Operator-Defined Reconfigurable Network OS for Software-Defined Networks

Jaehyun Nam, Hyeonseong Jo, Yeonkeun Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin

Abstract—**Barista is a novel architecture that seeks to enable flexible and customizable instantiations of network operating systems (NOSs) for software-defined networks (SDNs). As the NOS is the strategic control center of an SDN, implementing logic for management of network switches as well as higher-level applications, its design is critical to the welfare of the network. In this paper, we focus on three aspects of composable controller design: component synthesis, dynamic event control, and predictive NOS assessment. First, the modular design of the Barista enables flexible composition of functionalities prevalent in contemporary SDN controllers. Second, its event handling mechanism enables dynamic customization of control flows in a NOS. Third, its predictive NOS assessment helps to discover the optimal composition for the requirements specified by operators. These capabilities allow Barista operators to optimally select functionalities and dynamically handle events for their operating requirements while maximizing the resource utilization of the given system. Our results demonstrate that Barista can synthesize NOSs with many functionalities found in commodity controllers with competitive performance profiles.**

Index Terms—**Software-defined networking, component synthesis, dynamic event control, predictive NOS assessment.**

I. Introduction

SOFTWARE-defined networks have emerged as a compelling alternative to traditional, vertically integrated networks, which offer limited flexibility, programmability, and customizability. Among the motivations that drive SDNs is a desire to address the "islands of functionality" challenges that arises 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 offer a more agile platform on which diverse functions can be integrated in a more unified manner.

However, from interactions with SDN operators in industry and academia, it appears that stovepiped functions persist within SDNs, albeit in a different manifestation. For example, it is indeed 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 cannot be directly moved to OpenDaylight).

We posit that the design choices and rigidity in composition of contemporary SDN controllers significantly limit their ability to fully address the challenge of satisfying competing demands within enterprise networks. We further make the case for a new controller design that allows for customizable controllers while retaining a uniform programming API.

A NOS Overview. The NOS provides a unified abstraction of the SDN and implements the interface between higher-level applications and the software-defined data plane. Further, it provides global network programmability, enabling seamless deployment of dynamic and intelligent services across the network. A modern NOS is arguably the most critical component in the SDN stack, as it is responsible for both controlling the underlying network substrate (through the southbound API), and managing higher-level network applications (through the northbound API). Hence, the SDN control layer represents a vibrant area of active research and development, where significant software abstraction, security, and reliability challenges are centered.

There have been several notable efforts to develop NOSs both in academia (Onix [1], Beacon [2], Rosemary [3]) and industry (e.g., HP VAN [4] and Bigswitch BigNetwork Controller [5], ONOS [6], OpenDaylight [7], and FloodLight [8]). Interestingly, each of these systems specializes in different dimensions, offers varying system and performance capabilities, and differs in their appeal across different operational settings. For example, the Beacon [2] controller is highly optimized for providing maximal throughput from a single controller, while ONOS [6], Onix [1], and OpenDayLight [7] focus on distributed scalability. Alternatively, SE-FloodLight [9] attempts to address a range of security requirements that are imposed within sensitive network computing environments.

Barista. In this paper, we propose Barista, an event-driven composable SDN controller generation framework. Barista enables the rapid modular prototyping, customization, and fielding of control layer logic to meet a wide-range of operational requirements found in many diverse network environments. We discuss the internal logic of existing NOSs, their lack of customization, and the difficulties involved in embed-

Manuscript received July 16, 2018; revised March 1, 2019; accepted April 15, 2019; approved by IEEE/ACM Transactions on Networking Editor A. Dainotti. Date of publication May 27, 2019; date of current version June 14, 2019. This work was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korean Government (MSIP) (Cloud-Based Security Intelligence Technology Development for the Customized Security Service Provisioning) under Grant 2016-0-00078. *(Corresponding author: Seungwon Shin.)*

J. Nam, H. Jo, Y. Kim, and S. Shin are with the Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea (e-mail: claude@kaist.ac.kr).

P. Porras and V. Yegneswaran are with SRI International, Menlo Park, CA 94025 USA.

Digital Object Identifier 10.1109/TNET.2019.2914225

1063-6692 © 2019 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

ding new functions, and their inability to alter their functional behaviors when network operating conditions change. Barista incorporates an event handling model that provides a diversity of events. It offers a dynamic composition of event chaining, and policy-based event distribution. Through those features, operators can even customize the flows of event data through the NOS itself. Finally, we provide the analytic tool that assesses the impact of each piece. This tool enables operators to consider their requirements, and intelligently integrate components to produce a NOS instance that supports those requirements, while maximizing the resource utilization.

Contributions. In summary, the salient contributions of our work include the following.

• We present the design of a new SDN controller brewing framework, called Barista, which substantially accelerates the ability of the SDN research community to rapidly prototype and integrate new control layer functionality into a distributed NOS environment. Barista reduces the implementation costs required to create new modular NOS functional extensions while increasing the sharability of these components.

• We introduce a new SDN event handling framework that enables fine-grained control over events delivered to NOS components that implement the network control logic. Barista introduces an event broker model 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) express event distribution policies that control network event visibility per NOS component.

• We present an approach to NOS component performance assessment that operators can use to select the best NOS component compositions given target network conditions. This assessment approach stands in complement to NOS benchmarking services such as *Cbench* [10], which is widely used to evaluate throughput and latency performance of a NOS under various PACKET_IN volume streams. Here, we present an approach that assesses NOS performance under diverse event workloads, extends the assessment to consider NOS resource utilization (CPU, memory), and assists in identifying the best component composition given target performance constraints.

• We evaluate Barista against a set of diverse use cases, and demonstrate its modular composability, and its ability to address a range of operational use cases that no single current NOS can address.

II. Motivation for the Barista NOS

Over the past few years there has been a growing list of competing NOS software projects that have explored features that appeal to various network operator communities. Barista represents a unique design perspective among this growing spectrum of competitive NOS software projects. Here we motivate the need for a highly composable NOS compilation framework, which 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. Combined, we believe Barista offers a unique NOS approach, that is applicable to both the research community and to 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 software-defined network research. In surveying this work, researchers often employ an existing NOS as a base from which to explore new control-layer features. For example, Ravana [11] modifies Ryu [12], 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, such as [9] and [13]. 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 [6] and OpenDaylight [7]. Alternatively, other researchers have introduced complete ground-up NOS proposals, which focus on exploring specialized properties, such as robust application management with high performance [3].

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 (similar to the motivation that inspired the Click modular router [14]). For example, with Barista, one can rapidly devise and implement a new flow-rule conflict detection algorithm that does not require the 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 [9]). Barista-hosted NOS extensions offer modular components that do not require NOS-internal modifications to join its event pipeline. We believe that Barista's approach to a component-based NOS architecture provides the research community with a rapid development framework, which 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

When network operators deploy SDN-enabled networks, they commonly face a basic question of how to select the best NOS to match the operational requirements of the target network. For example, consider a case in which a network operator must manage two networks: *network A* consists of 1,000 switches and 100,000 hosts for web testing, and *network B* consists of 10 switches and 100 hosts that provide database services for a corporate-sensitive dataset. For this scenario, our operator may 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. Unfortunately, while managing two separate NOSs may offer the ability to match each environment with the most applicable NOS features, the deployment of two different NOS platforms will also impact the overall management cost.

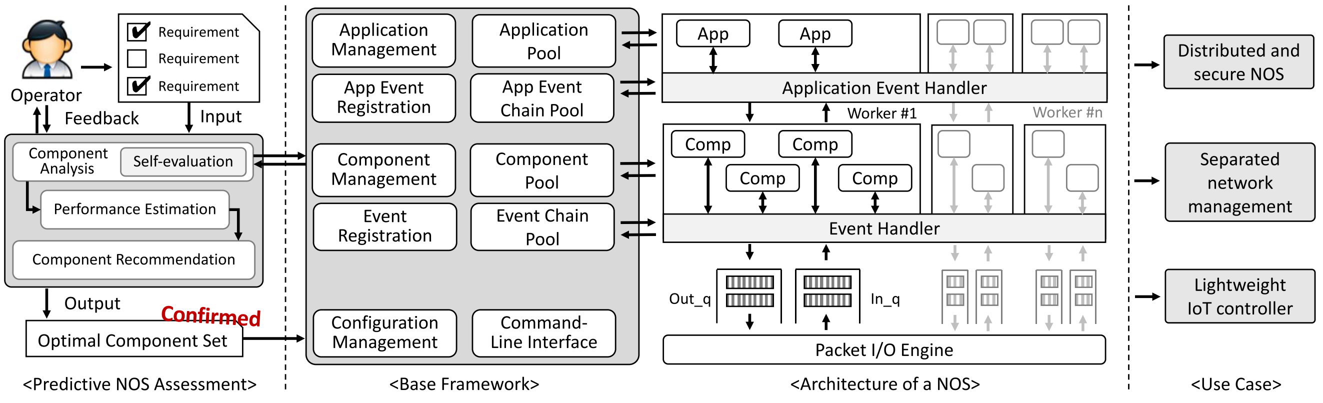


Fig. 1. Barista System Overview: Predictive NOS assessment, Base and Event Handling Frameworks, and Use Cases

Thus, the second motivation for Barista is to design a NOS compilation framework that enables the NOS to be customized at deployment time, with those critical features that best suite the target environment. Here, our network operator simply specifies the functional requirements that are present in each network, and produces two automated compilations of Barista that deliver the comparable functional services that are 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 compilation phase will produce a *NOS composition* that integrates those functional components that match the stated objectives.

III. System Design

In this section, we present the design of Barista and explain how it facilitates the development of 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 has focused on specialization of NOS features to better support target network environments. Rather, its goal is to enable operator-defined composability of NOS features that can be assembled under the constraints of available resources. Here, we first describe the composability through the base framework, and look into the flexible event handling framework in Section IV and the predictive NOS assessment tool in Section V.

A. Base Framework

The Barista base framework is composed of two key elements: *components* and *events*. A component represents the implementation of a specific NOS function. Here, the framework provides two classes of components. The first class is a general component (e.g., packet I/O engine and OpenFlow engine). General components are designed primarily to become a functional logic inside of a NOS. The second class is an autonomous component (e.g., statistics and resource managements). Autonomous components are quite similar to general components. However, they are intended to independently conduct certain actions without intervention.

In the framework, events play a central role by driving 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. Rather, all intercommunication between components is done through events. While this event-handling mechanism may increase the communication overhead compared to direct function calls, it enables components to be highly decoupled and enhances the ability for components to be combined automatically during the NOS build phase and even in runtime.

In terms of integrating NOS components, it provides a component designer with a degree of abstraction from the NOS internals as well as an abstraction for defining component composability. Since each component is isolated from others and events are the only way to communicate with each other, operators can easily add, remove, and substitute components.

While the base framework also provides applications and an application event handler as shown in Figure 1, their functionality is exactly the same with components and the event handler. The only difference is what kinds of information they deal with. Components focus more on lower-level data to manage connected switches and hosts, discovering a network topology and so forth. On the other hand, applications use application events to handle network flows in the data plane.

B. Component Development

The code template of a Barista component is illustrated in Figure 2. A component is composed of four pieces: main, cleanup, CLI (command-line interface), and handler functions. The main and cleanup functions have similar functionalities of a constructor and a deconstructor. The CLI function is the interface of a component for operators. It allows operators to interact with the component in runtime. The handler function plays a role of the core function of a component. It is responsible for receiving predefined (inbound) events in the configuration of a component.

```
int sample_main(int *activated, int argc, char **argv) {
      // initialize data structures, load configurations, etc.
      activate();
}

int sample_cleanup(int *activated) {
      deactivate();
      // clean up used data structures and de-allocate memory, etc.
}

int sample_cli(char **args) {
      // implement CLI-based interfaces
}

int sample_handler(const event_t *ev, event_out_t *ev_out) {
      switch (ev->type) {
      case EV_EVENT_TYPE_1:
            ev_event_type_3(
                  SAMPLE_ID, ...);
            break;
      case EV_EVENT_TYPE_2:
            // do something
            break;
      }
      return 0;
}
```

```
"name" : "sample",
"args" : "arguments",
"type" : [general | autonomous],
"site" : [internal | external],
"role" : [network | security | admin],
"perm" : [r | w | x],
"inbounds":   ["EV_EVENT_TYPE_1",
               "EV_EVENT_TYPE_2"],
"outbounds": ["EV_EVENT_TYPE_3"]
```

Fig. 2. Code template of a Barista component

In the configuration of a component, there are several fields besides the inbound events. Each component can be either general or autonomous. Based on the type of a component, the framework handles the component differently. The role field indicates what kinds of events a component can receive and trigger. By defining a role, the framework restricts the use of receiving and triggering events. It currently provides 5 roles: admin, security, management, network, and base. The permission field defines the ability of a component (reading the internal data of events, modifying the data, and cutting off the event distribution). Finally, the outbounds field describes what kinds of events will be triggered from a component. Through the configuration, the framework determines how a component would be integrated and operated.

To integrate a component into the framework, Barista provides a command-line interface. Thus, an operator dynamically (un)loads a component at runtime through its 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 or permission, it will be executed before other components that have a lower role or permission. In terms of event dependency, let us say that component *i* and *j* listen to the same event, *A*. At the same time, component *i* triggers event *B* to update some data, and component *j* triggers event *C* to read the data updated through event *B*. In this case, the former component will be executed first for event *A*. The framework also allows operators to adjust the execution order among components to give more flexibility to them.

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 3, the wrapper is composed of four functions: channel, event and component managers and an external event handler. The channel 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 event handler converts incoming messages to actual events. With those functions, external components can transparently communicate with other components.

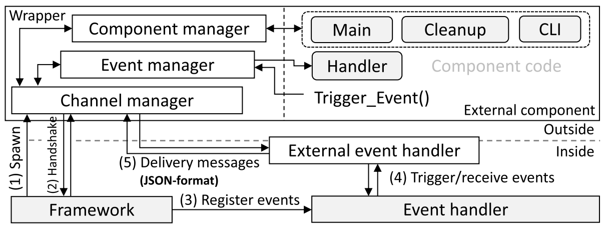


Fig. 3. Using wrappers for component portability

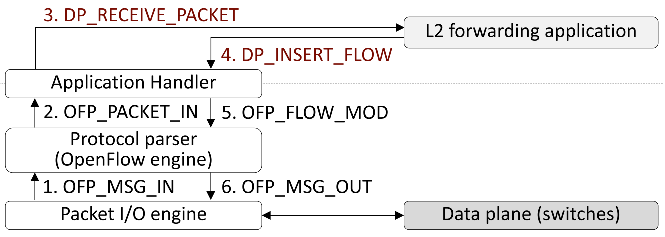


Fig. 4. A base component workflow example

Furthermore, Barista provides *a JSON-based message format* for communications between external components and the framework. Basically, in order to execute a component either inside or outside the framework, the development language of the component should be the same with that of the Barista framework. In this case, Barista may limit the freedom of development languages. Thus, with a more generalized message format (i.e., JSON format), Barista allows external components written in other languages (e.g., Java, Python) to be integrated while they cannot be directly embedded into the framework.

D. Component Composition

Barista provides a component pool (as summarized in Table I) that includes a set of pre-developed components supporting the functionalities of today's NOSs. Thus, operators can develop their own components by themselves or pick-and-choose components that they want from the component pool.

Base components: The Barista NOS requires a base set of components to function. Abstractly, a NOS requires a packet I/O engine to receive messages from the data plane, a parsing engine to interpret the contents of messages, and a decision maker to decide where packets should be forwarded to provide the minimum functionality of a NOS. Besides them, since those components do not provide a global network view, we include five additional management components (switch, host, topology, flow, and statistics) that maintain a range of network state information and pass that information to other components.

Figure 4 illustrates how the base components are combined with events, excluding management components for simplicity.

TABLE I
DESCRIPTION OF BASE AND EXTENSIBLE COMPONENTS CURRENTLY IMPLEMENTED IN BARISTA

Component	Description
(a) Base Components	
Packet I/O engine	Receives incoming messages from the data plane and sends outgoing messages to the data plane.
Protocol parser	Identifies the kinds of messages (e.g., OpenFlow, Netconf, and SMTP) and inspects messages using the corresponding parsing engines (Currently, we support an OpenFlow engine).
Application handler	Converts events to application events and vice versa.
Switch management	Manages connected switches and collects the switch-related information such as datapath ID, switch states, the number of ports, and so on. When switch alterations occur, it triggers notification events to other components.
Host management	Maintains connected hosts similar to switch management. It also supports dynamic host migration.
Topology management	Periodically sends LLDP packets to connected switches and receives the topology information from switches, and maintains the overall network topology as a graph.
Flow management	Maintains all active flow rules in switches.
Statistics management	Periodically requests current switch statistics and distributes the statistics to registered components.
Logging	Records any logging messages from components with priorities (FATAL, ERROR, WARNING, INFO, DEBUG).
(b) Extensible Components	
Cluster	Stores events into a distributed storage and polls events triggered in other Barista instances.
Static rule enforcement	Enforces predefined flow rules to switches when they are connected.
Flow-rule cache	Caches flow rules inserted to the switches. Candidate flows rules are vetted against this cached set prior to insertion into switch flow tables.
Control flow integrity	If a component becomes compromised at runtime, the component may trigger unexpected events or may access internal storages. It monitors event chains from a source to a destination and blocks abnormal events.
Internal message integrity	Generates the checksum of a new event and inspects all events to check any unexpected modifications. When it detects altered events, it alerts the component that modifies the events to operators.
OF message verification	It is possible that some fields of a control message are incorrect due to various application faults. This component provides a service for control message content vetting to identify "out-of-specification" content.
User authentication	Allows specific users to access the user interface of the Barista framework.
Application authentication	Provides a service for load-time verification of the integrity and authenticity of third-party applications.
Role-based authorization	Enables the operator to impose the limitations of triggering events on a per-group basis. It filters events issued by a component, allowing only access to pre-specified events based on the role of the component.
Component access control	The operations are similar to the role-based authorization. It enables the operator to impose the limitations of triggering events on a per-component basis.
Flow rule conflict resolution	Detect conflicts between newly enforced flow rules and existing flow rules. It generates all possible alias reduced rules (ARRs) introduced in FortNOX [15] and compares ARRs with existing rules.
Resource management	Monitors CPU and memory usages.
Control traffic management	Monitors inbound and outbound traffic.
Failure management	Since the base framework adopts a variety of components, it is possible that some component crashes may harm NOS operations. Thus, when component crashes occur, it recovers the crashed components or the whole NOS.
(c) Component Portability	
SBI isolation	If either the southbound layer or the core layer is crashed, all functionality of a NOS would halt. Thus, it allows recovering either layer while the other continues to work.
Application isolation	To prevent the core layer from any application faults, it separates applications from the core layer. It allows the core layer to maintain operations even when application crashes arise that would otherwise halt other components.

Each component has inbound and outbound events. For example, the protocol parser (i.e., OpenFlow engine) has two inbound events (i.e., OFP_MSG_IN and OFP_FLOW_MOD) and two outbound events (i.e., OFP_PACKET_IN and OFP_MSG_OUT). Once one of the inbound events is triggered, the protocol parser receives the events through the event handler. Then, when it generates one of the outbound events, the event handler takes and delivers it to the application handler and the packet I/O engine according to the event type. Similarly, the application handler triggers received events from the event handler to the application event handler. Then, the L2 forwarding application receives inbound events (DP_RECEIVE_PACKET) and generate outbound events (DP_INSERT_FLOW).

Extensible components: Besides the base components, we currently provide 14 extensible components for scalability, performance, confidentiality, integrity, and availability as shown in Table I-(b). Those components support specific functionalities to achieve each feature. For instance, the role-based authorization serves to improve the confidentiality of a NOS. The internal and OpenFlow message verification components keep the integrity of a NOS. Likewise, the combination of those components allows operators to support multiple capabilities depending on their needs at the same time.

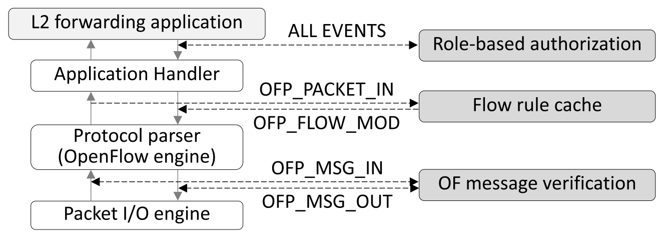


Fig. 5. Integrating extensible components into a NOS

To add extensible components at a NOS, the base framework generally employs the events of the base components. Figure 5 presents how the extensible components are integrated into the NOS. Since most of the components intervene in the control flows of a NOS, those components exploit the same events as the base components. For example, the flow rule cache registers the OFP_PACKET_IN event to check if it has any flow rule whose matching information is the same with that of the PACKET_IN message in the OFP_ PACKET_IN event. If it has one, it attempts to redirect the original control flow with the OFP_FLOW_MOD event containing the cached rule.

IV. SDN-specific Event Handling Model

Current NOS designs largely employ an event-driven programming model, wherein the SDN control logic defines the actions and OpenFlow protocol outputs that occur in response to notification events from the data plane. Within the topic of SDN event handling, most of the focus of prior work has involved issues of scalability in the presence of high-frequency events [16]. In modern NOSs, control layer event handling is primarily viewed as a notification interface between the data plane and components that implement network control logic.

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 customization process. 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

In the Barista framework we seek to extend the SDN event handling model to incorporate inter-component communications and event-broker-derived meta events as two additional event classes that Barista authors can define. Inter-component events provide a departure from the use of implementation-dependent function call exchanges to implementation-agnostic strategy for allowing components to interoperate. Meta events enable a novel strategy for altering the accessibility and behavior of Barista components based on live event stream characteristics observed by the event broker. We illustrate the processing paths of these three event classes in Figure 6.

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

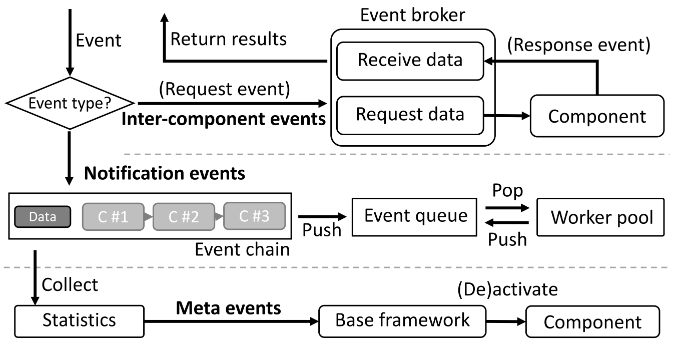


Fig. 6. Barista event processing logic

for receiving a subset of data plane events. When a registered event occurs the event handler forwards the event by invoking the callback function. Barista also provides a mechanism managing notification-based data-plane events. When a component is integrated into the framework, it registers event notification criteria, and the Barista event broker delivers these events via the component's notification interface.

Inter-component events: While today's NOSs provide dynamic and modular component designs through software services such as the use of OSGi [17] in ONOS, inter-component dependencies remain tightly coupled to component implementations. For these environments, events are used for message distribution, and direct function calls are used for component to component data exchanges. Indeed, even very modular components designs may require inter-component data sharing.

Let us define two terms: *contextual dependency* and *functional dependency*. A contextual dependency arises when one component *i* requires information that is produced from another component *j* in order for *i* to operate. A functional dependency arises when a component *i* must submit data to a component *j* (e.g. employing an API function call) in order for *j* to produce a result that is then consumed by *i*. Most of existing NOSs have functional dependencies among components. A highly modular system is one in which functional dependencies are minimized among the system components.

Barista replaces functional dependencies among components using the event broker, where inter-component communication occurs through the exchange of request and response events. A component simply triggers a request event to the event handler. Then, the event broker delivers the event to a target component while it holds the event of the original component. Once the target component triggers a reply event, the event broker replaces the request event to the reply event and releases the event of the original component. That is, the Barista event handling framework replaces functional dependencies between components with an inter-component event mechanism that removes component-specific implementation dependencies.

Meta events: Meta events are produced for live event statistics as observed by the event handler. Example event statistics include event volume, component level statistics regarding event consumption or production, and event type distribution statistics. Barista allows operators to define the thresholds under which trigger meta events, and to associate handling logic with produced meta events. For example, meta events may be configured to dynamically activate or deactivate components, or to filter certain events that are 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 offer the operator the ability to activate and deactivate components that are pre-deployed to address certain event production anomalies that may 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 arise. 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 may otherwise result in unwanted resource utilization. For example, if the volume of events, such as Packet-In events, rapidly increases, meta events may be configured to deploy DoS mitigation components while also selecting to suspend certain (*normal mode*) flow handling components.

B. Dynamic Component Event Chaining

An event chain arises when the event handler must service a group of components that are designed to consume a common event. Modern SDN controllers do not usually expose services to define chaining strategies among its components. Rather, such strategies must be defined within the code or through manual component priority configuration. The lack of event-chain support for components among NOSs may arise as they are often primarily concerned with non-interference. However, some components (e.g., flow rule conflict resolution, message verification, and integrity check) may require find-grained controls when defining event processing ordering. With this in mind, Barista facilitates explicit event chaining by default.

Unfortunately, there could be a conflict between components due to mutual event dependencies. Thus, to detect such conflict, we first define a dependency graph, a directed graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ denotes the set of components, and $E = \{e_1, e_2, \ldots, e_m\}$ represents the set of dependencies between components. We let $n = |V|$ and $m = |E|$ denote the number of components and dependencies respectively. Also, we define $P(v_i, v_j)$, a sequence (path) from v_i to v_j where $v_i, v_j \in V$. Then, when defining event processing ordering, Barista checks dependency conflicts between components based on the existence of $P(v_i, v_i)$ where $v_i \in V$. If $P(v_i, v_i)$ exists, there is a cycle from v_i, meaning that v_i has dependency conflicts with the components in $P(v_i, v_i)$. In this case, it notifies the conflicts and requires the conflict resolutions to an operator.

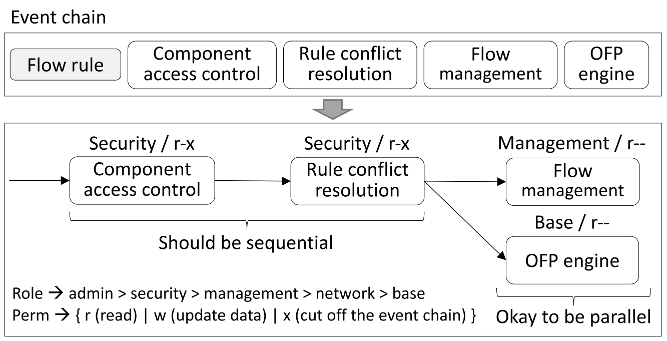


Fig. 7. Handling events in sequential and in parallel

Algorithm 1 Chaining components

```
# graph = dependencyGraph
# compList = the list of components that listen to an event
function CHAINCOMPONENTS(graph, compList)
    newList = []
    compList = sortByRole(compList)
    compList = sortByPermissionInSameRole(compList)
    for comp in compList do
        for otherComp in compList do
            if otherComp $\neq$ comp then
                if Path(graph, otherComp, comp) $\neq \emptyset$ then
                    if otherComp not in newList then
                        newList.append(otherComp)
        if comp not in newList then
            newList.append(comp)
        if comp.permission $\cap$ (W or X) $\neq \emptyset$ then
            comp.delivery = sequential
        else
            comp.delivery = parallel
    return newList
```

With respect to evaluating the event processing sequence, Barista has two ways to deliver events to components, as shown in Figure 7: *sequential delivery* and *parallel delivery*. Sequential delivery arises when a component is granted permission to potentially terminate the event-chaining sequences based on an internally-defined decision regarding the event, such as a filter criteria match. Parallel delivery is applicable when components consume events but do not impact the delivery of the event to other components.

Event sequence formulation begins by first ordering components based on their roles and authorities (i.e. permissions) as shown in Algorithm 1. Then, it checks if there is an event dependency among the given components. For each component, if there is a path from other components to the component in the dependency graph, it imposes sequencing such that the dependent components follow the event processing of the component. Once the order of the components is determined, it evaluates which component can be handled in parallel, or which component requires sequential delivery.

Algorithm 2 Handling event chains

```
function EVENTHANDLER
    while True do
        eventChain = popChain(eventQueue)
        if eventChain.reference_count > 0 then
            pushChain(eventQueue, eventChain)
        else if emptyChain(eventChain) then
            releaseChain(eventChain)
        else
            for component in eventChain do
                eventChain.reference_count += 1
                pushToWorker(eventChain, component)
                popComponent(eventChain, component)
                if component.delivery = sequential then
                    pushChain(eventQueue, eventChain)
                    break
function EVENTWORKER(eventChain, component)
    component.handler(eventChain.event)
    eventChain.reference_count -= 1
```

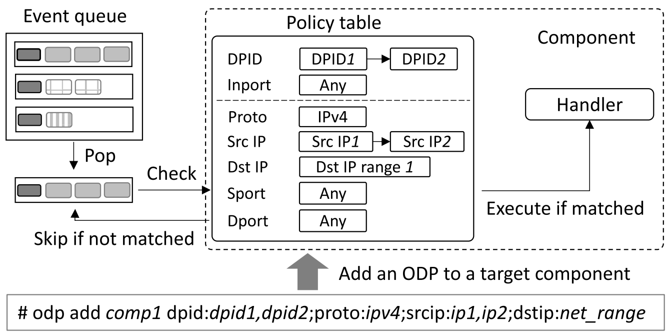
Finally, the adjusted component event chains are processed as shown in Algorithm 2.

This dynamic composition of event chaining enables a fine-grained control of component composition within the NOS, which is of particular relevance when integrating security and fault recovery components. While existing NOSs enhance network serviceability through various network components, they face some limitations when adding security features, due to their architectural design choices. For example, SE-Floodlight [9] required modification to the internal logic of Floodlight to embed a flow-rule conflict detection mechanism, as Floodlight provided no means to modularly incorporate this service as the *first* evaluation point for all notification events. In contrast, the Barista architecture is fundamentally designed to modularize not only components but also the event handling flows among components. It dramatically simplifies the ability for third-party research extensions to be integrated into the processing pipeline of notification and inter-component events.

C. Policy-based Event Distribution

Rather than intelligently utilizing event handling mechanisms, existing NOSs focus on adding more functions into their NOSs to satisfy ever increasing operating requirements. This approach is not scalable, and neither does its provide robustness to the overall NOS event handling services. In contrast, Barista takes a different approach that allows operators to not only select a set of components they wish to deploy, but also specify the event handling policy for the selected components.

To dynamically handle events, Barista provides operator-defined policies (ODPs). Each operator-defined policy is composed of 7 fields: datapath ID, in-port (incoming port of a switch), protocol, source/destination IP addresses and ports. While operators can define multiple values for each field, in the case of IP address fields, Barista currently supports matching a single IP address and the range of IP



```
# odp add comp1 dpid:dpid1,dpid2;proto:ipv4;srcip:ip1,ip2;dstip:net_range
```

Fig. 8. Policy-based event distribution

addresses based on a subnet mask. Once an operator defines an ODP, Barista updates the policy table in a target component as shown in Figure 8. Then, when the event handler identifies an incoming event, the event handler matches the event with the policy table of a current component before pushing it into the worker pool. If the event is matched with the component policy, it is pushed into the worker pool. Otherwise, the event handler skips it and deals with the next component in an event chain. Using ODPs, operators can provide different functionalities to different administrative networks. They can also be used to instantiate *preferred* functionalities to specific customers, as part of service level agreements (SLAs).

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 a 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 the distributed storage. At the same time, the cluster component keeps polling for new events from other instances and triggers those events to the event handler at its instance. The distributed storage maintains logical sequences to ensure the incoming events are ordered chronologically. Since the current cluster component uses a polling mechanism to get events from the distributed storage, it currently supports eventual consistency only. Supporting strong consistency is one of our future goals.

The information to be shared across Barista instances is dependent on what events the cluster component receives. For example, if an operator only wants to share the topology information, he simply needs to set switch and topology events at the cluster component. It is also possible that some events can be shared among specific instances rather than all instances. Thus, operators can strategically distribute events at each instance to others.

V. Predictive NOS Assessment

The predictive NOS assessment tool operates as an assistant to operators. It helps to estimate how much throughput they can achieve through the selected components and what kinds of functionalities the components will provide. It also identifies the component set that is best suited to satisfy the operating requirements specified by operators. This involves three major steps: component analysis, performance estimation, and component recommendation. Here, we describe each step in detail.

A. Component Analysis

First, the tool analyzes the functionality of each component and how the component affects the overall NOS system. The goal of this component analysis is to understand how differently each component influences on the NOS system. Since the overall throughput would vary based on hardware specifications and the system resource utilization would vary based on workloads, it conducts micro-evaluation of each component to develop a base knowledge.

The tool employs a self-evaluation mechanism that generates various virtual network environments and control messages. Operators can define the number of switches, the number of ports that each switch has, how many hosts (pairs of IP and MAC addresses) are connected to each switch and the amount of control messages per second. Using this mechanism, it automatically evaluates each component and generates a component assessment report using throughput, latency, CPU utilization, and memory utilization profiles for each component. Besides the influence of specific components on the NOS, it is also important to consider the impact of various event types on NOS performance, as performance can vary across different workloads. Hence, the prediction tool explicitly tracks the per-component overhead associated with each event type.

B. Performance Estimation

Next, the tool predicts the estimated latency and throughput when operators choose a set of desirable components with a specific workload. Current benchmark tools, (e.g., CBench [10]) are insufficient for our purpose as they are limited in the workload and set of event types that they generate. Thus, we develop an analytic benchmarking strategy, i.e., a performance model, with heterogeneous workloads, that captures the per-component impact of each event type.

Performance model: Our performance model is based on event dependencies. For example, as shown in Figure 4, OFP_PACKET_IN should be followed by OFP_MSG_IN, and OFP_MSG_OUT should be followed by OFP_FLOW_MOD. Likewise, according to event dependencies, we define a directed graph $G = (E, L)$, where $E = \{e_1, e_2, \ldots, e_n\}$ denotes the set of events, and $L = \{l_1, l_2, \ldots, l_k\}$ represents the dependencies between events. Here, we let $n = |E|$ and $k = |L|$ denote the number of events and links respectively. For an event $e_i \in E$, we denote p_i as a component list listening to e_i, and we use $expLatency(p_i)$ to get the approximated latency with given components in processing event e_i as shown in Algorithm 3. Here, the tool utilizes the results of the component analysis.

Latency estimation: To calculate the expected latency with selected components by operators, we first define $w_i = \{w_{i1}, w_{i2}, \ldots, w_{im}\}$ as the set of probability for each event type where $\sum_{j=1}^{n} w_{ij} = 1$, indicating how often each event is triggered from e_i. Since each event may have an dependency on another event, we need to calculate the conditional probability of each event. For this, we use $backwardWeight(e_i)$ (shown in Algorithm 4) to get the conditional probability of e_i. Here, we assume that there is no cycled dependency between

Algorithm 3 Get the expected latency of an event
```
1: function EXPLATENCY(componentList)
2:     latency = 0.0, max_parallel = 0.0
3:     for component in componentList do
4:         if component.delivery = sequential then
5:             if max_parallel $\neq$ 0.0 then
6:                 latency += max_parallel
7:                 max_parallel = 0.0
8:             latency += component.latency
9:             latency += sequential_overhead_by_eventHandler
10:        else
11:            if max_parallel $<$ component.latency then
12:                max_parallel = component.latency
13:            latency += parallel_overhead_by_eventHandler
14:    if max_parallel $\neq$ 0.0 then
15:        latency += max_parallel
16:    return latency
```

Algorithm 4 Get the conditional probability of an event
```
1: function BACKWARDWEIGHT($currEvent$)
2:     $prevWeight = 0.0$
3:     if $empty(prevEvents(currEvent))$ then
4:         $prevWeight = 1.0$
5:     else
6:         for $prevEvent$ in $prevEvents(currEvent)$ do
7:             $prevWeight$ += $backwardWeight(prevEvent)$
8:     return $prevWeight \times currEvent.weight$
```

events. Finally, the tool computes the expected latency with the given components as follows.

$$expLat = \sum_{i=1}^{n} backwardWeight(e_i) expLatency(p_i)$$

Throughput estimation: Unlike the latency estimation, we need to consider the parallelism of Barista architecture to predict the throughput with selected components, which is too complicated to consider all possible parameters. Thus, we use a heuristic method to get the expected throughput. Here, we define x as the number of workers, and Lat_{io} which is the packet receiving overhead of the packet I/O engine (i.e., the latency of OFP_MSG_IN). Also, we define T_1 which is the base throughput with a single worker.

As a large number of incoming messages arrive within a short time, the packet I/O engine will receive a batch of the messages from a packet buffer at once. In addition, since multiple workers process events in parallel, the overall latency will be reduced as well. Thus, we assume that the overall latency with burst traffic would become approximately $\frac{expLat - \frac{Lat_{io}}{2}}{2}$. In the same context, we also define $expLat \times T_1$ as the parallelized ratio based on the correlation between $expLat$ and T_1. Finally, the tool estimates the expected throughput as follows.

$$T_x = (expLat \times T_1)(\frac{expLat - \frac{Lat_{io}}{2}}{2}) ln(t) + T_1$$

TABLE II
REQUIREMENTS SUMMARY

#	Requirement	Component
1	Large number of switches and hosts	CL
2	Multiple networks with geographical distances	CL
3	Static routing and forwarding	SFE, FRC
4	Remote access to the NOS UI	UA
5	Adoption of 3rd-party applications	AA, RA
6	Adoption of 3rd-party NOS components	CAC, CFI, IMI
7	Privileged control of applications	RA
8	Authorization for access to internal storage	RA, CAC
9	Protection of internal storage integrity from unauthorized modification	CAC, IMI
10	Protection of integrity from malformed msgs	OMV
11	Protection of availability from app failures	FM, AI
12	Protection of availability from SBI failures	FM, SBI
13	Protection of availability from excess traffic	CTM, RM
14	Failure recovery of NOS core services	FM, RM
15	Policy conflict resolution	FCR

C. Component Recommendation

As the final step, this tool recommends a component set that is optimal to satisfy the operating requirements. Table II presents a list of requirements that operators can choose. Each requirement contains one or multiple components to support specific functions and each component may be associated with multiple requirements. With given requirements, we employ multiple-criteria decision analysis (MCDA) [18], which is commonly used to make a decision among multiple criteria, and linear programming techniques to discover an optimal set of components.

Here, we first define $R = \{r_1, r_2, \dots, r_n\}$ as the set of requirements and $C = \{c_1, c_2, \dots, c_m\}$ as the set of components. $n = |R|$ is the number of requirements, and $m = |C|$ is the number of components. Then, based on the selected requirements, we build a decision matrix, D where $d_{ij} \in \{0.001, 1\}$, $d_{ij} = 1$ if an operator selects requirement j and requirement j contains component i. To calculate the overall preference, we build a matrix, D', which normalizes the weight of a requirement across all components.

$$D' = \begin{pmatrix} r_{11} & \dots & r_{1n} \\ \dots & r_{ij} & \dots \\ r_{m1} & \dots & r_{mn} \end{pmatrix} \text{ where } r_{ij} = \frac{d_{ij}}{\sum_{k=1}^{m} d_{kj}}$$

Finally, we extract the relative weight of component i as follows.

$$w_i = \frac{\sum_{j=1}^{n} r_{ij}}{\sum_{k=1}^{m} \sum_{j=1}^{n} r_{kj}}$$

Now, to discover a component set that satisfies the given requirements while maximizing the resource utilization of the given system, we produce a model of the component selection problem. To achieve the optimal set, we define the objective function Z, and $x_i = 1$ if c_i is explicitly selected.

$$Z = \sum_{i=1}^{m} w_i x_i, \text{ where } x_i \in \{0, 1\}$$

Then, we add constraints for performance and resource utilization. Each constraint contributes to the evaluation of the best-effort component set.

$$T_{min} \leq T_x \text{ (default, 8 workers)}$$
$$\sum_{i=1}^{m} c_i x_i \leq cpulimit \text{ (default, 100\%)}$$
$$\sum_{i=1}^{m} m_i x_i \leq memlimit \text{ (default, physical mem size)}$$

We then use the PuLP constraint solver [19] to automatically discover an optimal component set for operators. Unfortunately, it is possible that the system cannot find any component set that satisfies the requirements. For example, the required throughput could be higher than that of the base framework. In such instances, the operator is notified and prompted to readjust the requirements until a satisfying component set may be found.

VI. System Implementation

A prototype of Barista has been implemented to evaluate the efficiency and effectiveness of its design. This implementation includes the base framework [20], a broad set of components and the predictive NOS assessment tool. The Barista prototype consists of over 19K lines of mostly C code and supporting Python scripts, and the source code of the base framework, pre-developed components, and the NOS assessment tool is available at https://github.com/sdx4u/barista.

The base framework maintains a component list, containing operator-defined configurations (JSON format). The event handler restricts the type of data per event to avoid type-unsafe boilerplate code. For handling events, we use worker pools, and events are processed in each worker. For event distribution across Barista instances, we use MariaDB and Galera Cluster [21] and make transactions in batch.

The predictive NOS assessment tool is implemented with NumPy [22] and PuLP [19]. To conduct component analysis, we developed the self-evaluation mechanism that automatically generates component configurations and executes the Barista prototype with those files. To generate control traffic, we modified Cbench [10] and integrated it with the mechanism. The modified CBench produces more diverse control messages with specific input parameters such as IP/MAC address range, the number of switch ports, and the number of messages per second.

VII. System Evaluation

We describe experimental results that validate the efficiency and effectiveness of the Barista prototype system. First, we present per-component microbenchmark results and then present the assessment results. Finally, we describe three scenarios illustrating the flexibility and usability of the system.

A. Test environment

Our experimental testbed comprised of six machines. Three machines ran Barista instances, each with Intel E5-2620 CPU (12 cores, 2.40 GHz) processors and 32 GB of RAM. Three other machines, each with an Intel i5 CPU (4 cores, 3.50GHz) and 16 GB of RAM were used for control message generation.

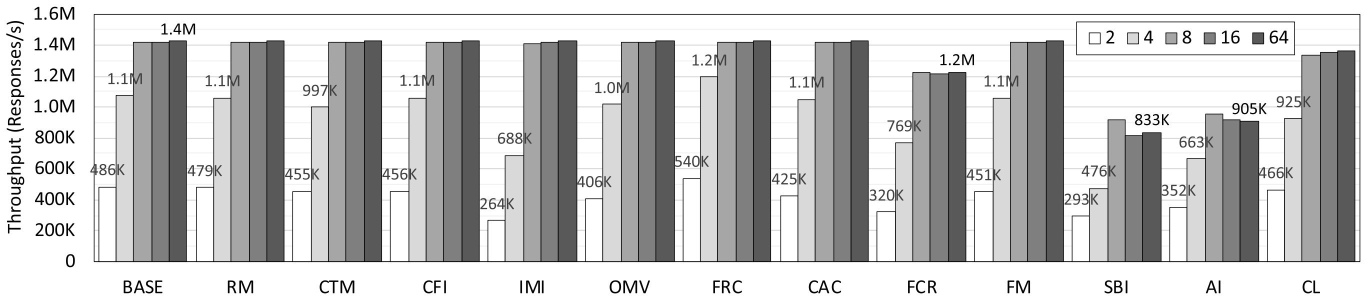


Fig. 9. Per-component throughput variations with different number of switches. Note) BASE: base components, RM: resource management, CTM: control traffic management, CFI: control flow integrity, IMI: internal message integrity, OMV: OpenFlow message verification, FRC: flow rule cache, CAC: component access control, FCR: flow-rule conflict resolution, FM: failure management, SBI: southbound-core isolation, AI: application-core isolation, CL: cluster

TABLE III
PER-COMPONENT CPU UTILIZATION

Component	Avg.	Max.	Component	Avg.	Max.
BASE	21.89	25.23	RM	22.81	25.85
CTM	21.23	23.05	CFI	25.60	28.76
IMI	27.98	34.15	OMV	22.50	25.94
FRC	20.83	24.30	CAC	22.70	26.51
FCR	26.67	30.60	FM	22.98	25.90
SBI	58.55	97.38	AI	68.41	80.46
CL	26.51	33.56	-	-	-

The topology involved 1,000 virtual hosts connected across 48 ports per switch. Hence, our self-evaluation mechanism generated 1,000 independent pairs of IP and MAC addresses for each switch.

B. Microbenchmarks

To understand how each component affects a NOS, we measure the throughput, CPU use, and memory use of a set of extensible components along with the base components. Figure 9 illustrates the throughputs for each component with different number of switches while we set the number of workers as 8. We find that the throughput of most components is comparable to that of the base components (from 462K to 1428K responses/s on average). The maximum throughput is saturated at 1.4M responses/s due to NIC bandwidth limitations (1 Gbps). When the number of switches is lower than the number of workers, the throughput suffers because of the imbalance of workers.

While most components have minimal impact on the overall throughput, there are 5 components whose throughputs are noticeably lower than that of the base components. In the case of Internal Message Integrity (IMI) component, the inspection overhead of all internal messages causes the overall performance degradation. Similarly, the conflict checks against all outgoing flow rules decrease the throughput of Flow-rule Conflict Resolution (FCR) component. In terms of external communications, the throughputs of Application (AI) and Southbound Isolation (SBI) components significantly dropped due to message passing overheads, causing the higher CPU uses (over 80%) than other components (23.9% on average) as shown in Table III. Cluster (CL) component also suffers performance degradations due to the DB query overheads.

The memory utilization for most components is static, with the notable exception of some components (e.g., host and switch management) that can dynamically allocate memory for new hosts and flow rules.

C. Dynamic Component Activation

One of Barista's contributions is dynamic component composition in runtime. To show this flexibility, as shown in Figure 10, we measure the throughput variations before and after some changes in component composition occur while feeding low volumes of traffic into the Barista NOS. When Barista runs with base components, it handles around 158.6 flows/s on average. After 5 seconds, we activate the internal message integrity, causing a noticeable performance degradation (95.6 flows/s). To get higher performance again, we remove the internal message integrity (at time 17), and then the number of flows gets back to the previous number. Likewise, Barista does not require any interruption in order to make changes in component composition. When an operator activates a new component, Barista first configures the component while it does not feed any events to it. Then, once the component is ready to handle events, it starts to feed events. When a component is deactivated, Barista first stops feeding events while it waits for the component to process pre-assigned events, and then detaches it. As a result, Barista can achieve dynamic component composition in runtime.

D. Performance Estimation

Next, we evaluate the accuracy of the performance estimation tool by comparing the difference between the actual throughput and estimated one. For this, we use three different machines: Xeon E5-2620 (2.40GHz, 24-cores), Xeon E5-1650 (3.50GHz, 12-cores), and i5-6600K (3.50GHz, 4-cores).

Figure 11 presents the actual and expected throughputs using our performance model. When the number of workers exceeds the number of cores, the actual throughput becomes saturated; thus, the expected throughput is no longer meaningful. Our measurement shows that the overall accuracy is 94.43% (from 87.3% to 97.6%) while the margins of those throughputs get larger as the number of workers increase. This

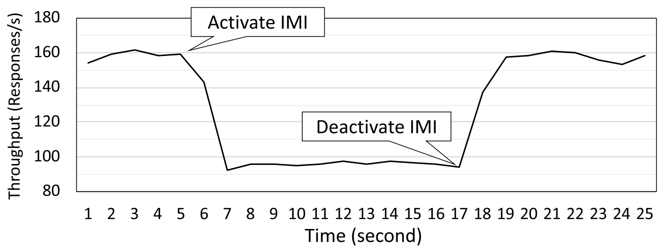


Fig. 10. Throughput variations during component composition

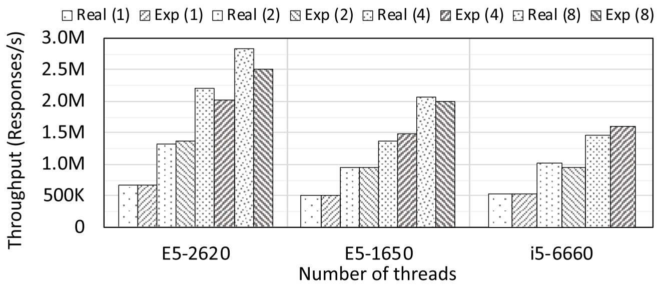


Fig. 11. Comparison of actual and estimated throughputs

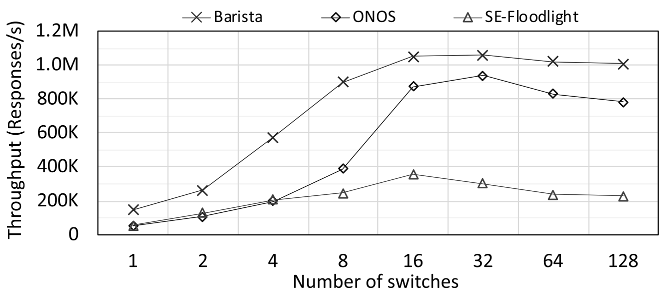


Fig. 12. Throughput comparison of the distributed and secure Barista NOS with representative NOSs for each feature

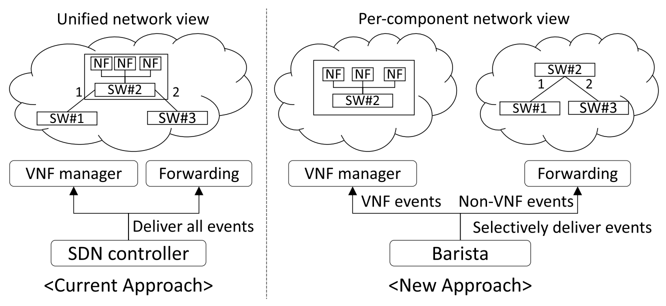


Fig. 13. Selective network management with ODPs

performance model does not incorporate hidden overheads such as context switching and network delays, leading to some of the variability in results between expected and observed throughput. However, the performance model can provide a basis for comparing the impact on system performance from various components.

E. Use Cases

In this section, we present the effectiveness of the Barista through three use cases: a distributed and secure NOS, separated network management using operator-defined policies, and a lightweight NOS for IoT environments.

1) Distributed and Secure NOS: Here, we describe how operators can build their own NOSs using the Barista. We assume that an operator wants to build a distributed but secure NOS for his network by selecting requirements 1, 5, 6, 10, and 15 in Table II. In addition, the operator sets the target CPU and memory utilization to be 60% and 1 GB respectively, and 1M responses/s as the minimum throughput with 8 workers. Since it does not select the flow rule conflict resolution, which is one of the key components in SE-Floodlight, during the recommendation phase we explicitly choose it to compare ours with SE-Floodlight. By inserting them into the assessment tool, it finally discovers the optimal component set, {Base, CTM, CL, SFE, UA, AA, RA, CAC, FCR, FM}. Figure 12 illustrates the throughput of the 3-node Barista cluster, 3-node ONOS cluster and SE-Floodlight with varying number of switches. Each ONOS instance saturates at 940K responses/s on average and SE-Floodlight saturates at 357K responses/s. In contrast, Barista instance has a maximum throughput of 1,059K responses/s.

2) 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 13, 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").

3) 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 [6], [7] are designed for server-side deployments and unsuitable to be run on small devices with limited resources. However, we found that a few controllers [8], [12], [23] to be readily executable on such devices (specifically the ODROID XU4 [24]).

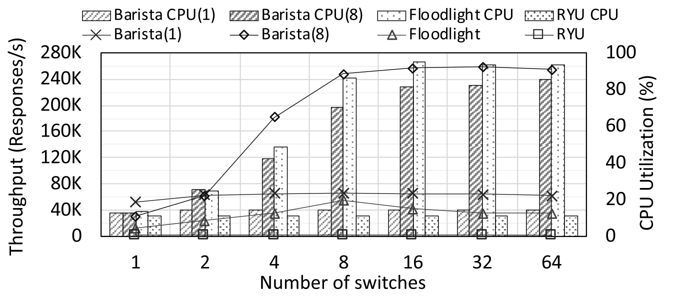


Fig. 14. Throughput comparison of the lightweight Barista NOS with existing NOSs on a single-board device

To demonstrate the flexiblity of Barista, we compare it with Ryu [12] and Floodlight [8]. For this, we use ODROID XU4 (ARM Cortex-A7, Octa-core, 2 GB of RAM) and a set of base components for the Barista cases. Figure 14 illustrates the throughput of each NOS (line graph in Figure 14). Ryu runs with a single thread; thus, its throughput does not scale up. The throughput is 2,058.4 responses/s on average. On the other hand, Floodlight runs with multiple threads; thus, throughput goes up to 55,005.0 responses/s with a 93.8% CPU utilization. In contrast, Barista shows much higher throughput. With a single worker, Barista performs up to 63,523.5 responses/s while consuming up to 14.5% of CPU utilization. With 8 workers, Barista performs up to 254,071.0 responses/s which is 117 and 4.7 times higher than Ryu and Floodlight respectively. With this measurement, we show that the Barista NOS is suitable for small computing devices such as IoT gateways.

VIII. Related Work

NOX [25] emerged as the first network operating system for SDNs and has since been ported to Python - POX [23]. Both Floodlight [8] and Beacon [2] 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 represents the first effort to address scalability by developing a distributed NOS platform [1]. ONOS [6], Hyperflow [26], Kandoo [27], and OpenDaylight [7] 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 [9], FortNOX [15], Rosemary [3], and LegoSDN [28] demonstrate the integration of multi-network-application security features into the SDN control layer. These projects focus on application consistency and robust application management to enhance NOS reliability when errant application-layer problems arise. They focus on addressing the needs of sensitive computing environments with less regard to the scalability and performance issues that are presented in other production environments. Some NOSs have adopted component-based architectures, such as ONOS [6], OpenDaylight [7], and Ryu [12]. However, while one can integrate new features into them, the method of attaching new components is quite complicated (e.g., ONOS [6] and OpenDaylight [7]). Ryu [12] provides a basic set of components (a subset of Barista components) and does not consider the issue of how to optimally select components based on operator-defined requirements.

Other approaches to flexibly build up SDN environments include FlowVisor [29], [30], CoVisor [31], NetIDE [32], Frentic [33], and Pyretic [34]. In terms of multi-controller environments, FlowVisor divides a SDN network into slices and enforces strict isolation between controllers running above it, while managing provisioning and shared resources. Similarly, CoVisor allows multiple controllers to cooperate on managing the same shared traffic while assembling network policies from these controllers into single policies for a physical network. In terms of dynamic composition, NetIDE focuses on application composition across multiple controllers by providing a composition semantics with filters and merge policies. On the other hand, Pyretic provides a higher-level runtime that resides "above" a single 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.

IX. Conclusion

The NOS is an integral piece of the SDN, and thus selecting the right controller is a crucial step in orchestrating networks for optimal security, performance and availability. We observe that contemporary NOS solutions tend to be limited to specific purposes making it challenging for operators to satisfy competing demands of disparate subnets using a homogeneous platform. Barista takes an important step toward addressing this problem by providing an event-driven NOS synthesis framework that simplifies integration of composable modules with the enhanced event handling mechanism and a predictive NOS assessment tool that intelligently selects an appropriate set of NOS modules. We evaluate the system against a range of commodity NOSs, finding that Barista simplifies instantiation of custom controllers with diverse feature combinations and efficiently replicates functionality commonly found in major controllers while delivering competitive performance.

References

[1] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proc. Symp. Operating Syst. Design Implement. (OSDI)*, 2010, pp. 1–14.
[2] D. Erickson, "The beacon openflow controller," in *Proc. ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2013, pp. 1–6.
[3] S. Shin *et al.*, "Rosemary: A robust, secure, and high-performance network operating system," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2014, pp. 78–89.
[4] HP. *HP VAN Controller*. [Online]. Available: https://support.hpe.com/hpsc/doc/public/display?docLocale=en_US&docId=emr_na-c03967699
[5] BigSwitch. *BigSwitch Network Controller*. Accessed: Mar. 1, 2019. [Online]. Available: http://www.bigswitch.com/sites/default/files/sdnresources/bncdatasheet.pdf
[6] OnLab. *Open Network OS*. [Online]. Available: https://onosproject.org

[7] OpenDayLight. *Open Source Network Controller*. Accessed: Mar. 1, 2019. [Online]. Available: http://www.opendaylight.org

[8] FloodLight. *Open SDN Controller*. Accessed: Mar. 1, 2019. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[9] P. A. Porras *et al.*, "Securing the software-defined network control layer," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 1–15.

[10] R. Sherwood and Y. Kok-Kiong, (2010). *Cbench: An Open-Flow Controller Benchmarker*. [Online]. Available: http://www.openflow.org/wk/index.php/Oflops

[11] N. Katta, H. Zhang, M. Freedman, and J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, p. 4.

[12] RYU. *RYU SDN Controller*. Accessed: Mar. 1, 2019. [Online]. Available:https://osrg.github.io/ryu

[13] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2015, pp. 1–16.

[14] E. Kohler *et al.*, "The Click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, 2000.

[15] P. Porras *et al.*, "A security enforcement kernel for OpenFlow networks," in *Proc. Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 121–126.

[16] S. H. Yeganeh and Y. Ganjali, "Turning the tortoise to the hare: An alternative perspective on event handling in SDN," in *Proc. ACM Int. Workshop Softw.-Defined Ecosyst.*, 2014, pp. 29–32.

[17] OSGi. *The Dynamic Module System for Java*. Accessed: Mar. 1, 2019. [Online]. Available: https://www.osgi.org/

[18] S. Greco, J. R. Figueira, and M. Ehrgott, *Multiple Criteria Decision Analysis*. Springer Science & Business Media, 2005.

[19] S. Mitchell, M. OSullivan, and I. Dunning. (2011). PuLP: A Linear Programming Toolkit for Python. The University of Auckland. Auckland, New Zealand. [Online]. Available: http://www.optimization-online.org/DB FILE/2011/09/3178.pdf

[20] J. Nam *et al.*, "Barista: An event-centric nos composition framework for software-defined networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2018, pp. 980–988.

[21] *MariaDB Galera Cluster*. Accessed: Mar. 1, 2019. [Online]. Available: https://mariadb.com/
kb/en/mariadb/galera-cluster/

[22] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, no. 2, p. 22, 2011.

[23] POX. *Python Network Controller*. Accessed: Mar. 1, 2019. [Online]. Available: https://noxrepo.github.io/pox-doc/html/

[24] Hardkernel. *ODROID XU4 Board*. Accessed: Mar. 1, 2019. [Online]. Available: https://wiki.odroid.com/odroid-xu4/odroid-xu4

[25] N. Gude *et al.*, "NOX: Towards an operating system for networks," in *Proc. ACM SIGCOMM Comput. Commun. Rev.*, 2008, pp. 1–6.

[26] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proc. Internet Netw. Manage. Conf. Res. Enterprise Netw.*, 2010, pp. 1–6.

[27] S. H. Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2012, pp. 19–24.

[28] B. Chandrasekaran and T. Benson, "Tolerating SDN application failures with LegoSDN," in *Proc. ACM Workshop Hot Topics Netw. (HotNet)*, 2014, p. 22.

[29] R. Sherwood *et al.*, "Can the production network be the testbed?" in *Proc. USENIX Conf. Operating Syst. Design Implement. (NSDI)*, 2010, pp. 1–14.

[30] A. Al-Shabibi *et al.*, "OpenVirteX: Make your virtual SDNs programmable," in *Proc. Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 25–30.

[31] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *Proc. 12th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2015, pp. 87–101.

[32] R. Doriguzzi-Corin *et al.*, "Reusability of software-defined networking applications: A runtime, multi-controller approach," in *Proc. 12th Int. Conf. Netw. Service Manage. (CNSM)*, 2016, pp. 209–215.

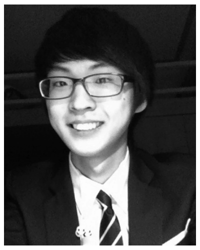
[33] N. Foster *et al.*, "Frenetic: A high-level language for OpenFlow networks," in *Proc. Workshop Program. Routers for Extensible Services Tomorrow*, 2010.

[34] C. Monsanto *et al.*, "Composing Software-defined Networks," in *Proc. USENIX Conf. Networked Syst. Design Implement. (NSDI)*, 2013, P. 6.

[35] NEC. *NEC Programmable Controller*. [Online]. Available: http://www.nec.com/en/global/prod/pflow/controller.html

Jaehyun Nam received the B.S. degree in computer science and engineering from Sogang University, South Korea, and the M.S. degree in information security from the Korea Advanced Institute of Science and Technology, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Security. His research interests focus on networked and distributed computing systems. He is especially interested in performance and security issues from software-defined networking (SDN) and network function virtualization (NFV).

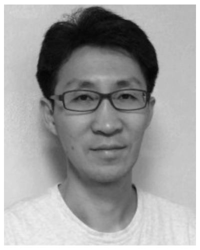
Hyeonseong Jo received the B.S. degree in information computer engineering from Ajou University, South Korea, and the M.S. degree in information security from the Korea Advanced Institute of Science and Technology, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Security. His research interests include finding security issue of software defined network.



Yeonkeun Kim received the B.S. degree in computer science engineering from the Ulsan National Institute of Science and Technology (UNIST), South Korea, and the M.S. degree in information security from the Korea Advanced Institute of Science and Technology, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Security. His research interests include network security issues of IoT and embedding systems.

Vinod Yegneswaran received the A.B. degree in computer science from the University of California at Berkeley, Berkeley, CA, USA, in 2000, and the Ph.D. degree in computer science from the University of Wisconsin–Madison, Madison, WI, USA, in 2006. He is currently a Senior Computer Scientist with SRI International, Menlo Park, CA, USA, pursuing advanced research in network and systems security. His current research interests include SDN security, malware analysis and anti-censorship technologies. He has served on several NSF panels and program committees of security and networking conferences, including the IEEE Security and Privacy Symposium.

Phillip Porras received the M.S. degree in computer science from the University of California at Santa Barbara, Santa Barbara, CA, USA, in 1992. He is currently an SRI Fellow and the Program Director of the Internet Security Group, SRI's Computer Science Laboratory, SRI International, Menlo Park, CA, USA. He has participated on numerous program committees and editorial boards, and participates on multiple commercial company technical advisory boards. He continues to publish and conduct technology development on numerous topics, including intrusion detection and alarm correlation, privacy, malware analytics, active and software defined networks, and wireless security.



Seungwon Shin received the B.S. and M.S. degrees in electrical and computer engineering from the Korea Advanced Institute of Science and Technology (KAIST), and the Ph.D. degree in computer engineering from the Electrical and Computer Engineering Department, Texas A&M University. He has spent nine years at industry, where he devised several mission critical networking systems. He is currently an Associate Professor with the School of Electrical Engineering, KAIST. His research interests span the areas of SDN security, IoT security, and Botnet analysis/detection.