Contents

8 End-Host Networking

8.1 Why End-Host Networking Matters

In a **datacenter network**, traffic does **not** exist on its own. Every packet is:

- **Generated** by an application;
- **Processed** by an application;
- **Terminated** by an application.

And **all applications run on end-hosts** (i.e., servers, VMs, containers). Therefore, **end-hosts are where network traffic begins and ends**. This may sound trivial, but it is the *key insight* behind the **end-host networking** approach.

What is an end-host?

An **End-Host** is simply a **server** (or VM, or container) inside a rack of a datacenter. It typically includes: CPUs, memory, one or more network interfaces (NICs), the operating system kernel and user-space applications. Unlike switches or routers:

- End-hosts **run applications**;
- End-hosts **execute the networking stack** (i.e., implement TCP/IP, UDP, etc.);
- End-hosts **interpret packet payloads** (i.e., they read and write application data).

Role of End-Hosts in the network. From the network's perspective, end-hosts are: **traffic sources** and **traffic sinks** (i.e., they generate and consume traffic). However, from the application's perspective, end-hosts are **the only place where application semantics exist**. Indeed, the switches in the network can forward packets, inspect headers, and make routing/load-balancing decisions, but they **cannot create** packets, **consume** packets, or **decide what** packets **mean** (i.e., application semantics).

Why does this matter for performance? Even if the **datacenter network fabric is perfect** (i.e., low latency, high bandwidth, no packet loss), packets still must:

1. Enter the server via the Network Interface Card (NIC);
2. Traverse PCIe (Peripheral Component Interconnect Express) bus to reach the host memory;
3. Be processed by the operating system kernel networking stack (e.g., Linux TCP/IP stack);
4. Be delivered to the application (e.g., web server, database).

If **any of these steps is slow**, then the **overall performance** of the networked application **suffers**, regardless of how good the network fabric is. Therefore, **end-host networking performance is critical** to the overall performance of datacenter applications.

Aspect	End-hosts	Switches / Routers
Run applications	✔	✖
Generate traffic	✔	✖
Interpret payload	✔	✖
Programmability	High	Limited
Deployment speed	Fast	Slow

Table 15: Key differences between end-hosts and in-network devices (switches/routers). The asymmetry in capabilities highlights why end-host networking is crucial for application performance. End-hosts are **software-driven** and network devices are **infrastructure-driven**.

Remark: What is the PCIe bus?

The **PCIe (Peripheral Component Interconnect Express)** is the **high-speed internal interconnect** that links the **Network Interface Card (NIC)** to the **CPU and memory of a server**. It is *not a networking protocol*, but it is **fundamental to networking performance at the end host**.

In other words, PCIe is the hardware bus that carries packets inside the server, moving data between the NIC and host memory/CPU at very high speed and low latency. The life of a packet at the host is as follows (simplified):

$$\text{Network} \rightarrow \text{NIC} \xrightarrow{\text{PCIe}} \text{Host Memory} \rightarrow \text{Kernel} \rightarrow \text{Application}$$

So PCIe is the **bridge between networking hardware and software**.

⚠ Why improving in-network hardware is hard in practice?

At first glance, improving the network seems like the obvious solution to improving application performance. After all, if the network is faster, then packets should arrive sooner, right? However, in **practice**, this turns out to be **very hard**, **slow**, and **expensive**. This is why *end-host networking* becomes attractive.

⚠ Operational complexity

- ✖ **Networks are shared infrastructure**. Datacenter networks are shared by **thousands of applications**, shared by **many teams**, and

operated under **strict reliability guarantees**. Any change to in-network hardware affects **all tenants**, **all applications**, potentially **the whole datacenter**. Even a small bug in a switch can cause **network-wide outages**.

- ✗ **Hardware bugs are catastrophic**. In-network devices operate at **line rate** (i.e., they forward packets at full speed), but they are also **hard to debug**, and often fail in **unpredictable ways**. If a switch drops packets silently, corrupts state or misroutes traffic, **everything breaks**, and it is **very hard to trace the root cause**.

⚠ Deployment time

- ✗ **Hardware innovation is slow**. Improving in-network hardware usually means new switch ASICs (Application-Specific Integrated Circuits), NIC firmware updates, drivers, or new control-plane software. All of these take **years** to design, test, manufacture, and deploy at scale. In contrast, end-host software can be updated in **days** or **weeks**.
- ✗ **Rollouts are painful**. Deploying new network hardware requires staged rollouts, compatibility testing, maintenance windows, and fallback plans. And often these rollouts require **recabling**, **topology changes**, or **downtime**, which are all costly and risky. This is the opposite of agile innovation.

⚠ Cost and compatibility

- ✗ **Financial cost**. Network hardware is extremely expensive, tightly budgeted, and amortized over many years. Replacing switches is not done lightly, and often requires capital expenditure approval. In contrast, end-host software changes are **low-cost** and **iterative**.
- ✗ **Compatibility constraints**. New in-network features must work with **existing protocols**, interoperate with **legacy devices**, and comply with vendor ecosystems. Often the innovation is constrained by **backward compatibility** and operators cannot deploy "experimental" features in production networks (e.g., new congestion control algorithms). End-host software, on the other hand, can be **customized** per application or team.

In summary, while improving in-network hardware seems appealing in theory, **practical challenges** make it **difficult**, **slow**, and **costly** in reality. This is why focusing on **end-host networking optimizations** is often the more viable path to improving application performance in datacenter environments.

✓ Advantages of End-Host Innovation

End-host innovation means **moving networking functionality closer to servers**, where applications run. This is not an accident or a workaround; it is a **deliberate design choice** adopted by modern datacenters for several reasons:

- **Easier Deployment**
  - ✔ **Software beats hardware**. End-hosts are general-purpose machines, software-driven, and under frequent update cycles. This makes deploying new networking features, like updating a kernel module or a driver, or deploying a user-space service, **much easier** than changing switch hardware or firmware. Because there is **no need to touch the physical network**.
  - ✔ **Smaller blast radius**. If something goes wrong, only a **subset of servers** is affected, rollback is easy and failures are **contained**. This makes experimentation and innovation **safe**.
- **Faster Iteration**
  - ✔ **End-hosts follow software timelines**. At end-host changes happen in **days or weeks**, while network hardware changes take **years**. Also, CI/CD pipelines, automated testing, and continuous deployment are standard practice for end-host software, enabling rapid iteration and improvement. In contrast, network hardware changes require lengthy testing, staged rollouts, and careful planning. End-host innovation matches the **pace of application development**.
  - ✔ **Easier debugging and testing**. At the end-host there is a full OS visibility, tracing tools (eBPF, tcpdump, perf), application-level context. This enables fine-grained performance tuning and rapid diagnosis of regressions.
- **Closer to applications**
  - ✔ **Access to semantics**. Only end-hosts know which application owns a packet, what a flow represents, and which latency or throughput matters. This **semantic knowledge** enables smarter optimizations that are **application-aware**, such as prioritizing critical flows, adapting to workload patterns, or implementing custom congestion control algorithms. In contrast, in-network devices operate blindly, without understanding application context, they can only make decisions based on packet headers and statistics (e.g., headers, flow size, etc.).
  - ✔ **Better cross-layer optimization**. End-hosts can jointly consider networking, CPU scheduling, memory hierarchy, and application logic. This is **impossible** inside the network fabric.
- **Less disruption to the network**. End-host innovation does not require network-wide coordination, preserves stability of the fabric and avoids vendor lock-in. Operators like this **a lot**.

So kernel bypass, programmable NICs, and eBPF are all examples of **end-host networking innovations** that leverage these advantages to improve application performance in datacenter environments.

8.2 Life of a Packet Inside a Server

When a packet arrives at a server, it does **not** go directly to the application. Instead, it must traverse a **layered processing pipeline**, which includes several steps:

- Hardware boundaries (NIC, bus, memory).
- Protection domains (kernel, user space).
- Software abstractions (network stack, sockets, libraries).

Each step adds **latency**, **CPU overhead**, and **potential bottlenecks**. Understanding this pipeline is crucial for optimizing network performance.

The baseline packet path

At a very high level, an incoming packet follows this path:

1. **Network** (i.e., the wire). It represents the external world where packets are transmitted.
2. **NIC (Network Interface Card)**. The **NIC** is the hardware component connected to the network. Its responsibilities include:
   - Receiving packets from the wire.
   - Storing them temporarily in NIC memory.
   - Performing basic checks (e.g., checksum offload).
   - Initializing data transfer to host memory.

   The NIC operates **independently of the CPU**, it cannot interpret application data. So it is the **first bottleneck** in the packet path: if it cannot sustain line rate, everything else is useless.
3. **PCIe bus** (Peripheral Component Interconnect Express, connecting NIC to memory). The NIC is **not directly connected** to the CPU or main memory. Packets must cross the **PCIe bus** using **DMA (Direct Memory Access)** to transfer data to host memory. The PCIe bus has its own **bandwidth and latency characteristics**, which can impact performance.
4. **NIC Driver** (i.e., software managing the NIC). The **driver** is kernel code that manages the NIC, programs hardware registers, handles interrupts and moves packets into kernel data structures. It acts as the **software interface between hardware and kernel**. It runs in **kernel context**, it is executed frequently and bugs or inefficiencies here can have a large impact, since it is on the **critical path** of packet processing (i.e., every packet must go through it).
5. **Kernel Networking Stack** (operating system's network processing). The **network stack** is a complex software layer that implements network protocols (e.g., TCP/IP). It is responsible for:

- Protocol handling (e.g., TCP state machine).
- Packet reassembly.
- Congestion control.
- Routing.

This is where security checks happen, congestion control lives, but also packet ordering is enforced. The kernel provides **generality and safety**, but at the cost of **performance overhead** due to context switches, data copies, and protocol processing.

6. **User-space Application** (i.e., the server application). Eventually, the packet reaches a socket buffer or a user-space application via a system call (e.g., `recvfrom()`). Crossing from kernel to user space involves **context switches** and **data copies**, which add latency and CPU overhead. The user-kernel boundary crossing is expensive and unavoidable in the baseline model.

This is the **baseline model**, all optimizations later in the notes aim to **reduce the cost of one or more of these steps**.

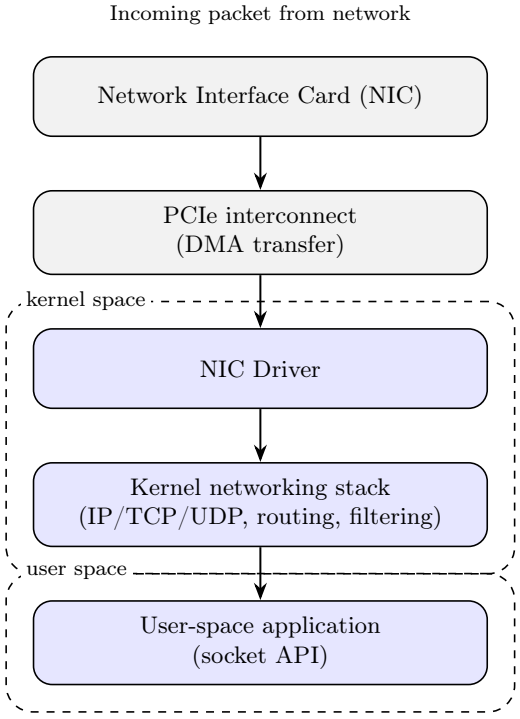

Figure 9: Baseline packet path inside a server.

Kernel packet buffers and descriptor rings

Now that we have a high-level understanding of the packet path, let's look at some important data structures used in the kernel networking stack to manage packets efficiently.

Our focus is on the **interaction between the NIC and the kernel**. Since the kernel mediates all packet transfers between the hardware and the applications, its interaction with the NIC determines the performance, scalability, and isolation. This makes the kernel the primary bottleneck and optimization target in end-host networking.

The NIC **cannot write packets wherever it wants** in host memory. Instead, the kernel:

1. **Pre-allocates *packet buffers*** in memory to hold incoming packets.
2. **Tells the NIC where they are** in memory.
3. **Uses *descriptor rings* to coordinate ownership** of these buffers between the NIC and the kernel.

This design avoids CPU involvement in the fast path, enables high throughput, and supports batching and Direct Memory Access (DMA).

? What are packet buffers? **Packet Buffers are regions of host (DRAM) memory allocated by the kernel used to store incoming packets**. For example, in Linux, these are typically `sk_buff` structures that hold packet metadata and data, or memory pages managed by the networking subsystem. It is important to note that buffers are allocated **before** packets arrive because this avoids dynamic allocation on the fast path.[15] As anticipated, pre-allocating buffers is crucial for performance because, without it, the kernel would require locks, causing cache misses and severely limiting throughput. **Pre-allocated packet buffers are essential for line-rate reception of packets**.

? What is a RX descriptor ring? **RX** stands for *receive* and refers to the **direction of traffic**:

- ← **RX** path: packet reception (incoming packets).
- → **TX** path: packet transmission (outgoing packets).

Obviously, there are **TX descriptor rings** as well, but we focus on RX here because it is more complex and performance-critical.

A **descriptor** is *not* a packet. It is a **small control structure** that describes *where* a packet should go. We can think of it as a **post-it note** attached to a buffer. Usually, a descriptor contains:

[15] "*Dynamic allocation on the fast path*" means allocating memory for each incoming packet as it arrives, which would introduce significant latency and overhead. By pre-allocating buffers, the system can quickly place incoming packets into these pre-reserved memory areas, allowing for higher throughput and lower latency. Here, fast path refers to the critical execution path (i.e., the sequence of operations that must be performed quickly to ensure efficient packet processing) that handles incoming packets with minimal delay.

- A **pointer** to a packet buffer in host memory (physical address).
- The **size** of that buffer.
- **Status flags** (empty, full, ownership, errors, etc.).

For example:

```
Descriptor:
    address = 0x1A2B3C4D  # Physical address of packet buffer
    length  = 2048        # Size of the buffer in bytes
    status  = EMPTY       # Status flag indicating buffer is empty
```

In simple terms, the descriptor **tells the NIC where to DMA-write incoming packets**.

Finally, a **ring** is just a **circular queue** with a fixed number of entries. A *circular queue* means that when we reach the end of the queue, we wrap around to the beginning. The circular nature allows for efficient use of memory (no reallocations), constant-time enqueue/dequeue operations, and perfect for hardware-software communication.

Putting it all together, an **RX Descriptor Ring** is a **circular queue of descriptors that tell the NIC where to place incoming packets in host memory**. The RX descriptor ring is:

- **Allocated** by the **kernel**.
- **Shared** with the **NIC**.
- **Accessed concurrently** by both the **kernel** and the **NIC**.

Relationship between NIC and Kernel memory via RX Descriptor Rings

1. **Initialization phase**
   (a) The kernel **allocates packet buffers** in DRAM, creates descriptors pointing to empty buffers, marks them as **available** and tells the NIC about them.
   (b) The driver programs the NIC with the address of the RX descriptor ring in host memory.
   (c) The NIC now knows where buffers are located in host memory for incoming packets.

   In this phase, the kernel and NIC set up the necessary data structures to enable efficient packet reception.

2. **Packet reception phase**. When a packet arrives, the NIC:
   (a) **Fetches** a descriptor from the RX ring (i.e., gets the address of an empty buffer) via **PCIe reads**.
   (b) **Performs** a **Direct Memory Access (DMA)** to **write** the incoming packet into the specified buffer in host memory.

(c) **Updates** the descriptor status to **indicate that the buffer is full** and ready for processing by the kernel.

The NIC does **not** allocate memory on the fly, call the CPU, or touch the kernel during this fast path. It simply uses DMA to write packets into pre-allocated buffers as indicated by the descriptors.

How does the kernel know when packets have arrived?

So far, the NIC did all the work of receiving packets and writing them into host memory. But the kernel **does not know** a packet arrived until the NIC notify the CPU. In the first naïve design, the NIC raises an **Interrupt Request (IRQ)** to a CPU core. Then, the kernel's NIC driver interrupt handler is invoked. This is the **first moment** the CPU is involved in packet processing.

Interrupts can be expensive!

Handling interrupts involves context switches, saving/restoring CPU state, and can lead to cache misses. If packets arrive at a high rate, the CPU can become overwhelmed with interrupts, leading to **interrupt livelock** (page 153), where it spends all its time handling interrupts and cannot process packets effectively. To mitigate this, techniques like *interrupt coalescing* (batching multiple packets per interrupt) and *polling* (the kernel periodically checks for new packets instead of relying on interrupts) are often employed, and we will see them later in the notes.

3. **Ownership transfer**. The ownership of a descriptor:
   (a) Start with the **kernel** (buffer is empty, step 1).
   (b) Moves to the **NIC** during DMA write (buffer is being filled, step 2).
   (c) Returns to the **kernel** once packet is written (buffer is full).

   This ownership transfer is crucial for synchronization between the NIC and kernel. It ensures that the NIC only writes to buffers that the kernel has marked as available, and the kernel only processes buffers that the NIC has filled with incoming packets. It prevents data races, avoids locks, and enables zero-copy transfers (i.e., no CPU copies needed).

4. **Kernel processing phase**. Upon receiving an interrupt (IRQ) from the NIC, the kernel:
   (a) **Checks** the RX descriptor ring to identify descriptors marked as **full** by the NIC.
   (b) **Processes** the packets stored in the corresponding buffers (e.g., protocol handling, socket buffering, statistics).
   (c) **Reclaims** the buffers by resetting the descriptors and marking them as **available** again for the NIC.

   This phase involves the kernel regaining ownership of the descriptors after packet reception. Buffer recycling is **essential** to sustain high throughput, as it allows the NIC to continue receiving packets without exhausting available buffers.

Step	NIC	PCIe	CPU (Kernel)
Packet arrival	✔	✘	✘
Descriptor fetch	✔	✔	✘
DMA transfer	✔	✔	✘
Interrupt	✔	✘	⚠ (interrupt)
Kernel processing	✘	✘	✔

Table 16: Summary of which components are involved in each step of the packet reception process using RX descriptor rings.

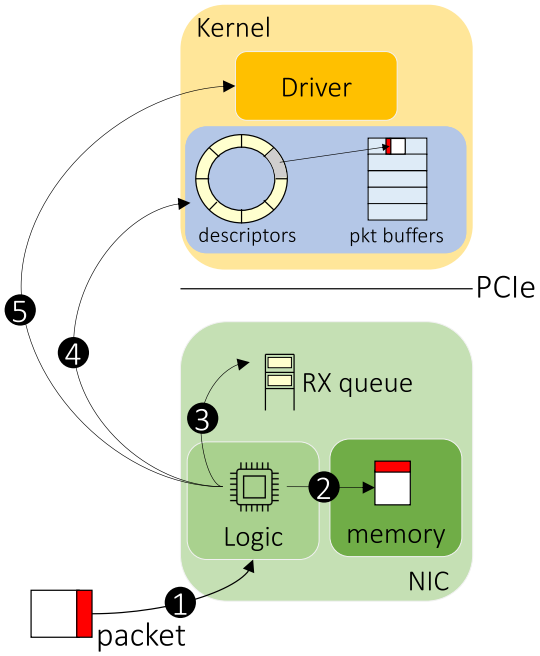

Figure 10: Packet reception at the end-host [4], step by step: (1) a packet arrives from the network and is received by the NIC; (2) the NIC fetches an RX descriptor to determine the address of an empty host packet buffer; (3) the NIC transfers the packet into host memory via DMA over PCIe and updates the descriptor status; (4) the NIC generates an interrupt (IRQ) to notify the CPU that packets are available; (5) the kernel driver and networking stack process the packet and eventually deliver it to the application. Finally, regarding the RX queue within the NIC and the memory buffers:

- The NIC has a **small internal memory** just enough to **hold packets briefly** before DMA transfer. This memory is **only for waiting**, not for storage.
- The NIC RX queue is just a small temporary waiting area inside the NIC that holds packets for a short time after they arrive from the network, until the NIC can copy them into main memory. It is separate from the kernel's descriptor ring and has nothing to do with kernel memory structures.

8.3 The Receive Livelock Problem

In the simplest receive model (page 147):

1. A packet arrives at the Network Interface Card (NIC).
2. The NIC DMA-writes the packet into main memory.
3. The NIC raises an **interrupt (IRQ)**.
4. The CPU stops what it is doing and handles the packet.

This happens **for every packet**.

⚠ Why does this become a problem? Interrupts are **expensive** because they preempt running tasks, flush CPU pipelines, pollute caches, and force context switches. At low packet rates this is fine and the overhead is negligible. However, at high packet rates (e.g., 10 Gbps and beyond), the CPU may spend **most of its time just handling interrupts** and very little time is left for *actual packet processing*. The CPU becomes the bottleneck, not the NIC. **Interrupt cost scales with packet rate, not with packet size**; many small packets are much worse than fewer large ones.

? What is Receive Livelock

Receive Livelock is a situation where the CPU spends all its time handling receive interrupts but makes no forward progress in processing packets. The system is busy, active, and consuming CPU, but **not productive**. It is called *livelock* because unlike a *deadlock* where the system is stuck doing nothing, here the system is busy doing something (handling interrupts) but not making progress. In other words, the cpu is *alive*, but **stuck reacting**.

? What is the CPU actually doing? In receive livelock, the CPU handles an interrupt, processes *very few* packets, immediately receives another interrupt, and repeats endlessly. It never gets enough uninterrupted time to drain the RX ring buffer and deliver packets to applications.

⚠ Why throughput can drop to zero

The most counterintuitive aspect of receive livelock is that **as the packet rate increases, the throughput can actually drop to zero**. This is counterintuitive because we might expect more incoming packets to lead to more processed packets. However, in receive livelock, the CPU becomes overwhelmed with interrupts and is unable to process packets effectively. The vicious cycle is:

1. High packet arrival rate.
2. NIC generates many interrupts.
3. CPU spends most cycles on interrupt handling.
4. Very little packet processing is completed.
5. RX ring fills up.

6. NIC cannot DMA new packets.
7. Packets get dropped.

As result, despite a high arrival rate, the effective throughput (packets successfully processed) can **plummet to zero** because the CPU is too busy handling interrupts to make any progress on actual packet processing. So **the system is interrupt-bound, not bandwidth-bound**. This is why faster NICs alone do **not** solve the problem.

Historical Context

Receive livelock was identified in the late 1990s and early 2000s on networks that were much slower than today's 10/40/100 Gbps links. However, modern NICs are 1,000 times faster, yet CPUs did not scale in interrupt efficiency at the same rate. Today, we have 25/40/100+ Gbps NICs, microservices with many small packets, and virtualized, multi-tenant systems. However, the **conditions for livelock are easier to reach than ever before**.

In summary, receive livelock is a critical challenge in high-speed networking where the CPU becomes overwhelmed with interrupts, leading to a situation where it is busy but not productive. In the next sections, we will explore various techniques to mitigate this problem and improve packet processing efficiency.

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.

[2] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.

[3] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Nsdi*, volume 16, pages 523–535, 2016.

[4] Antichi Gianni. Network Computing. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[5] Albert Greenberg, Dave Maltz, Guohan Lu, Jiaxin Cao, Ratul Mahajan, and Yibo Zhu. Packet-level telemetry in large datacenter networks. In *SIGCOMM'15*, August 2015.

[6] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev.*, 45(4):139–152, August 2015.

[7] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[8] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 311–324, USA, 2016. USENIX Association.

[9] Weiwu Pang, Sourav Panda, Jehangir Amjad, Christophe Diot, and Ramesh Govindan. CloudCluster: Unearthing the functional structure of a cloud service. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1213–1230, Renton, WA, April 2022. USENIX Association.

[10] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[11] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang,

Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 76–89, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

Index