

Wanted: Systems abstractions for SDN

Sapan Bhatia
sapanb@cs.princeton.edu
Princeton University

Andy Bavier
acb@cs.princeton.edu
Princeton University

Larry Peterson
llp@cs.princeton.edu
Princeton University

Abstract

This paper presents a case for applying the principles of Software-Defined Networking (SDN) to middleboxes and end hosts. The challenges of configuring networking on network hosts resemble those addressed by SDN – numerous multi-vendor components, each with its own syntax and idiosyncratic corner cases, must be orchestrated smoothly. We have developed a prototype called NativeClick, a novel use of the Click Modular Router language, to orchestrate Linux networking tools. NativeClick demonstrates that, while existing SDN efforts have produced insufficient Abstractions to cover a wide range of networking behavior, SDN-like abstractions can make host configurations modular.

1 Introduction

Software Defined Networking (SDN) is unraveling the complexities of today’s networks. While the term SDN is sometimes used to refer to a specific set of technologies [6], we believe that its power lies in its basic idea, which is to *enable modular network configuration through the use of software abstractions* [12]. SDN achieves modularity by cleanly separating the network “control plane” from its “data plane” through a well-defined interface. This modularity makes control plane configurations amenable to standard software engineering practices, and potentially leads to implementations that are simpler to debug, reason about, and extend.

This paper argues that while SDN has so far been applied to the configuration and management of components *within the network*, it can be taken much further. The problem of network configuration does not stop at the network interface (be it virtual or physical); to be truly end-to-end, it must also include the ability to configure the network subsystems of the end hosts and middleboxes that connect to the network.

Network hosts typically run commodity operating systems. This is an understandable choice, as these OSes contain full-featured, robust, and flexible networking stacks. They provide a wide variety of established

and experimental transport protocols, L2/L3 forwarding, packet translation (e.g., NAT), packet filtering, traffic shaping, tunneling, and more. The portability of UNIX-based OSes such as Linux and NetBSD makes them a good choice for network appliances such as wireless access points, enterprise gateways, web caches, transparent web proxies, and traffic monitors. Most have distributed, open source development models that lead to fast integration with new technological standards as well as cutting edge techniques from academia and industry.

The power of commodity OS networking comes at the price of complexity. For example, Linux presents system builders with a rich variety of standard networking tools: iptables, iproute, tc, ip tunnel, dnsmasq, ebtables various traffic generators, load balancers, and transparent proxies. These tools are stable and offer immense flexibility. However, most of these tools have been developed independently; each tool incorporates idiosyncratic concepts and syntax; and many are poorly documented. The task of assembling heterogeneous tools is laborious, and leads to legacy configurations that are hard to modify and evolve.

At first glance it may seem unnatural to conflate the problem of configuring network hosts with the problem of configuring switches in the network – each involves a different set of technologies and its own development ecosystem. Our experience managing network hosts has led us to think that the problems with configuring them are strongly reminiscent of those facing traditional, pre-SDN network configuration. There are many parallels between the two worlds:

- It is a requirement to leverage existing, mature components that are currently in production use.
- Components from multiple vendors or developers need to coordinate to forward packets.
- Each component is configured in isolation using its own specific configuration method and syntax.
- It is difficult to visualize network packet flow at a high level, such as in a graph representation.

A unified programming model that runs end-to-end would address these concerns that are common to both settings. In the specific context of *end-host configuration*, what we mean by *unified network configuration* is something that meets two requirements:

1. In the spirit of SDN, it must provide a clean, abstract, and most importantly, *holistic* view of the network stack’s configuration.
2. It must leverage the existing, high-performance, production-quality, full-featured, widely-used OS networking stacks and configuration tools, as these tools are the main reason the OSes were chosen in the first place.

A number of systems have attempted to enable holistic network configuration in the past, among them the Click Modular Router [5]. Click abstractly represents network configuration as a graph of modular elements that perform specific packet processing functions; packets flow along the edges of the graph. We believe that Click can aptly satisfy our first requirement. Unfortunately, Click as originally proposed does not meet the second requirement: Click’s runtime layer replaces the existing networking stack. It seems almost certain that the network stack of the Linux kernel, with its thousands of developers and widespread production use, is more stable, portable, and standards-compliant than Click.

Seeing this limitation, we pose a question: is it possible to use the *Click language* to configure the *Linux networking stack*? To answer this question, we designed and developed a prototype called NativeClick that addresses the key challenges of using Click without its runtime layer. NativeClick compiles a Click specification to a collection of scripts that invokes standard Linux tools. A key requirement of our approach is to be able to modularize Linux tools as Click elements and to direct traffic between the elements. We achieve this through the use of a novel mechanism that combines OS-level containers and virtual point-to-point (VPPP) interfaces. When used together, these ideas modularize the Linux networking stack into a Click-like packet flow graph.

NativeClick solves a specific problem, but in doing so it adopts SDN’s mission of using clean, modular abstractions to orchestrate production-quality networking components. The fact that existing SDN efforts currently lack similar abstractions raises several questions:

- What is the appropriate user interface to visualize and configure networks end-to-end?
- Is it possible to define an SDN framework on end hosts that plugs into OpenFlow – a key enabler of SDN in the network?

- Can Click be used as a unified, end-to-end network programming language?
- Can the Click language be used to specify networks that leverage OpenFlow as a low level packet forwarding mechanism?

We devote much of the remainder of the paper to discussing specific challenges on end hosts, presenting the design of NativeClick along with a preliminary performance analysis. We then revisit our long term vision and discuss open problems.

2 Challenges

Our ultimate goal is to define a programming framework in which it is possible to implement complex configurations with some programming skills and a good understanding of basic network concepts—arcane knowledge of vendor-specific tools should not be a requirement. The Linux networking stack and the tools that configure it, as they stand, are far removed from this level of usability. We have anecdotal evidence to suggest that the challenges facing Linux also apply to other UNIX-based operating systems. While we think it would be interesting to evaluate the applicability of our work to other OSes, for now we defer it as future work and focus our attention on the Linux network stack.

In this section, we enumerate the challenges in the path of achieving our goal. Each challenge is accompanied by an example to illustrate its impact in practice, along with a solution based on a software abstraction. The examples reflect some of our own experiences using the Linux networking stack in practice. The abstractions collectively make up the design of NativeClick.

Challenge: *Orchestrating multiple network management tools is laborious and error prone.*

On Linux, networking is implemented through an interplay of management tools. For example, the device configuration tool sets the IP address and queue lengths of various network devices, the iptables tool configures the network stack with packet filtering and classification rules, and the tc tool shapes traffic. The tools have dependencies; in this example, iptables may classify traffic based on the address assigned to the network device and tc might use the ensuing classes to shape traffic. Keeping track of these dependencies can be a daunting task, and the resulting configurations fragile. Each individual tool has its own specific method of configuration and its own syntax.

Example: When the IP address of a network interface is changed, all entries in the route table and the traffic policy table that involve that device are silently dropped, requiring a restart of those services.

Solution: *Consolidated configuration using the Click modular router.*

As mentioned earlier, our solution to this problem is to consolidate the configuration of all aspects of networking through a unified programming framework inspired by the Click Modular Router. Click is an elegant approach to modular network configuration, modeling network processing as a packet flow graph of self-contained elements.

The Click design consists of an architecture, a language, and a runtime system. In the Click *architecture*, a network configuration is defined as a graph of fine-grained processing operations called *elements*. The basic interface of an element is a set of input and output ports. Packets are handed off by an element via its output ports to other elements via their input ports. The basic interface can be extended to expose properties of the element's state, such as buffer sizes and queue lengths. The Click *language* allows the user to specify a graph of elements using a simple and intuitive syntax. Although it contains many advanced features, any Click configuration can be written using three constructs for defining elements, ports, and the connections between ports. Finally the Click *runtime* implements a library of useful elements, along with the architectural framework required to compose them into a graph.

NativeClick adopts the architecture of Click and its language, but discards its runtime. The architecture of Click satisfies our needs for configuring network processing in an abstract and holistic manner. The Click language helps specify an entire networking configuration in a single program with an intuitive syntax, as opposed to Linux, in which configurations are scattered across multiple files. In Click, the dependencies between elements are explicit – so for instance in the example above, elements with dependencies could be reloaded when one of the dependencies changed.

Challenge: *Traffic flow is implicit, inflexible and hard to visualize.*

The flow of packets through the Linux network stack is implicit and hard to visualize. It may be argued that the underlying problem is scant documentation – and that exactly what path packets follow through various modules in the network stack can be discerned from source code. Even though this may be true for a specific version of Linux, since the notion of a path is not formally defined, it is problematic to ensure that the same paths are preserved between versions. Moreover, the path that packets take is inflexible. It is impossible to reverse the direction of packets between two processing steps, even if both directions preserve the semantics of the protocols involved. The ability to visualize traffic flow at an abstract, tool-independent level would be a significant step

towards bringing this type of configuration to a wider audience, in addition to simplifying it for experts.

Example: Packet filtering in the Linux network stack does not affect UDP packets generated by some DHCP daemons. This is because the daemons generate these packets over raw sockets, and the raw sockets receive the packets before the packet filtering modules have acted on them.

In a Click specification, by contrast, the path that packets take can be traced precisely from the traffic source to terminal processing elements. Users have complete freedom in changing this path. They can do so simply by changing the connections between element ports. The explicit hand off of packets from one element to the next also gives Click the familiarity of a general purpose programming language. Elements in Click act on a packet like functions and procedures act on input values in a general purpose program.

The challenge for NativeClick is to start with a Click specification containing explicit packet processing paths, and enforce these same processing paths in the Linux OS.

Solution: *Use virtual point-to-point (VPPP) links.*

A virtual link is a pair of point to point devices that acts like a local tunnel. Packets entering one device are emitted on the other and vice versa. Since all of the Linux networking tools have the ability to operate on a given network device, tools that need to be connected can simply be placed on the two sides of a virtual link.

For NativeClick, VPPP links implement the ports and graph edges of Click's architecture. A port equals a VPPP device inside a container, and the VPPP links connect the containers into a network. Packets flow across a VPPP link into a container, are processed by the element that resides there, and are forwarded on a different VPPP link or another network device.

Challenge: *Linux tools can interfere in unpredictable ways due to the lack of modular interfaces.*

Linux tools do not have a well-defined interface that they can use to interoperate safely. Instead, they interoperate by committing changes to the global state of the network stack. These changes can conflict, and as a result, compositions of tools can behave in unpredictable ways.

Example: The network stack can be configured via several global configuration parameters. One of these parameters `rp_filter` enables defense against IP spoofing, and is turned on by several firewalls. We have encountered cases in which IP spoofing was used as an exploratory mechanism. Painstaking debugging was necessary to reveal the conflict.

Click addresses this problem by clearly defining fine-grained modular units of processing called elements that perform a single function, such as packet encapsulation or address rewriting. The challenge for NativeClick in modularizing Linux tools as Click elements is twofold: to break down complex tools into simple, Click-like operations, and to encapsulate these operations in a modular interface.

Solution: *Modularize elements as executable scripts in OS-level containers.*

NativeClick elements are executable scripts that can be written in a general-purpose language. The scripts invoke the networking tools in simple ways – for example, an element might invoke a single iptables rule or set up a single traffic shaping queue with tc. Therefore NativeClick elements have roughly the same processing granularity as those of Click. To start an element, NativeClick runs the appropriate script inside a container with several command-line arguments: the start flag, a set of input and output ports (mapping to virtual links already created in the container by NativeClick), and a set of named configuration parameters.

NativeClick enforces isolation between elements by encapsulating them in OS-level containers. In Linux, an OS-level container is an enhanced process with its own virtualized network stack, file system mounts, system variables such as the hostname, etc. – in addition to having its own copies of resources virtualized by traditional processes, e.g. virtual memory and the file descriptor table. The virtualized network stack includes network interfaces, the route table, stack configuration flags, and traffic filtering and shaping rules. The greatest benefit of using this type of virtualization in NativeClick is its low overhead, which is comparable to that of traditional processes.

Placing each element in a container gives elements free rein to configure the stack in arbitrary ways, while preventing elements from interacting with each other. In essence, containers give NativeClick the modularity of the Click architecture.

To sum up, Figure 1 illustrates the basic ideas behind NativeClick. Part (a) shows a portion of a standard Click graph with elements connected by ports; the Click runtime implements these components in C++. Part (b) shows an equivalent graph in NativeClick, with elements implemented by Python scripts running inside OS containers and wrapping Linux networking tools, and ports mapping to VPPP links.

3 Performance

There are two possible sources of overhead in NativeClick: the packaging of processing operations as el-

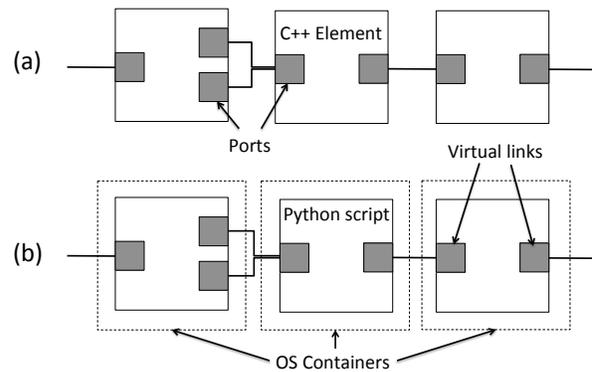


Figure 1: Intuition behind NativeClick

ements, and the interfaces that connect elements. NativeClick elements do not add any new processing to the data path; the performance of an element simply reflects the performance of the underlying mechanism it wraps. In some cases, it may be less efficient to compose simple elements to perform a function rather than directly using a specialized element designed for the purpose. Such tradeoffs were examined in the original Click work, and were found to be insignificant in their implementation. In the future we plan to evaluate similar tradeoffs and possible optimizations for NativeClick.

Unlike elements themselves, interfaces between elements do constitute additional processing in the data path. Interfaces help realize the explicit hand off of packets, which is indispensable for enabling the use of the graph paradigm, so they cannot be fully eliminated. The best we can do is to keep their overhead low, and to that end, we make use of lightweight constructs, namely OS level containers and virtual links.

Preliminary microbenchmarks show that these overheads are not prohibitively high. Our apparatus consisted of a single 12 core Xeon server with 48GB of RAM running the 3.x series of the Linux kernel. Our experiment created chains of NativeClick elements of varying lengths, and tested UDP throughput through these chains using iperf.

For a chain length of 0 (i.e. no interfaces) the throughput in one direction was 810Mbps. This amounts to 1.6Gbps of total packet processing throughput, since both client and server were running on a single host. For a chain length of 10, the throughput reduced to 745Mbps, or 8.6% below the original throughput, which we believe does not constitute an unacceptable performance degradation.

We see two opportunities for optimizing NativeClick’s performance. The first involves avoiding the use of con-

tainers and virtual links when they are not required. For example, in cases in which the path of packets in the Linux network stack exactly matches the expected path between two elements, the elements can be merged. The second consists of optimizations enabled by virtue of having a global view of networking. Many such optimizations are already covered by the Click work [4].

From this preliminary testing, we conclude that it is feasible to pursue NativeClick’s approach to imposing modularity on the Linux network stack and tools. We plan to perform a more thorough evaluation through macrobenchmarks and more varied traffic in the future.

4 Related Work

Open vSwitch (OVS) [10] is a software implementation of an OpenFlow switch, now a part of the Linux kernel, that is designed for dynamic, multi-server virtualization environments (a.k.a. “the Cloud”). Its goal is to enable cluster-wide logical abstractions by providing a unified framework for configuring packet forwarding, traffic shaping, tunneling, and flow monitoring on end hosts. OVS only subsumes functionality related to L2/L3 packet forwarding; in contrast, NativeClick provides a holistic view of the entire Linux networking stack.

Modular abstractions for programmable host networking go back at least 20 years. The x-kernel [8] was a modular framework for implementing network protocols within a commodity OS. A follow-on project, the Scout OS [7], used the path abstraction to make processing of packet flows explicit via chains of modules. Our thinking about network configuration was inspired by these systems as well as by Click.

Due to their negligible I/O overhead, OS containers are a natural choice for virtualizing the networking stack on end hosts. The VINI testbed [1, 2] leverages OS containers as lightweight virtual machines that can act as programmable routers. The Mininet simulation framework [3] uses network namespaces and virtual links to build virtual software-defined networks of up to 4096 hosts and switches on a single PC. We expect that NativeClick will be able to support configurations consisting of thousands of elements.

5 Beyond the Network Host

The key principle of SDN is to define modular interfaces through software abstractions that run across network components. Our approach with NativeClick directly applies this principle: we overlay abstractions developed in the Click Modular Router (Click elements and ports) onto Linux mechanisms (containers, virtual links,

executable scripts) to modularize the Linux toolset. For this reason, we place our work in the broad SDN effort.

The commodity OSes that run on these hosts play a significant role in the creation of networks today: their problems are therefore also the problems of networking at large. We have examined a small part of this problem by modularizing the Linux network stack in the form of Click packet flow graphs. In doing so, we hope to have demonstrated that networking on OSes has problems similar to the ones on networks, and that these problems are not solved using networking abstractions.

A good illustration of this point is the Open vSwitch (OVS) software [11], which is a virtual OpenFlow switch implementation for Linux. One of the goals of OVS is to bring SDN principles to the end host. While there are compelling benefits to using OVS, especially in the realm of VM orchestration in Cloud environments, these benefits are restricted to the networking side of the problem. OVS does not replace the rich library of powerful and mature tools available on commodity OSes, and the decision not to aim for this objective is by design [9].

Given the full span of problems that run from network switches and routers to end host network stacks, finding the abstractions for a programming model that truly takes a holistic view without neglecting the extremities is an open problem. One of the key messages of this paper is that SDN currently lacks abstractions that cover network host functionalities and that it would be beneficial for the systems community to explore such abstractions.

Moving forward, we would like to explore the idea of using the Click language as a framework for holistic programming that includes components in the network as well as tools on the end host. NativeClick takes two important steps in this direction. First, it implements the host side of the programming environment. Second, it unbundles the Click language from its runtime. We do not fully understand the challenges involved in packaging network components as Click elements, but we now have a framework for evaluating various possibilities, such as connecting via the OpenFlow interface or defining vendor-specific elements.

References

- [1] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proceedings ACM SIGCOMM 2006 Conference* (Pisa, Italy, Sep 2006).
- [2] BHATIA, S., MOTIWALA, M., MUHLBAUER, W., MUNDADA, Y., VALANCIUS, V., BAVIER, A., FEAMSTER, N., PETERSON, L., AND REXFORD, J. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proc. ROADS 2008/CoNEXT 2008* (Madrid, Spain, Dec 2008).
- [3] HANDIGOL, N., HELLER, B., JEYAKUMAR, V., LANTZ, B., AND MCKEOWN, N. Reproducible Network Experiments using

- Container-Based Emulation. In *Proceedings of CoNEXT 2012* (Nice, France, Dec 2012).
- [4] KOHLER, E., MORRIS, R., AND CHEN, B. Programming Language Optimizations for Modular Router Configurations. In *Proceedings of the 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, California, Oct 2002).
 - [5] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (Aug 2000), 263–297.
 - [6] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.
 - [7] MOSBERGER, D., AND PETERSON, L. L. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)* (Seattle, Washington, Oct 1996), pp. 153–167.
 - [8] PETERSON, L. x-kernel Home Page. <http://www.cs.arizona.edu/xkernel/>.
 - [9] PETIT, J. Openswitch mailing list. Jun 2011. <http://openswitch.org/pipermail/discuss/2011-June/005341.html>.
 - [10] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking into the Virtualization Layer. In *Proceedings of HotNets–VIII* (New York, New York, Oct 2009).
 - [11] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. *Proc. HotNets (October 2009)* (2009).
 - [12] SHENKER, S. The Future of Networking, and the Past of Protocols. <http://opennetsummit.org/talks/shenker-tue.pdf>.