDP port

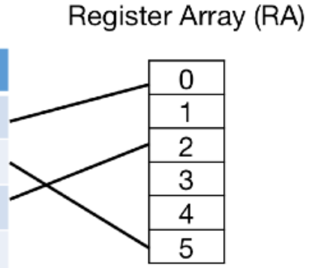


UDP is not reliable: upon packet loss, retry! Designing a **reliable transport** protocol for in-network computing is still an open challenge!

In-network key-value storage

Match-Action Table

Match	Action
Key = X	Read/Write RA[0]
Key = Y	Read/Write RA[5]
Key = Z	Read/Write RA[2]
Default	Drop()



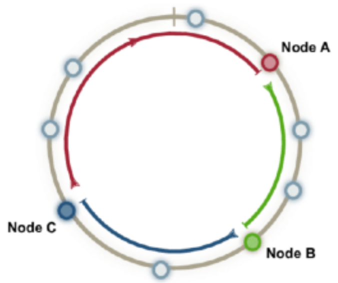
Key-value store in a single switch

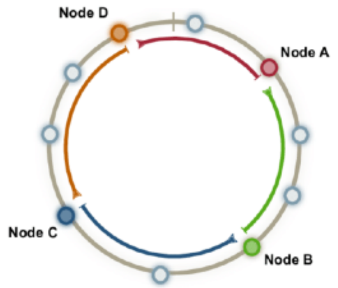
- Store and serve key-value items using register arrays

Key-value store in the network

- Data partitioning with consistent hashing and virtual nodes

Use a hash function to hash both the virtual nodes and the keys to a ring

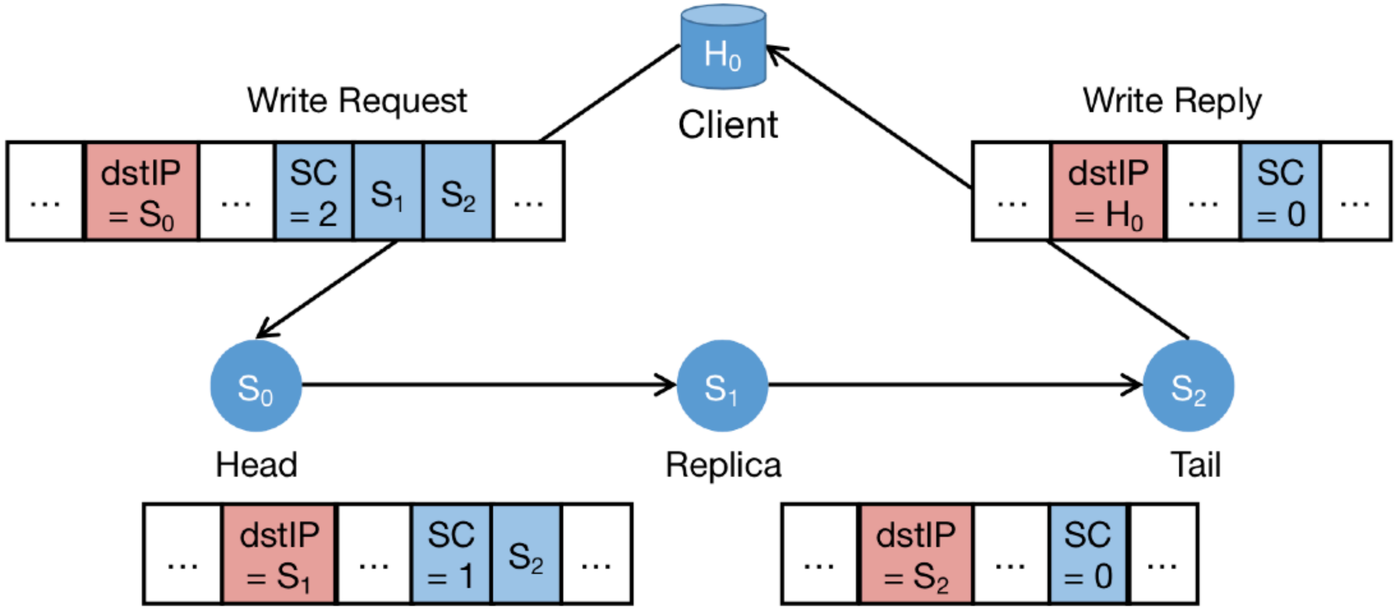




Virtual nodes are mapped to physical nodes (switches) with load balance; keys assigned to a virtual node are replicated on $f+1$ virtual nodes that do not share physical nodes.

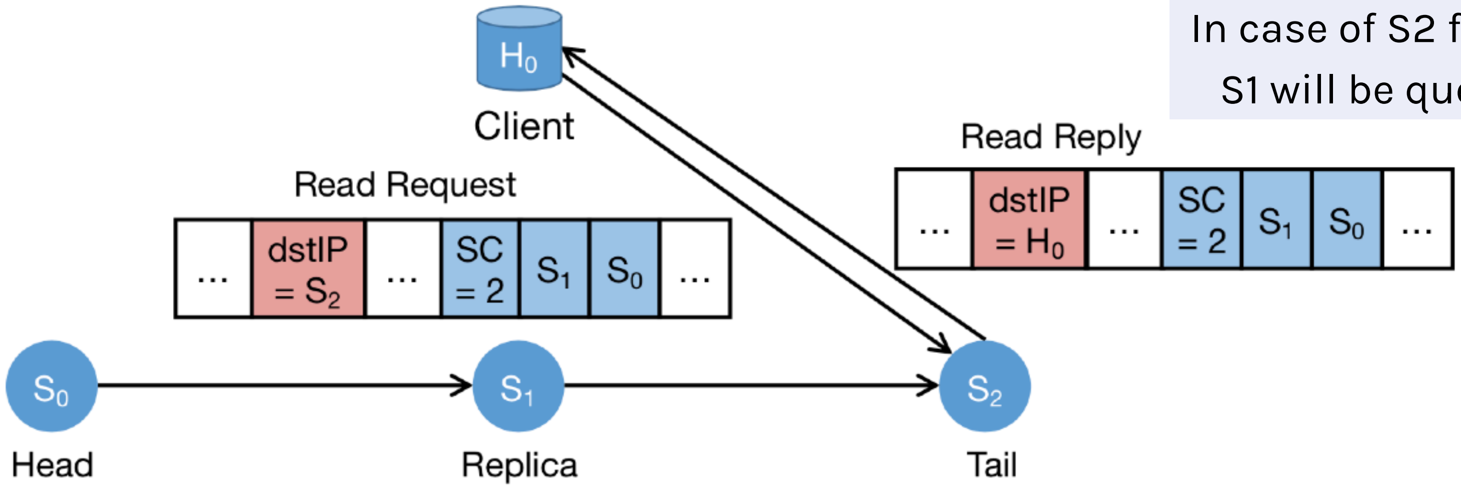
NetChain routing - write requests

Segment routing according to the chain structure



Write from the head and update through the chain until the tail

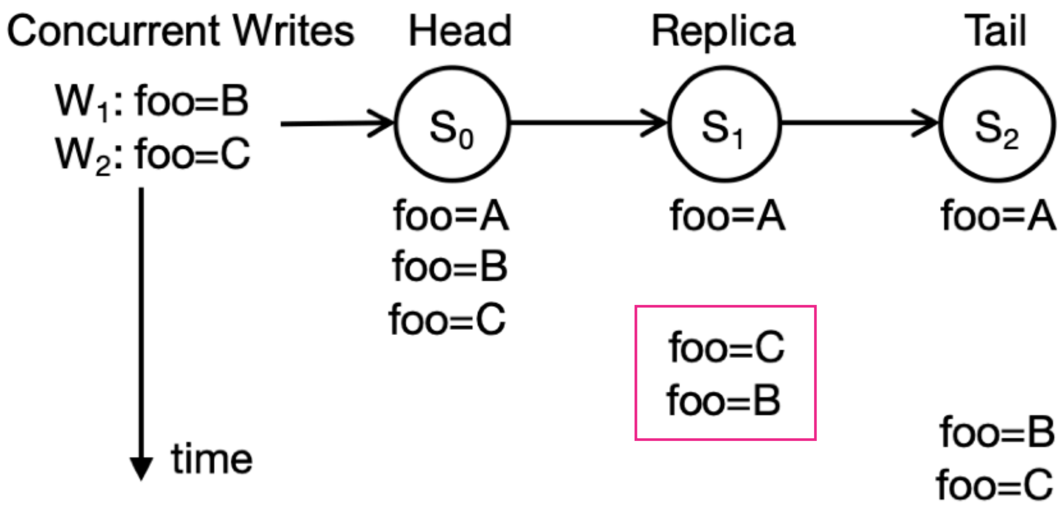
NetChain routing - read requests



In case of S2 failure,
S1 will be queried

Always read from the tail

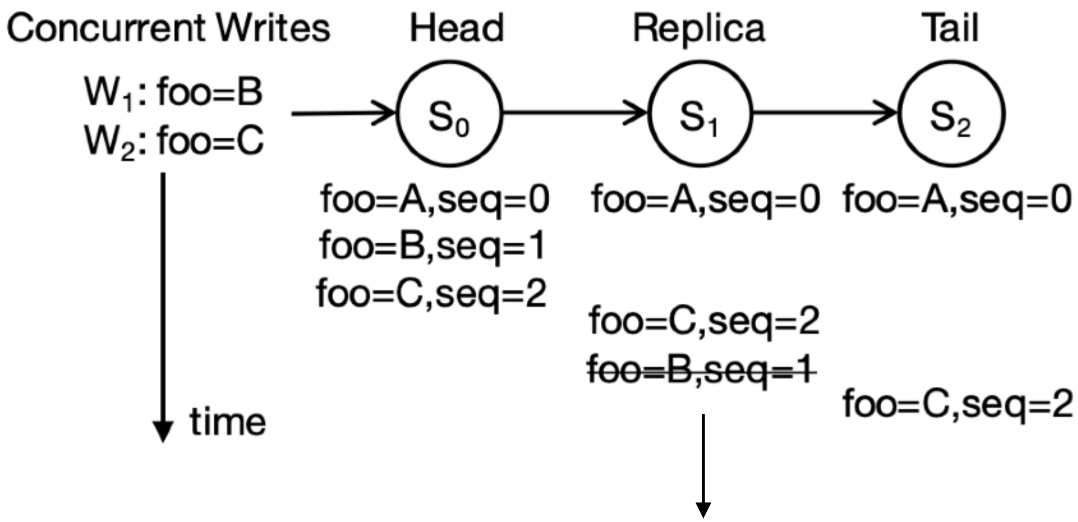
NetChain out of order delivery



W2 arrived before W1: out of order delivery on the network → **inconsistent state**

NetChain out of order delivery

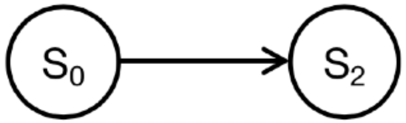
Serialization with sequence number!



Obsolete updates (with lower sequence numbers) will be dropped

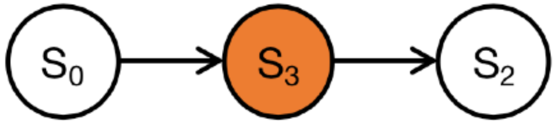
Handling switch failures

Fast failover



- Failover to remaining f nodes
- Tolerate $f-1$ failures
- Efficiency: only need to update neighbor switches of failed switch

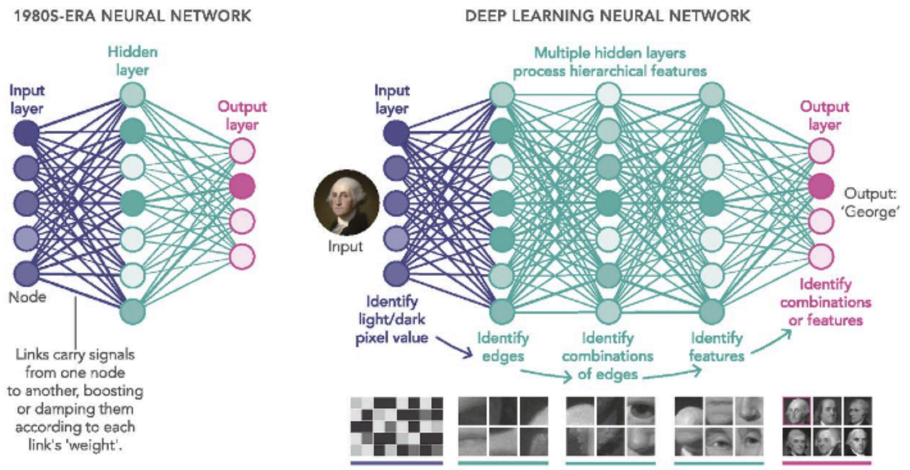
Failure recovery



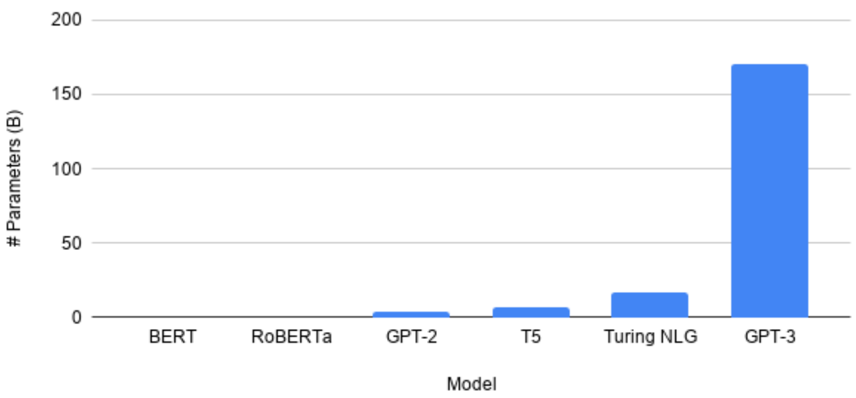
- Add another switch
- Tolerate $f+1$ failures again
- Consistency: two-phase atomic switching
- Minimize disruption: virtual groups

Using in-network computing to accelerate distributed machine learning

Machine learning in data centers

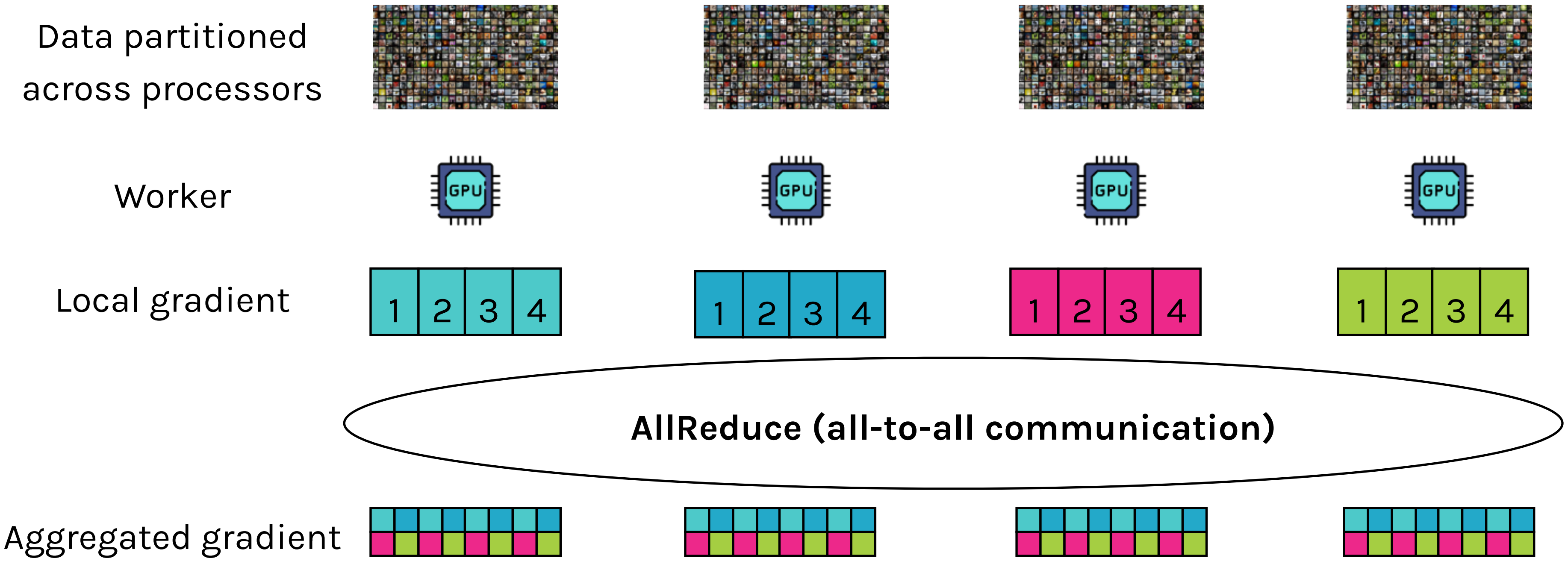


Modern DNNs consist of up to hundreds of layers



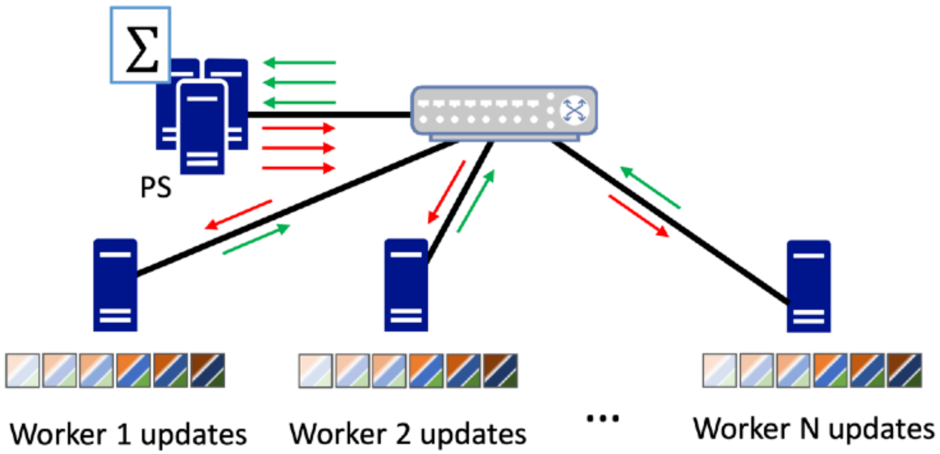
Typically billions of parameters

Data-parallel distributed machine learning

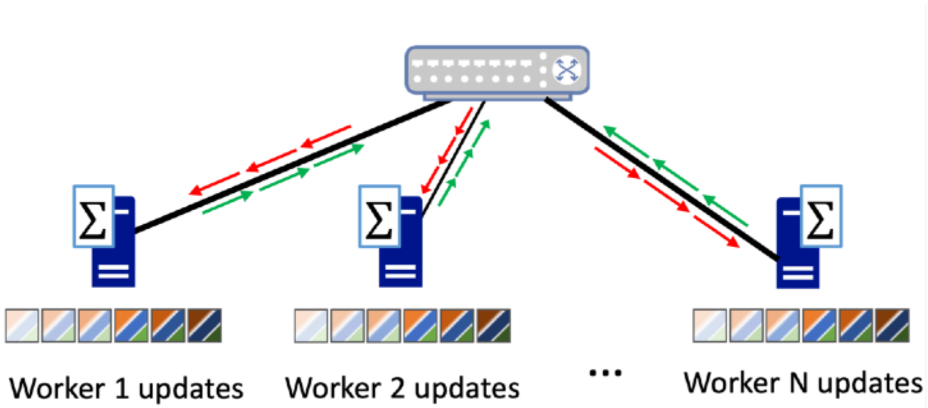


Aggregation (100s of MBs to GBs) has to be performed in every iteration →
Network becomes the bottleneck in the training speed!

Two existing approaches



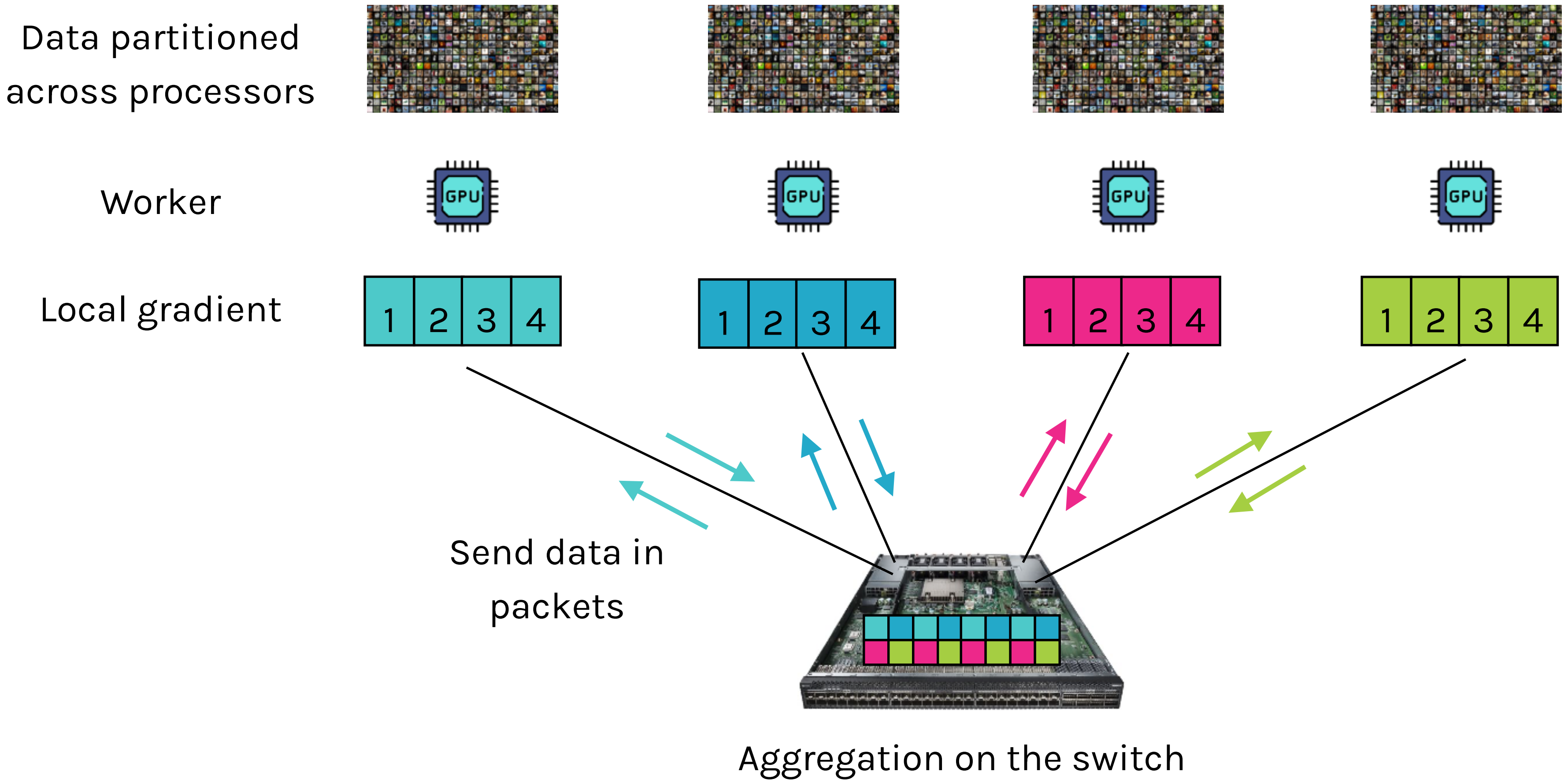
Parameter server (PS)



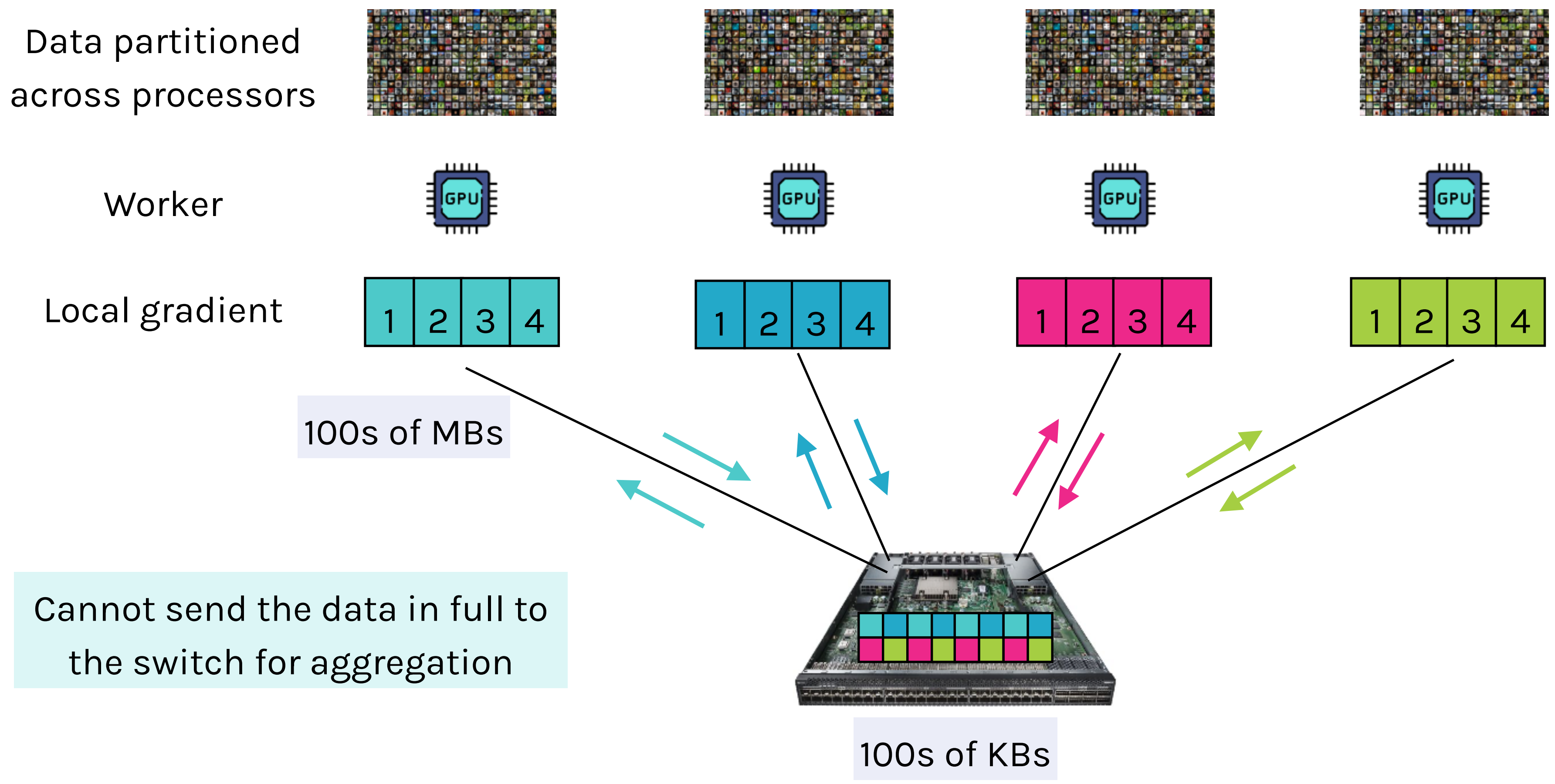
AllReduce (ring)

The network remains a performance bottleneck in scaling distributed machine learning.

Turn the network into an ML accelerator

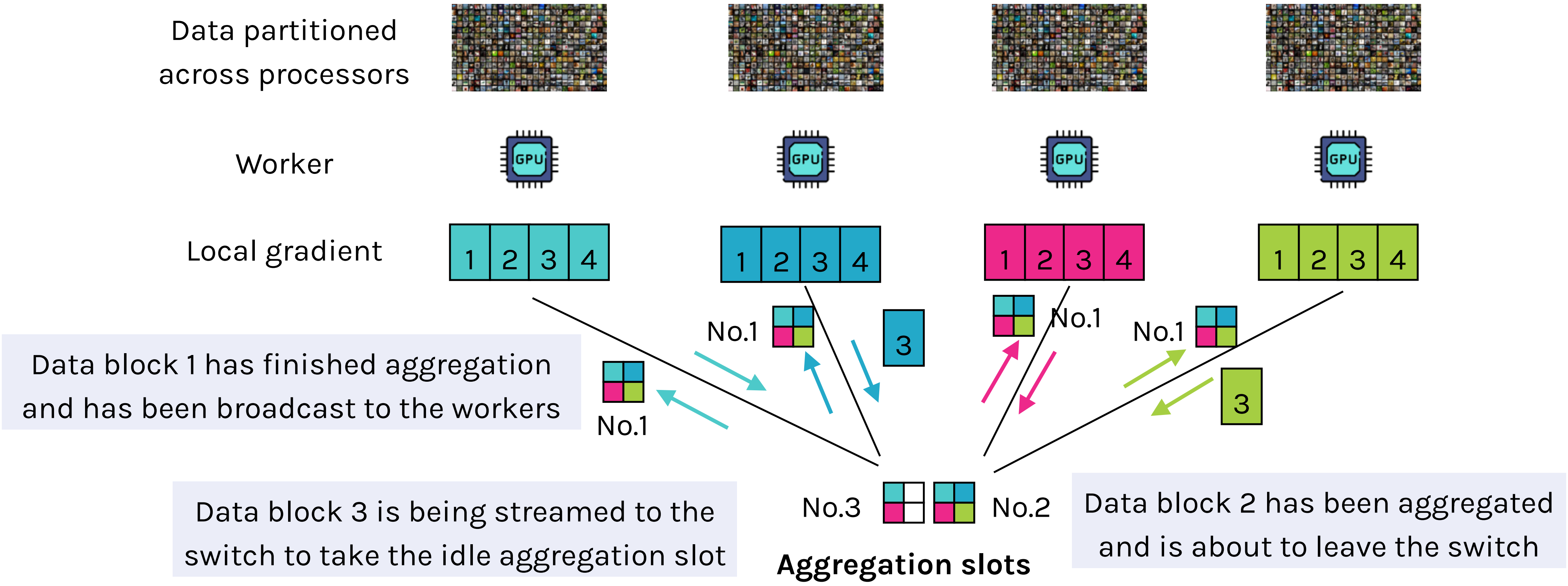


Challenges



Cannot send the data in full to the switch for aggregation

Streaming aggregation



Questions to think about

How to craft packets to send to the switch for SwitchML?

How much data to send in each packet?

How to perform aggregation on the switch?

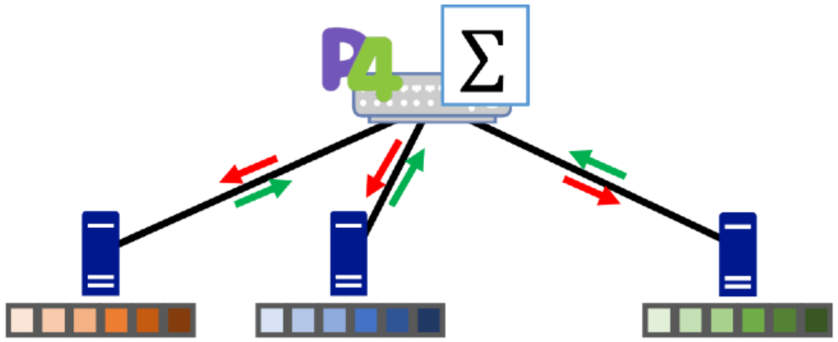
- Respecting the Tofino switch register access restrictions

How to ensure reliability?

- Handling packet loss

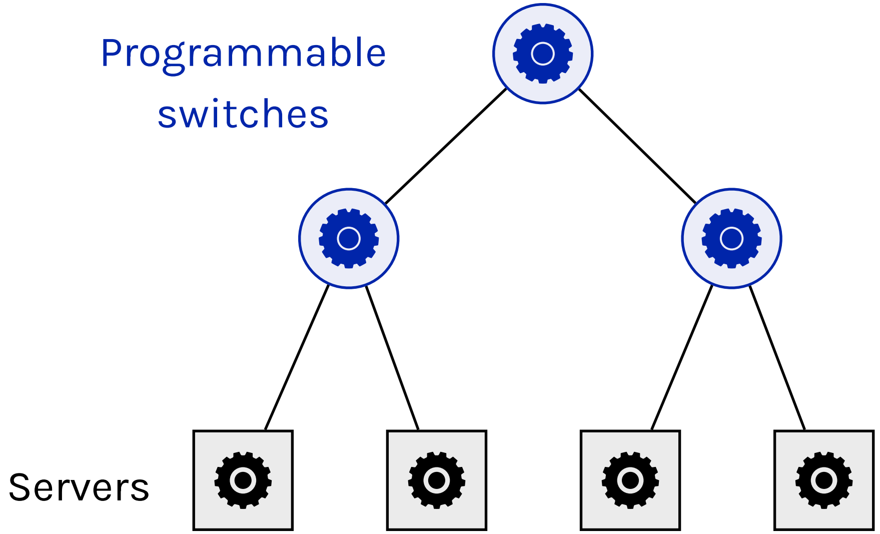
How to achieve maximum throughput?

- What is the optimal streaming rate? How to calculate it?



Lab5: Switches Do Dream of Machine Learning! (+ bonus)

Summary



Leverage switches for in-network computing: in-network caching, in-network coordination, in-network AllReduce

Reading material

NetCache: Balancing Key-Value Stores with Fast In-Network Caching

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Robert Soulé[2,4], Jeongkeun Lee[2], Nate Foster[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]Johns Hopkins University, [2]Barefoot Networks, [3]Princeton University, [4]Università della Svizzera italiana, [5]Cornell University, [6] UC Berkeley

ABSTRACT

We present NetCache, a new key-value store architecture that leverages the power and flexibility of new-generation programmable switches to handle queries on hot items and balance the load across storage nodes. NetCache provides high aggregate throughput and low latency even under highly-skewed and rapidly-changing workloads. The core of NetCache is a packet-processing pipeline that exploits the capabilities of modern programmable switch ASICs to efficiently detect, index, cache and serve hot key-value items in the switch data plane. Additionally, our solution guarantees cache coherence with minimal overhead. We implement a NetCache prototype on Barefoot Tofino switches and commodity servers and demonstrate that a single switch can process 2+ billion queries per second for 64K items with 16-byte keys and 128-byte values, while only consuming a small portion of its hardware resources. To the best of our knowledge,

KEYWORDS

Key-value stores; Programmable switches; Caching

ACM Reference Format:
Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of SOSP '17, Shanghai, China, October 28, 2017*, 17 pages. https://doi.org/10.1145/3132747.3132764

1 INTRODUCTION

Modern Internet services, such as search, social networking and e-commerce, critically depend on high-performance key-value stores. Rendering even a single web page often requires hundreds or even thousands of storage accesses [34]. So, as these services scale to billions of users, system operators increasingly rely on *in-memory* key-value stores to meet the

NetChain: Scale-Free Sub-RTT Coordination

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Nate Foster[2,4], Jeongkeun Lee[2], Robert Soulé[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]*Johns Hopkins University,* [2]*Barefoot Networks,* [3]*Princeton University,* [4]*Cornell University,* [5]*Università della Svizzera italiana,* [6] *UC Berkeley*

Abstract

Coordination services are a fundamental building block of modern cloud systems, providing critical functionalities like configuration management and distributed locking. The major challenge is to achieve low latency and high throughput while providing strong consistency and fault-tolerance. Traditional server-based solutions require multiple round-trip times (RTTs) to process a query. This paper presents NetChain, a new approach that provides scale-free sub-RTT coordination in datacenters. NetChain exploits recent advances in programmable switches to store data and process queries entirely in the network data plane. This eliminates the query processing at coordination servers and cuts the end-to-end latency to as little as half of an RTT—clients only experience processing delay from their own soft-

DrTM [6], which can process hundreds of millions of transactions per second with a latency of tens of microseconds, crucially depend on fast distributed locking to mediate concurrent access to data partitioned in multiple servers. Unfortunately, acquiring locks becomes a significant bottleneck which severely limits the transaction throughput [7]. This is because servers have to spend their resources on (*i*) processing locking requests and (*ii*) aborting transactions that cannot acquire all locks under high-contention workloads, which can be otherwise used to execute and commit transactions. This is one of the main factors that led to relaxing consistency semantics in many recent large-scale distributed systems [8, 9], and the recent efforts to avoid coordination by leveraging application semantics [10, 11]. While these systems are successful in achieving high throughput, unfortunately, they restrict the programming model and complicate the

Scaling Distributed Machine Learning with In-Network Aggregation

Amedeo Sapio* *KAUST*; Marco Canini* *KAUST*; Chen-Yu Ho *KAUST*; Jacob Nelson *Microsoft*; Panos Kalnis *KAUST*; Changhoon Kim *Barefoot Networks*; Arvind Krishnamurthy *University of Washington*; Masoud Moshref *Barefoot Networks*; Dan R. K. Ports *Microsoft*; Peter Richtárik *KAUST*

Abstract

Training machine learning models in parallel is an increasingly important workload. We accelerate distributed parallel training by designing a communication primitive that uses a programmable switch dataplane to execute a key step of the training process. Our approach, SwitchML, reduces the volume of exchanged data by aggregating the model updates from multiple workers in the network. We co-design the switch processing with the end-host protocols and ML frameworks to provide an efficient solution that speeds up training by up to 5.5× for a number of real-world benchmark models.

1 Introduction

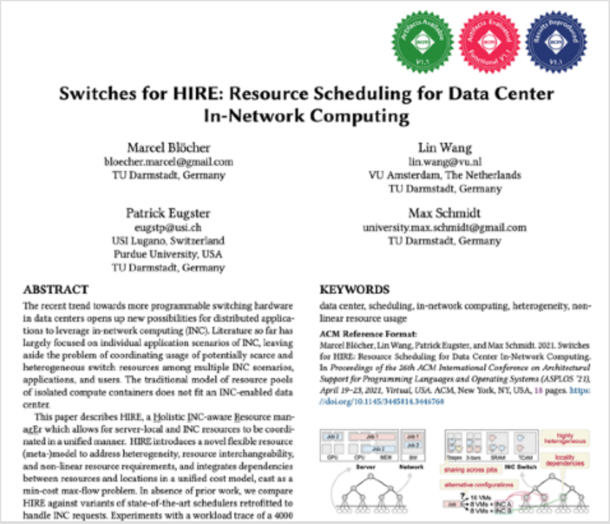
Today's machine learning (ML) solutions' remarkable success derives from the ability to build increasingly sophisticated models on increasingly large data sets. To cope with the resulting increase in training time, ML practitioners use distributed training [1, 22]. Large-scale clusters use hundreds of nodes,

aggregation primitive can accelerate distributed ML workloads, and can be implemented using programmable switch hardware [5, 10]. Aggregation reduces the amount of data transmitted during synchronization phases, which increases throughput, diminishes latency, and speeds up training time.

Building an in-network aggregation primitive using programmable switches presents many challenges. First, the per-packet processing capabilities are limited, and so is on-chip memory. We must limit our resource usage so that the switch can perform its primary function of conveying packets. Second, the computing units inside a programmable switch operate on integer values, whereas ML frameworks and models operate on floating-point values. Finally, the in-network aggregation primitive is an all-to-all primitive, unlike traditional unicast or multicast communication patterns. As a result, in-network aggregation requires mechanisms for synchronizing workers and detecting and recovering from packet loss.

We address these challenges in SwitchML, showing that it is indeed possible for a programmable network device to perform in-network aggregation at line rate. SwitchML is

Our own work on in-network computing



Switches for HIRE: Resource Scheduling for Data Center In-Network Computing

Marcel Blöcher
bloecher.marcel@gmail.com
TU Darmstadt, Germany

Lin Wang
lin.wang@vu.nl
VU Amsterdam, The Netherlands
TU Darmstadt, Germany

Patrick Eugster
eugstp@usi.ch
USI Lugano, Switzerland
Purdue University, USA
TU Darmstadt, Germany

Max Schmidt
university.max.schmidt@gmail.com
TU Darmstadt, Germany

ABSTRACT

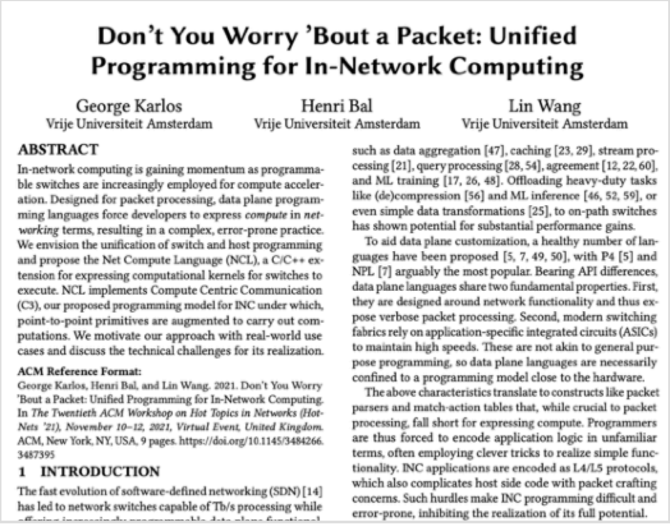
The recent trend towards more programmable switching hardware in data centers opens up new possibilities for distributed applications to leverage in-network computing (INC). Literature so far has largely focused on individual application scenarios of INC, leaving aside the problem of coordinating usage of potentially scarce and heterogeneous switch resources among multiple INC scenarios, applications, and users. The traditional model of resource pools of isolated compute containers does not fit an INC-enabled data center.

This paper describes HIRE, a Holistic INC-aware Resource managEr which allows for server-local and INC resources to be coordinated in a unified manner. HIRE introduces a novel flexible resource (meta-)model to address heterogeneity, resource interchangeability, and non-linear resource requirements, and integrates dependencies between resources and locations in a unified cost model, cast as a min-cost max-flow problem. In absence of prior work, we compare HIRE against variants of state-of-the-art schedulers retrofitted to handle INC requests. Experiments with a workload trace of a 4000

KEYWORDS

data center, scheduling, in-network computing, heterogeneity, non-linear resource usage

ACM Reference Format:
Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. 2021. Switches for HIRE: Resource Scheduling for Data Center In-Network Computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3445814.3446760



Don't You Worry 'Bout a Packet: Unified Programming for In-Network Computing

George Karlos, Vrije Universiteit Amsterdam
Henri Bal, Vrije Universiteit Amsterdam
Lin Wang, Vrije Universiteit Amsterdam

ABSTRACT

In-network computing is gaining momentum as programmable switches are increasingly employed for compute acceleration. Designed for packet processing, data plane programming languages force developers to express *compute* in *networking* terms, resulting in a complex, error-prone practice. We envision the unification of switch and host programming and propose the Net Compute Language (NCL), a C/C++ extension for expressing computational kernels for switches to execute. NCL implements Compute Centric Communication (C3), our proposed programming model for INC under which, point-to-point primitives are augmented to carry out computations. We motivate our approach with real-world use cases and discuss the technical challenges for its realization.

ACM Reference Format:
George Karlos, Henri Bal, and Lin Wang. 2021. Don't You Worry 'Bout a Packet: Unified Programming for In-Network Computing. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3484266.3487395

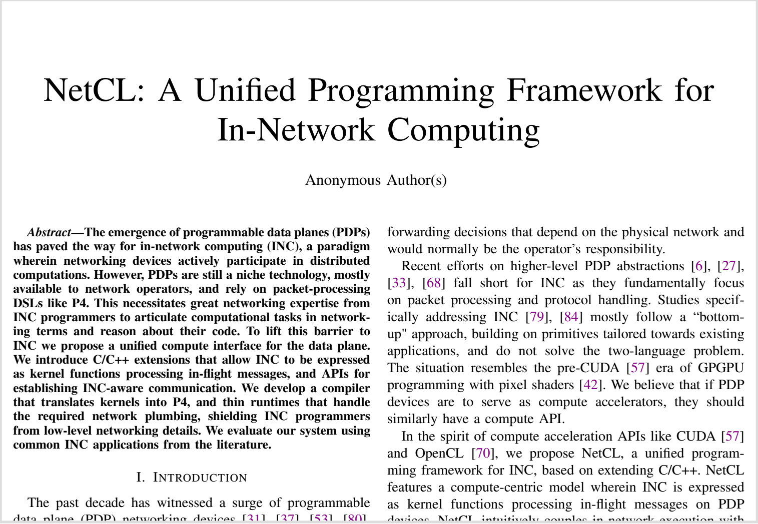
1 INTRODUCTION

The fast evolution of software-defined networking (SDN) [14] has led to network switches capable of Tb/s processing while

such as data aggregation [47], caching [23, 29], stream processing [21], query processing [28, 54], agreement [12, 22, 60], and ML training [17, 26, 48]. Offloading heavy-duty tasks like (de)compression [56] and ML inference [46, 52, 59], or even simple data transformations [25], to on-path switches has shown potential for substantial performance gains.

To aid data plane customization, a healthy number of languages have been proposed [5, 7, 49, 50], with P4 [5] and NPL [7] arguably the most popular. Bearing API differences, data plane languages share two fundamental properties. First, they are designed around network functionality and thus expose verbose packet processing. Second, modern switching fabrics rely on application-specific integrated circuits (ASICs) to maintain high speeds. These are not akin to general purpose programming, so data plane languages are necessarily confined to a programming model close to the hardware.

The above characteristics translate to constructs like packet parsers and match-action tables that, while crucial to packet processing, fall short for expressing compute. Programmers are thus forced to encode application logic in unfamiliar terms, often employing clever tricks to realize simple functionality. INC applications are encoded as L4/L5 protocols, which also complicates host side code with packet crafting concerns. Such hurdles make INC programming difficult and error-prone, inhibiting the realization of its full potential.



NetCL: A Unified Programming Framework for In-Network Computing

Anonymous Author(s)

***Abstract*—The emergence of programmable data planes (PDPs) has paved the way for in-network computing (INC), a paradigm wherein networking devices actively participate in distributed computations. However, PDPs are still a niche technology, mostly available to network operators, and rely on packet-processing DSLs like P4. This necessitates great networking expertise from INC programmers to articulate computational tasks in networking terms and reason about their code. To lift this barrier to INC we propose a unified compute interface for the data plane. We introduce C/C++ extensions that allow INC to be expressed as kernel functions processing in-flight messages, and APIs for establishing INC-aware communication. We develop a compiler that translates kernels into P4, and thin runtimes that handle the required network plumbing, shielding INC programmers from low-level networking details. We evaluate our system using common INC applications from the literature.**

I. Introduction

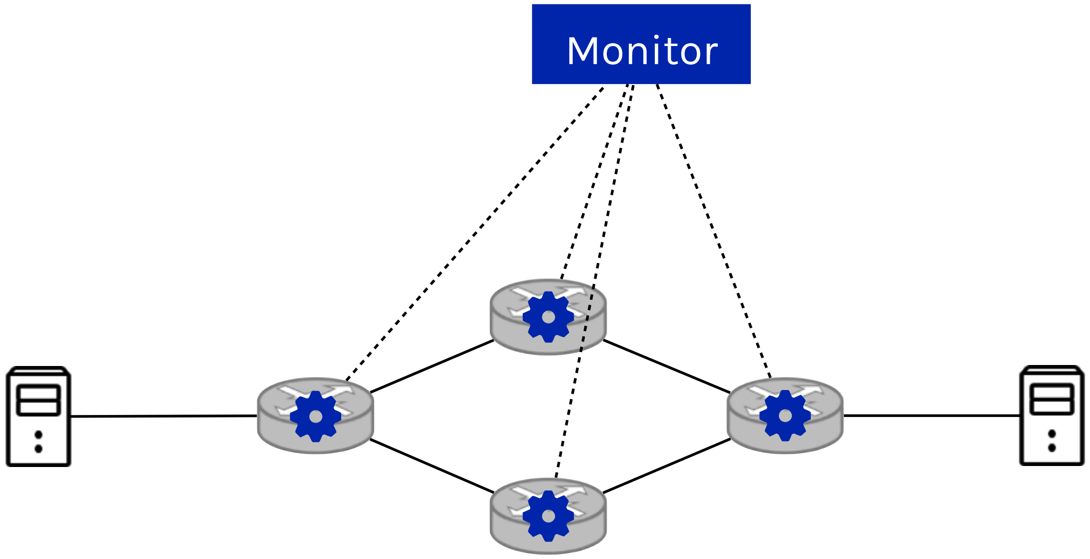
The past decade has witnessed a surge of programmable

forwarding decisions that depend on the physical network and would normally be the operator's responsibility.

Recent efforts on higher-level PDP abstractions [6], [27], [33], [68] fall short for INC as they fundamentally focus on packet processing and protocol handling. Studies specifically addressing INC [79], [84] mostly follow a "bottom-up" approach, building on primitives tailored towards existing applications, and do not solve the two-language problem. The situation resembles the pre-CUDA [57] era of GPGPU programming with pixel shaders [42]. We believe that if PDP devices are to serve as compute accelerators, they should similarly have a compute API.

In the spirit of compute acceleration APIs like CUDA [57] and OpenCL [70], we propose NetCL, a unified programming framework for INC, based on extending C/C++. NetCL features a compute-centric model wherein INC is expressed as kernel functions processing in-flight messages on PDP

Next time: network monitoring



How to achieve **fast** and **accurate** network monitoring with programmable data plane?