Barista: An Event-centric NOS Composition Framework for Software-Defined Networks

Jaehyun Nam†, Hyeonseong Jo†, Yeonkeun Kim†, Phillip Porras‡,
Vinod Yegneswaran‡, and Seungwon Shin†
KAIST, Daejeon, Korea†, SRI International, CA, USA‡

***Abstract*—As the network operating system (NOS) is the strategic control center of a software-defined network (SDN), its design is critical to the welfare of the network. Contemporary research has largely focused on specialized NOSs that seek to optimize controller design across one or a few dimensions (e.g., scalability, performance, or security) due to fundamental differences in architectural trade-offs needed to support competing demands. We thus designed Barista, as a new framework that enables flexible and customizable instantiations of network operating systems (NOSs) supporting diverse design choices. The Barista framework incorporates two mechanisms to harmonize architectural differences across design choices: *component synthesis* and *dynamic event control*. First, the modular design of the Barista framework enables flexible composition of functionalities prevalent in contemporary SDN controllers. Second, its event-handling mechanism enables dynamic adjustment of control flows in a NOS. These capabilities allow operators to easily enable functionalities and dynamically handle associated events, thereby satisfying network operating requirements. Our results demonstrate that Barista can synthesize NOSs with many functionalities found in commodity NOSs with competitive performance profiles.**

I. Introduction

Software-defined networks have emerged as a compelling alternative to vertically integrated networks, which offer limited flexibility, programmability, and customizability. Among SDNs driving motivations is a desire to address the "islands-of-functionality" challenge in traditional networks, in which disparate functions must be independently deployed, separately managed, and require distinct policy control. In theory, the programmable control layer of SDNs offers a more agile platform which facilitates the unified integration and management of diverse functions.

In the SDN stack, a network operating system (NOS) stands as the strategic control center since it determines the functionalities of SDN by implementing logic for network management and providing a control interface for higher-level applications. The complexity associated with their design leads to significant software abstraction, scalability, security, and reliability challenges. To address these challenges, research in network and security communities has expended considerable effort toward designing NOSs focused on high availability, robustness and security control. As a result, contemporary NOSs have become specialized toward specific objectives. For example, Beacon [13] is highly optimized for providing maximal throughput from a single controller; ONOS [25] and OpenDaylight [5] focus on distributed scalability; and SE-Floodlight [26] attempts to address a range of security requirements imposed within sensitive network computing environments.

However, based on interactions with SDN operators in both industry and academia, stove-piped functions persist with SDNs, albeit in a different manifestation. For example, it is common for network administrators to manage multiple SDN sub-networks with disparate requirements and policies using distinct network verticals managed by different controllers (e.g., a campus network consisting of department networks with differing management policies). This invariably leads to increased management costs, because each controller has its own programming architecture and software APIs (e.g., an SDN application running on ONOS [25] cannot be directly moved to OpenDaylight [5]).

We posit that the rigidity in a composition of contemporary SDN controllers significantly limits their ability to fully address the challenge of satisfying competing demands within enterprise, cloud, and data center networks. Furthermore, while operators might want to achieve the functionalities present in two different NOSs, it is quite challenging to integrate these capabilities due to fundamental architectural differences and conflicting design principles. For instance, the distributed ONOS architecture emphasizes parallelism for large-scale networks, while SE-Floodlight focuses on centralized monitoring of all control flows inside a NOS. Thus, we make the case for a new NOS design that allows for customizable controllers while retaining a uniform programming API.

To address architectural limitations, we designed a novel NOS framework, called Barista, that supports the flexible composition of the functionalities found in commodity NOSs. Barista facilitates rapid modular prototyping, customization, and fielding of control layer logic to meet diverse operational requirements. We focused on two key aspects of Barista's design: *component synthesis* and *dynamic event control*. First, the modular design of the Barista framework enables operator-defined composability of various NOS functionalities (e.g., clustering, role-based authorization, and flow-rule conflict resolution) as portable component extensions. Second, Barista allows operators to customize the control flows across NOS components. Barista's event handling framework supports a diversity of event types, component chaining, and policy-based event distribution. Barista also introduces a meta-event class that can dynamically modify the set of active components based on current operating conditions. These capabilities collectively enable the dynamic composition and synthesis of custom NOSs based on operational requirements.

Contributions. In sum, we made the following contributions:

- Design of a new NOS architecture, called Barista, which substantially accelerates the ability of the SDN research

community to rapidly prototype and integrate new NOS functionalities into a distributed NOS environment.
- Development of a new SDN event-handling framework that enables fine-grained control over events delivered to NOS components through a diverse set of event types, dynamic chaining among components, and policy-based event distribution.
- Evaluation of Barista in three use cases that demonstrate its composability, and its ability to address a range of operational scenarios that no current NOS can singularly address.
- Release of Barista as an open-source project.

II. Background and Motivation

Over the past few years, a growing list of competing NOS software projects have explored features that appeal to various network operator communities. Barista represents a unique design perspective among this spectrum of competitive NOS software projects. Here, we are motivated by the need for a highly composable NOS compilation framework, that enables the rapid integration of new NOS features and extensions, while allowing operators to flexibly compose the most appropriate features to match the needs of their individual target environments. Barista offers a unique NOS approach, that is applicable to both research communities and operators in a wide range of operational settings.

A. The Academic Case for Barista

The SDN control layer has garnered much of the focus among those involved in SDN research. In surveying this work, we found that researchers often employ an existing NOS as a base from which to explore new control-layer features. For example, Ravana [19] modifies Ryu [8], a well-known open-source NOS, to introduce fault-tolerant features to the SDN control layer. Other groups have integrated extensions, such as strong security features, into established NOSs [18], [26]. Unlike the proprietary nature of legacy networks, SDN researchers enjoy access to a wide range of opensource platforms from which to experiment, including some of the most visible and widely used NOSs, such as ONOS [25] and OpenDaylight [5]. Alternatively, other researchers have introduced complete ground-up NOS proposals, which focus on exploring specialized properties, such as robust application management with high performance [30].

Barista seeks to further extend the ease with which new NOS components can be designed and integrated. It minimizes the effort from which components, including feature extensions to an existing NOS instance can be modularly constructed and deployed. For example, with Barista, one can rapidly devise and implement a new flow-rule conflict-detection algorithm that does not require in-depth analysis or modification of the NOS internal architecture (e.g., the implementation of such an algorithm required internal modifications of the Floodlight NOS [26]). Barista-hosted NOS extensions offer modular components that do not require internal NOS modifications to join its event pipeline. Barista's approach to a component-based NOS architecture provides research communities with a rapid development framework, that 1) significantly accelerates experimentation by reducing the implementation cost, and 2) produces cleanly-composable NOS functional extensions that are easily shared.

B. The Industrial Case for Barista

Network operators who deploy SDN-enabled networks face the challenge of selecting the best NOS to match the operational requirements of the target network. For example, consider a network operator who manages two networks: *network A* consists of 1K switches and 100K hosts for web testing, and *network B* consists of 10 switches and 100 hosts that provide database services for a corporate-sensitive dataset. In this scenario, an operator might conclude that a NOS designed for network scalability and high-performance would best suite network A, while a NOS that offers strong security policy enforcement would best suite network B. Although, managing two separate NOSs might match each environment with the most applicable NOS features, the deployment of two different NOS platforms will also impact the overall management cost.

Thus, the second motivation for Barista is to design a NOS compilation framework that enables the customization of the NOS at deployment time, with the features that best suite the target environment. Here, the network operator simply specifies the functional requirements for each network, and produces two automated compilations of Barista that deliver the comparable functional services provided by the two independent NOS platforms. Once the functional requirements are specified (i.e., performance, security, fault recovery protections, scalability) for each network, the Barista framework will produce a NOS composition that integrates the functional components that match the stated objectives.

C. Motivating Example

Consider the case where an operator maintains a large number of computing servers and network devices in a datacenter network. He needs a scalable NOS to manage the entire set of entities. However, since the NOS manages the whole network infrastructure, a compromise of the NOS affects all other services. As a result, the operator needs a scalable and secure NOS to satisfy the operating requirements. Without an available controller, the operator might customize ONOS [25] by adding the security features introduced by SE-Floodlight [26] and LegoSDN [10].

Most security features (e.g., role-based authorization, flow-rule conflict resolution) require inline hooking of inter-component communication. However, modern NOSs (including ONOS) use direct communication between components for high performance, which means that components communicate with each other without any control of a NOS. In turn, the operator needs to embed security logic into all locations where manual security inspection is required. Furthermore, whenever ONOS code is updated, the operator needs to check and make corresponding updates to the security logic. Although security features can be integrated into ONOS, its underlying framework (i.e., OSGi [6]) cannot support complete isolation among components (bundles) due to the shared computing environment where a component can lead to compromises that affect the entire system [22], [30].

Our Solution: Barista allows operators to deploy components supporting diverse functionalities as extensions. For full isolation, Barista replaces direct function calls between components

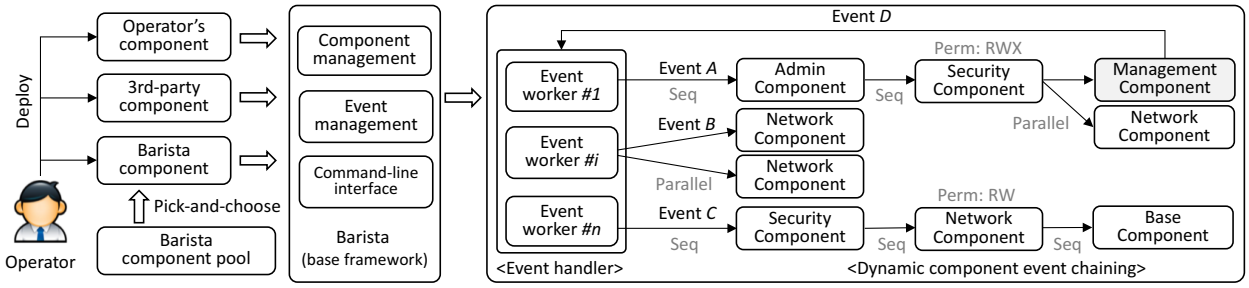


Fig. 1: Barista System Overview: Base Framework and Event Handling Framework

with request-response events handled by its centralized event handler that brokers call between components. Furthermore, Barista dynamically enforces the component event-processing sequence based on the characteristics of deployed components. As a result, the operator can compose scalability and security functionalities on top of an isolated computing environment with the dynamic event-processing chain regardless of changes to the internal logic of components.

III. System Design

This section presents the design of Barista and explains how it facilitates the development of various NOS functionalities as component extensions that are assembled to build customized NOS instances. The Barista design sits on the opposite spectrum of prior NOS work that focused on the specialization of NOS functionalities to better support target network environments. Rather, Barista's goal is to enable operator-defined composability of NOS functionalities that can be assembled in its isolated per-component environment.

A. Base Framework Overview

Figure 1 illustrates the two key elements of the Barista framework: components and events. A component represents the implementation of a specific NOS function. The concept of modular composition is inspired by Click [20] and Exokernel [12]. We also consider applications as components in this work. The only difference is in the kinds of information they deal with. The framework provides two classes of components. The first class is a general component (e.g., packet I/O engine and OpenFlow engine). These are designed primarily to become a functional logic inside a NOS. The second class is an autonomous component (e.g., statistics and resource managements). These components are similar to general components, however, they are intended to independently conduct certain actions without intervention.

In the Barista framework, events drive the component-to-component information flow. All communication between components is managed in an event handler. To communicate with other components, the incoming and outgoing events of each component should be first registered at the event handler. Then, the event handler identifies which components have registered for an event and delivers the event to those components.

Barista does not support non-event-based interaction between components; all intercommunication between components is done through events. While this event-handling mechanism may increase the communication overhead compared to direct function calls, in terms of integrating NOS components, it provides a a degree of abstraction from the NOS internals for defining component composability. Moreover, it allows insertion of security functionalities to inspect all communications (e.g., API permission check and data integrity checks). Thus, our design approach emphasizes flexible composition over high performance.

B. Component Specification

Barista's implementation of a component is similar to that of a general program. A component is composed of four pieces: main, cleanup, command-line interface (CLI), and handler functions. The main and cleanup functions are analogous to a constructor and a deconstructor. Through those functions, a component can be attached and/or detached from the framework. The CLI function is the interface to a component for operators and allows operators to interact with the component in runtime. The handler function acts as the core function of a component that is responsible for receiving predefined (inbound) events.

Component configuration includes several fields besides the inbound events. A component can be either general or autonomous, which determines how the framework handles it. The role (e.g., admin, security, management, and network) field indicates the kinds of events a component can receive and trigger. The permission field defines the capabilities of a component (reading the internal data of events, modifying data, and cutting off event distribution). Finally, the outbound field describes the kinds of events that will be triggered from a component.

To integrate a component into the framework, an operator can dynamically (un)load a component at runtime through Barista's CLI. Once a component is integrated into the framework, its execution order is automatically determined according to its role, permission, and dependencies on other events. If a component has a higher role (admin $>$ security $>$ management $>$ network) or permission (r:4, w:2, x:1), it will be executed before other components with lower roles or permissions. Suppose that component i and j listen to the same event A. Also, component i triggers event B to update some data, and component j triggers event C to read the data updated by component i. In this case, component i will be executed first for event A since it can trigger event B based on event A. The framework also allows advanced operators to adjust the execution order among components through the CLI for additional flexibility.

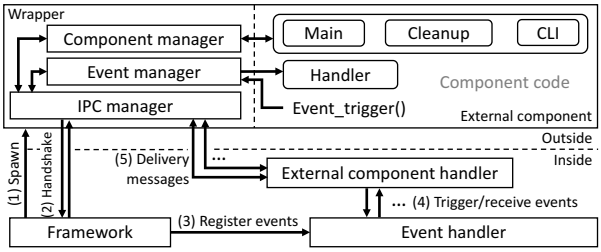


Fig. 2: Using wrappers for component portability

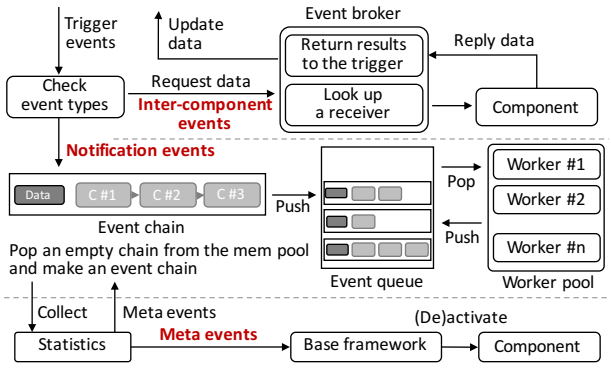


Fig. 3: Barista event processing logic - The upper layer shows inter-component event handling through direct interaction with the event broker. The middle layer shows notification events, which are handled through an event queue that feeds events to registered components. The bottom layer shows meta events, which are derived from notification event statistics and delivered to the base framework.

C. Component Portability

Barista allows a component to be executed either inside or outside the framework by providing a wrapper between external components and the framework. The wrapper allows the same code of a component to be used, no matter where it is executed, without any modification. As shown in Figure 2, the wrapper is composed of four functions: inter-process communication (IPC), event and component managers and an external component handler. The IPC manager coordinates all messages between an external component and the framework (including events) and delivers the messages to the corresponding managers. The event and component managers emulate the behavior of the framework based on the given messages. The external component handler converts messages coming through IPC channels to actual events. With those functions, external components can transparently communicate with other components.

This portability yields two major benefits. First, a component can freely introduce 3rd-party libraries and systems. For example, many modern controllers [5], [25] use the OSGi framework [6] to achieve dynamic development and deployment. While this framework is used to develop new OSGi-based modules and dynamically integrate them together, its integration is limited to OSGi-based modules. If a 3rd-party system does not provide OSGi-based libraries, a developer has to embed whole libraries into a bundle or make a valid OSGi bundle from the libraries. Barista, however, does not restrict the integration of other libraries, which means that developers can build components using 3rd-party libraries as independent applications. Second, a component can be fully isolated. While a few NOSs [30] adopt a micro NOS approach (e.g., [12]) to isolate functionality, most NOSs [1], [5], [25] still follow a monolithic approach; Even though each component is modularized, the execution environments are shared. This monolithic approach allows a small component failure to spread across the whole system in a cascade effect [11]. The failure of a component in Barista does not affect the whole operation of a NOS, although some partial functionalities might be temporarily unavailable. As soon as a component fails, it can be restarted and recover its original states depending on the implementation of the component.

D. Component Pool

Barista provides a component pool that includes a set of 25 self-developed components supporting the functionalities of contemporary NOSs from which developers can easily pick-and-choose based on their operating needs.

Barista provides the necessary components (packet I/O engine, protocol parser (e.g., OpenFlow), and application handler), management components (switch, host, topology, flow, and statistics), and logging as *base components* to implement a NOS. As *component extensions*, the cluster component internally integrates a 3rd-party distributed storage for scalability. For performance improvement, static flow-rule enforcement and a flow-rule cache may be deployed. For security, application authentication, role-based authorization, component access control, control flow and internal message integrity, data-plane message verification, and flow-rule conflict resolution components may be embedded into the control flows of a NOS. Monitoring components for system resources and the control channel between a NOS and the data plane can be used to detect abnormal NOS behavior. The failure management and isolation for southbound interfaces and applications can increase the robustness of a NOS.

IV. SDN Event Handling

The Barista event-handling framework seeks to service a broader range of component composition strategies and expose event-handling configuration as part of the NOS operation. For example, Barista introduces an event broker that enables the NOS author to 1) associate components to a diverse set of event types, 2) define dynamic event chaining among components, and 3) offer policy-based event distribution. This section presents these three event brokerage issues as they are addressed within the Barista framework.

A. Handling Diverse Event Classes

Barista extends the SDN event handling model by incorporating inter-component communications and event-broker-derived meta events as additional event classes that Barista authors can define. We illustrate the processing paths of those three event classes in Figure 3.

Notification Events: The event-handling mechanisms used by today's SDN controllers are quite straightforward. A component first registers a callback function to an event handler

for receiving a subset of data-plane events (e.g., PACKET_IN messages). When a registered event occurs, the event handler forwards the event by invoking the callback function. Barista also provides a mechanism for managing notification-based data-plane events.

Inter-component Events: While today's NOSs provide dynamic and modular component designs through software services (e.g., OSGi [6]), inter-component dependencies remain tightly coupled to component implementations. For these environments, events are used for message distribution, and direct function (API) calls are used for component-to-component data exchanges (e.g., getting switch details). Even modular components designs may require inter-component data sharing.

Here, we define two terms: *contextual dependency* and *functional dependency*. A contextual dependency arises when one component, *i*, needs the information produced from another component, *j*, for *i* to operate. A functional dependency arises when a component, *i*, must submit data to a component, *j*, (e.g. employing an API call) for *j* to produce a result that is then consumed by *i*. A highly modularized system is one in which functional dependencies are minimized among system components while retaining contextual dependencies.

Barista uses the event broker to replace functional dependencies among components with inter-component communications involving the exchange of request and response events, as shown in Figure 3. When a component triggers a request event to the event handler, the event broker delivers the event to a target component instead of the original component. When the target component triggers a reply event, the event broker replaces the request event with the reply event. Thus, the Barista event framework replaces functional dependencies between components with an inter-component event mechanism that removes component-specific implementation dependencies.

Meta Events: Barista produces meta events for live event statistics such as event volume, component-level statistics regarding event consumption or production, and event-type distribution statistics. Barista also allows operators to define the thresholds that trigger meta events, and to associate handling logic with produced meta events. Meta events can be configured to dynamically activate and deactivate components, or to filter certain events otherwise delivered to specific components. The event handler automatically triggers the predefined handling logic as defined by the operator when meta events are produced.

Meta events offer a novel mechanism to define specialized handling components to deal with dynamic event-stream conditions, such as dynamically activated component logic to deal with *flashmob traffic* or other unexpected saturation events. Meta events allow the operator to activate and deactivate components that are pre-deployed to address certain event production anomalies that can arise from a wide-range of anticipated operating conditions. This meta-event-handling service represents an extension beyond existing NOSs, which may dynamically load and unload components, but require human intervention to do so as anomalous event production patterns occur. Meta events offer administrators a means to express conditional component activation in advance, and to deactivate such logic when event-production patterns indicate that such conditions are no longer present. Meta events can also impose event handler filtering adjustments, such as the filtering of specific events to certain components that could result in unwanted resource utilization.

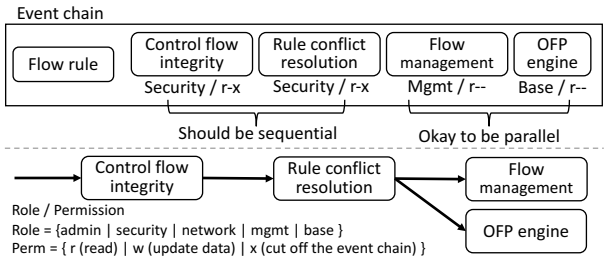


Fig. 4: Handling events in sequence and in parallel

B. Dynamic Component-Event Chaining

An event chain occurs when the event handler must service a group of components that are designed to consume a common event. Modern NOSs do not usually expose services to define chaining strategies among its components; such strategies must be defined within the code or through manual priority configuration. The lack of event-chain support for components among NOSs is likely because they are primarily concerned with non-interference. However, some components, especially security components (e.g., a flow-rule conflict resolver, or an integrity checker), require fine-grained controls with even-processing orders [27]. To accommodate this requirement, Barista facilitates explicit event chaining by default.

Barista has two ways to deliver events to components, as shown in Figure 4: *sequential delivery* and *parallel delivery*. Sequential delivery is used when a component is granted permission to terminate event-chaining sequences based on an internally-defined decision regarding the event, such as a filter-criteria match (e.g., update, cut-off permissions). Parallel delivery is used when components consume events but do not impact the delivery of the event to other components (e.g., read-only permission). Event-sequence formulation begins with Barista ordering component event delivery based on each component role and delivering events to components that have higher authority first. Barista then evaluates which component can be delivered events in parallel or sequential. Next, it checks for contextual dependencies among the consumers. If so, it imposes sequencing such that the dependent component follows the event processing of the independent components. Finally, it processes the adjusted event chains.

This dynamic composition of event chaining enables a fine-grained control of component composition within the NOS, which is particularly relevant to the integration of security and fault-recovery components. While existing NOSs enhance network serviceability through various network components, their architectural designs impose limitations on adding security features. For example, SE-Floodlight [26] requires modification to its internal logic to embed a flow-rule conflict-detection mechanism. In contrast, Barista's architecture is designed to modularize components and event-handling flows. This design simplifies the integration of research extensions into the processing pipeline of NOS notification and inter-component events.

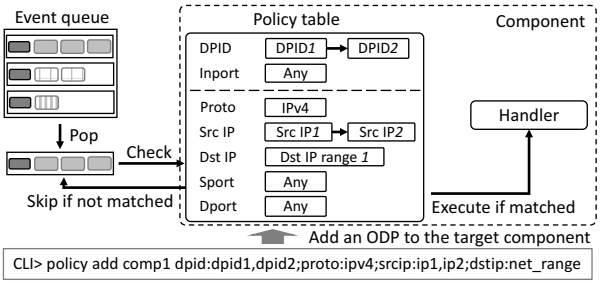


Fig. 5: Policy-based event distribution

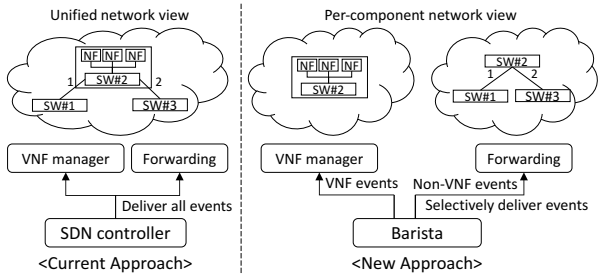


Fig. 6: Selective network management with per-component policy specification

C. Policy-based Event Distribution

Rather than employing event-handling mechanisms, existing NOSs focus on adding more functions to satisfy ever-increasing operating requirements. This approach is neither scalable nor robust. In contrast, Barista provides operator-defined policies (ODPs). that allow operators to select the set of components that they wish to deploy and specify the event-handling policy for the deployed components.

In the current Barista prototype, each ODP is composed of seven fields: datapath ID, in-port (incoming switch port), protocol, source/destination IP addresses and ports. This can be extended based on future operator needs. Once an operator defines an ODP, Barista updates the policy table in a target component as shown in Figure 5. Then, when the event handler identifies an incoming event, it matches the event with the policy table of a target component before delivery. If the event does not match component policy, the event handler proceeds to the next component in the event chain.

In Figure 6, an SDN allows an operator to manage network traffic from a global perspective over the switching infrastructure. Using one or more SDN applications hosted on the NOS, each network flow can be managed by an individual application or among multiple applications. However, an operator may need to dynamically alter the network flow management policy depending on runtime criteria. To illustrate this point, let us consider a deployment in which the SDN operator employs two flow management applications: a flow-forwarding application and a virtual-network-function (VNF) manager, which selectively directs certain flows to one or more VNF box(es). While the VNF manager locally routes traffic through VNFs (i.e., *VNF chaining*), the forwarding application handles traffic globally. However, current NOSs do not provide mechanisms to selectively differentiate between such applications; both applications receive messages for all traffic, even though some messages are not relevant. Hence, the operator chooses to deploy those applications independently. Unlike current NOSs, Barista's ODPs can filter messages so that the VNF manager only handles traffic bound to each VNF box, and the forwarding application handles all traffic except for the traffic handled by the VNF manager.

D. Event Distribution across Instances

The Barista event handler uses the cluster component to deliver events to other instances. The cluster component at each instance shares its events through distributed storage. When indicated events are triggered at one of the instances, the cluster component receives them from the event handler and stores them in distributed storage. At the same time, the cluster component continues polling for new events from other instances and triggers those events at its instance. The distributed storage maintains logical sequences to ensure that incoming events are ordered chronologically. The polling mechanism gets events from the distributed storage and currently supports eventual consistency; supporting strong consistency is a future goal.

The information to be shared across Barista instances depends on the events received by the cluster component. For example, if operators only want to share topology information, they can set switch and topology events at the cluster component. Some events can be shared among specific instances rather than all instances allowing operators to strategically distribute events at each instance.

V. System Implementation

A prototype of Barista has been implemented to evaluate the efficiency and effectiveness of its design, including the base framework and a broad set of components. The Barista prototype consists of over 17K lines of mostly C code and supporting Python scripts.

The base framework maintains a component list containing operator-defined configurations (JSON format). Event distribution across Barista instances uses MariaDB and Galera Cluster [3] and makes batch transactions. The wrappers for component portability use POSIX message queues and shared memories. We are currently developing the wrappers in more languages to provide additional options to Barista developers. To generate control traffic for evaluation, we modified Cbench [29] to produce more diverse control messages with specific input parameters, such as the range of IP/MAC address pairs and messages per second. All implementation details and source code will be available at https://github.com/sdx4u/barista.

VI. Use Cases

This section presents three use cases that demonstrate the effectiveness of Barista: A) a distributed and secure NOS, B) separated network management using operator-defined policies and C) an IoT use case.

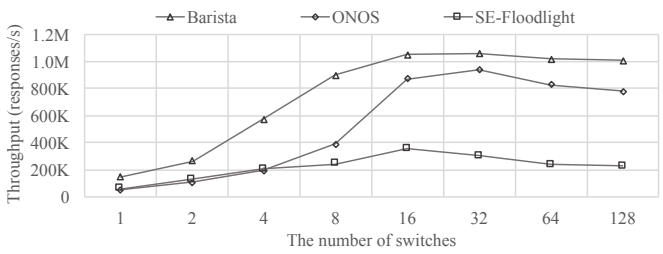


Fig. 7: Throughput comparison of Barista, ONOS, and SE-Floodlight

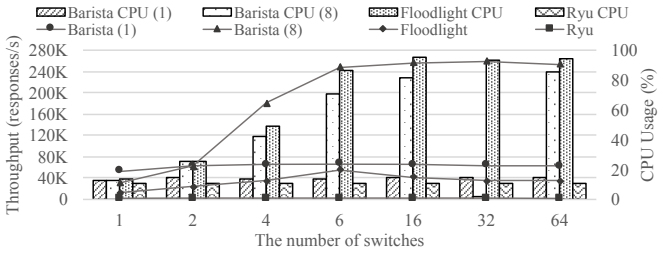


Fig. 8: Throughput comparison of Barista, Ryu, and Floodlight on a single-board device

A. Distributed and Secure NOS

This use case describes how an operator can brew a scalable, secure NOS using the Barista framework as discussed in Section II-C. The first step considers the functionalities of a scalable NOS (i.e., ONOS) and a secure NOS (i.e., SE-Floodlight). ONOS achieves scalability with its distributed mechanisms (e.g., distributed storage and raft-based consistency model). In the case of SE-Floodlight's key features are role-based authorization, component authentication, and flow-rule conflict resolution. An operator can simply enable those functionalities (i.e., cluster, role-based authorization, component access control, and flow-rule conflict resolution) with the base components in the Barista framework.

To show the effectiveness of the Barista NOS composition, we compared the throughputs of Barista, ONOS, and SE-Floodlight with default configurations. The measurements used Cbench with 1,000 virtual hosts connected across 48 ports per switch. Figure 7 illustrates the per-instance throughput of the 3-node Barista cluster, 3-node ONOS cluster, and SE-Floodlight with varying numbers of switches. The reason for per-instance throughput is because SE-Floodlight only supports a single instance. Each ONOS instance saturated at an average of 940K responses/s, and SE-Floodlight saturated at 357K responses/s. The Barista instance supported a maximum throughput of 1,059K responses/s. Although the Barista instance lost some throughput due to the high computation overhead coming from checking flow-rule conflicts, it showed almost three times higher throughput than that of SE-Floodlight and was comparable with ONOS. With this throughput rate, the Barista framework allows operators to compose required functionalities with competitive performance.

B. Separation of Network Management

Barista allows operators to define operator-defined policies at a per-component level (i.e., they can use policies to affect the set of flows seen by each Barista component). To illustrate this capability, we instantiated a simple network, shown in Figure 6, that is managed with a Barista controller. Here, an operator wants to separately manage network traffic with a forwarding application and local traffic at the VNF box with a VNF manager. Achieving this requires the definition of three ODPs. To handle traffic on the physical network using the forwarding application, the operator defines two ODPs: *"forwarding dpid:!2"* and *"forwarding dpid:2; port:1,2; proto:lldp"*. To handle local traffic at the VNF box using the VNF manager, the operator defines a third ODP: *"vnf_manger dpid:2"*. These ODPs allow Barista to filters out events so that each application sees only the events defined by ODPs. This requirement can be satisfied without any application modification by applying ODPs that control event flows inside the controller. Thus, Barista empowers operators by providing the ability to dynamically control flow handling within the controller through well-defined policies. As another example, operators may define policies to scale up the throughput of a component by increasing the number of component instances and assigning a subset of traffic to each one (i.e., "forwarding_1 dstip:192.168.0.0/25" and "forwarding_2 dstip:192.168.0.128/25").

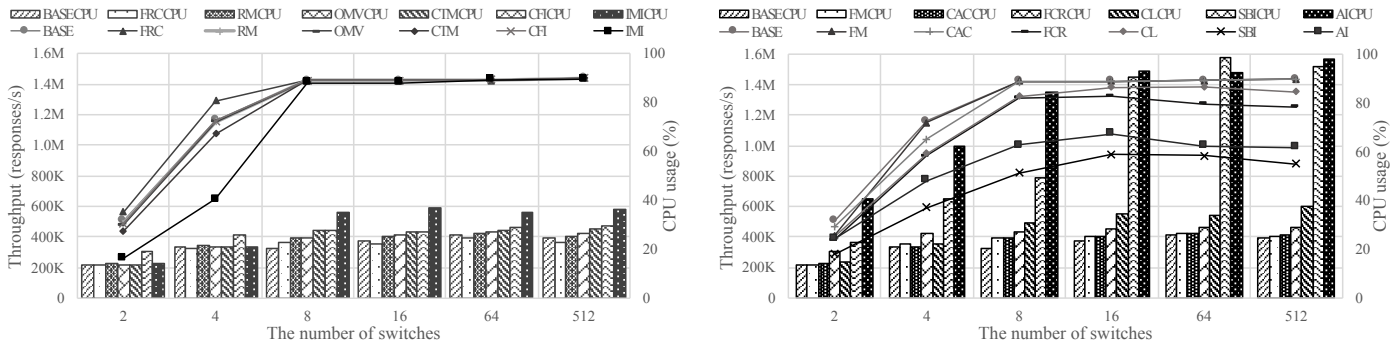
C. Lightweight NOS for IoT Environments

IoT devices require a lightweight NOS as they have much lower computing power than commodity servers. Unfortunately, most contemporary NOSs [5], [25] are designed for server-side deployments and unsuitable to be run on small devices with limited resources. However, a few controllers [1], [7], [8] are readily executable on such devices (specifically the ODROID XU4 [16]).

To demonstrate the applicability of Barista in this situation, we compared it with Ryu [8] and Floodlight [1] (default configurations). We used ODROID XU4 (ARM Cortex-A7, Octa-core, 2 GB of RAM) and a set of base components for the Barista cases. Figure 8 illustrates the throughput of each NOS (lines). The throughput of Ryu is 2,058.4 responses/s on average. Since Ryu is a single-thread controller and threading is only available for application tasks, its throughput does not scale up and is much lower than others. Because Floodlight can run with multiple threads, its throughput goes up to 55,005.0 responses/s with a 93.8% CPU usage. However, Barista shows even higher throughput. With a single worker, Barista performs up to 63,523.5 responses/s while consuming up to 14.5% of CPU resources. With eight workers, Barista performs up to 254K responses/s. This demonstrates that the Barista NOS is also suitable for small computing devices such as IoT gateways with low resource consumption.

VII. System Evaluation

This section describes experimental results that validate the efficiency and effectiveness of the Barista prototype system. Our testbed comprised six machines. Three machines ran Barista instances, each with Intel E5-2620 CPU (6 cores, 2.40 GHz) and 32 GB of RAM. Three other machines, each with an Intel i5-6600K CPU (4 cores, 3.50GHz) and 16 GB of RAM



Base: base components (described in Section III-D), FRC: flow rule cache, RM: resource management, OMV: OF message verification, CTM: control traffic management, CFI: control flow integrity, IMI: internal message integrity, FM: failure management, CAC: component access control, FCR: flow conflict resolution, CL: cluster, SBI: southbound isolation, AI: application isolation (application authentication, static flow rule enforcement are excluded)

Fig. 9: Microbenchmarks of throughput and CPU usage

were used for control-message generation. For the evaluation, we set Cbench to generate 1,000 unique hosts for each switch.

Component Microbenchmarks: To understand how each component affects a NOS, we measured the throughput and CPU usage of a set of extensible components along with the base components. Figure 9 illustrates the throughput and CPU usage for each component as we varied the number of switches. We present the cases with 16 event-handling workers finding that the throughput of most components (from 428K to 1,338K responses/s on average) is comparable to that of the base components (from 505K to 1,438K responses/s). The throughput is saturated at around 1.4M responses/s due to NIC bandwidth limitations (1 Gbps). When the number of switches is low, the throughput suffers because of the workload imbalance across workers (we will optimize the framework in the near future). In most cases, each component has minimal impact on the overall throughput; however, as more components are integrated into the framework, the CPU usage also increases. For example, the CPU overhead of AI and SBI is significantly higher (up to 98.3%) than the others (up to 37.8%).

The cases with two and four switches in Figure 9 show the internal message integrity at 260K and 645K responses/s (-48.4% and -44.5% compared to the throughputs of the base components), respectively, because of the checksum-generation overhead. The control-traffic management sightly decreases the throughputs (438K and 1,158K responses/s, -13.3% and -7.5%) since it monitors all incoming and outgoing messages. However, as the number of workers increases, their overheads are covered with sufficient computation resources. Unlike the other components, the throughputs of the flow-rule cache increase (561K and 1,152K responses/s, +11.1%). Figure 9 shows that the throughputs of four components (cluster, flow conflict resolution, application isolation, and southbound isolation) visibly decrease. The throughput of the cluster (-5.8% to -20.2%) is reduced due to the read and write overheads from a distributed storage. The flow-conflict resolution compares all possible alias-reduced rules (ARRs introduced in [26]) with existing flow rules, resulting in high computation overhead (-7.0% to -24.2%). The performance degradation of SBI and AI mainly occurs due to the IPC overhead. Since some data-plane messages (e.g., PORT_STATUS, and PACKET_IN messages related to LLDP packets) are not delivered to the application layer, the throughput of SBI (-33.8% to -48.9%) is relatively lower than that of AI (-23.1% to -33.5%).

VIII. Related Work

A growing list of competing NOS software projects have explored features that appeal to various network operator communities. NOX [15] emerged as the first network operating system for SDNs and has since been ported to Python - POX [7]. Both Floodlight [1] and Beacon [13] followed the release of NOX and were primarily optimized to maximize connection throughput. Later NOS architectures have extended early NOS functions to address the growing interest in SDNs for managing large and dynamic (virtual) network environments. Onix [21] represents the first effort to address scalability by developing a distributed NOS platform. ONOS [25], Hyperflow [31], Kandoo [17], OpenDaylight [5], and Beehive [32] incorporate similar objectives. They are designed as distributed platforms to support large numbers of requests in wide-area environments and emphasize the need for greater scalability while maintaining high performance. However, these platforms do not focus on security or robust network application management in their designs.

SE-Floodlight [26], FortNOX [27], Rosemary [30], LegoSDN [10], and Ravana [19] demonstrate the integration of multi-network-application security and robustness features into the SDN control layer. Those controllers focus on application consistency and robust application management to enhance NOS reliability in sensitive computing environments with less regard to the scalability and performance issues that are presented in other production environments. While Corybantic [23] optimizes the modularized management of controller applications, it does not consider the adoption of any security mechanisms. Other NOSs have adopted component-based architectures [5], [8], [25], but they do not consider the issue of how to compose components based on operator-defined requirements. While new features could be integrated into ONOS [25] and OpenDaylight [5], the interface for component extension is not straightforward. Ryu [8] provides

a basic set of components that is a strict subset of Barista's component library.

Other approaches to deal with multi-controller environments include FlowVisor [9], [28], Frentic [14], and Pyretic [24]. FlowVisor divides a network into slices and enforces strict isolation between controllers running above it, while managing provisioning and shared resources. FlowVisor could similarly be used to manage Barista instances while providing a more homogeneous northbound API for applications. Pyretic provides a higher-level runtime that resides "above" the controller providing compositional operators for querying and transforming network streams. In fact, Barista's sequential and parallel composition functions are inspired by Pyretic. Such runtimes could be ported to run over Barista controllers. Finally, commercial products such as HP-VAN [2] and the NEC controller [4] are proprietary closed-source systems. We have limited insight into their implementation, but they do not support our approach of specification-driven NOS synthesis.

IX. Conclusion

The NOS is an integral piece of SDN, and its functionalities significantly influence the whole network. While scalability, security, and reliability challenges should be addressed in the design of a NOS, we observe that contemporary NOS solutions tend to be focused on specific dimensions, and the integration of the functionalities across NOSs is challenging due to incompatible design principles. Barista takes an important step toward addressing this problem by providing a NOS component-synthesis framework that simplifies the integration of composable NOS modules with a dynamic event-handling mechanism. We evaluated the system against a range of commodity NOSs with useful scenarios, and found that Barista simplifies instantiation of a NOS with diverse feature combinations and efficiently replicates functionalities found in major NOSs while delivering competitive performance.

Acknowledgement

KAIST was supported by Institute for Information & Communications Technology Promotion (IITP) grants funded by the Korean government (MSIT) (No. 2015-0-00575, Global SDN/NFV Open-Source Software Core Module/Function Development). SRI International was supported by the National Science Foundation under Grant No. 1446426. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Floodlight. http://floodlight.openflowhub.org.

[2] HP VAN. https://www.hpe.com/us/en/networking/management.html.

[3] MariaDB. https://mariadb.com/kb/en/mariadb/galera-cluster.

[4] NEC Controller. https://www.necam.com/sdn/Software/SDNController.

[5] OpenDayLight. http://www.opendaylight.org.

[6] OSGi: The Dynamic Module System for Java. https://www.osgi.org/.

[7] POX. http://www.noxrepo.org/pox.

[8] RYU Controller. https://osrg.github.io/ryu/.

[9] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshiba, G. Parulkar, E. Salvador, and B. Snow. OpenVirteX: Make Your Virtual SDNs Programmable. In *Proceedings of the Workshop on Hot Topics in Software Defined Networking*, 2014.

[10] B. Chandrasekaran and T. Benson. Tolerating SDN application failures with LegoSDN. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2014.

[11] S. L. Changhoon Yoon. Attacking SDN Infrastructure: Are We Ready for the Next-Gen Networking? *Blackhat USA*, 2016.

[12] D. R. Engler, M. F. Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*. ACM, 1995.

[13] D. Erickson. The Beacon Openflow Controller. In *Proceedings of the Workshop on Hot Topics in Software Defined Networking*, 2013.

[14] N. Foster, M. J. Freedman, R. Harrison, J. Rexford, M. L. Meola, and D. Walker. Frenetic: a high-level language for OpenFlow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, 2010.

[15] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *Proceedings of ACM SIGCOMM Computer Communication Review*, 2008.

[16] Hardkernel. ODROID XU4 Board. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G143452239825.

[17] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proceedings of the Workshop on Hot Topics in Software Defined Networks*, 2012.

[18] S. Hong, L. Xu, H. Wang, and G. Gu. Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures. In *Proceedings of the Annual Network and Distributed System Security Symposium*, 2015.

[19] N. Katta, H. Zhang, M. Freedman, and J. Rexford. Ravana: Controller Fault-tolerance in Software-defined Networking. In *Proceedings of the Symposium on Software Defined Networking Research*, 2015.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 2000.

[21] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2010.

[22] S. Lee, C. Yoon, C. Lee, S. Shin, V. Yegneswaran, and P. Porras. DELTA: A Security Assessment Framework for Software-Defined Networks. In *Proceedings of the Annual Network and Distributed System Security Symposium*, 2017.

[23] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner. Corybantic: towards the modular composition of SDN control programs. In *Proceedings of the ACM Workshop on Hot Topics in Networks*, 2013.

[24] C. Monsanto, J. Reich, N. Foster, J. Wexford, and D. Walker. Composing Software-defined Networks.

[25] OnLab.us. Open Network OS. http://onlab.us/tools.html#os.

[26] P. Porras, S. Cheung, M. Fong, K. Skinner, and V. Yegneswaran. Securing the Software-Defined Network Control Layer. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[27] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A security enforcement kernel for OpenFlow networks. In *Proceedings of the workshop on Hot Topics in Software Defined Networks*, 2012.

[28] R. Sherwood, G. Gibe, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2010.

[29] R. Sherwood and Y. Kok-Kiong. Cbench: an open-flow controller benchmarker, 2010.

[30] S. Shin, Y. Song, T. Lee, S. Lee, J. Chung, P. Porras, V. Yegneswaran, J. Noh, and B. B. Kang. Rosemary: A Robust, Secure, and High-performance Network Operating System. In *Proceedings of the Conference on Computer and Communications Security*, 2014.

[31] A. Tootoonchian and Y. Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the Internet Network Management Conference on Research on Enterprise Networking*, 2010.

[32] S. H. Yeganeh and Y. Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, 2016.