NISA: A Network Instruction Set Architecture for Active Networking

Vishal Satish and Aditya Srivastava
University of California, Berkeley

ABSTRACT

Active networking is a networking paradigm in which "active" switches execute instructions encoded in incoming packets as opposed to simply enacting static protocols. This ability to dynamically execute a rich set of instructions for different incoming packets enables a host of opportunities for the network, such as better network introspection and monitoring, network debugging, and novel traffic engineering. The active networking vision previously failed due to the lack of a killer app and the difficulties inherent in deployment. However, with the advent of programmable switches, we envision a new age for active networking. In this paper, we develop a network instruction set architecture (NISA) to demonstrate the utility of active networks and provide a set of primitives to program them [6]. Additionally, we show how easy NISA is to use by building a simple network monitoring and debuggability tool on top of it.

1 INTRODUCTION

In today's datacenters, network events occur on extremely short timescales on the order of microseconds due to the bursty nature of the workloads that run on top of them [3]. These same workloads demand high throughput and low latency; however, traditional networking paradigms for congestion control fail to guarantee this as they are unable to properly characterize events occurring on such short timescales and appropriately adapt to them. This is because traditional networking paradigms were designed in a static fashion with much longer timescales of WANs such as the Internet in mind. In such cases, it was enough for the control plane to monitor the network at a coarse timescale granularity and only periodically adapt it by updating static routing tables as necessary [4]. With newer bursty datacenter workloads, the network must be able to adapt at the individual packet level - it must be able to field packets differently based on the current network status quo such as switch buffer capacity.

This notion of "active" networking is nothing new - Tennenhouse and Wetherall proposed a similar concept over 25 years ago where packets carry code to be executed at active switches in the network [9]. They posited that such an approach would enable highly adaptive dynamic scheduling. Two roadblocks prevented the widespread adoption of such an approach when it was first introduced: 1) "Active" switch hardware was scarce, and when present it paled in performance when compared to its dumb static counterpart 2) The lack of a killer app to motivate highly adaptive scheduling. For the latter, we must keep in mind that networking at the time was based around the Internet which involved interactions on much larger timescales across large global WANs compared to the short timescales seen in datacenters today. There was just no need for such an adaptive network.

With the recent emergence of bursty datacenter workloads, it become clear that traditional static networking was not going to cut it as it just could not adapt to such rapid changes in the network. By the time a monitoring period had finished and a control plane decision was to be made, hundreds or even thousands of short-lived congestions could have occurred and significantly hampered application performance [3]. Furthermore, as opposed to the Internet, today's datacenters are offered as a premium service in a competitive market and thus customers care very much about network quality. Chronic poor network performance due to congestion can result in lost customers and in turn significant losses in revenue on the order of millions or even billions of dollars for today's large cloud providers such as AWS and Azure.

Now that the importance of active networking has been established, all that's left is to actually implement it. Until recently, this was made difficult due to the lack of active switch hardware that could provide performance on par with traditional static hardware. Now however, recent advances in programmable switching such as the Barefoot Networks Tofino programmable ASIC switch [5] have resulted in programmable hardware that can match the performance of traditional static switches and is affordable, especially given the large budgets of major cloud providers that wish to stay ahead of the game and provide their customers with the best networking experience [2].

Given the recent significance and feasibility of active networking, we propose the Network Instruction Set Architecture (NISA) for active networking - a set of instruction primitives to be executed at programmable switches in the network on top of which higher-level primitives and applications can easily be constructed. Although we acknowledge that our implementation of the NISA is far from perfect due to our limited networking knowledge, by establishing a narrow waist we and future collaborators can continue to improve

and expand it independent of downstream applications. To this end, our contributions are:

(1) A Network Instruction Set Architecture (NISA) consisting of 11 primitives upon which higher-level primitives and applications can be built.
(2) A simple network introspection tool built on top of the NISA that showcases its ease-of-use and versatility for network operators.

2 RELATED WORK

2.1 Active Networking

Active networking is a networking paradigm first proposed by Tennenhouse and Wetherall over 25 years ago where packets carry code to be executed at active switches in the network [9]. Compared to traditional static networking where switches simply enact fixed routing protocols using preconfigured routing tables, in active networking switches interact more closely with packets and execute instructions encoded in their headers. These instructions can run the gamut from toggling deflection-based routing based on dynamically specified buffer capacities to introspection of switch metrics such as average queueing delay. The former end of the spectrum enables fast adapting network protocols that can handle network events on microsecond timescales. Traditional static networking is nowhere near as reactive as modifications to the routing protocol must come from the control plane which cannot tractably be queried on such short timescales.

2.2 Programmable Switching

In recent years, programmable switch ASICs such as the Barefoot Networks Tofino [5] have seen increased adoption as they lower the barrier to entry for testing and later productionizing new networking protocols. Instead of having to overhaul existing switch infrastructure with newer switches that support more SOTA functionality, existing programmable switches can merely be reprogrammed. One early drawback of programmable switches that hampered their adoption was poor performance when compared to traditional static switches. This was because the underlying ASIC hardware had not yet matured. In recent years, switches such as the Tofino have shown to be as performant as their static counterparts. Another factor stunting earlier adoption was the high price of such switches; however, with newer fabrication methods and amortized costs due to mass production, such switches are now much more affordable, especially for large cloud providers such as AWS and Azure.

Programmable switches such as the Tofino expose a Match-Action style interface where incoming packet headers are matched to corresponding actions that should be executed on the packets. Programmers can define custom headers and actions to match against. One unified language for programmable switching is P4 [2] which adopts an Ingress/Egress pipeline where programmers define processors that can be invoked in either or both halves to pre-process headers and select actions.

```
header instr_t {
    bit<8> opcode;
    bit<32> id;
    bit<32> rd;
    bit<32> rs1;
    bit<32> rs2;
}
```

Figure 1: NISA Packet Header Format

3 SYSTEM DESIGN

In this section we will describe the design of the components built in order to realize the NISA. At a high level, we simulate a pod of a fattree network using the *mininet* network simulator [7]. In this environment, we then use the *P*4 language to program a set of programmable switches to support the NISA [2]. Finally, we test and validate that the programmable switches implement the NISA faithfully by writing a *scapy* based Python CLI tool. We will now dive into each of these components in greater detail [8].

3.1 Network Environment

To test the behavior of the NISA, we decided on a pod of a fattree network as our topology since this topology is in common use in real datacenter networks and provides sufficient connectivity to test trickier aspects of the NISA [1]. This topology was implemented in the *mininet* network simulator and we conducted preliminary tests to ensure connectivity of the various hosts and switches.

3.2 Network ISA

Since we envision that this NISA will be supported by programmable switches which expose a match-action interface and utilized by network operators who inject custom instructions into packet headers for execution at switches, we first defined the custom packet header format in the P4 language as seen in Figure 1.

The *opcode* field uniquely identifies the instruction to be executed, serving the same function as in other ISAs.

The *id*, if populated, serves two functions. It can hold the IPV4 address of the switch which should execute the instruction in the case that we want the instruction to execute at a particular switch. It can also hold the IPV4 address of the switch which did in fact execute the instruction. This last case can also serve as a marker for switches indicating that

```
opcode : 0x00;
id     : switch IP;
rd     : 0x1 if instruction has been executed;
rs1    : undefined;
rs2    : undefined;
```

Figure 2: Return to Sender

the instruction has been executed and can be interpreted differently based on whether we want an instruction to only be executed once or if it's fine for an instruction to be executed at multiple switches. For example, consider an instruction to determine the queue length at a particular switch. In this case, we would populate *id* with the IPV4 address of that switch to ensure that only that switch executes it. On the other hand, we may want to find the maximum queue length in a path in the network, perhaps to determine the most congested switch. In this case, we would not populate the *id* field, but instead each switch would in turn execute the instruction and update *id* with its own IPV4 address if its queue length is greater than the maximum seen on the path so far.

The remaining fields, *rd*, *rs*1, and *rs*2, are inspired by the RISCV ISA. The *rd* field can optionally hold the return value for an executed instruction, such as the queue length at a switch. The *rs*1 and *rs*2 fields can both hold optional arguments to an instruction.

We will now describe the three broad categories which can be used to understand the instructions we have defined.

3.2.1 Debugging and Introspection Instructions: The first class of instructions are meant for actively debugging and introspecting the network. The packets with these headers are not meant to carry any meaningful data, but instead function as a medium of communication between a debugging host and one or more switches.

Figures 2, 3, and 4 demonstrate this class of instructions. In Figure 2, we have a return to sender instruction which is executed at the specified switch. This can be used to debug host to switch connectivity and to see if a certain switch is in the path from one host to another. Figures 3 and 4 are return to sender instructions which are conditionally executed if the queue length or queued time at a switch exceeds the specified amount respectively. These instructions can be useful for interactively determine switch characteristics from a debugging host.

While we categorize this set of instructions as meant for debugging and introspection of the network, it is also conceivable that they could be used at a higher level to shape or adapt traffic in the network as well.

```
opcode : 0x01;
id     : switch IP;
rd     : Queue Length at switch;
rs1    : X;
rs2    : undefined;
```

Figure 3: Return to Sender If Queue Length > X

```
opcode : 0x02;
id     : switch IP;
rd     : Queued Time at switch;
rs1    : X;
rs2    : undefined;
```

Figure 4: Return to Sender If Queued Time > X

```
opcode : 0x03;
id     : switch IP;
rd     : Queue Length at switch;
rs1    : X;
rs2    : P;
```

Figure 5: Deflect Packet to port P if Queue Length > X

```
opcode : 0x04;
id     : switch IP;
rd     : Queue Time at switch;
rs1    : X;
rs2    : P;
```

Figure 6: Deflect Packet to port P if Queued Time > X

3.2.2 Traffic Engineering Instructions: Our second class of instructions are intended to help with traffic engineering in the network. For now, we have two instructions which implement conditional deflection switching, which could be useful for reducing load at bottleneck switches in a datacenter network and instead utilize redundant and underutilized paths. Figure 5 demonstrates such an instruction which deflects the incoming packet to a specified port P if the queue length at a switch exceeds a specified value X. Figure 6 demonstrates a similar instruction which conditionally deflects the packet based on queued time.

3.2.3 Passive Monitoring and Adaptation Instructions: Our final class of instructions are meant for passively monitoring the network and perhaps using that information to adapt application traffic. These rely on the fact that programmable switches can generate control messages and send these to hosts. What distinguishes these instructions from the debugging and introspection instructions is that the packets here are meant to carry data from one host to another,

```
opcode : 0x05;
id     : switch IP;
rd     : Queue Length at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 7: Max Queue Length

```
opcode : 0x06;
id     : switch IP;
rd     : Queue Length at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 8: Min Queue Length

```
opcode : 0x07;
id     : switch IP;
rd     : Queued Time at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 9: Max Queued Time

```
opcode : 0x08;
id     : switch IP;
rd     : Queued Time at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 10: Min Queued Time

while the switch is responsible for generating the control message to send back to the host if required.

For example, Figure 11 demonstrates such an instruction, where the specified switch generates a control message containing its current queue length to send back to the sending host. The packet itself will continue to the receiving host undisturbed. Figures 7-12 demonstrate this class of instructions.

3.3 Network Introspection Tool

To test and validate the NISA, we built a network introspection tool. This tool crafts packets with the specified header format encapsulated in Ethernet and IPV4 headers and can send messages between any pair of hosts in the network and with the optional switch *id*. Using this tool, we were able to verify the correct implementation of the NISA on the programmable switches in our network.

```
opcode : 0x09;
id     : switch IP;
rd     : Queue Length at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 11: Queue Length at Switch

```
opcode : 0x0a;
id     : switch IP;
rd     : Queued Time at switch;
rs1    : undefined;
rs2    : undefined;
```

Figure 12: Queued Time at Switch

4 IMPLEMENTATION

There were two main components in our implementation and validation of the NISA which we describe below.

4.1 Programmable Switches

After building and validating the basic pod fattree network topology described previously, we programmed the programmable switches using the P4 language. P4 exposes a match-action interface such that incoming packet headers are matched on certain fields and then specific actions can be executed for these matches.

As an example, let's dive into the implementation of the instruction shown in Figure 5. First, the switch matches on the instruction opcode and invokes the specific function for Deflect Packet to port P if Queue Length > X. At this point, the switch checks its own statistics, and if it's queue length is greater than X, it then sets the packet's egress port to P and sends it on its way. Each instruction described here is similarly implemented.

4.2 Network Introspection Tool

We built the network introspection tool as a simple Python CLI which uses the *scapy* library to construct the appropriate NISA packet given the correct arguments.

5 CONCLUSION

In this paper, we developed an ISA for the network and demonstrated that this network ISA, or NISA, can be easily programmed into programmable switches using the P4 language and be used as a sufficient interface for developing sophisticated higher level network functions, as is the goal of any useful ISA. This NISA brings us one step closer to realizing the active networking vision and addressing some of the more difficult problems in datacenter workloads today

by bringing programmability to the network at the *network* timescale.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review* 38, 4 (2008), 63–74.

[2] Pat Bosshart et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[3] Qiao Zhang et al. 2017. High-resolution measurement of data center microbursts. *Proceedings of the 2017 Internet Measurement Conference* (2017).

[4] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 87–98.

[5] Intel. 2021. Intel Tofino. (2021). https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html Last accessed 12 May 2021.

[6] Vishal Satish and Aditya Srivastava. 2021. NISA. (2021). https://github.com/indianwolverine/nisa Last accessed 12 May 2021.

[7] Mininet Dev Team. 2021. Mininet. (2021). http://mininet.org/ Last accessed 12 May 2021.

[8] Scapy Dev Team. 2021. Scapy. (2021). https://scapy.net/ Last accessed 12 May 2021.

[9] D. L. Tennenhouse and D. J. Wetherall. 1996. Towards an Active Network Architecture. *ACM SIGCOMM Computer Communication Review* 26, 2 (1996), 5–18.