Using Deep Programmability to Put Network Owners in Control

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

Guru Parulkar
Open Networking Foundation
Stanford University
guru@opennetworking.org

Larry Peterson
Open Networking Foundation
Princeton University
llp@opennetworking.org

Oguz Sunay
Open Networking Foundation
oguz@opennetworking.org

This article is an editorial note submitted to CCR. It has NOT been peer reviewed.
The authors take full responsibility for this article's technical content. Comments can be posted through CCR Online.

ABSTRACT

Controlling an opaque system by reading some "dials" and setting some "knobs," without really knowing what they do, is a hazardous and fruitless endeavor, particularly at scale. What we need are transparent networks, that start at the top with a high-level intent and map all the way down, through the control plane to the data plane. If we can specify the behavior we want in software, then we can check that the system behaves as we expect. This is impossible if the implementation is opaque. We therefore need to use open-source software or write it ourselves (or both), and have mechanisms for checking actual behavior against the specified intent. With fine-grain checking (e.g., every packet, every state variable), we can build networks that are more reliable, secure, and performant. In the limit, we can build networks that run autonomously under verifiable, closed-loop control. We believe this vision, while ambitious, is finally within our reach, due to *deep programmability* across the stack, both vertically (control and data plane) and horizontally (end to end). It will emerge naturally in some networks, as network owners take control of their software and engage in open-source efforts; whereas in enterprise networks it may take longer. In 5G access networks, there is a pressing need for our community to engage, so these networks, too, can operate autonomously under verifiable, closed-loop control.

CCS CONCEPTS

• **Networks → Network architectures**; **Network types**;

KEYWORDS

Software Defined Networks (SDN), Programmable Networks, Network Verification, Telemetry

1 INTRODUCTION

Over the last fifteen years we have witnessed an extraordinary change in how networks are designed and built. This has taken place in two main phases. In phase one, network owners took charge of the software that controls their networks. In 2005, cloud providers built their networks using closed, proprietary networking equipment designed for enterprise and ISP networks. Today, the largest cloud providers and data-center owners build their own networking equipment based on merchant switching ASICs, a Linux-based switch OS, and a homegrown management and control system. While often referred to as *disaggregation* or *Software-Defined Networking* (SDN), this revolution has really been about who is in charge. *"Who gets to decide what functions, protocols, and features are supported by a network: Is it the equipment vendors (e.g., router manufacturers), the technology providers (e.g., chip and optics manufacturers), or those who own and operate networks?"* [4].

In phase two, network owners started specifying how packets are processed, by programming the forwarding behavior in switches, middleboxes, smartNICs, and end-host stacks. This has been made possible by P4-programmable switches (e.g., Tofino), NICs (e.g., Pensando, Xilinx), and the ability to write efficient packet-processing code in software (e.g., XDP, DPDK, VPP). The motivation, again, is about who is in charge. The primary purpose of a network is to forward packets from one place to another; if network owners cannot control packet forwarding, they are not really in charge.

Together, these two pieces—control and data planes, with the behavior of both prescribed by the network owner—recast the network as a *deeply programmable platform* that can be controlled by network owners to suit their needs. This view has several natural consequences. First, network owners will be able to specify the desired behavior at the "top," and the specification will be partitioned and compiled "down" to dictate how the network is controlled and packets are processed. Second, if network owners *can* tailor the network to suit their needs, they likely *will*. Rather than allowing vendors to determine how their network should operate, network operators will deploy novel designs that are more reliable, secure, and performant, using specific insights based on the knowledge they have gained by running large networks.

As a consequence, most large networks will work differently from others. Behaviors that were formerly realized using standard protocols—routing, congestion control, access control, virtualization—will instead be thought of as custom programs, partitioned across the platform by a compiler. Done right, interoperability will arise top-down, from compiling the same programs to different network elements, rather than being driven bottom-up via standardization. In practice, standardization will likely trail software development.

It will also mean that networking students, researchers, and practitioners will need to learn how to program a network top-down, as a distributed computing platform.

While perhaps a little controversial to some, we take everything up to this point as our starting point. Indeed, we believe it is somewhat inevitable. The goal of this paper is to describe what we believe is now possible, as a direct consequence of the emergence of *deep programmability* for networks. This paper is also a call to arms to our community to explore *how* networks should be programmed and for *what* purpose. The time is ripe to help shape how networks will be designed in the future and to create new frameworks and methodologies to assist network owners, allowing them to decide where to place each piece of functionality.

A particular opportunity enabled by deep programmability is the ability to *observe* and *verify* each packet. We imagine that packets will carry information about their provenance (the path they took, the rules they followed, the delays they encountered, etc.) and network state will be monitored, too (the contents of forwarding tables, address translation, overlay tunnels, and topology). Packets and state will be checked against the specified intent to answer questions like: Did the packet take the correct path? Is it allowed to be at this point in the network? Is the forwarding state consistent with a set of invariant properties derived from the owner's original specification? And if packets, state, or code are found to be in error, they can be automatically repaired and re-verified to ensure the problem is resolved.

In the past, network owners controlled their networks however they *could*—wherever they were allowed to read "dials" and set "knobs." However, with deep programmability "across the stack," they can place functionality where it belongs and verify that it is implemented correctly. As a community, we have not been thinking about these issues as a primary focus, but we should. We need to think more deeply about which functionality belongs in the data plane (e.g., for high speed, low overhead, and quick adaptation), or in the distributed control-plane software (e.g., for greater flexibility without the overhead and delay of going to a central controller), or in a logically-centralized SDN controller (e.g., for network-wide visibility and to run centralized algorithms) [24]. And we need to design effective ways to synthesize the partitioning of functionality and then verify that the whole system behaves as intended. We believe that these goals, however ambitious, are now within our reach. Done right, we believe the answers will eventually lead to networks that are "programmed by many, operated by few."

2 RELATIONSHIP TO RECENT TRENDS

The skeptical reader might ask how our position differs from the past decade of research on logically-centralized SDN controllers [23, 26] or recent efforts related to intent-based networking [31], self-driving networks [10, 32], or in-network computing [30]. The key difference is that deep programmability "across the stack" focuses not just on a single component, but is instead a comprehensive approach that allows network owners to put functionality wherever it belongs.

Software-Defined Networking (SDN): The push to SDN architectures was an important step toward deep programmability—it opened up control-plane APIs and allowed network owners to program directly against them. But as radical as SDN may have seemed at the time, it is ultimately only a part of the solution. Today's SDN applications are sometimes deployed on a centralized controller not because a centralized design is necessarily the best choice, but rather because that is where network owners are *allowed* to deploy new functionality. For instance, backhauling measurements to a central controller and computing new configurations to push individual devices, limits the responsiveness of the control loop. But it is too hard to realize better strategies when the underlying devices are closed and inflexible.

Intent-based Networking: Another recent trend, intent-based networking, advocates for specifying behavior in terms of high-level network intents. We completely agree. However, this industry trend assumes an *opaque* model in which network owners lack visibility into the code that determines behavior. Under this regime, one must somehow configure opaque boxes without a precise understanding of what the configuration does, without a clear specification of low-level behavior to check against, and without a means to fix behaviors when they are found to be broken. While the idea of having a top-down specification of intent is important, it seems unlikely (to us) that large networks will be built using opaque models, because they do not afford the required level of transparency and control to the network owner.

Self-Driving Networks: Self-driving networks informed by machine-learning analytics, have also been touted as a promising future approach for building networks. ML is effective at making inferences about data that is too complex or too unpredictable for humans to enumerate. For example, self-driving cars use ML to process complex sensor data to try and understand previously unseen behaviors. But we cannot effectively use machine learning without the right "dials" and "knobs" for observing and controlling the network. Mogul observes that it would have been impossible to automate control of a 1965 Corvair, because the relationship between the car's controls and its behavior were opaque or ill-defined [27]. Self-driving cars became possible only when vehicles could be controlled precisely in software. The manufacturer of a self-driving car would not trust opaque software supplied by a vendor that could only be controlled by configuring an intent. Nor should we.

In-network Computing: The past few years have seen the emergence of commodity data planes that support programmable packet processing at line rate. This has created great interest in supporting a wide range of network functionality in the data plane, from fine-grained network telemetry to support for distributed services (e.g., key-value stores, load balancers, and consensus protocols). Yet, just because we *can* implement sophisticated functionality in the data plane does not mean that we *should* [25]. Often the data plane lacks sufficient resources (i.e., processing or memory) to implement a function well, and sometimes relocating functionality from end hosts to the network introduces significant complexity with only negligible performance gains. Clearly, programmable data planes are important enablers, but the community is only just starting to develop an understanding of what functionality belongs there and what should remain in the control plane, or stay out of the network entirely.

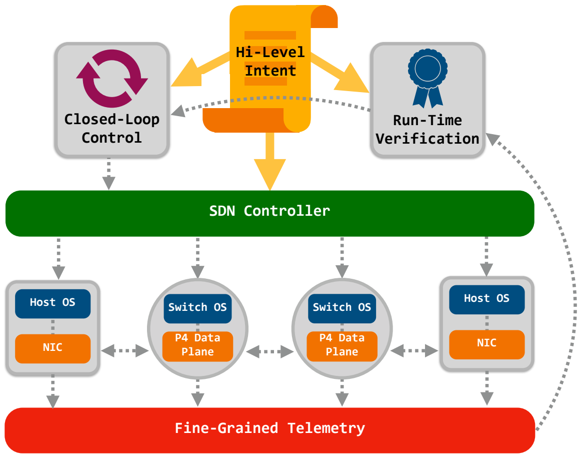

Figure 1: Deep programmable network architecture. The network owner specifies top-level behavior as a program that is compiled to code for the controller, switch/host OSes, and data planes. Telemetry data is collected end-to-end and used for run-time verification and closed-loop control. Note that verification and closed-loop control may be implemented in the controller (as shown) or in the switch OS or the data plane (not shown).

3 FORM SHOULD FOLLOW FUNCTION

This section uses traffic engineering (TE) as a running example to illustrate the challenges and opportunities that arise with deep programmability at all layers of the network. We picked TE because of the fundamental role it plays in many networks, and because of the long history of developing TE solutions. Of course, deep programmability and verifiable closed-loop control apply to many other important applications as well (e.g., congestion control, denial-of-service attack mitigation, diffusing microbursts, etc.).

3.1 Architecture

Figure 1 depicts the architecture of a *deep programmable* network platform, with different "dials" and "knobs" at each level, and verifiable closed-loop control using fine-grained telemetry data collected along end-to-end paths.

Data Plane: Programmable data planes, whether a P4-programmable switch or a SmartNIC, processes packets at line rate and can "turn on a dime," reacting to changes in local state and updating forwarding behavior instantaneously. However, data planes can only maintain a limited amount of state and perform simple computations on each packet.

Switch & Host OSes: The switch OS is equipped with a general-purpose CPU with sufficient processing power and memory resources to implement sophisticated protocols (e.g., BGP, OSPF, P4Runtime, etc.) and algorithms, and to aggregate measurement data collected from the data plane. However, its view of network state is fundamentally local, and the CPU resources tend to lag behind high-end servers. The host networking stack has similar capabilities (and limitations).

SDN controller: Finally, the SDN controller maintains a network-wide view of the current topology and operating conditions, and runs sophisticated algorithms that would difficult to express as a distributed computation. A centralized controller can also incorporate other data (such as properties of the physical layer) that are not readily visible to the underlying switches. However, its ability to update network behavior is limited by the speed at which it can propagate updates to network devices—it could be anywhere from less than a second to a few minutes, depending on the size and complexity of the update.

Given the unique strengths and limitations of each layer, it is natural to ask which functionality should run where. We explore this question next, using TE as an example.

3.2 Example: Traffic Engineering

Traffic engineering—adapting routing to the prevailing traffic demands—is a fundamental network function that has been studied extensively over the years. Different approaches leverage programmability at different layers, leading to different adaptation timescales and optimization objectives, as summarized in Table 1. Centralized solutions make it easier to incorporate sophisticated objectives and additional data (e.g., shared-risk link groups at the physical layer) and ensure stability, while distributed solutions enable much faster adaptation to traffic shifts and device failures.

Protocol configuration: In traditional distributed routing protocols like OSPF and IS-IS, the switch OSes compute shortest paths based on configurable link weights. Lacking the ability to change the switch OS software, let alone the data-plane functionality, network owners resorted to tuning the link weights to coerce the switches into selecting the desired paths. The paths would be chosen to robustly optimize for a given objective (e.g., minimizing the maximum link utilization) given traffic demands and failure scenarios extrapolated from historical data. The "control loop," as it were, operated on human timescales—hours or days—and was sometimes implemented manually. In essence, the only "program" being run was the centralized selection of the link weights themselves. The weight-selection algorithms relied on abstract models of the control plane would behave, with no ability to verify those behaviors. In addition, expressing routing policy in terms of link weights is both limiting (some combinations of paths cannot be expressed) and inefficient (optimizing link weights is NP-hard), even though the techniques performed reasonably well in practice.

Distributed control plane: Distributed load-sensitive routing, with the switch OSes exchanging dynamic link weights and adjusting forwarding accordingly, were explored as early as the ARPAnet [19]. While the early protocols worked well under low-to-moderate loads, preventing oscillations proved much more difficult when network load increased. After a long "dry spell," load-sensitive routing saw a resurgence, including traffic-engineering extensions to protocols like OSPF and IS-IS [17] as a way to create tunnels that "pin" traffic on paths with good performance. Newer research results [11, 15], drawing on techniques from control theory and optimization theory, showed it was possible to design distributed control planes with good convergence properties. These solutions relied on customized multipath protocols and telemetry techniques. Unfortunately, without the ability to change the "knobs" and "dials" in the control-plane

TE Solution	Network	Programmable Component	Architecture	Objective(s)	Timescale
Tuning IGP weights [9]	WAN	Protocol configuration	Centralized	Congestion	Hours or days
TeXCP [15], OSPF-TE [17]	WAN	Switch OS agent	Distributed	Congestion, Failures, Stability	Seconds
SWAN [12], B4 [14]	WAN	SDN controller	Centralized	Utilization, Latency	Minutes
HULA [16], Contra [13]	DC	P4 data plane	Distributed	Congestion, Failures	Milliseconds

Table 1: Representative historical traffic engineering approaches.

software on commercial switches, these ideas were never widely deployed.

Centralized SDN controller: Shortly after the first SDN platforms emerged about a decade ago, cloud providers begin experimenting with SDN-based approaches to traffic engineering. Systems such as SWAN [12] and B4 [14] exploited the network-wide visibility and direct control over forwarding paths offered by the SDN control plane to implement complex, multi-objective TE schemes. For instance, B4 is designed to carry latency-sensitive customer-facing traffic on short paths while utilizing the excess capacity on other paths to carry lower-priority internal traffic and also quickly reacting to failures. Unlike pre-SDN centralized approaches, these systems operate under closed-loop control: they continuously monitor the load and compute a set of forwarding paths that optimize for the desired objective. Their main limitation is the high latency of the control loop—because the controller is centralized, its view of the network may be out of date. In addition, using SDN protocols such as OpenFlow to update forwarding tables can require minutes to propagate changes, which places a limit on the responsiveness.

Distributed data-plane approaches: Several recent systems have proposed using fine-grained telemetry to enable pure data-plane TE solutions [2, 13, 16]. The key idea is to record information about the current network conditions on packets to inform forwarding decisions for other packets. For instance, each packet might record the congestion it "saw" as it traversed the network [2] or probes might "pick up" statistics kept locally on each switch, to track of the performance of downstream paths in real time [13, 16]. Switches can then use this information to select the path(s) with the best performance for forwarding data traffic. Different networks, and different classes of traffic, may want to optimize different path performance metrics (such as path length, propagation delay, and link load), while also imposing constraints on which paths are considered (such as a sequence of middleboxes to traverse) [13]. Compared to the other approaches just discussed, the "control loop" in these data-plane approaches to TE is more responsive—there is no need to wait for a central controller, or even the switch OS, to intervene. However, there are trade-offs—e.g., unlike centralized approaches, simultaneously optimizing for multiple objectives would be difficult. Also, running these distributed protocols on arbitrary topologies (as opposed to the tree-like topologies common in data centers) requires protocol mechanisms for preventing forwarding loops [13].

Discussion: While research on TE has changed over time, one trend is clear: as more and more components of the network became programmable, researchers were able to implement more sophisticated solutions. We are not arguing that any of these solutions are better than the others—there are genuine trade-offs, and we expect that different considerations would take priority, depending on the specific needs of the network owner. What's important is that the network platform offer the right "dials" and "knobs" so network owners can take charge and implement the solution they want. Also, we need a methodology for making principled choices about which function should go where.

4 PROGRAMMING NETWORK WITH INTENTS

Requiring programmers to write code for each component in a deep programmable platform would be a tall order. We argue that network owners should be able to specify their high-level intent and let a compiler synthesize the distributed set of programs that run in the data and control planes.

For example, the intent for a traffic-engineering application might consist of path performance metrics (for ranking the paths) and regular expressions on paths (to identify which paths are permitted, and how their ranks are computed). A compiler would then synthesize the control code and forwarding code software to implement performance-aware routing that realizes the specified policy [13]. As another example, consider the problem of mitigating denial-of-service attacks that rely on DNS amplification. The network operator could specify a high-level query that identifies DNS amplification attacks (e.g., flagging destination IP addresses that receive DNS response packets from an unusually large number of distinct sources), coupled with the appropriate action (e.g., drop or rate limit) to take on the offending traffic. The compiler would then synthesize the data-plane programs that collect, analyze, and act on the traffic accordingly. Hence, the network software would be generated in a "correct by construction" fashion. Moreover, the compiler's output could be validated using run-time techniques—see Section 5.

One challenge that arises when designing any distributed system is reasoning about complex dynamics. A failed link can trigger the BGP convergence process, packet loss triggers TCP congestion control, and poor performance along the current path triggers performance-aware routing schemes to shift traffic to another path. We need to know that each control loops will behave well—*i.e.,*, converge, or at least avoid significant disruptions or performance degradation due to oscillations. Often, multiple interacting control loops need some sort of "timescale separation" to prevent bad behavior, or some sort of damping to avoid overreacting to new information. We believe that determining how to verify these properties of network control loops, or how to synthesize control loops with known safety or stability properties, is an exciting avenue for future work. For example, the body of work on "optimization decomposition" [5] shows how to decompose a variety of network-wide optimization problems into distributed algorithms that solve these problems—with provable convergence and correctness properties.

With deep programmability, these optimization-based techniques now have a plausible path to deployment.

5 VERIFIABLE NETWORKS

Having deep programmability "across the stack" also creates an exciting opportunity for network owners to verify that the network works as intended. By verifiability, we mean the ability to check that every packet in the network follows a specified path and provides the expected performance.

The community has already made important progress on network verification, including static verification of forwarding rules [18, 20] and analysis of existing distributed protocols [3, 8]. Yet, this early work has been limited in several important ways. One issue is that they usually work with models—e.g., control-plane snapshots of the intended forwarding rules, or abstract models of legacy protocols—that elide important information about the state of the switches or the operation of the protocols. Discrepancies can arise between the model and the actual behavior, especially when failures occur or when the network is being reconfigured.

Deep programmability enables observing "ground-truth" behavior at the packet level, which can be used to build direct solutions to many verification problems. In particular, by collecting fine-grained telemetry data for every packet, the network owner can answer questions that, while easy to ask, have traditionally been difficult to answer.

How did this packet get here? Imagine we pick a packet out of the network and ask how it got here. In other words, we want to know the set of switches and middleboxes the packet visited along its path. The goal is to answer the question for *this very packet*, not using a `traceroute` or `ping` probe, or an inference drawn from a dump of the forwarding configuration, but rather based on ground-truth observations of the path itself. If the packet carries a list of switch IDs (e.g., using In-band Network Telemetry (INT) [21]), we can check to see if the path is correct. In fact, we can check for correctness at several different levels in the hierarchy of control abstractions. At the top, we can check the packet's path against the control plane's topology map for the network (which might have caused an otherwise correct routing protocol to pick the wrong path, in which case the corrective action might be to fix the topology map state, or identify a bug in the code that identifies the topology). We can also check the path against the forwarding state in one or more switches: The topology might be correct, but the routing protocol may have written the wrong forwarding entry. Or perhaps the state was initially correct, but has since been corrupted by errant or malicious software or failed hardware.

Why did the packet get here? While path information tells us a lot, we can carry the example a step further. Imagine that we also ask why the packet got here. The packet could carry with it the sequence of forwarding rules that it followed at each switch along the path. For example, it might tell us that at the first switch it matched on a rule, which identified the second switch to go to next. Armed with this list of rules, our analysis and verification tools can identify which rules caused the packet to arrive here. This can help pinpoint the cause of the error in the control-plane software, the routing protocol, an access control policy, or in the switch forwarding table. We can also ask what *execution path* the packet took through the data-plane program at each switch in its path [22]. If the answer violates our expectations, the program (or compiler!) may have a bug that needs to be fixed.

How long was the packet queued at each hop? The opportunities become even more interesting if we consider performance measures, too. Imagine we can ask our packet how long it was queued at each hop in its journey. If we can observe the delay at each hop, we can automatically and immediately determine whether or not the network is meeting service level objectives (SLOs). Furthermore, with similar information about other packets traversing some of the same links and the same time, the network can identify where most of the delay takes place and (more importantly) why. It might point to a microburst, which would now be identified in place and time. Or it might point to a shortcoming in the traffic engineering or congestion control algorithms. It might even point to a denial-of-service attack that is overloading some of the links. Whatever the reason for the degradation, the control loop can take an appropriate corrective action.

What is the performance of this path? In addition to measuring the performance experienced by each packet, the network can track the performance of paths through the network. This is possible by aggregating performance statistics across packets that traverse the same path. Alternatively, the network can combine link-level statistics (such as link utilization and queue depth) from the data plane across multiple links, to compute statistics (such as the maximum utilization or total queuing delay across all links in a path) even for paths that do not (yet) carry any traffic [16]. These data-plane measurements enable the network to verify that the paths offer the expected performance, or to identify alternate paths that would offer better performance.

Fortunately, once the network knows a correctness or performance property is violated, it is often relatively easy to decide what to do about it—such as reporting, marking, dropping, rate limiting, or rerouting some portion of the traffic. This makes it relatively straightforward for the network to enforce high-level policy goals by using telemetry results to trigger simple actions—often directly in the data plane to react quickly and minimize overhead. Packets that generate errors can be reported (to enable further diagnosis) and dropped (to prevent security violations). If a queue has a backlog, arriving packets can be probabilistically marked or dropped in proportion to the size of their flow. Or if a path has poor performance, the switch can divert a portion of the traffic to less-loaded paths.

6 END-TO-END PROGRAMMABILITY

Deep programmability—and the opportunity to build networks that run autonomously under verifiable, closed-loop control—is opening up, with data-center fabrics and the backbones interconnecting those data centers at the vanguard. But the biggest opportunity, both in terms of technical challenges and societal impact, is the access network. This is a particularly critical moment in time for the mobile cellular network, with the early stage transition to 5G opening the door to deep programmability being truly end-to-end.

Historically one of the most opaque networks imaginable, the cellular network is being re-designed according to SDN principles with the emergence of 5G [6, 7]. The new architecture embraces disaggregation, with the packet and signal-processing pipeline embedded in the base stations of the *Radio Access Network (RAN)* split into multiple, distributed subsystems, each controlled from an edge cloud-hosted control plane. This centralized controller, often referred to as the *Near Real-Time RAN Intelligent Controller (RIC)*, hosts a set of control applications that make global (RAN-wide) decisions, as opposed to local (per base station) decisions. This includes handover control, link aggregation control, radio interference management, and load balancing. Even some aspects of the user plane responsible for forwarding packets between the Radio Units and the Internet are being prototyped as P4 programs running in programmable switches.

This opportunity for deep programmability is coupled with a networking paradigm that fundamentally depends on the fidelity of telemetry data. Real-time decisions about how to schedule transmissions in a way that makes the most efficient use of the available spectrum is based on frequent (≤ 1ms) *Channel Quality Indicators* sent from user devices to base stations. Also influencing these decisions are information on the respective loads on the virtualized components of the disaggregated RAN, as well as the throughput and latency information on the links that interconnect them. This fine-grain information is aggregated and feeds into RAN-wide control decisions about when it is better to serve a user device from the current base station versus a neighboring base station, versus multiple base stations simultaneously. Near-real-time control decisions take the interference of the shared radio spectrum and the need to support device mobility into account. Still higher-level control loops then make telemetry-informed decisions about how to allocate spectrum resources to meet delay and throughput requirements, from low-power IoT devices to enhanced mobile broadband with multi-gigabit peak rates to mission-critical applications requiring predictable latencies.

If the control loops that steer the mobile cellular network were not challenging enough in their own right, 5G has laid out an ambitious roadmap for the next decade, including support for (i) a massive *Internet of Things*, including devices with ultra-low energy (10+ years of battery life), ultra-low complexity (10s of bits-per-sec), and ultra-high density (1 million nodes per square km); (ii) *mission-critical control*, including ultra-low latency (as low as 1 ms), ultra-high predictability (hitting the latency target 99.999% of the time), and extreme mobility (up to 100 km/h); and (iii) *enhanced mobile broadband*, with extreme capacity (10 Tbps per square km) and data rates (multi-Gbps peak, 100+ Mbps sustained).

The stakes could not be higher: 5G will be an integral part of our cyber-physical world. Disaggregation opens the door to deep programmability, but using that opportunity to bring verifiable closed-loop control to access networks is on us.

7 PRONTO EXEMPLAR

The paper lays out an ambitious agenda, but one we believe is now within reach. To this end, we are building a full-featured network, called *Pronto*, under *verifiable closed-loop control*. Our goal is for Pronto to serve as an exemplar for others to replicate and improve upon.

The Pronto network consists of a 5G-enabled edge cloud constructed from the software components and programmable forwarding elements shown in Figure 2:

- P4-programmable switches based on the Protocol Independent Switch Architecture; e.g., Barefoot Tofino.
- P4-programmable NICs; e.g., Xilinx smartNICs.
- P4-programmable vSwitch in the end host, directing packets to the correct VM/container.
- An open source SDN stack that includes a thin switch OS (Stratum), an SDN Controller (ONOS), and a suite of control applications that implement a leaf-spine switching fabric (Trellis) [28].
- A disaggregated and software-defined, 3GPP-compliant RAN (SD-RAN) and Mobile Core (SD-Core) [29].
- A runtime control portal and lifecycle management toolchain that operationalizes the network (not shown in Figure).

The open-source software that controls Pronto is called *Aether* and is being deployed and operated by the Open Network Foundation [1].

Pronto adds the measurement, code generation, and verification elements needed for verifiable closed-loop control to this programmable foundation. Fine-grained measurements are recorded using INT, allowing for packets to be stamped by the forwarding elements to indicate the path it took, the queuing delay it experienced, and the rules it matched. These measurements are then fed back into the verification and code generation tools.

We package the combination of software and hardware shown in Figure 2 as an edge cloud pod, called the *Aether Compute Edge (ACE)*, which we then replicate in enterprises around the world. Aether includes a cloud-hosted management platform (not shown) that supports ACE as a managed service, providing the lifecycle management and runtime control needed to make Pronto operationally complete. An Aether management portal also gives enterprises the ability to deploy innovative edge services on their ACE pod, taking advantage of the local 5G connectivity.

8 CONCLUSION

We believe it is inevitable that networks will become deeply programmable, from top to bottom and end to end, opening up a new era in network design. While some network owners will be more tentative, designing networks that replicate the behavior of the opaque systems of the past, others will be bolder, taking the opportunity to re-design their networks to transparently implement the behavior they really want, with all the visibility (dials) and controls (knobs) they need, where they need them. As they construct their networks as a single coherent, distributed system, they will also build-in verification at multiple levels of abstraction, meaning networks will be worthy of the trust society increasingly places in them.

As a community, we could contribute to this change in a piecemeal fashion, to help it along from the sidelines. But we believe there is a much bigger opportunity—in fact, a pressing need—for the research community to develop new methodologies for system design, considering the network as one large, programmable distributed system. In these early days, there is ample opportunity to shape how future networks are designed. A community

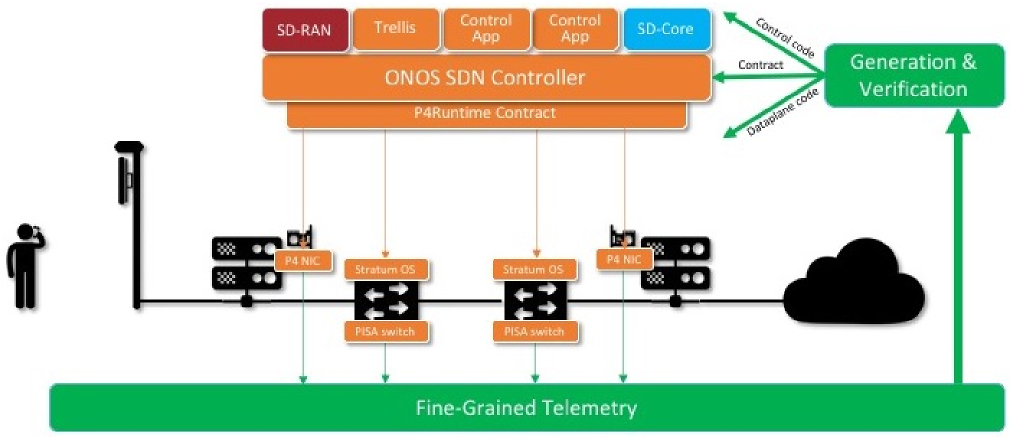

Figure 2: Pronto: An exemplar network built using commercially available programmable hardware, open-source software, and new measurement, code generation, and verification elements needed for verifiable closed-loop control.

effort to define new abstractions and new frameworks, to assist network owners in their work, will lead to a vibrant ecosystem in which we can all share the code that implements the necessary and non-differentiating infrastructure, while leaving room for network owners to differentiate by introducing their own innovative ideas. This opportunity exists for all networks, from ISPs to cloud providers and from home WiFi to the enterprise. But the need is particularly urgent (and technically challenging) for emerging access networks, particularly 5G, to ensure they can be designed using a top-down, transparent methodology, and operated as a verifiable, closed-loop system.

REFERENCES

[1] Aether: Managed 5G-Enabled Edge Cloud for Enterprises, April 2020. https://www.opennetworking.org/aether/.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*, 2014.

[3] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration analysis. In *ACM SIGCOMM*, August 2017.

[4] M. Casado, N. McKeown, and S. Shenker. From Ethane to SDN and beyond. *ACM SIGCOMM Computer Communications Review*, 49(5):92–95, Nov. 2019.

[5] M. Chiang, S. Low, R. Calderbank, and J. Doyle. Layering as optimization decomposition: A mathematical theory of network protocols. *Proceedings of the IEEE*, 95(1), January 2007.

[6] I. Chih-Lin and S. Katti. O-RAN: Towards an Open and Smart RAN, October 2018. https://www.o-ran.org/s/O-RAN-WP-FInal-181017.pdf.

[7] T. Czichy. 5G RAN optimization using the O-RAN software community's RIC (RAN Intelligent Controller). In *Open Networking Summit, Europe*, September 2019. Slides available at https://wiki.o-ran-sc.org/pages/viewpage.action?pageId=10715420&preview=/10715420/10715422/Near_RT_RIC_for_ONS.pdf.

[8] A. Fogel, S. Fund, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX Networked Systems Design and Implementation*, May 2015.

[9] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40(10):118–124, 2002.

[10] Y. Geng, S. Liu, F. Wang, Z. Yin, B. Prabhakar, and M. Rosenblum. Self-programming networks: Architecture and algorithms. In *Annual Allerton Conference on Communication, Control, and Computing*, 2017.

[11] J. He, M. Suchara, M. Bresler, J. Rexford, and M. Chiang. Rethinking internet traffic management: From multiple decompositions to a practical protocol. In *ACM SIGCOMM CoNext Conference*, December 2007.

[12] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, page 15–26, 2013.

[13] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. In *USENIX Networked Systems Design and Implementation*, February 2020.

[14] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, page 3–14.

[15] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, Philadelphia, PA, August 2005.

[16] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *ACM SIGCOMM Symposium on SDN Research*, 2016.

[17] D. Katz, K. Kompella, and D. Yeung. Traffic engineering (TE) extensions to OSPF version 2, September 2003. RFC 3630.

[18] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX Networked Systems Design and Implementation*, San Jose, CA, May 2012.

[19] A. Khanna and J. Zinky. The revised ARPANET routing metric. In *ACM SIGCOMM*, pages 45–56, September 1989.

[20] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX Conference on Networked Systems Design and Implementation*, Apr. 2013.

[21] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie. In-band Network Telemetry (INT), June 2016. https://p4.org/assets/INT-current-spec.pdf.

[22] S. Kodeswaran, M. T. Arashloo, P. Tammana, and J. Rexford. Tracking P4 program execution in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, March 2020.

[23] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *USENIX Conference on Operating Systems Design and Implementation*, page 351–364, 2010.

[24] N. Matni, A. Tang, and J. C. Doyle. A case study in network architecture tradeoffs. In *ACM SIGCOMM Symposium on SDN Research*, 2015.

[25] J. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Computer Communication Review*, 49(1), January 2019.

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[27] J. Mogul. Unsafe at any speed: Self-driving networks without self-crashing networks. In *ACM SIGCOMM Workshop on Self-Driving Networks*.

[28] L. Peterson, C. Cascone, B. O'Connor, and T. Vachuska. *Software-Defined Networks: A Systems Approach*. 2020. https://sdn.systemsapproach.org.

[29] L. Peterson and O. Sunay. *5G Mobile Networks: A Systems Approach*. 2020. https://5g.systemsapproach.org.

[30] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *ACM Workshop on Hot Topics in Networks*, page 150–156, 2017.

[31] Y. Tsuzaki and Y. Okabe. Reactive configuration updating for intent-based networking. In *International Conference on Information Networking (ICOIN)*, pages 97–102, 2017.

[32] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang. Machine learning for networking: Workflow, advances and opportunities. *IEEE Network*, 32(2):92–99, 2018.