Making Decisions at Data Plane Speeds

Srinivas Narayana
Rutgers University, New Brunswick, NJ, USA

ABSTRACT

Feedback control loops to implement self-driving networks constitute *data collection* to sense the network, and *control algorithms* to make decisions driving the network. High-quality data is necessary for smart decisions. Yet, high-quality data is hard to obtain from the network data plane, due to insufficient visibility and large data volumes stemming from high packet rates. This paper distills principles to collect high-quality data arising from our own research experience: (i) filter and aggregate data as close to the source as possible; (ii) identify broad families of statistics that are measurable with bounded inaccuracy; (iii) don't assume low-level data plane software is easy to instrument, but instead (iv) apportion software flexibility by the time scales of the computation; and (v) prefer in-band approaches where possible for timely and efficient reactivity. We call the community to act upon these principles to leverage emerging opportunities using safely-extensible network stacks.

1. INTRODUCTION

Feedback control is an integral part of self-driving systems. Networks have conventionally incorporated feedback control at several layers of the stack to drive themselves. Classic examples include congestion control, medium access control, and IP traffic engineering. Feedback control includes two components: *data collection* to sense the network in the data plane, and *control algorithms* either in the data or the control plane, to drive the network based on the data that was collected.

Regardless of the smartness of decision-making algorithms, bad data can lead to poor decisions. Hence, it is paramount to have access to high-quality data from the data plane. However, there are several reasons why obtaining good data is challenging. To make our discussion concrete, we focus on scenarios in data center networks for the rest of this paper.

Why is it hard to collect high-quality data?

(1) *Insufficient visibility:* Designing feedback control to respond to anomalies in performance requires access to fine-grained, low-level network performance data directly measured at the bottlenecks. Examples include determining the queue lengths at routers and servers and the contributions of individual connections to those hotspots. However, such raw performance signals are often hard to measure in the data plane, because deployed hardware and software are simply incapable of the introspection required for such observations. The emergence of In-Band Network Telemetry on programmable dataplanes alleviates these problems to some extent, but it does not solve the visibility problem, especially for application-level metrics (§2.1).

(2) *Large data volumes.* When raw signals (e.g., queue sizes) are indeed available on a packet by packet basis, the speed at which such signals are generated poses a significant challenge. Naive attempts to collect such signals "out of band" using storage systems could double the number of packets processed by the network. Instead, either the per-packet signals must be sampled or aggregated to reduce the packet rate of outgoing signals, or bandwidth that could otherwise be used for actual network traffic must be repurposed to carry signals in-band, typically requiring server changes and new infrastructure. A related difficulty is the design of algorithms that can aggregate per-packet signals into useful "buckets" cutting across protocol layers, for example, organizing connections into a histogram of application-level response latencies. However, network data planes are traditionally only capable of simple stateful aggregations at a low protocol level.

2. LESSONS FROM THREE STORIES

Given the difficulties of obtaining high-quality data, how should one go about designing algorithms to collect network data for self driving? In this section, we distill some principles from our own prior research efforts.

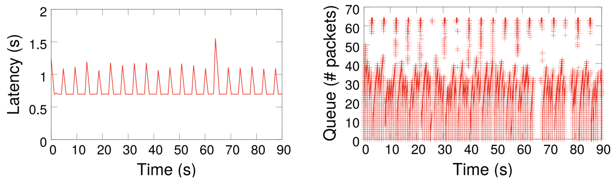
2.1 Network Performance Diagnosis

In the Marple system [3], our goal was to diagnose anomalies in network performance. An example of such an anomaly is *microbursts*: short-time-scale bursts of packets arising from traffic sources that display an ON/OFF transmission pattern, increasing the queueing delays transiently but recurrently for other latency-sensitive traffic sharing the network. Identifying the perpetrators of such microbursts is challenging, since they are not major contributors to traffic on the network and it is unclear which switch and queue in the network is the site of the microburst. At the time, the Tofino programmable switches had just introduced the capability to observe queue sizes as metadata on a per-packet basis on the switch pipeline. However, this raw data is arriving at the same high speed as the packets on the switch.

Principle 1 [Pushdown]. *Filter and aggregate data as close to the source of the data as possible.*

This principle is well known in database systems where the cost of moving data, say between machines or shuffling rows

Self-Driving Networks Workshop 2023 Orlando, Florida



(a) HTTP latency.

(b) Queue sizes.

Figure 1: Microbursts: (a) A victim HTTP flow experiencing frequent spikes in response latency. (b) The time evolution of queueing delays experienced by packets traversing a queue with a microburst-perpetrating traffic source.

for database joins, is significant. Reordering database operations to eliminate irrelevant data earlier in the processing (e.g., matching on predicates or projecting specific columns) can significantly improve efficiency. Inspired by this principle, we designed primitives that operate directly in the switch data plane, at line rate, to implement filtering and aggregation through user-defined keys. For the microburst scenario, this enables the ability to (i) identify switch queues and packets which experience large queueing delays at those queues; and (ii) identify traffic sources (say, aggregated at the level of transport 5-tuples) that contribute ON/OFF traffic, by counting the number of bursts of packets separated in time by a user-defined threshold. This enables not only determining where the microburst-perpetrating sources are active, but also the sources themselves.

Principle 2 [Identify accurate families]. *Identify families of statistics measurable with bounded (or zero) inaccuracy, and design algorithms customized to those.*

The extensive literature on sketching algorithms adheres to this principle. However, it is much more generally applicable to data collection even for statistics not typically captured with sketches, for example, the number of packets considered out-of-order in a TCP connection. Concurrently with the Marple work, there existed hardware switching chips collecting aggregated network performance metrics, for example average packet latency per 5-tuple flow. However, when the switch experiences an uptick in the number of flows (e.g., under a flash crowd or a TCP SYN flood), memory size limitations would force the switch to evict existing flows from its memory. The policies used for eviction from the switch made it unclear how the data that is retained on the switch compares in accuracy to an ideal lossless measurement. However, the problem goes beyond the shortcomings of one platform: there was a fundamental lack of understanding of how a switch should collect measurements not easily summarized with limited memory.

In the Marple work, we identified a class of statistics for which it is possible to obtain accurate data despite the eviction of data from a switch under high memory pressure. The trick is that we use a slower, but larger and more persistent memory than a switch, to *merge* any partial measurements evicted from a switch with an authoritative measurement residing in the larger memory. A multi-tier memory architecture for measurement dovetails well with the existence of plentiful memory on servers outside of switches. We identified that statistics s whose per-packet update takes the functional form $s \triangleq \alpha(\vec{p}) \cdot s + \beta(\vec{p})$, where α and β can be any switch-implementable functions over a recent bounded history of packets $\vec{p}$, can be merged with 100% accurate results. This seemingly simple functional form captures diverse statistics, for example the number of out-of-order packets in each TCP connection.

2.2 Programming Congestion Control

Congestion control is a classic example of self-driving, with a rich research literature. In our work on the Congestion Control Plane (CCP [2]), we were inspired by the need to prototype and evaluate a complex congestion control protocol [1]—one that involves signal processing algorithms such as Discrete Fourier Transforms—in realistic settings.

Principle 3 [Low-level software changes slowly]. *Software is not arbitrarily fungible. In particular, data plane software is not easily changed, for reasons surrounding stability and performance.*

Traditionally, TCP congestion control is implemented using Linux kernel modules, which limit what developers are allowed to do. For example, floating point computations are challenging inside the kernel. Invoking some unsafe numerical operations could easily crash the kernel (e.g., division by zero). Further, the emergence of many kernel-bypass software platforms necessitated the implementation of the same protocol logic on diverse software platforms such as Intel's Data Plane Development Kit (DPDK) and Google's QUIC, each with their own relatively-static programming APIs.

In addition to asking if it is possible to ease development and experimentation for congestion control within Linux, we also wondered if it is possible to develop such logic once and have it run everywhere.

Principle 4 [Flexibility $\propto$ Available Compute Time]. *The flexibility accorded to a software layer should be proportional to the time available to compute at that layer.*

It was tempting to introduce a highly-flexible programming API to help develop complex functionality directly within the Linux kernel and emerging kernel-bypass frameworks. Specifically, the API could allow the maintenance of arbitrary state over which arbitrary computation could occur. However, high-speed packet processing is highly sensitive to the performance of the memory subsystem. For context, with 100Gbit Ethernet, it is necessary to admit a new minimum-sized Ethernet packet approximately every 6 nanoseconds to keep up with the packet arrival rate. In such contexts, the hit rates at the fastest cache layers are critical to performance—a single L2 cache miss could consume the entire time budget available to process a packet and slow the entire system down. Hence, the size of the memory maintained across packets must be limited, as should the compute over that memory. Not all complex functionality can or should go into the data plane.

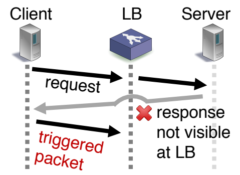
In the CCP system, we observed that the nature of congestion control makes it neither necessary nor useful to implement complex congestion control computation for each packet in the data plane. Instead, the natural computational time scale for congestion control is the round-trip time (RTT) of the connection, which is much longer than the time to admit a new packet. Our design choice was to enable developers to write flexible yet simple *fold functions* in the data plane to maintain only the summaries of per-packet signals for each connection. These summaries would be relayed once every RTT to a much more flexible control plane component running in user space. The data plane and the control plane components have asymmetric flexibility that is proportional to the natural time scales over which computations occur in those components.

2.3 Server Load Balancing

In large-scale Internet services, it is standard practice to implement a layer of *server load balancing* to distribute the incoming workload of client requests across a pool of servers offering the service. The oft-stated goal of load balancing is to spread load, preventing hotspots or failures on any one server from impacting client requests. In our ongoing work on server load balancing [4], we ask whether load balancing could be used proactively to improve service performance, by redirecting more requests to the better-performing servers in the pool. Existing solutions that use server performance implement an *agent-based model,* where a software agent on the server (running either as a daemon or as a part of a library incorporated into the application) relays feedback on load and queue occupancies to the load balancer. Such explicit feedback is crucial since, in many deployments, the responses to clients from the servers skip the load balancer on the return path. This idea, known as *direct server return (DSR)*, significantly reduces the workload on the load balancer relative to processing both requests and responses.

With the advent of microservices, serverless, and nanoscale computing, there is now a move towards increasingly-finer granularity of computing per request. Shrinking compute times significantly hurt the usability of agent-based feedback. First, request processing becomes highly vulnerable to variability within the system, for example due to process scheduling. Second, server agents completely miss network delays. As the per-request compute time approaches the client connection's RTT, the network delay contributes half of the total client-visible response latency.

Principle 5 [In-Band Feedback Control]. *To design highly-reactive systems, avoid staleness and big data problems through in-band feedback control.*



Relevant data must be made available as quickly and as accurately as possible at the point where self-driving decisions are made. In performance-aware server load balancing, one relevant piece of data is an estimate of the up-to-date response latency offered by each server. Rather than siphoning data from server agents to load balancers through an out-of-band stream, or through a centralized data collection system, it is appealing if load balancers can measure the response latencies directly. However, the load balancer's visibility into client traffic is asymmetric: with DSR, the load balancer only sees the requests and not the responses, making it challenging to measure response latencies by correlating them with the requests.

Our key insight is that it is possible to substitute the measurement of the delay between request and response by the delay between the request and a packet that a client transmits due to the response—a packet we call a *causally-triggered transmission.* There are many examples of causally-triggered transmissions, most commonly TCP acknowledgments. We show that such transmissions can be detected while only observing requests but not responses [4].

3. A CALL TO ACTION

We believe there are significant opportunities ahead to design novel self-driving networked systems, by leveraging emerging *safely-extensible data plane* software in the network stack. Concretely:

1. *Verified kernel extensions* (eBPF) allow user-developed code to be attached with safety guarantees to specific "hooks" (function calls or execution sites) in the Linux kernel. Examples of hooks include the net device driver, packet scheduler, congestion control, and system calls. Safety in the eBPF context means that programs have a bounded running time, contain only instructions that do not crash (e.g., no division by zero), and all memory accesses are within safe bounds permitted by the kernel.
2. *Service proxies* (e.g., Envoy, Linkerd) are a new software layer in the container networking stack, refactoring common communication-related capabilities needed in containerized applications into a reusable component. For example, service proxies implement load balancing policy and failure detection and recovery logic common to multiple applications. Some service proxies such as Envoy are safely extensible at run time, including WebAssembly (WASM) sandboxes.

These extensible software layers enable the design of novel algorithms for data collection and feedback control operating directly in the packet-processing software path, with well-designed channels to communicate out-of-band with a flexible control plane. For example, one could incorporate application-specific customizations for congestion control, packet scheduling, or high-speed packet forwarding. Extensible data plane software is naturally amenable to applying principled data collection and feedback control techniques (§2) that overcome the fundamental challenges of data collection (§1). We believe that the prospects of designing self-driving networks have never before been as bright.

However, to make those prospects viable, the community must address wide-ranging challenges to enable the effective use of these emerging extensible network layers.

(1) Designing algorithms under safety constraints: Any program run within an extensible network layer must be 'safe'—a term whose definition depends on the context (e.g., which kernel version and which hook are we extending?), and is evolving. This brings up questions like: What is the scope of algorithms that can be implemented safely within extensible network layers? What programming abstractions could make it easy to design such algorithms?

(2) Performance: Achieving high performance within extensible software layers is crucial since these are on the critical path of packet processing. How should the performance of an algorithm be optimized while retaining its safety guarantees? How should we design optimizing compilers? Is there scope for workload-driven optimizations?

We call upon the community to act on these challenges to help realize novel self-driving networked systems.

4. REFERENCES

[1] Prateesh Goyal et al. Elasticity detection: A building block for internet congestion control. In *SIGCOMM*, 2022.

[2] Akshay Narayan et al. Restructuring endpoint congestion control. In *SIGCOMM*, 2018.

[3] Srinivas Narayana et al. Language-directed hardware design for network performance monitoring. In *SIGCOMM*, 2017.

[4] Bhavana Vannarth Shobhana et al. Load balancers need in-band feedback control. In *ACM HotNets*, 2022.