Trident
Towards a Unified SDN Programming Framework with Automatic Updates

Author: Taishi Nojima
Advisor: Prof. Richard Yang

May 8, 2018
Abstract

Software Defined Networking (SDN) realizes fine-grained, consistent network control by allowing network engineers to define the network’s behavior through centralized policies. However, despite its recent progress, the scope of existing SDN programming is mostly limited to its core networking functionalities, routing and resource allocation. Thus, even though most enterprise networks contain more sophisticated network functions, such as middleboxes, virtualized network functions and host agents, integrating SDN with those network functions remains complex, leading to errors, redundancy and inefficiency. In this project, as a joint-collaboration between myself and Kai Gao from Tsinghua University, we conduct the first systematic study on designing a unified SDN programming framework to accommodate network functions.

The major challenge in realizing the unified framework is how to correctly and efficiently handle fine-grained information that are asynchronously and continuously provided by network functions. To overcome this limitation, we introduce the initial design of Trident, a unified SDN/NF programming framework. Trident introduces three novel concepts to achieve fine-grained packet selection, well-structured route computation, and automatic dependency tracking of user network policy: flow attributes with 3-valued logic to model results from network functions; route algebra as a powerful abstraction of routing strategies; and live variable as a generic data model that provides semantic dependency tracking and automatic change propagation. While our performance evaluation has not been fully done yet, we demonstrate that Trident can be applied in many real world use cases using any standard programming language. This final report summarizes our working paper, Trident: Towards a Unified SDN Programming Framework with Automatic Updates, which we submitted to SIGCOMM 2018.

1 Introduction

Software-Defined Networking (SDN) emerged more than a decade ago and has drawn a lot of attention in providing fine-grained consistent network control.[3] Unlike traditional network management systems, SDN allows a network to customize its behaviors through centralized policies at a conceptually centralized network controller.[7] In recent years, significant progress has been made in this field to realize a more simple, centralized network
control system: for example, Openflow established a standard data-plane abstraction for distributed switches[7]; Maple simplified SDN programming by allowing a programmer to use a standard programming language in defining the behaviors of an entire network[7]; and ONOS and Open Daylight provided scalable, reliable and flexible network operating systems to realize state-of-the-art SDN control platforms.[4][5]

Despite the recent progress, a major remaining problem is its complex integration with other networking technologies: the scope of SDN programming is mostly limited to its core networking functionalities — routing and resource allocation — and does not support simple integration with other sophisticated networking technologies. These technologies, as we call “network functions,” such as security appliances, middleboxes and Network Function Virtualization are heavily deployed in most of modern enterprise networks of any size.[6] Due to this restriction of SDN programming, network engineers are still forced to manage and configure network functions separately from their SDN systems, leading to errors, redundancy, and inefficiency.

Thus, a unified SDN programming framework that can accommodate network functions would offer substantial benefits, reducing existing inefficiencies in today’s network management and bringing out the best of both SDN and network function technologies to achieve more coherent, flexible and secure network management system.

As the first step to achieve this goal, this paper introduces the initial design of Trident, a unified SDN programming framework that can accommodate networking functions. Note that this paper is an abridged version of our working paper, Trident: Towards a Unified SDN Programming Framework with Automatic Updates.[3]

2 Challenges

Designing a unified framework is non-trivial; in particular, there are several gaps between SDN and network functions that cannot be integrated easily. SDN systems have a fundamentally simple “Match-Action” paradigm, as shown in Figure 1, where “match” selects the packets and the “actions”, mostly routing decisions, are applied to these packets. However, existing solutions have major limitations in integrating feature-rich network functions into a single coherent SDN programming framework.
2.1 Incomplete Packet Selection Functionality

The first challenge is that existing SDN systems have incomplete packet selection functionalities. SDN programming is based on a per-packet, stateless, synchronous model: it makes a routing decision immediately based on the information available from each arriving packet, without keeping track of states across multiple packets. Thus, the current SDN systems can support routing policies that depend on the global network state and packet header fields, such as source IP address or destination port, but can’t handle more higher-level, stateful information, such as results from firewalls or IDS.

On the other hand, network functions are stateful, whose results are often provided asynchronously and may take non-deterministic time to be available. For example, consider an Intrusion Detection System that triggers an alert on a HTTP connection to a specific URL. It has to internally keep track of states across packets to classify them into the corresponding HTTP flow; and it can provide results only asynchronously since the HTTP URL parameter is not included in all packets and may be large enough that it has to span across multiple packets.

Due to the asynchronous, stateful nature of network functions, it’s difficult to extend the existing packet selection to accommodate the ability to match on the information provided by network functions: a binary-boolean programming model would be functionally incomplete and cannot work. For example, consider the URL of an HTTP request again, which is not known in the first few packets. In this case, writing something as simple as the following snippet might lead to ambiguities:

```java
if (pkt.http_url == "www.xyz.com") {
    // let it through
} else {
    // drop it
}
```
Without proper packet selection capabilities, all packets will be dropped even if it is about to make a request to www.xyz.com, since packets to establish a TCP connection don’t have the HTTP URL information.

2.2 Lack of a Proper Abstraction of Routing Strategies

The second major challenge in designing the unified framework is the lack of a proper abstraction for route selection. While many SDN programming languages provide routing as a built-in functionality, they generally do not support advanced, efficient re-routing capabilities. In other words, it is often too complex or impossible to represent and combine different routing strategies to handle flexible routing; for example, simple concatenation of two routes, a route computed using a shortest path algorithm and another using traffic engineering, is often impractical and computationally inefficient in most SDN systems today. And without the ability to represent and combine routing strategies in a systematic, flexible way, it would be difficult to take full advantage of the new packet selection capabilities that can handle feature-rich results from network functions. Thus, it is essential to devise a proper abstraction of routing strategies with well-defined semantics.

2.3 Inefficient Dependency Tracking

The third issue is dependency tracking. In traditional SDN systems, since the network state changes infrequently and the control plane state is relatively stable, full recompilation of routing policies upon state change often doesn’t compromise the performance. However, when integrated with network functions whose outputs will be updated frequently and constantly, full recompilation becomes impractical; thus, it is essential to devise an efficient way to track data dependencies to achieve fine-grained, incremental policy reconciliation.

3 Trident Approach

To overcome these limitations, Trident introduces three novel components to achieve the unified SDN/NF programming framework: 1) Flow Attribute, 2) Route Algebra, and 3) Live Variable.
Flow attribute is a general abstraction for modeling results from network functions, with unknown variables and Kleene’s 3-valued logic\cite{2} to handle asynchronous data with provable functional completeness in packet selection. Route algebra addresses the second challenge, providing a powerful abstraction of routing strategies with well-defined semantics and operations to support advanced re-routing capabilities. And Live Variable offers a generic data model that provides semantic dependency tracking and automatic change propagation.

An overview of Trident is shown in Figure 2. Live variable is considered as a basic element in our programming framework upon which extended packet selectors and route algebra system are built. We provide the abstractions of flow attributes and route algebra to accommodate the effects of network functions in the “match-action” paradigm, so that DevOps need not work with live variables themselves. Under the hood, Trident uses an intermediate structure to represent the “match-action” paradigm, called a binding. A binding contains the results of a packet selector and the results of a route set (user-defined routes in the network). For each binding, the packets that are selected by a packet selector are forwarded using the corresponding routes bound to that selector.
4 Trident Design

4.1 Live Variable

The foundation of Trident is a new type called live variable, which stores not only a value, but also how the value is computed. Thus, whenever the value of a live variable changes, Trident knows exactly which other variables are dependent on this variable and how they need to be recomputed.

For example, consider the following example for basic live variables:

\[ a = b + c \]

where \( a, b \) and \( c \) are all integer numbers. In this case, the live variable \( a \) retains not only its current value but also how this value is computed, namely \( b + c \). Thus, whenever the value of either \( b \) or \( c \) changes, Trident automatically detects that change and recomputes and updates the value of \( a \).

The notion of live variables can be extended to model the continuous and asynchronous data from remote network functions. We call it Remote Live Variable, which reads from and depends on a data stream connected to a remote network function, so that whenever the network function’s output changes, Trident can recompute and update the values of all variables that depend on the output of the network function.

To visualize how remote live variables are integrated in a network policy in Trident, consider the scenario in Figure 3. Remote live variable \( G \) is collected from networking devices \( (S_1, S_2) \), while remote live variables \texttt{http_url} and \texttt{flow_rate} are collected from both networking devices and an IDS, such as Bro and Suricata. And \( f \) is a network policy function that uses this information to determine how the packets should be controlled. Now, we could see that \( v_1, v_2, v_3 \) are remote live variables that depend on remote network devices while \( v_4 \) is a live variable that depends on \( v_1, v_2 \) and \( v_3 \). When the values of any \( v_1, v_2 \) or \( v_3 \) change, Trident knows that the policy \( v_4 \) needs to be invalidated and recomputed.

4.2 Flow Attribute and Packet Selector

A fundamental problem of existing SDN programming is the limited packet selection capability: it cannot handle information collected from remote network functions. To realize a more complete packet selection, Trident inter-
nally adopts the abstraction of a packet that represents both the header fields and flow attributes as a live variable.

The header fields represent the packet’s headers, including Ethernet source and destination addresses and IP source and destination addresses, as well as deterministic information about the packet such as its ingress port and the current location. In other words, they are the fields about each individual packet and that can be handled by existing SDN programming languages for packet selection.

To accommodate the ability to match on the fine-grained information provided by network functions, Trident introduces flow attributes, which are remote live variables that represent asynchronous attributes about a network flow provided by network functions. The flow attributes can virtually represent any information about the flow as long as they have a consistent data type, which could be from the HTTP URL (string) to whether it triggered an IDS alert (bool) to the statistics about the network state (float). And the values of flow attributes can be updated asynchronously. For example, consider a scenario where a network function can provide the URL of a HTTP flow and store it in the flow attribute attr. Since the URL information is not available for the first several packets of the flow (such as TCP handshake packets), the flow attribute attr initially has the value unknown.
But when the network function processes its HTTP request packet with the URL parameter and outputs the URL to the stream, the value of attr is automatically updated accordingly.

On top of the data models defined above, Trident supports a powerful packet selector that can match packets based not only on the header fields but also on the flow attributes, providing a more complete packet selection capability. A user can define a selector simply as a function that takes a packet object and returns a live variable of Boolean type.

4.3 Route Algebra

To provide a proper abstraction of routing strategies with automatic dependency tracking, Trident models a network and route objects as live variables. Each network element, such as a port of a network device and a link, is represented as remote live variables since their values are based on the physical status of the element. And as the smallest component for a routing strategy, a route object is defined as a sequence of port pairs in the network, which is essentially a tuple of live variables.

On top of this data model, Trident introduces the special operators of route algebra that support the systematic construction of routes. The operators include concatenation, inversion, union, intersection, difference, aggregation, selection and reflection, among others, all of which support automatic dependency tracking based on live variables. For example, when concatenation is applied to two sets of route objects, it constructs a new set by first concatenating all possible route pairs from the two sets and then removing the invalid ones. Each route object in the resulting set retains the dependency graph so that Trident can efficiently and incrementally recompute routes when its dependent route is invalidated.

4.4 Runtime System

To support the aforementioned concepts and achieve the unified SDN programming framework, Trident comes with a novel and efficient runtime system. Figure 4 demonstrates the basic workflow of the Trident runtime system, which mainly consists of three stages: 1) Initialization, 2) Asynchronous feedback updates, and 3) Automatic invalidation and recovery.

At the initialization stage, network policies are submitted to the runtime system by a user. The runtime system then evaluates these policies to con-
struct bindings of packet selector and route objects with the corresponding data dependency graphs, which are sent to the unified data store and data plane compilers. The deplane compilers generate corresponding data plane configurations and deploy them on networking devices.

Once these configurations are installed, the networking devices can properly serve packets entering and traversing the network according to the user policies. However, as these packets are processed by switches and network functions, the value of certain flow attributes and network state might change, which can eventually affect the output of the user policies. To correctly handle such changes, the value updates that are asynchronously provided by network functions are sent back to the unified data store. (Asynchronous feedback updates)

Upon the arrival of the asynchronous feedback updates, the Trident runtime system immediately checks whether the current bindings are still consistent with the user policies and recompute bindings if necessary: the system automatically removes routes, bindings and the corresponding data plane configurations that are no longer valid, while it computes and installs new configurations that are consistent with the new value.
5 Evaluation and Future Work

There are two main questions in evaluating Trident:

1. How useful and expressive is the combination of packet selectors and route algebra in real network management scenarios?

2. What is the overall performance of Trident as a SDN controller?

5.1 Use Cases

The potential use cases of Trident in real world networks are extensive. On top of basic network programs that existing SDN programming can handle with, such as L3 Routing, a user can realize a complex networking environment with various network functions, such as Cache Access Violation and SDMZ, with a few lines of code using the Trident programming framework and manage in a coherent, flexible and reliable manner.

To give you a sense of what programming with Trident looks like, Listing 1 shows the implementation of Cache Access Violation, which comes from the FlowTag paper.[1] Note that Trident network policies can be implemented in any standard programming language, although this listing is in Scala-like pseudocode. As shown in Figure 5, the traffic from hosts, $h_1$ or $h_2$, first goes to a proxy server and sees if there is a local cache copy. Otherwise, it should pass a firewall and fetch the data from the Internet. Host 2 ($h_2$) is not allowed to access www.xyz.com by the global policy.

Line 2 defines a live variable representing the network state, called label, so that the program can look up network devices in the network by their labels.
class WebAccess extends Policy {
    val label = NetworkState[Vertex, String]("label")
    val http_url = FlowAttr3V[String]("http_url", 5 TUPLE)

    val host = V where { v => v.label == "h1" or v.label == "h2" }
    val proxy = V where { v => v.label == "proxy" }
    val fw = V where { v => v.label == "firewall" }
    val gw = V where { v => v.label == "internet" }
    val route = WaypointRoute(host <=> proxy <=> fw <=> gw)

    val h2 = "10.0.0.3"

    override def apply(pkt: Packet)(implicit ctx: Context) {
        if (pkt.src_ip == h2) {
            if (pkt/http_url == "www.xyz.com") {
                drop()
                return
            }
        }
        use route
    }
}


Line 3 defines a flow attribute (live variable) called http_url, which represents the URL of a HTTP flow and whose value is provided by a network function in the network. The second argument, 5 TUPLE, specifies how to classify packets into each flow represented by this flow attribute; in the case of a HTTP flow, packets with the TCP 5 tuple can be considered in the same flow. Note that in order for this policy to work properly, a user needs to set up the network function capable of parsing a HTTP flow in the network and configure it so that it updates the http_url information for a TCP connection through the Trident client API.

Line 5-8 simply define networking devices (remote live variables as explained in the route algebra section above) and look up each by its label from the network.

In Line 9, a user-defined function called WaypointRoute constructs the route used for the policy. Although the implementation of the WaypointRoute function is omitted, this could be implemented in any way as long as it follows the special operations defined in the route algebra so that the correctness and consistency of its dependent variables are enforced.

In Line 12-21, the policy is defined: it checks whether the packet has a source IP address of 10.0.0.2 and also has a known http_url value of
www.xyz.com. If so, the packet is dropped, and otherwise, it’s forwarded using the route.

5.2 Performance Evaluation (Future Work)

Unfortunately, we are still in the process of implementing the initial prototype of Trident, so we have not been able to comprehensively evaluate its performance. However, we recognize several limitations in the system that we need to address in the future. The main limitation is in the process of generating and updating data plane configurations: the problem is how to generate flow rules when there are flow attributes. Since an Openflow switch can’t recognize flow attributes, Trident needs to update the flow rules when the attribute changes. However, it would be impractical to react to all flow attribute changes, because of the limitations in the processing power of the controller, the number of flow entries that can be registered in switches, and the link capacity between the controller and network devices. We don’t have a scalable, efficient solution to this problem yet and are working together to come up with a reasonable solution.
Bibliography


