Lecture 21: Programmable networks

Anirudh Sivaraman

2017/11/29

This lecture will be about *programmability* within networks: the ability to continuosly change a network's features/functionalities based on what the network's operator wants. Recall from our first few lectures that networks consist of two components: *end hosts* and *routers*. End hosts are typically server machines that use a general-purpose processor such as an x86 or ARM processor. The use of a general-purpose processor makes these end hosts immediately programmable: if you want to change the functionality of an end host, you code up new functionality in a programming language of your choice, and then compile this into an executable file that consists of assembly instructions for that end hosts' processor.[1]

Given that end hosts are more or less programmable by default, the only remaining component required to enable a programmable network is a mechanism to program the routers within the network. So when the research literature talks about a programmable network, it's really a reference to programmability within the routers (or switches or access points) inside the network. Similarly, we'll focus on how we make routers programmable in this lecture. We'll again start with the history of programmable networks and then talk about two recent approaches to network programmability in increasing order of the scope of their programmability: *software-defined networking* and *programmable data planes*.

1 An abridged history of network programmability

We'll provide a brief overview of the history of network programmability. A more detailed version of this history can be found elsewhere [7, 11].

What's the simplest way to build a router that can take packets coming in on one port and send it to another port? Let's assume we are provided with a mechanism to receive packets on any given port and a mechanism to send packets on any given port. This mechanism could concretely take the form of a peripheral that you can plug into your computer; your computer would then communicate with this peripheral to send and receive packets. Assuming the existence of this peripheral, how would you build a router?

One simple solution is to take a general-purpose processor and implement the router's functionalities as a program on this general-purpose processor. These functionalities include:

1. Running the routing protocol (e.g., link-state or distance-vector routing) to compute routing/forwarding tables for each router

2. Receiving packets

3. Looking up a packet's destination address in the routing/forwarding tables we just computed to find an output port for this packet

4. Sending the packet to that output port

5. Queueing the packet either at the input or output depending on whether the router is input/output queued

6. Dequeueing the packet from its queue and sending it out.

[1] Not all end hosts are *easily* programmable even if they use a general-purpose processor underneath. IoT devices, which we talked about last lecture, are one example. Most IoT devices use an embedded low-power processor, but typically don't allow the user to program these processors. Instead, users interact with these IoT devices through pre-defined applications.

The early routers were built exactly this way. The first device, which would retrospectively come to be called a router, was the interface message processor (IMP) [9] in 1969 on the ARPANET, which was one of the networks on the original Internet. The IMP could support tens of ports, where each port ran at around 50 kbit/s. It ran on top of a minicomputer, a large cabinet-sized machine, which ran at a few MHz!

Routers were built this way up through the mid 1990s. The upside of a router built this way was that it was eminently programmable: you could reload the router with a different program and add a different set of functionalities to the router. However, at some point, general-purpose processors could no longer keep up with the high packet forwarding rates that were expected out of routers because of the explosion of the Internet in the mid 90s. The Juniper M40 in 1998 was probably the first example of using dedicated hardware in the form of a dedicated digital chip in silicon (much like a GPU chip) to solve the problem of increasing packet rates.

The M40 was also likely the first router to distinguish two kinds of router functionalities (the control and the data planes) that run at two different time scales—and to use that distinction gainfully in its system design. The M40 used a dedicated chip for the data plane, which needed to process every packet, but continued to use a general-purpose processor chip for the control plane. As a result of this new control vs. date plane split, the M40 ran an order of magnitude faster than its contemporaries and ushered in an era of hardware routers. Most high-performance routers today use a dedicated hardware chip.

The downside of a dedicated hardware chip is rigidity. Once the data plane has been baked into a silicon chip, it can not be changed—unless you can convince your router vendor to change it. Asking a router vendor to change their data plane chip is like asking Intel or NVIDIA to add a new instruction to their CPU or GPU. It's unlikely that Intel or NVIDIA will do it right away because it can take a few years to produce a new chip or a modified version of an existing chip. It's also extremely costly and the costs can be up to a few million dollars for a single chip production effort. The other downside of a dedicated hardware chip is the inability to fix bugs easily. Most bug fixes are no different from new feature requests at the hardware level and have to go through the same lengthy and costly chip production process.

This rigidity also affects network operators, who continuosly want more out of their network for reasons such as improved network monitoring and troubleshooting, the ability to enforce access control rules for multiple tenants sharing a network, or the ability to prioritize one kind of traffic over another. With a rigid data plane chip, there is no way to add these features unless these features already happen to be present in the chip.

To be fair, router data plane chips have progressively added more features over the last twenty years, but they always seem to lag behind what the operators want because it's a hard task to predict new use cases. This lead to either operator frustration or cumbersome and brittle workarounds. Furthemore, the interface between the router's control and data planes, which was used by the control processor to access these features, was highly vendor specific requiring operators to deal with different interfaces depending on which vendor's routers they were dealing with.

2 Software-defined networking

The first move towards a more programmable network in the era of hardware routers (by which we mean routers with data plane chips) was a movement that later came to be called software-defined networking (SDN). SDN has its roots in internet service provider networks such as AT&T's backbone network [8]. The need for a more programmable network arose from a desire to control the exact path taken by different packets that went beyond vanilla destination-address-based forwarding (a process called policy routing, which we briefly alluded to in lectures 8 and 11).

One example of policy routing is traffic engineering: the ability to split traffic volume in some ratio across different paths in the network. Another example is to make sure one tenant's traffic does not cross over into the part of a network that belongs to a different tenant, a use case that is common in multi-tenant networks such as those found in cloud computing.

The standard practice then was to somehow coerce a distributed routing protocol (such as a link-state or distance-vector routing protocol) to achieve these policy routing objectives using some unintuitive knobs in the routing protocol.

Software-defined networking introduced two concepts to alleviate this problem:

1. *A logically centralized control plane*: SDN lifted the control plane out of the routers and on to a pool of central servers. This central pool of servers would compute the routes for the entire network and then populate the individual router forwarding tables with the results of the route computation. As an example, you could imagine this central pool of servers running Dijkstra's algorithm to compute each router's forwarding table and then sending the routers their forwarding tables alone [6].

2. *A standardized interface between the control and data planes*: SDN further allowed this central pool of route computation servers to communicate the final fowarding tables to *any* router by developing a standardized interface between the control and the data planes, which allowed the control plane to modify router forwarding tables (in the data plane) in a uniform vendor-independent manner.

SDN as an idea went back to the early 2000s at least, but it took the rise of large scale datacenters in the late 2000s for it to really take off. These datacenters, as we have discussed earlier, could house up to 100K servers. Assuming a 10:1 server:router ratio, this translates to 10K routers. Configuring a routing policy on these 10K routers using unintuitive knobs within a distributed routing is extremely error prone and fatigue inducing. In other words, several large-scale network operators (e.g., Google) adopted SDN-like ideas not because it was a good idea (which it was), but because they had no other alternative at that scale.

Another factor that helped speed up adoption was the standardized interface between the control and data plane. The most popular version of this inteface was known as OpenFlow [10], developed in 2008, which took the common miminum denominator of functionality across different router data planes and standardized an interface to that functionality alone as part of OpenFlow. This functionality was the ability for a control plane to add and remove network prefixes into the data plane. It turned out OpenFlow was good enough for most policy routing use cases.

3 Programmable data planes

Soon enough, people started noticing shortcomings with OpenFlow. OpenFlow was developed as the common minimum denominator across *existing* router functionality. In other words, OpenFlow did not add any new router functionality nor did it provide a way to do so. All OpenFlow did (and this was a pretty big step forward) was to standardize an interface to existing router chips so that the route computation servers could communicate with different router chips in the same manner once these chips supported the OpenFlow standard.

Put differently, the hardware within the data-plane chip was still rigid and fixed function: if you did not already have a particular feature in the data plane such as the ability to support different priorities for different traffic classes, you were still out of luck. One place where this need was felt was in the standardization of new protocol formats. The protocol format is a specification of which bits in the packet belong to which packet header, e.g., the first 48 bits of an Ethernet packet correspond to the Ethernet source MAC address header. Existing routers hardcoded the set of protocol formats and could not be tweaked to support a new protocol format that a network operator may have found useful.

In response to this, many router vendors (around the 2013 time frame) started building programmable data-plane chips [1, 4, 2, 5] that provided a small instruction set in the router's hardware, which could then be programmed in software to realize different kinds of router functionality, e.g., the ability to support a new protocol format or the ability to perform user-specified network measurement in the router's data plane. The key difference from a general-purpose processor is that these router chips provide much higher performance (10 to 100x greater packet forwarding rates), but a more limited form of programmability.

You can think of this transition from fixed and rigid data-plane chips to programmable data-plane chips as analogus to the transition from fixed-function GPUs to user-programmable GPUs back in the mid 2000s. During this transition, GPUs went from being a collection of fixed pieces of functionality (the term fixed-function is from the GPU world) to a user-programmable device where the user could specify what this functionality would be. Today user programmability of GPUs has reached the point where many of the user-programmed use cases of GPUs (e.g., deep learning) have nothing to do with graphics any more.

The transition to programmable data-plane chips is still ongoing and it is still a bit early to say how it will all play out. At the very least, it seems clear that programmable data-plane chips will benefit vendors of these chips for at least two reasons: (1) the ability to add new features in software or firmware in a few weeks instead of spending a few years and a few million dollars on a new chip, (2) the ability to fix bugs in software or firmware without necessitating costly chip recalls [3] or chip redesign.

References

[1] Barefoot: The World's Fastest and Most Programmable Networks. `https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf`.

[2] Intel FlexPipe. `http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf`.

[3] Pentium FDIV bug. `https://en.wikipedia.org/wiki/Pentium_FDIV_bug`.

[4] XPliant^{TM} Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[6] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, 2005.

[7] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Computer Communication Review*, 2014.

[8] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Computer Communication Review*, 2005.

[9] Frank E Heart, Robert E Kahn, SM Ornstein, WR Crowther, and David C Walden. The interface message processor for the ARPA computer network. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, 1970.

[10] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[11] Anirudh Sivaraman. *Designing Fast and Programmable Routers*. PhD thesis, Massachusetts Institute of Technology, September 2017.