DOI:10.1145/2661061.2661063

New abstractions are critical for achieving SDN goals.

BY MARTIN CASADO, NATE FOSTER, AND ARJUN GUHA

Abstractions for Software-Defined Networks

SOFTWARE-DEFINED NETWORKING (SDN) has received a lot of attention in recent years as a means of addressing some of the long-standing challenges in networking. SDN starts from two simple ideas: generalize network hardware so it provides a standard collection of packet-processing functions instead of a fixed set of narrow features, and decouple the software that controls the network from the devices that implement it. This design makes it possible to evolve the network without having to change the underlying hardware and enables expressing network algorithms in terms of appropriate abstractions for particular applications.

Figure 1 contrasts the architectures of traditional networks and SDN. In SDN, one or more controller machines execute a general-purpose program that responds to events such as changes in network topology, connections initiated by end hosts, shifts in traffic load, or messages from other controllers, by computing a collection of packet-forwarding rules. The controllers then push these rules to the switches, which implement the required functionality efficiently using specialized hardware.

Because SDN does not specify how controllers are implemented, it can be used to implement a variety of network algorithms, including simple ones such as shortest-path routing, and more sophisticated ones such as traffic engineering.

Many novel applications have been implemented with SDN including policy-based access control, adaptive traffic monitoring, wide-area traffic engineering, network virtualization, and others.[6,9,16,18–20,44] In principle, it would be possible to implement any of these applications in a traditional network, but it would not be easy: the programmer would have to design new distributed protocols and also address practical issues because traditional switches cannot be easily controlled by third-party programs.

Early SDN controller platforms exposed a rudimentary programming interface that provided little more than a thin wrapper around the features of the underlying hardware. Where there were higher-level abstractions, they reflected structures already found in traditional networks such as topology or link-state information. However, there is now a growing body of work exploring how SDN can change not only which control algorithms can be

» **key insights**

- **SDN is a new network architecture that decouples the software that controls a network from the devices that implement it.**
- **By providing global visibility into network state, SDN can dramatically simplify the way that many network algorithms are expressed.**
- **SDN also makes it possible to evolve the functionality of a network without having to change the underlying hardware.**
- **SDN is enabling the development of new network programming models, systems abstractions, and verification tools.**

IMAGE BY AVN PHOTO LAB

thateasily expressed, but how they can best be written. Just as modern operating systems provide rich abstractions for managing hardware-level resources, we believe that similar abstractions will be needed for networks to fully realize the vision of SDN.

These abstractions are the topic of this article. We review recent and ongoing work on improving SDN programming models and abstractions, focusing on the following areas:

Network-wide structures: SDN controllers are built using relatively small collections of tightly-coupled servers, which makes them amenable to distributed algorithms that maintain consistent versions of network-wide structures such as topology, traffic statistics, and others.

Distributed updates: SDN controllers manage the entire network, so they must often change rules on multiple switches. Update mechanisms that provide consistency guarantees during periods of transition can simplify the development of dynamic programs.

Modular composition: Many network programs naturally decompose into several modules. Controllers that provide compositional programming interfaces make it easy to specify orthogonal aspects of network behavior in terms of modular components.

Virtualization: Decoupling application logic from the physical topology simplifies programs, ensures isolation, and provides portability. Virtual network abstractions can also provide enhanced scalability and fault tolerance.

Formal verification: To help programmers write correct programs, some controllers provide tools for automatically checking formal properties and diagnosing problems when unexpected errors occur.

Here, we explore these abstractions in further detail. To provide a common basis for discussion, we begin by introducing OpenFlow as a concrete instance of SDN.

OpenFlow

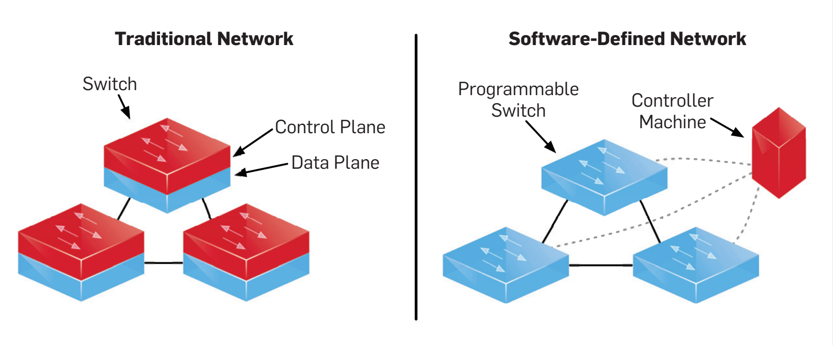
The OpenFlow specification defines a standard collection of features switches must provide, as well as an interface controllers can use to communicate with switches: instructions for installing and deleting forwarding rules, and notifications about flows, topology, and traffic statistics.[31]

An OpenFlow switch maintains a *forwarding table* that contains a list of prioritized *rules*. Each rule has a *pattern* that describes a set of packets and *actions* that describe transformations on packets. When a packet arrives at a switch, the switch finds a rule whose pattern matches the packet headers and applies the associated actions. If multiple rules match, the switch applies the actions of the highest priority rule, while if no rules match, the switch encapsulates the packet in an OpenFlow message and sends it to the controllers. The controllers can either process the packet directly, or send messages back to the switch instructing it to install or delete rules in its forwarding table. The maximum size of a table is determined by hardware constraints, but most switches have space for at least several thousand rules.

To support traffic monitoring, every rule has associated counters that keep track of basic statistics such as the number and total size of all packets processed with that rule. Controllers can read these counters using OpenFlow messages. They can also configure the physical ports on a switch by creating queues that rate-limit traffic or provide minimum bandwidth guarantees—features that are useful for implementing traffic engineering applications.

As an example, consider the accompanying table: Read from top to bottom, these rules block all SSH traffic, forward non-SSH traffic destined for hosts 10.0.0.1 and 10.0.0.2 out ports 1 and 2 respectively, and divert all other traffic to the controller for further processing.

Figure 1. Traditional and software-defined architectures.



Example of OpenFlow forwarding table.

Priority	Pattern	Action	Counters
30	TcpDstPort = 22	Drop	⟨7156, 124⟩
20	IpDstAddr = 10.0.0.1	Forward 1	⟨2648, 38⟩
10	IpDstAddr = 10.0.0.2	Forward 2	⟨14184, 246⟩
0	*	Controller	⟨1686, 14⟩

Network-wide Structures

A major advantage of SDN is that the controllers can compute network-wide structures that give global visibility into network state, using distributed algorithms that provide strong guarantees about the consistency of these structures across controllers. It would be practically infeasible to maintain these network-wide structures in a traditional network where control is distributed across a larger number of devices, but by using them the logic of many applications can become much simpler. For example, shortest path routing can be implemented by evaluating Dijkstra's algorithm over the structure representing the topology.[41]

Example. To illustrate, consider the task of maintaining a spanning tree that connects the switches in the network. Such a tree could be used to forward broadcast traffic without any danger of forwarding loops. Designing a distributed algorithm to construct and maintain a spanning tree is surprisingly difficult because it must work correctly in arbitrary topologies and rapidly reconverge to a new tree when events such as unexpected device or link failures occur.

Traditional solution. The classic way to build a spanning tree is to use the *spanning tree protocol*[36]—a fully distributed protocol, in which the switches periodically exchange information with their neighbors using pairwise announcements. The switches agree on a root node by running a distributed leader election protocol, and then construct the spanning tree incrementally from that node, enabling and disabling links to select the shortest path to the root, and breaking ties using switch identifiers. Note that an implementation of the spanning tree protocol requires neighbor discovery, leader election, as well as the actual tree construction algorithm, but because these components are specific to the protocol, their logic cannot be easily reused by other protocols that require similar functionality. Moreover, when the topology changes, the time to calculate a new tree scales with the size of the longest loop-free path.

SDN solution. Most SDN controllers provide a suite of common functions that arise in many applications such as topology discovery and link fault detection and also maintain structures that keep track of information about the state of the network such as host locations, link capacities, the traffic matrix etc. The database that stores this information is often called a Network Information Base (NIB).[25] Using a NIB, an SDN implementation of spanning tree can be dramatically simpler than its distributed counterpart: whenever the topology changes, it simply computes a spanning tree from the topology using Prim's algorithm, and installs rules on switches that forward along the tree.

Richer applications. By providing programmers with information about the state of the entire network, the NIB also makes it easy to implement richer applications such as traffic engineering that would be difficult to realize in traditional networks.[11] For example, the B4 and SWAN systems use SDN to balance load across the wide-area links between datacenters, achieving much higher utilization than was possible with traditional approaches.[18,19] These applications require distributed controllers that automatically manage data replicated across many controllers through the NIB.[26]

SDN can make many network programs vastly simpler by providing network-wide structures and allowing common distributed programming abstractions to be implemented once and reused across many applications.

Using multiple controllers addresses important issues such as scalability and fault tolerance—for example, one controller can take over for another if its load becomes high, or if its links with the switches fail. However, because the number of controllers is typically small, these controllers can use algorithms such as Paxos—something that would not scale in fully distributed settings. Hence, although controllers do use distributed algorithms, they are simpler and often converge faster than traditional protocols since there are fewer controllers than switches.

Discussion. SDN can make many network programs vastly simpler by providing network-wide structures and allowing common distributed programming abstractions to be implemented once and reused across many applications. Such reuse is effectively impossible in traditional networks, where forwarding and control are tightly coupled on each device, implementations of functions such as leader election are tied to specific protocols, and devices have varying CPU, memory, and storage capabilities.

Distributed Updates

In traditional networks, it is often acceptable for configuration updates to be merely eventually consistent. For example, if the network configuration is recalculated due to a link failure, a packet may traverse a switch once in the original state and a second time in the updated state. This can lead to behaviors such as forwarding loops or dropping packets, but since most networks only provide best-effort delivery, as long as the network eventually converges to the new state, transient errors during the transition may be acceptable. However, eventually consistent updates do not always suffice in SDN. For example, an SDN controller might manage filtering rules in addition to forwarding rules, and these rules may be critical for ensuring invariants such as access control or isolation between the traffic of tenants sharing the network. If configuration updates are propagated to switches in a merely eventually consistent manner, these invariants can easily be violated during periods of transition.

Programmers can sometimes work around these problems by carefully ordering updates so that packets only traverse paths whose configurations have been fully propagated into the network. For example, a programmer might update the ingress switches first, and check that all partially updated paths in the interior of the network are otherwise unreachable during the transition. But calculating orderings manually is complicated and makes updates slow to roll out. Recent work has investigated abstractions that provide general mechanisms for handling distributed updates as well as guarantees ensuring packets never "see" a partially updated path. The idea is to attach versions to configurations and carefully design update protocols that ensure every packet (or set of related packets) is processed by a single consistent version.

Examples. The need for configuration updates that provide strong consistency is a significant departure from traditional networks. To demonstrate they are not only of academic interest, consider the following scenarios:

- **Shortest-path routing:** Initially the network is configured to forward along shortest paths. Then the operator decides to take several switches down for maintenance. The controller generates a new network-wide configuration that forwards along a different set of paths. At all times, the network is expected to provide connectivity and be free of forwarding loops.
- **Distributed access control:** Initially the network is configured to filter a set of "forbidden" packets and otherwise forward along shortest paths. Because the filtering rules are too large to fit into a single forwarding table, the rules are distributed across several switches in the network. The configuration is carefully constructed to ensure each packet traverses the appropriate switches containing the necessary filtering rules. Later, the operator decides to rearrange the rules, maintaining the same policy but placing filtering rules on different switches. At all times, the network is expected to filter forbidden packets and forward other packets to their destinations.
- **Server load balancing:** Initially the network is configured to redirect incoming requests to several back-end server replicas. At some point, more servers are brought online. The controller then generates a new configuration that balances the load among the new set of servers. At all times, the network is expected to forward incoming traffic to one of the back-end servers while ensuring connection affinity—all packets in a connection should be sent to the same server.

Beyond the basic abstraction of versioning, the state update subsystem of the controller can expose multiple consistency models to the application.

In each of these scenarios, computing the initial and final configurations is straightforward, but transitioning between them while preserving the desired invariants is not. In particular, because the controller lacks the ability to update the state of the entire network atomically, packets traversing the network will necessarily be processed by old, new, or even intermediate configurations containing a mixture of forwarding rules from both configurations.

Update abstractions. Consistent update abstractions allow a controller to update the forwarding state of the entire network while ensuring a packet will never traverse a path that is in transition between two states. The abstractions themselves are straightforward to describe: the controller program specifies the version of the state being pushed into the network and the update subsystem guarantees that each packet traversing the network only "sees" a consistent version of the state. Beyond the basic abstraction of versioning, the state update subsystem of the controller can expose multiple consistency models to the application.

One possible model is per-packet consistency: each packet is processed using a single version of the forwarding state.[39] That is, every packet is either processed with the old network-wide configuration, or the new configuration, but not a mixture of the two. Another model is per-flow consistency: every set of related packets is processed using a single configuration version.[39] Other extensions consider bandwidth and attempt to avoid creating additional congestion during the transition.[18,27]

Update mechanisms. A general mechanism for implementing consistent updates is to use a two-phase update. As its name suggests, a two-phase update proceeds in two steps: the controller modifies the new configuration by instrumenting the forwarding rules so they only match packets stamped with a tag corresponding to the new version, and installs it on every switch; the controller updates the rules at the perimeter of the network to stamp packets with the new version tag, and uninstalls the old configuration from every switch. Although the network contains a mixture of rules from the old and new configurations during the transition, these rules have the property that any given packet will be processed according to a single version. Similar mechanisms can be used to implement per-flow consistency.[39]

In many situations, optimized mechanisms can be used in place of two-phase update. For example, if the update only adds paths, then only rules that impinge on those paths need to be updated. Likewise, if the update only affects a subset of the switches (and the policy has the property that it never forwards traffic across those switches more than once) then the other switches do not need to be updated at all. These optimized mechanisms generate fewer messages, use less rule space on switches, or complete the transition more rapidly than full two-phase update. Consistent updates can also be implemented incrementally[21] or by diverting some packets to the controller.[31]

Discussion. Updates are a fundamental abstraction for any SDN controller. But despite some promising initial results, many open questions remain. An obvious concern is efficiency: the mechanisms just described require substantial space for rules and a large number of control messages to implement transitions. In large networks, the costs of these mechanisms would be prohibitive. The optimizations discussed here are a good start, but a more comprehensive investigation is needed. Another important issue is the responsiveness of updates. The abstractions described in this section make no guarantees about how long an update will take to complete. For planned changes, this may be acceptable, but when reacting to failures, a fast response is essential.[38] It would be interesting to explore abstractions that trade off weaker guarantees for more responsive update mechanisms. For example, an abstraction that only guarantees packets ultimately reach their final destination and do not traverse loops seems natural, and would admit more efficient implementations. Finally, it may be useful to synthesize updates from application-specific invariants.[28,35]

Modular Composition

In operating systems, processes allow multiple users to share the available hardware resources on a single machine. Each process is associated with a thread of execution, along with system resources such as memory, locks, file descriptors, and sockets. The operating system requires all interactions between processes take place over well-specified interfaces. For example, memory allocated to one process cannot be tampered with by another, unless it has explicitly been shared by the first process. Although SDN controllers have been compared to "network operating systems," current controllers lack abstractions analogous to processes.[13] Instead, most controllers give applications unfettered access to the forwarding tables on every switch in the network, which makes it difficult to write programs in a modular way.

This is unfortunate, because network programming should lend itself naturally to modularization. SDN applications are commonly built out of standard building blocks such as routing, broadcast, monitoring, and access control. However, the lack of modularity in most SDN controllers forces programmers to reimplement these fundamental services from scratch in each new application instead of simply obtaining them from libraries.

Examples. The following scenarios illustrate why modularity can be difficult to achieve in current SDN controllers.

Forwarding and monitoring: The network implements forwarding and traffic monitoring. Because switch tables implement both features, the rules must be carefully crafted to forward and monitor certain packets but only forward or monitor others. If the programmer executes standard forwarding and monitoring programs side-by-side, the programs may install overlapping rules and the overall behavior of the system will be unpredictable.

Forwarding with isolation: The network is partitioned into two sets of hosts. Each set is isolated from the other, but the network forwards traffic between pairs of hosts in the same set. As with the previous example, the program decomposes into two orthogonal functions: isolation and forwarding. However, the programmer must consider both functions at once as rules generated by one module could easily forward traffic to hosts in the other set, violating the intended policy.

Low-latency video and bulk data transfer: The network provides low-latency service to a videoconferencing application and allows a backup application to forward traffic along several different paths, as long as there is sufficient bandwidth. The programmer must consider both functions simultaneously, to ensure the service-level requirements of each application are met.

Although these examples involve different applications, the problems share a common cause: allowing programs to manipulate low-level network state directly makes it effectively impossible to develop SDN applications in a modular way.

Programming language abstractions. One way to make SDN applications more modular is to change the programming interface they use. Rather than explicitly managing low-level forwarding rules on switches, SDN programmers could use a high-level language that compiles to OpenFlow. Such a language should allow programmers to develop and test modules independently without worrying about unintended interactions. A programmer could even replace a module with another that provides the same functionality.

The NetKAT[2] language (and its predecessor NetCore[14,32,33]) provides a collection of high-level programming constructs including operators for composing independent programs. In the first example, the forwarding and monitoring modules could be composed using its union operator, which would yield a module that both forwards and monitors, as desired. The NetKAT compiler takes this policy and generates equivalent forwarding rules that can be installed on the switches by its runtime system. The Maple controller[43] allows programmers to write modules as packet-processing functions in Java or Haskell and thus use the modularity mechanisms those languages provide. Maple uses a form of runtime tracing to record program decisions and create optimized OpenFlow rules.

Isolated slices. In certain situations, programmers need to ensure the programs being combined will not interfere with each other. For example, in the traffic isolation scenario, the two forwarding modules must be non-interfering.

Combining them using union would be incorrect—the modules might interact by sending packets to each other. One way to guarantee isolation is by using an abstraction that allows multiple programs to execute side-by-side while restricting each to its own isolated "slice" of the network. FlowVisor interposes a hypervisor between the controller and the switches, inspecting each event and control message to ensure the program and its traffic is confined to its own segment of the network.[42] The FortNOX controller also provides strong isolation between applications, using a framework based on role-based authentication.[37] A recent extension to NetKAT also provides a programming construct analogous to slices.[2,15]

Participatory networking. Combining behaviors from multiple modules sometimes leads to conflicts. For example, if one module reserves all the bandwidth available on a link, other modules will not be able to use that link. The PANE controller[10] allows network administrators to specify module-specific quotas and access control policies on network resources. PANE leverages this mechanism to provide an API that allows end-host applications to request network resources. For example, a videoconferencing application can easily be modified to use the PANE API to reserve bandwidth for a high-quality video call. PANE ensures its bandwidth request does not exceed limits set by the administrator and does not starve other applications of resources.

Discussion. Abstractions for decomposing complex applications into simple modules are critical technology for SDN. Without them, programmers have to write programs in a monolithic style, developing, testing, and reasoning about the potential interactions between each piece of the program simultaneously. The abstractions provided by high-level languages such as NetKAT and Maple, hypervisors such as FlowVisor and FortNOX, and controllers such as PANE, make it possible to build applications in a modular way. But although these abstractions are a promising first step, much more work is needed. For example, developers need intuitive reasoning principles for establishing properties of programs built out of separate modules—for example, whether one module can be replaced by another without affecting the behavior of the overall program. They also need better ways of expressing and resolving conflicts, especially for properties involving security and resource constraints.

Virtualization

SDN decouples the software that controls the network from the underlying forwarding elements. But it does not decouple the forwarding logic from the underlying physical network topology. This means a program that implements shortest-path routing must maintain a complete representation of the topology and it must recompute paths whenever the topology changes. To address this issue, some SDN controllers now provide primitives for writing applications in terms of virtual network elements. Decoupling programs from topology also creates opportunities for making SDN applications more scalable and fault tolerant.

Examples. As motivation for virtualization, consider these scenarios:

Access control: Access control is typically implemented by encoding information such as MAC or IP addresses into configurations. Unfortunately, this means topology changes such as a host moving from one location to another can undermine security. If access control lists are instead configured in terms of a virtual switch that is connected to each host, then the policy remains stable even if the topology changes.

Multi-tenant datacenter: In datacenters, one often wants to allow multiple tenants to impose different policies on devices in a shared physical network. However, overlapping addresses and services (Ethernet vs. IP) lead to complicated forwarding tables, and it is difficult to guarantee that traffic generated by one tenant will be isolated from other tenants. Using virtual switches, each tenant can be provided with a virtual network they can configure however they like without interfering with other tenants.

Scale-out router: In large networks, it can be necessary to make a collection of physical switches behave like a single logical switch. For example, a large set of low-cost commodity switches could be assembled into a single carrier-grade router. Besides simplifying the forwarding logic for individual applications, this approach can also be used to obtain scalability—because such a router only exists at the logical level, it can be dynamically augmented with additional physical switches as needed.

As these examples show, virtualization can make applications more portable and scalable, by decoupling their forwarding logic from specific physical topologies.

Virtualization abstractions. The most prominent example of a virtual network abstraction for SDN is VMware's Network Virtualization platform (NSX).[7,9] The Pyretic controller supports similar abstractions.[33] These controllers expose the same fundamental structure to programmers at the virtual and physical levels—a graph representing the network topology—which allows programs written for the physical network to be used at the virtual level, and vice versa.

To define a virtual network, the programmer specifies a mapping between the elements in the logical network and the elements in the physical network. For example, to create a single "big switch" out of an arbitrary topology, they would map all of the switches in the physical network onto the single virtual switch and hide all internal links.[7,33]

Virtualization mechanisms. Virtualization abstractions are easy to describe, but their implementations are far from simple. Platforms such as

NSX are based on a controller hypervisor that maps events and control messages at the logical down to the physical level, and vice versa. To streamline the bookkeeping needed to implement virtualization, most platforms stamp incoming packets with a tag (for example, a VLAN tag or MPLS label) that explicitly associates it with one or more virtual networks.

Packet processing in these systems proceeds in several steps. First, the system identifies the logical context of the packet—that is, its location in the virtual network consisting of a switch and a port. Second, it processes the packet according to the policy for its logical context, which relocates the packet into a different logical context (and possibly generates additional packets). Finally, it maps the packet down to the physical level. The hypervisor typically generates physical-level forwarding rules that implement all three steps simultaneously. One challenge concerns the rule space available on physical switches. Depending on the number of virtual networks and the size of their policies, the hypervisor may not be able to accommodate the complete set of rules needed to realize these policies on the switches. Hence, just as in memory management in an ordinary operating system, the hypervisor typically implements a form of "paging," moving rules onto and off of physical switches dynamically.

Discussion. Virtualization abstractions are an important component of modern SDN controllers. Decoupling programs from the physical topology simplifies applications and also enables sharing the network among several different programs without interference. However, although several production controllers already support virtualization, many open questions remain. One issue concerns the level of detail that should be exposed at the logical level. Current implementations of SDN virtualization provide the same programming interface at the logical and physical levels, eliding resources such as link capacities, queues, and local switch capacity. Another question is how to combine virtualization with other abstractions such as consistent updates. Doing this combination directly is not always possible as both abstractions are commonly implemented using tagging schemes. Finally, current platforms do not support efficient nested virtualization. Semantically there are no deep issues, but there are practical ramifications of implementing nested virtualization using hypervisors.

Decoupling programs from the physical topology simplifies applications and also enables sharing the network among several different programs without interference.

Formal Verification

Today's network operators typically work with low-level network configurations by hand. Unsurprisingly, this leads to configuration errors that make many networks unreliable and insecure. By standardizing the interface to network hardware, SDN offers a tremendous opportunity to develop methods and tools that make it much easier to build and operate reliable networks. There are many critical invariants that arise in networks, several of which are described here. These properties can be checked automatically using static or dynamic tools that formally model the state of the network and controller.

Examples. Many network properties are topology-specific, so they can only be stated and verified given a model of the structure of the network.

Connectivity: Packets emitted by any host in the network are eventually delivered to their intended destinations, except possibly due to congestion or failures.

Loop freedom: No packet is ever forwarded along a loop back to a location in the network where it was previously processed with the same headers and contents.

Waypointing: Packets emitted by untrustworthy hosts traverse a middlebox that scans for malicious traffic before being forwarded to their intended destinations.

Bandwidth: The network provides the minimum bandwidth specified in service-level agreements with tenants.

Other properties are either entirely topology-agnostic or hold for large classes of topologies. These properties capture general correctness criteria for applications that are intended to be executed on many different networks:

Access control: The network blocks all traffic emitted by unauthorized hosts, as specified by an access control list.

Host-learning: The controller eventually learns the location of all hosts and the network forwards packets directly to their intended destinations.

Spanning tree: The network forwards broadcast traffic along a tree that contains every switch (if the network is connected).

Both types of properties have been difficult to establish in traditional networks, as they require reasoning about complex state distributed across many heterogeneous devices. Building on the uniform interfaces provided by SDN, several recent tools have made it possible to verify many network properties automatically.

Verifying configurations. Verifying properties such as loop freedom and connectivity, among others, requires modeling both the topology and switch configurations. Header Space Analysis[23] models switches and the topology as functions in an *n*-dimensional space, where points represent the vector of packet headers. This model can be used to generate test packets that provide coverage for each rule in the overall configuration[46] and extensions can check configurations incrementally.[22] FlowChecker is based on similar ideas, but encodes policies as binary-decision diagrams.[1] Anteater[29] encodes switch configurations as Boolean SAT instances, building on an encoding originally developed by Xie et al.[45] VeriFlow[24] develops domain-specific representations and algorithms for checking properties in real time, which is important because the forwarding behavior of an SDN can rapidly evolve, especially if the controller is reacting to changing network conditions. Finally, NetKAT[2] includes a sound, complete, and decidable equational reasoning system for proving equivalences between network programs.

Verifying controllers. In addition to tools that can verify properties of configurations, some recent efforts have focused on tools that can verify control programs themselves, often focusing on topology-independent properties. NICE[5] uses a combination of symbolic execution and model checking to verify several important properties, including the absence of race conditions and bugs akin to switch memory leaks. Another tool developed by Scott et al. checks whether abstractions provided by SDN controllers are correctly realized in switch-level configurations.[17] Guha et al. describe a framework for establishing controller correctness using a proof assistant, as well as a machine-verified implementation of the NetCore language against a detailed operational model of OpenFlow.[14] VeriCon shows that Hoare-style verification is possible for controllers written as simple imperative programs[3] and has been applied successfully to a number of examples adapted from the SDN literature (for example, firewalls, routing algorithms, and so on). Nelson, et al. present a Datalog-based SDN programming language, called Flowlog, that they also use to write and verify several canonical properties.[34] Because Flowlog is designed to be finite-state, it is amenable to automatic verification without the need for complex programmer-supplied assertions.

There is a tremendous need for tools that can provide rigorous guarantees about the behavior, performance, reliability, and security of networked systems.

Discussion. There is a tremendous need for tools that can provide rigorous guarantees about the behavior, performance, reliability, and security of networked systems. By standardizing the interfaces for controlling networks, SDN makes it feasible to build tools for verifying configurations and controllers against precise formal models. Some possible next steps in this area include developing custom logics and decision procedures for expressing and checking properties, enriching models with additional features such as latency and bandwidth, and better integrating property checking and debugging tools into SDN controller platforms.

Related Work

Enormous momentum has gathered behind SDN in recent years, but the ideas behind SDN build on many previous efforts. Tempest,[40] an architecture developed at Cambridge in the mid-1990s, was an early attempt to decouple forwarding and control in the context of ATM networks. Several features from Tempest can be found in SDN today including an emphasis on open interfaces and support for virtualization. Similarly, the IETF ForCES working group defined a standard protocol that a controller could use to manage multiple heterogeneous devices in a single network.[8] The Soft-Router project explored the benefits of separating forwarding and control in terms of extensibility, scalability, reliability, security, and cost.[26]

The Routing Control Platform,[4] developed at AT&T, demonstrated that logical centralization could be used to dramatically simplify routing algorithms while still providing good performance. These ideas were later expanded in the 4D platform,[12] which introduced the distinction between management and control planes. The benefits of expressing algorithms using network-wide data structures instead of using distributed algorithms in SDN can also be seen in this work.

The most immediate predecessor of SDN was Ethane,[6] a system aimed at providing fine-grained in-network access control. Ethane provided a high-level language for defining security policies, and a controller program that implemented those policies by installing and uninstalling custom forwarding rules in programmable network switches. The NOX controller was based on Ethane,[13] and the protocol used by the Ethane controller to communicate with switches later evolved into the first version of the OpenFlow standard.[31]

Conclusion

Many of the initial efforts around SDN have focused on architectural concerns—making it possible to evolve the network and develop rich applications. But the growth of this new software ecosystem has also led to the development of fundamental new abstractions that exploit the ability to write network control software on standard servers with a less constrained state distribution model. We believe these abstractions are critical for achieving the goals of SDN and may prove to be some of its most lasting legacies.

Acknowledgments. The authors wish to thank Shrutarshi Basu, Andrew Ferguson, Anil Madhavapeddy, mark Reitblatt, Jennifer Rexford, Mooly Sagiv, Steffen Smolka, Robert Soulé, David Walker, and *Communications*' reviewers for helpful comments. Our work is supported by NSF grant CNS-1111698, ONR award N00014-12-1-0757, a Sloan Research Fellowship, and a Google Research Award.

References

1. Al-Shaer, E. and Al-Haj, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of SafeConfig*, 2010.
2. Anderson, C.J., Foster, N. Guha, A., Jeannin, J-B, Kozen, D., Schlesinger, C. and Walker, D. NetKAT: Semantic foundations for networks. In *Proceedings of POPL*, 2014.
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M. and Valadarsky, A. VeriCon: Towards verifying controller programs in software-defined networks. In *Proceedings of PLDI*, 2014.
4. Caesar, M., Caldwell, D.F., Feamster, N., Rexford, J., Shaikh, A.and van der Merwe, J.E. Design and implementation of a routing control platform. In *Proceedings of NSDI*, 2005.
5. Canini, M., Venzano, D., Perešíni, P., Kostić, D. and Rexford, J. A NICE way to test OpenFlow applications. In *Proceedings of NDSI*, 2012.
6. Casado, M., Freedman, M.J. Pettit, J., Luo, J., McKeown, N. and Shenker, S. Ethane: Taking control of the enterprise. In *Proceedings of SIGCOMM*, 2007.
7. Casado, M., Koponen, T., Ramanathan, R. and Shenker, S. Virtualizing the network forwarding plane. In *Proceedings of PRESTO*, 2010.
8. Doria, A., Hadi Salim, J., Haas, R., Khosravi, H., Wang, W. Dong, L., Gopal, R. and Halpern, J. Forwarding and control element separation (ForCES), 2010. IETF RFC 5810.
9. Koponen, T. et al. Network virtualization in multi-tenant datacenters. In *Proceedings of NSDI*, 2014.
10. Ferguson, A.D., Guha, A., Liang, C., Fonseca, R. and Krishnamurthi, S. Participatory Networking: An API for application control of SDNs. In *Proceedings of SIGCOMM*, 2013.
11. Fortz, B., Rexford, J. and Thorup, M. Traffic engineering with traditional IP routing protocols. *IEEE Commun.* (Oct. 2002).
12. Greenberg, A.G., Hjálmtýsson, G., Maltz, D.A., Myers, A., Rexford, J., Xie, G.G., Yan, H., Zhan, J. and Zhang, H. A clean slate 4D approach to network control and management. *SIGCOMM CCR 35*, 5 (2005).
13. Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N. and Shenker, S. NOX: Towards an operating system for networks. *ACM SIGCOMM CCR 38*, 3 (2008).
14. Guha, A., Reitblatt, M. and Foster, N. Machine-verified network controllers. In *Proceedings of PLDI*, 2013.
15. Gutz, S., Story, A., Schlesinger, C. and Foster, N. Splendid isolation: A slice abstraction for software-defined networks. In *Proceedings of HotSDN*, 2012.
16. Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S. and McKeown, N. ElasticTree: Saving energy in datacenter networks. In *Proceedings of NSDI*, 2010.
17. Heller, B. et al. Leveraging SDN layering to systematically troubleshoot networks. In *Proceedings of HotSDN*, 2013.
18. Hong, C-Y, Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M. and Wattenhofer, R. Achieving high utilization with software-driven WAN. In *Proceedings of SIGCOMM*, 2013.
19. Jain, S. et al. B4: Experience with a globally deployed software defined WAN. In *Proceedings of SIGCOMM*, 2013.
20. Jose, L., Yu, M. and Rexford, J. Online measurement of large traffic aggregates on commodity switches. In *Proceedings of HotICE*, 2011.
21. Katta, N.P., Rexford, J., and Walker, D. Incremental consistent updates. In *Proceedings of HotSDN*, 2013.
22. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N. and Whyte, S. Real-time network policy checking using Header Space Analysis. In *Proceedings of NSDI*, 2013.
23. Kazemian, P., Varghese, G. and McKeown, N. Header space analysis: Static checking for networks. In *Proceedings of NSDI*, 2012.
24. Khurshid, A., Zhou, W., Caesar, M. and Godfrey, B. VeriFlow: Verifying network-wide invariants in real time. In *Proceedings of NSDI*, 2013.
25. Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T. and Shenker, S. Onix: A distributed control platform for large-scale production networks. In *Proceedings of OSDI*, 2010.
26. Lakshman, T.V., Nandagopal, T., Ramjee, R., Sabnani, K. and Woo, T. The SoftRouter architecture. In *Proceedings of HotNets*, 2004.
27. Liu, H.H., Wu, X., Zhang, M., Yuan, L., Wattenhofer, R. and Maltz, D. zUpdate: Updating datacenter networks with zero loss. In *Proceedings of SIGCOMM*, 2013.
28. Mahajan, R. and Wattenhofer, R. On consistent updates in software-defined networks. In *Proceedings of HotNets*, 2013.
29. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B. and King, S.T. Debugging the data plane with Anteater. In *Proceedings of SICOMM*, 2011.
30. McGeer, R. A safe, efficient update protocol for OpenFlow networks. In *Proceedings of HotSDN*, 2012.
31. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. and Turner, J. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR 38*, 2 (2008).
32. Monsanto, C., Foster, N., Harrison, R. and Walker, D. A compiler and run-time system for network programming languages. In *Proceedings of POPL*, 2012.
33. Monsanto, C., Reich, J., Foster, N., Rexford, J. and Walker, D. Composing software defined networks. In *Proceedings of NSDI*, 2013.
34. Nelson, T., Ferguson, A., Scheer, M., and Krishnamurthi, S. Tierless programming and reasoning for software-defined networks. In *Proceedings of NSDI*, 2014.
35. Noyes, A., Warszawski, T., Cerny, P. and Foster, N. Toward synthesis of network updates. In *Proceedings of SYNT*, 2013.
36. Perlman, R. An algorithm for distributed computation of a spanning tree in an extended LAN. *SIGCOMM CCR 15*, 4 (1985).
37. Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M. and Gu, G. A security enforcement kernel for OpenFlow networks. In *Proceedings of HotSDN*, 2012.
38. Reitblatt, M., Canini, M., Foster, N. and Guha, A. Fattire: Declarative fault-tolerance for software-defined networks. In *Proceedings of HotSDN*, 2013.
39. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C. and Walker, D. Abstractions for network update. In *Proceedings of SIGCOMM*, 2012.
40. Rooney, S., van der Merwe, J.E., Crosby, S.A. and Leslie, I.M. The Tempest: A framework for safe, resource assured, programmable networks. *IEEE Commun. 36*, 10 (1998).
41. Shenker, S., Casado, M., Koponen, T. and McKeown, N. The future of networking and the past of protocols. Invited talk at Open Networking Summit, Oct. 2011.
42. Sherwood, R. et al. Carving research slices out of your production networks with OpenFlow. *SIGCOMM CCR 40*, 1 (2010).
43. Voellmy, A., Wang, J., Yang, Y.R., Ford, B. and Hudak, P. Maple: Simplifying SDN programming using algorithmic policies. In *Proceedings of SIGCOMM*, 2013.
44. Wang, R., Butnariu, D. and Rexford, J. OpenFlow-based server load balancing gone wild. In *Proceedings of HotICE*, 2011.
45. Xie, G.G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A.G., Hjálmtýsson, G. and Rexford, J. On static reachability analysis of IP networks. In *Proceedings of INFOCOM*, 2005.
46. Zeng, H., Kazemian, P., Varghese, G. and McKeown, N. Automatic test packet generation. In *Proceedings of CoNext*, 2012.

Martin Casado (mcasado@vmware.com) is fellow, senior vice president and general manager of networking and security at VMware and was the co-founder and CTO of Nicira Networks, Palo Alto, CA.

Nate Foster (jnfoster@cs.cornell.edu) is an assistant professor of computer science at Cornell University, Ithaca, NY.

Ajun Guha (arjun@cs.umass.edu) is an assistant professor of computer science at the University of Massachusetts, Amherst, MA.