applied sciences

MDPI

*Article*

# An Enhanced Message Distribution Mechanism for Northbound Interfaces in the SDN Environment

**Chenhui Wang [1,2], Hong Ni [1,2] and Lei Liu [1,2,*]**

1 National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences No. 21, North Fourth Ring Road, Haidian District, Beijing 100190, China; wangch@dsp.ac.cn (C.W.); nih@dsp.ac.cn (H.N.)
2 School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences No. 19(A), Yuquan Road, Shijingshan District, Beijing 100049, China
* Correspondence: liul@dsp.ac.cn; Tel.: +86-0108-254-7633

**Abstract:** Software-Defined Network (SDN), which is recommended as a new generation of the network, a substitute for TCP/IP network, has the characteristics of separation of data plane and control plane. Although the separation of the control plane brings a high degree of freedom and simple operation and maintenance, it also increases the cost of north–south communication. There are many additional modules for SDN to modify and enhance the basic functions of SDN. This paper proposes a message queue-based northbound communication mechanism, which pre-categorizes messages from the data plane and accurately pushes them to the apps potentially interested. This mechanism improves the efficiency of northbound communication and apps' execution. Furthermore, it supports both OpenFlow and the protocol-independent southbound interface, and it has strong compatibility. Experiments have proved that this mechanism can reduce the control-response latency by up to 41% when compared with the normal controller northbound communication system, and it also improves the network situation of the data plane, such as real-time bandwidth.

**Keywords:** SDN; northbound interface; message distribution mechanism; message queue

check for updates

**Citation:** Wang, C.; Ni, H.; Liu, L. An Enhanced Message Distribution Mechanism for Northbound Interfaces in the SDN Environment. *Appl. Sci.* **2021**, *11*, 4346. https://doi.org/10.3390/app11104346

Academic Editor: Carla Raffaelli

Received: 19 April 2021
Accepted: 7 May 2021
Published: 11 May 2021

## 1. Introduction

Today's Internet services are becoming increasingly complex, and all kinds of apps are growing in size, which brings a great challenge to the physical networks. Different functional apps work independently while maintaining a certain degree of coupling, such as traffic engineering, failure detection and recovery, and security component. All of these make the network inefficient and difficult to manage for ISP (internet service provider). For this situation, Software-Defined Network (SDN) [1] came into being and caught people's attention after a short time. SDN is an advanced network paradigm and its two most important features are decoupling of control and forwarding and using a logically centralized controller to collect global information and formulate strategies [2,3]. The former can be realized by means of a generic interface between the switches and controller, which is also called southbound interface. OpenFlow is the most commonly used one [1,4]. Furthermore, the decoupling is the key to the desired flexibility and making the network easier to introduce and add new abstractions and functions [5]; meanwhile, it simplifies network maintenance and management. The latter emphasizes that the controller is logical concentration rather than physical concentration. In other words, it is not wise to claim physical concentration for such a huge network size because of the need to guarantee network's performance and scalability [6].

In the SDN environment, all kinds of abstractions are realized through apps and the northbound interface (Hereinafter referred to as NBI) that is the communication channel of apps and the controller. The controller pushes the message to the apps, which is collected from the network and southbound interface. After that, the apps develop a strategy and inform the forwarding plane via the controller. Different apps need different information to complete the strategy and, with the increasing need of users, a number of apps have to run at the same time, resulting in a large number of communication costs [7]. Unlike the southbound, which already has a widely accepted proposal (OpenFlow, POF [8], and so on), there is no clear NBI standard [9]. Although many useful NBI exist in academia, such as ad-hoc APIs and RESTful APIs [10], their capabilities and usage can be much different because of the controller's features and apps demands. Many researches [2,5,9,10] argue that it may be a little bit early to define a standard proposal for NBI, as the apps are still being working out.

However, before there is a general standard or rule, it is necessary to develop a highly efficient and universal communication mechanism to ensure the performance of the controller. Some state-of-the-art controllers are not the same during communication with the apps, such as centralized controllers represented by NOX [11], Beacon [12] and Ryu [13]. Most of these controllers use an event mechanism based on OpenFlow messages to notify apps, for example, Ryu's app_manager (A major component of the Ryu program) will notify the registered apps one by one after parsing the received OpenFlow messages. This mechanism works well when the controller is running a few apps, but, as the number of apps increases, the performance of the controller drops significantly. Both the excellent performance Beacon coded by Java and the convenient POX (NOX's python version) coded by Python have the same problem, as well as the distributed controller ONOS [14].

In response to this problem, we modify the northbound communication of several exiting controllers, further distribute the events based not only on OpenFlow message types, but also on apps' interest, and push the events to each app using the messages of interest instead of all messages, which greatly improve multi-apps controller's performance on responsiveness. The main contributions are as follows:

- we abstract the message processing of the NBI into a mathematical model, provide an analysis in a qualitative way, and compare the performance changes of several controllers in the case of multi-app running (Section 3);
- ee propose an enhanced message distribution mechanism, named EMD, which adds the message queue to the NBI as an enhancement function of SDN to improve the performance of northbound communication (Section 4.1);
- we propose a protocol-independent method for message feature extraction and classification to enhance the compatibility of apps, and additionally add the network invariants checker for high availability and robustness (Sections 4.2 and 4.3); and,
- we evaluate the model from three aspects: north–south communication efficiency (Section 5.1), data plane performance (Section 5.2), and compatibility (Section 5.3) through experiments, and point out the app's scope of the model (Section 5.4).

## 2. Related Works

As a result of the lack of clear standards, the existing optimizations for controllers and apps mainly focus on stability, safety, and scalability. Zhou et al. [15] indicate that the global network-wide optimizations are designed to address deficiencies in the SDN ecosystem and provide general optimizations for SDN apps. They call these optimizations SDN Enhancements that are totally different from SDN apps, being roughly classified as conflict-resolver, TCAM-optimizer, invariant checker, and fault-tolerant provider. We focus on SDN Enhancement for controllers and apps in this paper, so others for SDN switches and southbound interface are not in our consideration.

Part of SDN Enhancements aims to resolve the conflict between apps, such as resource allocation and strategy conflict. Ferguson et al. [16] propose the concept of participatory networking referred to as PANE, in which the network provides a configuration API to apps and its users. In terms of conflict resolution, PANE represents a policy tree using Hierarchical Flow Tables [17] and three types of conflict-resolution operators to merge the conflict policy and generate the final action. Statesman [18] uses last-writer-wins and priority-based locking for resolving the conflict of each app's strategy meanwhile rejecting partial conflicting variables that are uncontrollable in the target state. GitFlow [19] runs a repository server to maintain the authoritative copy of the flow configuration state and track additional metadata for each evolving snapshot of the flow state, which resolves conflict while updating incrementally.

In the face of uncertainty of multiple apps running, some SDN Enhancements need to ensure the minimum stability and security of program operation. Statesman [18] proposes two network invariants—connectivity and network capacity—and the checker in the statesman system will check whether the target state meets the requirements of invariants before updating the state to the data plane. VeriFlow [20] generates a forwarding graph for each equivalence class, which is much similar to the definition of flow and checks the forwarding graphs' reachability, no-loop, and so on. In order to improve failure tolerance, LegoSDN [21] prototype proposes that it is better to sacrifice the availability of the flow that is dependent on a single switch than to sacrifice the connectivity of other flows dependent on the entire network. When it is unacceptable for some invariant, LegoSDN will use a host of policy checker to ensure that the network maintains a set of "NO-Compromise" invariants, just like VeriFlow [20] or the module in [22].

In addition to the above SDN Enhancements for conflict resolution and network invariants, FatTire [23] designs a new language for writing fault-tolerant SDN programs that provide paths as basic programming construct and applies to switches that support OpenFlow to achieve rapid fault recovery. Many existing SDN enhancements sacrifice the performance of SDN apps in order to ensure versatility. It is pointed out by [15] that this effect may reduce app performance by as much as 28%. At the same time, many enhancements do not consider the impact of the number of apps on functions and performance, which further reduces the performance during a large-scale deployment.

## 3. Motivation

### *3.1. Overview of Controller's NBI*

Most of the existing controllers use event processing mechanisms that are based on OpenFlow messages. In conjunction with Figure 1, after the controller receives the OpenFlow message from the southbound data plane, it triggers the corresponding event handler based on the message type after completing the preliminary analysis. At this time, all of the apps that have completed the registration will receive this message, and then the app can choose to either process the message or ignore it according to its own needs.

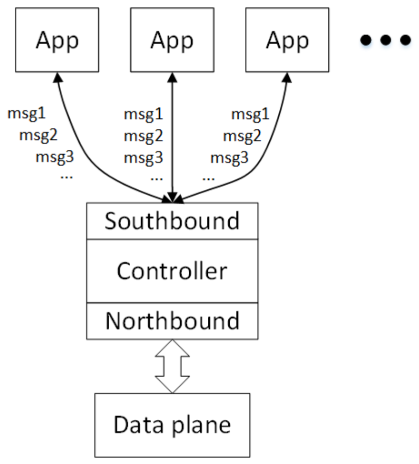


**Figure 1.** The Common SDN Architecture.

It can be seen that messages are pushed to all apps indiscriminately. When the number of apps is large, frequent triggers, and callbacks will seriously affect the overall performance of the controller.

### *3.2. Problem Description*

A complete packet in-out (sometimes using flow-mod) process starts from sending a message by the switch. We focus on northbound communication, so we divide it into the following five stages:

1. the southbound interface of the controller receives the data packet; and the core module completes the preliminary analysis;
2. push to all apps that have registered the message type;
3. apps complete the logic processing;
4. encapsulate the strategy and send it to the controller core module; and,
5. the controller calls the southbound interface to issue a policy message.

We abstract the packet in-out process into a mathematical problem for easy analysis. Firstly, items 1 and 5 depend on the performance of the controller. The values of these two items will not change if the controller has been selected, the hardware equipment is the same, and the data plane environment is always the same. We define it as a constant determined by objective factors, as:

$$T_{rcv} = C_1(c, net) \tag{1}$$

$$T_{snd} = C_2(c, net) \tag{2}$$

Secondly, item 2 is related to the processing capacity and queue length of each app. Without considering other time consuming, the time that it takes is:

$$T_{rq} = \frac{l_r}{e} \tag{3}$$

where $l_r$ is the length of the receive queue and $e$ is the efficiency of app processing.

Subsequently, item 4 is similar to item 2, but, if multiple apps share a single-process core module of the controller, the sending queue will be shared, which is:

$$T_{sq} = \frac{\sum_i l_i}{E} \tag{4}$$

where $l_i$ is the length of apps' send queue and $E$ is the average processing efficiency of core modules.

Finally, the analysis of item 3 is divided into two parts. One part is the time that is consumed by the callback function and the trigger event. We assume that these time consumings of all apps are the same, which is defined as $t_{complex}$; the other part is the processing consumption of each app. When the strategies of all apps are not conflicting and effective, the processing time should be calculated until the slowest app completes the strategy. Accordingly, the total processing time is:

$$T_p = \max_n \tau_i + n * t_{complex} \tag{5}$$

where $\tau_i$ is the processing time of app $i$ and $n$ is the number of valid apps.

In summary, the packet in-out time is:

$$T = T_{rcv} + T_{rq} + T_p + T_{sq} + T_{snd} \tag{6}$$

Additionally, the optimization goal of northbound communication is:

$$\min T \tag{7}$$

The next part of the article will analyze Equation (7) in a qualitative way.

### *3.3. Impact between Apps*

When the SDN controller has a large number of registered apps, although the time complexity of a single app is low enough, the overall processing latency will increase due to various reasons, such as thread switching or event triggering. We now present empirical data to quantify the impact of Apps' quantity on SDN controllers. The environment of the simulation experiment is two virtual machines that are running on a desktop computer with a 3.6 Ghz quad core and 16 GB of memory, one of which runs the controller and the other runs cbench. We choose representative Ryu and ONOS as test controllers and study three Apps for contrast.

1. simple_switch: an app for simple layer 2 switch, which aims to route the packet. This app is preset in the Ryu project;
2. Hedera: the traffic-enginnering SDN app. We rewrite this app using Python for adapting Ryu controller;
3. Reactive Forwarding: A native ONOS app for routing just like simple_switch; and,
4. redundancy: a blank program, which ends after executing 10 times plus-one calculations.

In Figure 2, *simple* and *Hedera*, respectively, represent the simple_switch app an-dlHedera app running by Ryu controller, and only one app is running at one time; *RF* represents the Reactive Forwarding app running alone by ONOS; and, *sth.* + *red* means that the given app and the redundancy app are running at the same time.

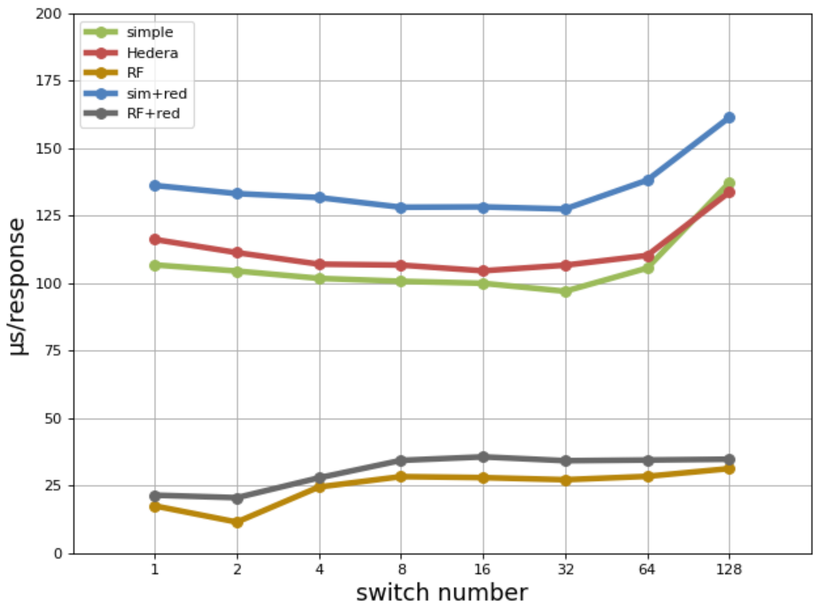


**Figure 2.** Latency Test under Various App Combinations And Switch Numbers.

The response latency of simple analog layer 2 switch app on ONOS is approximately 80 μs lower than that on Ryu. In addition, when the number of switches increases, the performance of ONOS does not decreases obviously. We observe that, just adding one blank registration app still causes Ryu's response latency to rise up by 31.4% (from 97.028 μs to 127.52 μs when there are 32 switches). ONOS's situation is slightly better, but it has also increased by 20.5% on average. It is noted that Figure 2 is just a small test to verify our conjecture; greater data comparison is seen in Section 5.1.

## 4. Message Distribution Mechanism

After receiving the message from the data plane, most of the SDN controllers complete the preliminary analysis and push the message to the registered apps. We believe that it is not such a simple delivery of the message as apps have their own interest, for example, the topology discovery app just needs the LLDP or ARP message instead of others in packet-in messages. Accordingly, we propose an SDN enhancement for the NBI, hoping to ensure certain network invariants while improving communication efficiency.

### *4.1. Message Queue Model*

We notice that the cause of the problem that is described in Section 3.2 is that the NBI ignores the apps' demand for messages. Taking the controller that supports OpenFlow as an example, most of the controllers can distribute events according to OpenFlow message types, so each app can register a handler to receive the corresponding event when the app is implemented. We integrate the message queue into the NBI of the controller to further divide the types of messages. The app subscribes to topics of interest by the message queue, then, in addition to general messages, the controller will only send messages under this topic to the app.

EMD runs in parallel with the core module of the controller in the controller program, as shown in Figure 3. We do not make too many changes to the core module of the conventional controller, but just take over the delivery mode of data messages, which is the most in the NBI through the message queue. The details will be further introduced in the form of pseudo-code in the follow-up.

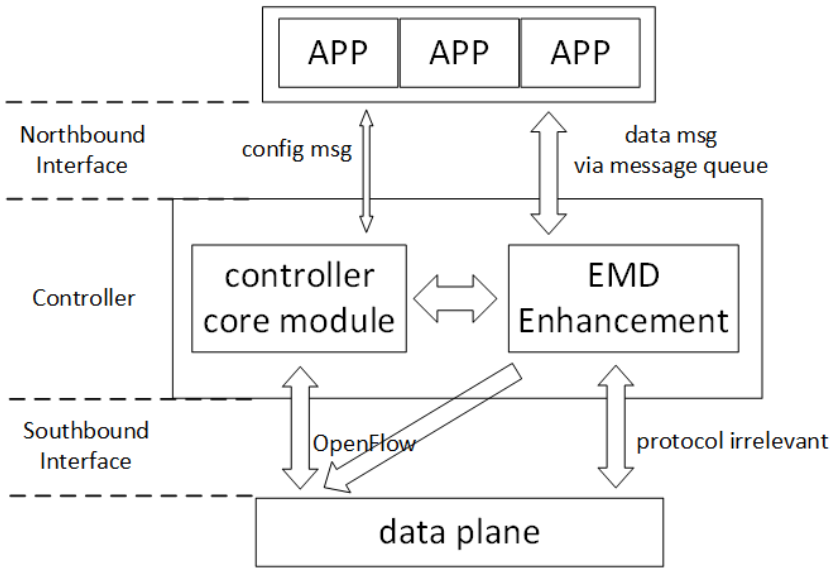


**Figure 3.** The Overall Structure of SDN with EMD Module.

Taking the most important asynchronous message packet-in in OpenFlow as an example, Algorithm 1 shows the improved northbound message distribution mechanism. And Table 1 is an explanation of some symbols and nouns in Algorithm 1

**Table 1.** The symbols of Algorithm 1.

| Symbol | Description |
|---|---|
| *msg* | The structure of south-north control message |
| *match* | The match field of OpenFlow protocol |
| $\{xid : multi\_app\}$ | A dictionary that the key is *msg.xid* and the value is a bool value indicating whether to send to multiple apps |
| *FOR_ALL* | A match collection of messages that need to send to all apps |
| *REPLY* | A match collection of simple messages that can be directly responded to |
| *send_msg_all* | Function that sends messages to all apps |
| *send_msg_topic* | Function that sends prefix topic message |

The main idea of Algorithm 1 is to use the publish–subscribe model of the message queue to subdivide the message distribution, which is more like a pre-classifier for the NBI. The first part (which is from Line 6 to Line 11) is mainly for the processing of general messages, which is the same as the usual controller; the second part (which is from Line 13 to Line 28) uses the match field, part of the message in OpenFlow protocol, to define the topic of the message queue. When the Apps are installed and the subscription action is then completed, the controller is responsible for parsing the message's match field, and the message queue is responsible for completing the task of distribution by category (using match field).

**Algorithm 1** Enhanced Message Distribution Mechanism.

```
Input: $msg$
Output: $\{xid : multi\_app\}$
 1: initialization:
 2: apps register and add the topics to the list $TOPIC$.
 3: end initialization
 4: for each $msg$ in southbound interface do
 5:   parsing the $msg$ in core module and set the $match$ filed
 6:   if $msg$ is in [Handshake, Config, Status, or other specific type] then
 7:     call for correspond event handler
 8:     continue
 9:   else if $msg$ is $FOR\_ALL$ then
10:     $send\_msg\_all(msg)$
11:     continue
12:   else
13:     for each $match$ in $msg$ do
14:       if $match$ is in $REPLY$ then
15:         generate the reply $msg$ named $resp\_msg$
16:         $packet\_out(resp\_msg)$
17:         $notify\_core\_module(dpid, match, msg.xid)$
18:         $msg\_topic = NULL$
19:         break
20:       else if $match$ is in $TOPIC$ then
21:         $msg\_topic \mid = match$
22:       end if
23:     end for
24:     if $msg\_topic \neq NULL$ then
25:       $send\_msg\_topic(msg\_topic)$
26:       $send\_msg(msg)$
27:     end if
28:     add $(msg.xid,\ 1\ if\ msg\_topic \geq 2\ else\ 0)$ to $\{xid : multi\_app\}$
29:   end if
30:   update the data plane's topology
31: end for
32: return $\{xid : multi\_app\}$
```

We believe that the use of pre-classification can improve the efficiency of northbound communication, especially when the controller has a large number of apps. Because most of the controllers that support OpenFlow [11–14] have completed message parsing before distribution, the extraction of message features will not bring too much burden; on the other hand, the time complexity of message feature matching is the message level, but it is still relatively lower when compared to the calling and processing of the apps.

## *4.2. Protocol Irrelevance*

When extracting the message features, OpenFlow's match field cannot cover all of the protocols. [24] points out that the number of the matching fields has been increased from 12 in OpenFlow 1.0 to 44 in OpenFlow 1.5, which makes the compatibility of our enhancement very poor. Taking the idea of protocol independence into account, we incorporate part of the POF protocol into the EMD, and apps can subscribe to the topic of interest in an offset-length way instead of the match field.

In Figure 4, we take DNS protocol data packets as an example to briefly introduce the protocol-irrelevant classification methods (ignoring the handshake and configuration phases). First, after the controller receives the message from the southbound data plane, it does not actively complete the analysis, but it directly pushes the message into the EMD enhancement; second, the EMD matches the topic list of the registered apps with the data of length *LENGTH* offset by *OFFSET* from the head of the message according to the *MASK*, and records the set of having been matched apps; finally, the message is published to the apps of interest through the message queue.

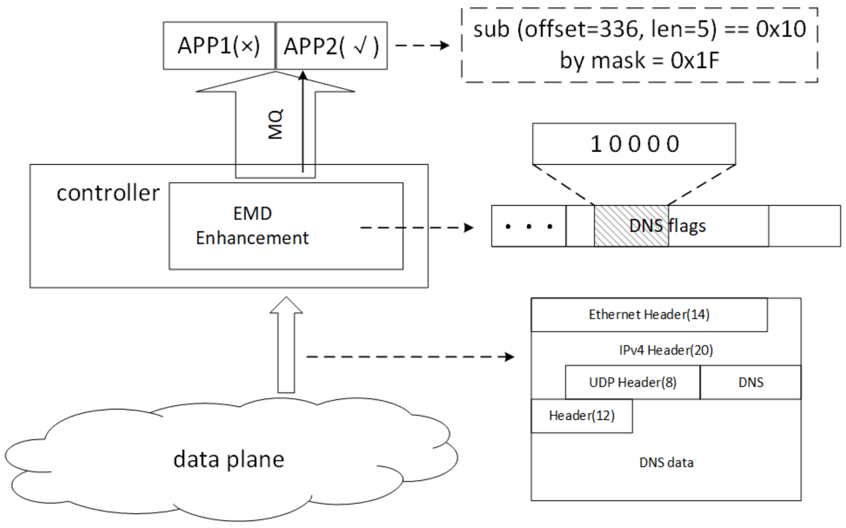


**Figure 4.** Distribution Mechanism by Protocol-irrelevant Way.

The protocol-independent approach not only solves the problem of insufficient fields and poor scalability, but it also provides more flexible implementation methods for complex strategies of apps. However, there are still two tiny problems, one is that the controller needs to support protocol-independent interface (such as POF), and the other is that apps need to parse the required fields of the data packet by themselves. We will improve the second point in the future to reduce the app's parsing for common fields.

## *4.3. Network Invariants Checker*

When the app plane lacks global information, it is prone to competition and conflict. This is an obvious problem in the SDN architecture. Mutual perception between apps will bring great development difficulties, and mutual transparency will bring conflicts between apps [25]. For the network, in addition to degrading network performance, conflicts between apps can paralyze the entire network. Routing loops and black holes are the most common serious problems.

Take Figure 5 as an example to briefly introduce the causes of routing loops and black holes. The host under the $A$ switch sends the data packet to the host under the $F$ switch. The pre-routing is $A \rightarrow B \rightarrow D \rightarrow F$. Because of the $X$ app's strategy, which plans to use the link $D \rightarrow C \rightarrow E \rightarrow F$, so it sends the flow-mod message to switch $D$ and adds the flow table of routing strategy, leading to overwrites the previous policy (may not be issued to the $C/E/F$ switch at the same time). When the data packet arrives at $C$, the $Y$ app considers the path $C \rightarrow B \rightarrow D \rightarrow F$ to be reasonable and continues to send the flow table to switch $C$ to cover the policy. At this time, the data packet will be routed to switch $B$. This is an example of a routing loop because of the inter-app conflict. The black holes generally occur when some links fail down, or a node is dormant due to the energy-saving or the placed app, and the data packets that are routed to that node by other apps will cause a black hole phenomenon.

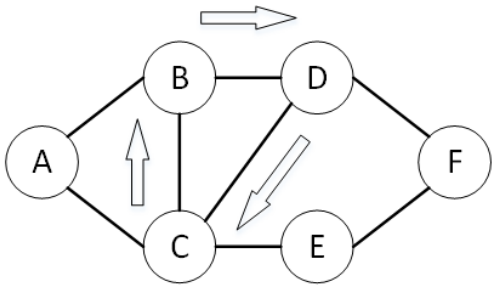


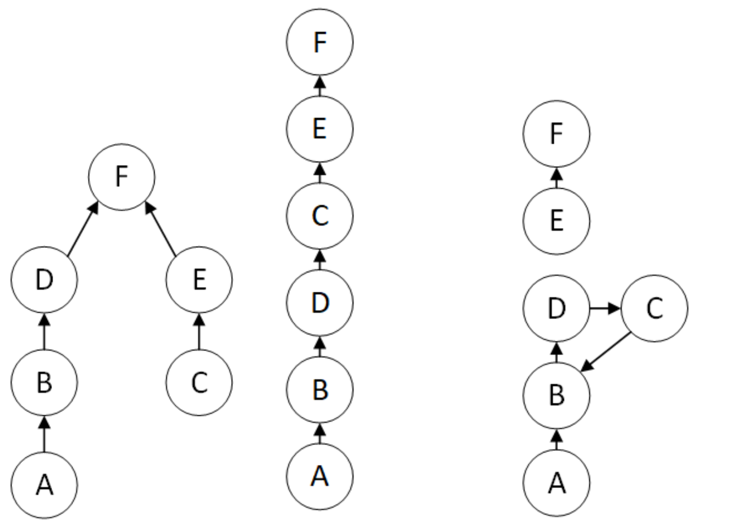
**Figure 5.** An Example of Routing Loop.

The location of our EMD enhancement has a natural advantage and it can guarantee the basic functions of the network, because it can receive all of the apps' response messages and get the global view of the data plane. We set two network invariants, namely no-loop and no-blackhole. When these two functions are enabled, our module will follow Algorithm 2 to check the control messages that are issued by each app to ensure that there are no routing loops and black holes in the network.

**Algorithm 2** Network Invariants Checker.

```
Input: $pkt\_out, flow\_mod$
Output: none
1:  initialization:
2:  set the dictionary strut $FLAG = \{xid : \{'mutil\_app' : bool, 'dealt' : list\}\}$, $FT = [\,]$.
3:  end initialization
4:  for each $pkt\_out$ in the NBI do
5:    if $FLAG[pkt\_out.xid]['mutil\_app']$ is not $TRUE$ then
6:      pass the $pkt\_out$ to the core module
7:    else if $pkt\_out.dpid$ or $pkt\_out.output.dpid$ is in $FLAG[pkt\_out.xid]['dealt']$ then
8:      drop the $pkt\_out$
9:    else
10:     if $check\_alive(pkt\_out.output.dpid)$ is $TRUE$ then
11:       pass the $pkt\_out$ to the core module
12:       put the $pkt\_out.dpid$ into $FLAG[pkt\_out.xid]['dealt']$
13:     else
14:       update the network topo to apps
15:     end if
16:   end if
17: end for
18: for each $flow\_mod$ in the NBI do
19:   $deepcopy(exp\_ft, FT[flow\_mod.dst])$
20:   change $exp\_ft$ according to $flow\_mod.actions$
21:   if $check\_loop(exp\_ft)$ is $TRUE$ then
22:     drop the $flow\_mod$
23:   else
24:     pass the $flow\_mod$ to the core module
25:     update the $FT[flow\_mod.dst]$ using $exp\_ft$
26:   end if
27: end for
```

The $FT$ in Algorithm 2 is a forwarding tree that takes the destination node as the root node, then $check\_alive()$ and $check\_loop()$ are functions to check whether the node is alive and check whether there is a loop in the forwarding tree, respectively. Taking Figure 5 as an example, the forwarding tree of node $F$ that is based on the shortest-path-first is Figure 6a. $X$ app modifies the forwarding tree by changing $flow\_mod$ at node $D$, shown as Figure 6b, and there is no routing loop at this time. After $Y$ app modifies the $flow\_mod$ at point $C$, the forwarding tree is modified to Figure 6c. At this time, a loop has been formed, so we delete this *flow_mod* to prevent routing loops.



(**a**) The OSPF's FT of *F* (**b**) FT Changed by X (**c**) FT Changed by Y

**Figure 6.** The Migration of The Forwarding Tree of Figure 5.

We only check the network invariants for *pkt_out* and *flow_mod*. The reason is that the proportion of these two is the largest in the controller-switch message and it will have a direct impact on the forwarding strategy. Processing these two messages can obtain the greatest stability, and meet the low latency requirements at a certain level.

## 5. Simulation and Evaluation

### *5.1. Latency and Throughput*

In this section, we show more detailed experimental data to verify the effectiveness of the EMD module, in which the experimental environment is similar to Section 3.2. The test software is cbench running on a virtual machine, and the tested one is a Ryu controller on another. The data of the latency experiment are the average of three random experiments that complete thirty consecutive packets in-out at least; the throughput experiment lasts more than 5 min. and is repeated three times.

The example label shown in Figure 7 is basically the same as the previous one; *simple* and *Hedera* represent the simple_switch app and Hereda app running alone; $+red$ means that there is also a redundant app running at the same time, $+3red$ means three redundant apps; and, $+EMD$ means the controller runs with the EMD module in this experiment. Each cluster of bars represents a set of experiments, using cbench to simulate 1/4/16/64 switches for experiments.

The redundant app only registers with the controller and listens to the event of the packet-in message; after receiving the event notification, it does not analyze the data packet, but it simply performs ten plus-one operations in a loop, and then ends. We use this method to simulate the certain apps, which may be more computationally expensive when processing uninteresting messages. It can be seen that, when comparing clusters 1/2/3 or clusters 6/7/8, the latency will increase significantly with the increase of redundant apps, increased up to 93.1% at most (from 99.02 µs to 191.2 µs by simple_switch when the number of switches is 16). This is what we do not want to see; in many cases, the controller is required to have high performance, even when multiple apps are running in parallel [26].

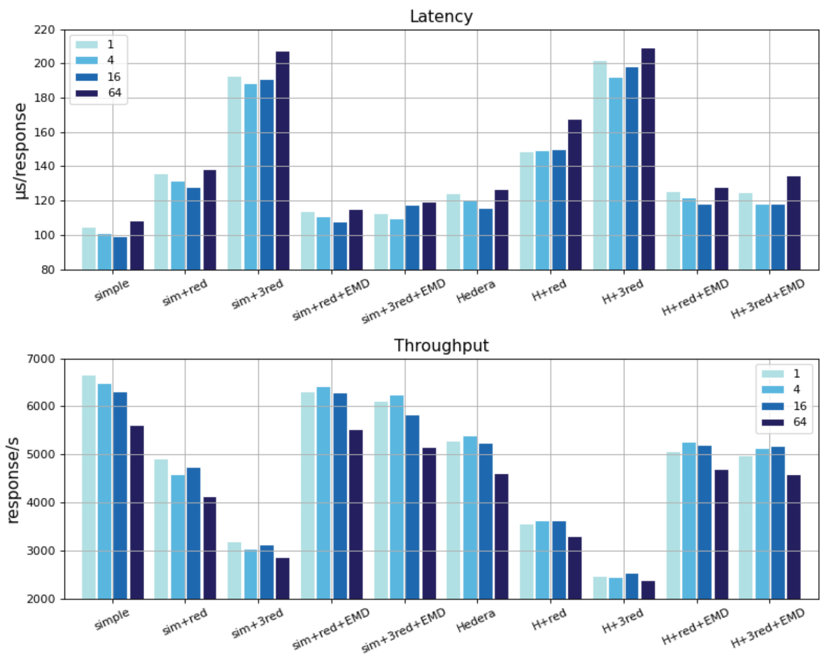


**Figure 7.** Latency & Throughput Test of Ryu under Various App Combinations.

When we add redundant apps, we make the controller run using a EMD module at the same time, which is the 4/5/9/10 cluster bar in Figure 7. It can be concluded that EMD can eliminate the performance degradation that is caused by multi-app parallelism to a certain extent. Take the comparative experiments of $sim$, $sim + 3red$, and $sim + 3red + EMD$ in latency experiment as an example. When Ryu runs the simple_switch app alone, the average response latency is 103.443 µs, but it rises to 195.135 µs after adding three redundant apps with an increase of 88.6%; the average response latency after using the EMD module was only 114.842 µs, an increase of only 11.02%. The latency is reduced by 41.15% when compared with the case of no EMD.

The throughput, also known as the response number per second, also has a corresponding trend. When it is increased to three redundant apps, the performance drops drastically. Based on the two experiments, whether it is the single packet in-out in the idle state or the maximum responses under the stress test, EMD has greatly improved as compared with the original NBI.

## *5.2. Comprehensive Network Performance*

The cbench test method that was used in the previous chapter can only verify the performance under idle (latency experiment) or full load (throughput experiment) conditions. The actual SDN network is complex and difficult to predict; [27] points out that there are key values of parameters in which the target function has first-generation breaks. When considering the comprehensive evaluation and practicality of the SDN network, more network parameters are needed, such as delay, delay variation (jitter), bandwidth, and packet loss parameters on a network [28]. Therefore, this section uses mininet to construct a virtual network to further test the effectiveness of the EMD.

In the experiment, the tested network is a pod-4 fat-tree network constructed by mininet, running on a ubuntu18.04 virtual machine with a 4-core processor and 4 GB of memory; the tested controller is a Ryu controller running on another machine with a 2-core processor and 4 GB of memory. Furthermore, *shortest_path* is an app that is based on the shortest-path-first algorithm of the number of hops, while *Hedera* is a routing algorithm that is based on the network status, the other example labels in figure are the same as in Section 4.1.

The experiment shown in Figure 8 is a bandwidth test performed on a switch in a virtual network built by mininet, and the test software is iperf. The blue part is the bandwidth test between hosts under the same aggregation switch, it can be seen that the algorithm that is based on the network status can slightly increase the bandwidth of the network by approximately 5.4% (from 2871.56 Kbps to 3027.63 Kbps). However, after adding redundant apps, due to the increase in single response time, the network bandwidth also drops from 3027.63 Kbps to a minimum of 1201.49 Kbps, a decrease of 60.3%. When the EMD is selected, the network bandwidth is 2887.68 Kbps, a drop of only 4.6%, which is even higher than shortest_path app. Similarly, in the orange part, the core switch test can also draw the same conclusion. The gray part is a random test between switches. We use stacked bars to make the disparity more obvious.

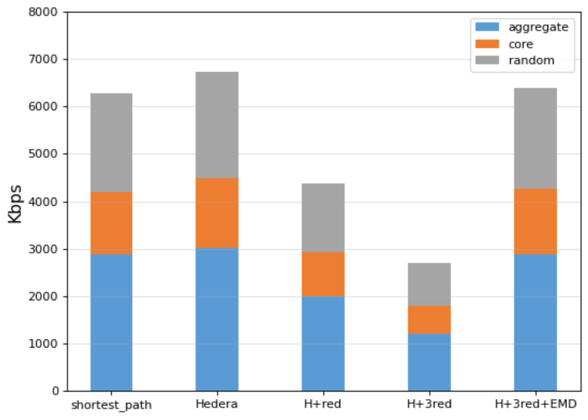


**Figure 8.** Bandwidth Test of A Fat-tree Data Center.

Furthermore, we use the tcp mode in iperf to test the time that it takes to transmit 1 MB of data on the network, and the conclusions are very similar (as shown in Figure 9). At the same time, other network parameters of using EMD also have a little advantage, such as jitter and packet loss rate. It can be concluded that the EMD is effective for the improvement of the SDN network's performance.

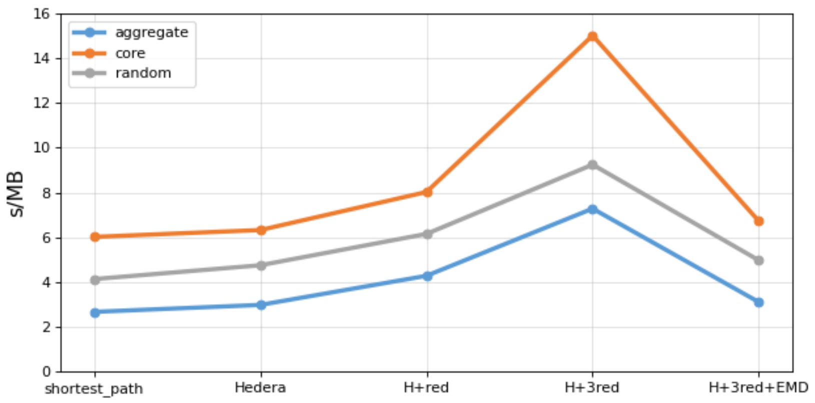


**Figure 9.** Transmission Time Test of A Fat-tree Data Center.

Although EMD can only directly affect the efficiency of northbound communication, when the impact of each packet in-out accumulates, the impact on the performance of the data plane also becomes important. It can be concluded that EMD indirectly improves the performance of the data plane.

### *5.3. The Experiment of Protocol-Irrelevant Mode*

The protocol-independent acceleration strategy in the EMD module needs the support of protocol-independent controllers and data plane devices. However, most of the existing simulation tools only support OpenFlow devices. In terms of controllers, POFOX [29] and PNPL [30] are quite immature, and the projects have stopped maintenance and updates; in terms of the data plane, the software-based POF switch prototype [31] has ceased to support, and commercial hardware switches (e.g., Huawei NE40E and NE5000E) are difficult for completing experimental verification under a certain scale. As a result, we use an alternative plan and still use the virtual network constructed by mininet as the tested network, in which the node is OpenvSwitch, which supports OpenFlow 1.3. During the experiment, OpenFlow is used to complete the handshake phase and the topology discovery. When running normally and getting the southbound message event, the parsing module is not called, but the feature matching is directly completed by the EMD module.

In addition, in order to ignore the influence of apps' routing decisions, the experiment in this section uses a chain network and an end-to-end delay to measure the effectiveness of protocol-independent strategies, in which independent variable is the path length.

In Figure 10, $+i$ means the controller uses the protocol-irrelevant mode and the others are the same meanings as before. Because of a chain network, the path-finding algorithm consumes basically the same time, so the delay difference can be considered to be the communication cost between the switch and the controller. The left picture in Figure 10 is the overall picture of the end-to-end delay under various path lengths, it is not so intuitive enough that we use the simple_switch app's delay as a benchmark to redraw the growth rate under the three modes into the right picture. It can be seen that, when the path length is small, the protocol-independent mode is not significantly effective. However, when the path is long, this mode can reduce up to 220.192 µs from the Openflow one, a decrease rate of 5.9% (from 4233.782 µs to 4013.59 µs when the path length is 16 hops).

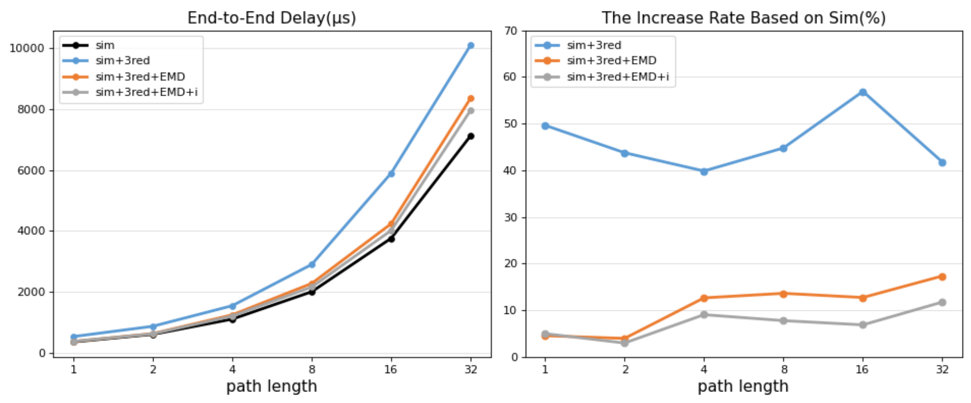


**Figure 10.** End-to-End Delay Test under Various Forwarding Path Length.

In addition to the reduction of delay, scalability and flexibility are the main advantages of protocol-irrelevance. Unfortunately, there is no suitable experiment to visually demonstrate. However, we believe that protocol-irrelevant methods with certain performance guarantees will be more favored by developers.

### *5.4. The Impact of Subscription Rates*

In our previous experiments, the default redundant app is not interested in packet-in packets at all; this extreme assumption makes our EMD have the best performance. Unfortunately, this is not possible. Ref. [32] points out that, in order to further meet the requirements of function and performance, SDN networks will have various complex combinations of apps running in parallel. In addition, no detailed research on the number and types of SDN apps has been found in the existing literature. Accordingly, in this section, we want to evaluate the performance changes of the EMD under different app subscription rates through simulation.

It should be noted that the subscription rate mentioned in this experiment refers to the proportion of apps that are interested in a certain topic, among all apps. In the case of multiple topics, the maximum subscription rate of all topics is taken.

$$r_{sub} = \max_{\forall t \in topic} \frac{n_t}{n_{reg}} \tag{8}$$

where $r_{sub}$ is the subscription rate; $n_t$ is the number of apps subscribed to the t topic; and, $n_{reg}$ is the total number of registered apps.

For example, among ten apps, No.1 to No.3 are interested in match_1, No.5 to No.8 are interested in match_2, and No.8 to No.10 are interested in match_3. In this case, the subscription rate is 40% instead of 90%. We have completed supplementary experiments, the performance in the multi-topic scenario is basically equivalent to that in the single-topic scenario according to the above rules.

This experiment environment is basically the same as Section 4.1. The test software is cbench using both latency mode and throughput mode, and the controller is Ryu. The difference is that some redundant apps also subscribe to messages to implement experimental environments with different subscription rates.

In Figure 11, the bar graph is the throughput test result and another is the latency test. In the throughput test, we can see that, when the subscription rate is low, the effect of the EMD is extremely good. The effect gradually decreases as the subscription rate increases, but it is still better than the normal model before the subscription rate is 70%. This situation is actually in line with our expectations. When the subscription rate is too high, the classification process of the EMD becomes useless, even a burden, because the message needs to be forwarded to almost all apps.

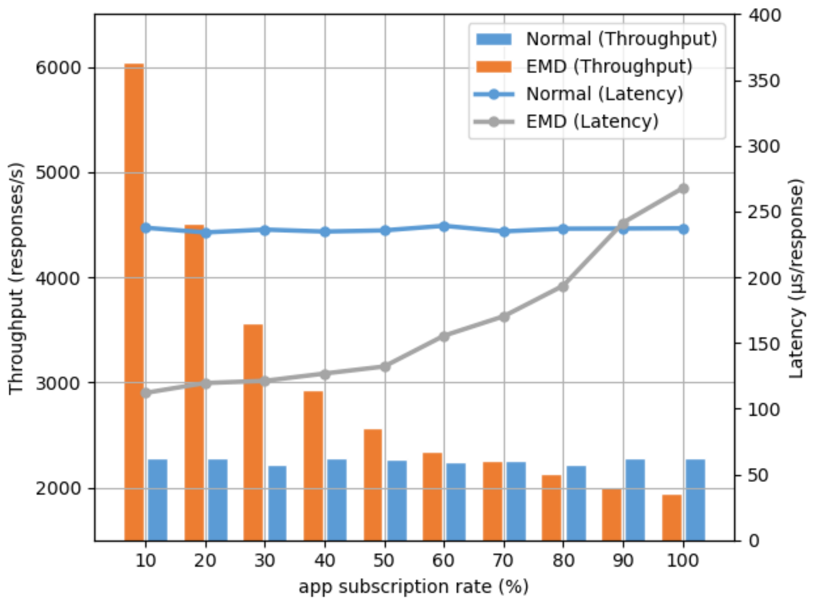


**Figure 11.** Latency And Throughput Test under Various Subscription Rates.

Similarly, the latency test also obtains almost the same result. When the subscription rate is about 90% and higher, the single response latency of the EMD is higher than that of the normal mode. In addition, when comparing the two experiments, it can be seen that, when the subscription rate is low, as compared with the throughput, the single response latency deteriorates slightly with the increase of the subscription rate.

Our original intention in designing EMD is to avoid processing uninteresting messages, so pre-classification is our advantage. When the subscription rate of a topic is too high, we can consider directly forwarding it without classifying it, which can obtain the maximum benefit.

## 6. Future Works

### *6.1. Faster Match Mechanism and Algorithm*

The advantage of the EMD is to complete message parsing and classification in advance, so the time complexity of the related classification mechanism and algorithm directly affects the overall performance. Our existing classification mechanism is based on the southbound communication protocol OpenFlow and it directly uses the match field after message parsing by the controller. Although this mechanism is simple and easy to implement, it cannot control the time complexity of analysis by ourselves. The matching distribution algorithm is based on the priority of topics, but the historical information of the controller is not used well. The subsequent matching mechanism can be combined with the matching algorithm, which is based on popularity and using-space.

### *6.2. Compatibility*

Compatibility has always been one of the main problems of NBI. There is no unified northbound standard, and some of them do not even have a clear division of the app plane and the control plane, as mentioned in the previous part. The implementation in this article is also heavily coupled with the controller, such as Ryu and ONOS. There is no modular program to adapt to most of the existing controllers (in that case, the difficulty is quite high). This part is one of our hopes, which to complete the modularization after the NBI has a unified standard.

### *6.3. Flexible Mode Switching*

The experiment shown in Section 5.4 shows that our mechanism has poor performance when the subscription rate of a certain topic is too high. In the future, we will consider adding a flexible selection mechanism, which switches to normal mode when the subscription rate of some topics is higher than the threshold for a certain period of time.

## 7. Conclusions

Although there is no unified NBI standard so far, we still hope to enhance the efficiency of SDN northbound communication on the existing basis. We find that the existing northbound communication has a rather rough classification of messages, which causes some apps to frequently process messages that they are not interested in. We use the idea of message queues for northbound communication to further subdivide the classification of messages and provide accurate message delivery for apps. We have also optimized the two different methods of OpenFlow and protocol-independent in southbound communication, so that our method is highly efficient and compatible at the same time. On this basis, we correspondingly increased the monitoring of two network invariants according to our position in SDN, which further improved the stability of the entire system. Finally, our EMD is designed to further subdivide app demand, and there are still certain shortcomings when the subscription rate is high. We plan to optimize this aspect in the next step.

**Author Contributions:** Conceptualization, C.W., H.N. and L.L.; methodology, C.W., H.N. and L.L.; software, C.W.; validation, C.W., H.N. and L.L.; writing—original draft preparation, C.W.; writing—review and editing, C.W., H.N. and L.L.; visualization, C.W.; supervision, H.N. and L.L.; project administration, L.L.; funding acquisition, H.N. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was funded by Strategic Leadership Project of Chinese Academy of Sciences: SEANET Technology Standardization Research System Development (Project No. XDC02070100).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Not applicable.

**Acknowledgments:** We would like to express our gratitude to Hong Ni, Lei Liu, Xiaodong Zhu, Yuwei Wang and Ce Zheng for their meaningful support for this work.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
2. Kreutz, D.; Ramos, F.M.; Verissimo, P.E.; Rothenberg, C.E.; Azodolmolky, S.; Uhlig, S. Software-defined networking: A comprehensive survey. *Proc. IEEE* **2014**, *103*, 14–76. [CrossRef]
3. Kim, H.; Feamster, N. Improving network management with software defined networking. *IEEE Commun. Mag.* **2013**, *51*, 114–119. [CrossRef]
4. Lara, A.; Kolasani, A.; Ramamurthy, B. Network innovation using openflow: A survey. *IEEE Commun. Surv. Tutor.* **2013**, *16*, 493–512. [CrossRef]
5. Mousa, M.; Bahaa-Eldin, A.M.; Sobh, M. Software Defined Networking concepts and challenges. In Proceedings of the 2016 11th International Conference on Computer Engineering & Systems (ICCES), Cairo, Egypt, 20–21 December 2016; pp. 79–90.
6. Koponen, T.; Casado, M.; Gude, N.; Stribling, J.; Poutievski, L.; Zhu, M.; Ramanathan, R.; Iwata, Y.; Inoue, H.; Hama, T.; et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*; USENIX Association: Canada, CA, USA, 2010; Volume 10, pp. 1–6.
7. Deng, Y.; Yiming, J.; Weiwei, H. A Method for Aggregately Deploying Services in SDN. *J. Netw. New Media* **2018**, *7*, 13–17.
8. Song, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 127–132.
9. Cox, J.H.; Chung, J.; Donovan, S.; Ivey, J.; Clark, R.J.; Riley, G.; Owen, H.L. Advancing software-defined networks: A survey. *IEEE Access* **2017**, *5*, 25487–25526. [CrossRef]
10. Richardson, L.; Ruby, S. *RESTful Web Services*; O'Reilly Media, Inc.: Massachusetts MA, USA 2008.
11. Tootoonchian, A.; Gorbunov, S.; Ganjali, Y.; Casado, M.; Sherwood, R. On Controller Performance in Software-Defined Networks. In Proceedings of the 2nd {USENIX} Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 12), San Jose, CA, USA, 24 April 2012.
12. Erickson, D. The beacon openflow controller. In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, Hong Kong, China, 16 August 2013; pp. 13–18.
13. Morita, K.; Yamahata, I.; Linux, V. Ryu: Network operating system. In Proceedings of the OpenStack Design Summit & Conference, San Francisco, CA, USA, 16–20 April 2012.
14. Krishnaswamy, U.; Berde, P.; Hart, J.; Kobayashi, M.; Radoslavov, P.; Lindberg, T.; Sverdlov, R.; Zhang, S.; Snow, W.; Parulkar, G. ONOS: An open source distributed SDN OS. In *HotSDN' 14*; Association for Computing Machinery: New York, NY, USA, 2014.
15. Zhou, Z.; Benson, T.A. Composing SDN Controller Enhancements with Mozart. In Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, 20–23 November 2019; pp. 351–363.
16. Ferguson, A.D.; Guha, A.; Liang, C.; Fonseca, R.; Krishnamurthi, S. Participatory networking: An API for application control of SDNs. *ACM SIGCOMM Comput. Commun. Rev.* **2013**, *43*, 327–338. [CrossRef]
17. Ferguson, A.D.; Guha, A.; Liang, C.; Fonseca, R.; Krishnamurthi, S. Hierarchical policies for software defined networks. In Proceedings of the First Workshop on Hot Topics in Software Defined Networks, Helsinki, Finland, 13 August 2012; pp. 37–42.
18. Sun, P.; Mahajan, R.; Rexford, J.; Yuan, L.; Zhang, M.; Arefin, A. A network-state management service. In Proceedings of the 2014 ACM Conference on SIGCOMM, Chicago, IL, USA, 17–22 August 2014.; pp. 563–574.
19. Dwaraki, A.; Seetharaman, S.; Natarajan, S.; Wolf, T. GitFlow: Flow revision management for software-defined networks. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, Santa Clara, CA, USA, 17–18 June 2015; pp. 1–6.
20. Khurshid, A.; Zou, X.; Zhou, W.; Caesar, M.; Godfrey, P.B. Veriflow: Verifying network-wide invariants in real time. In Proceedings of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13), Lombard, IL, USA, 2–5 April 2013; pp. 15–27.
21. Chandrasekaran, B.; Benson, T. Tolerating SDN application failures with LegoSDN. In Proceedings of the 13th ACM Workshop on Hot Topics in Networks, Los Angeles, CA, USA, 27–28 October 2014; pp. 1–7.
22. Scott, C.; Wundsam, A.; Raghavan, B.; Panda, A.; Or, A.; Lai, J.; Huang, E.; Liu, Z.; El-Hassany, A.; Whitlock, S.; et al. Troubleshooting blackbox SDN control software with minimal causal sequences. In Proceedings of the 2014 ACM Conference on SIGCOMM, Chicago, IL, USA, 17–22 August 2014; pp. 395–406.
23. Reitblatt, M.; Canini, M.; Guha, A.; Foster, N. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*; Association for Computing Machinery: New York, NY, USA 2013; pp. 109–114.

24. Hu, D.; Li, S.; Xue, N.; Chen, C.; Ma, S.; Fang, W.; Zhu, Z. Design and demonstration of SDN-based flexible flow converging with protocol-oblivious forwarding (POF). In Proceedings of the 2015 IEEE Global Communications Conference (GLOBECOM), San Diego, CA, USA, 6–10 December 2015; pp. 1–6.
25. Li, Y.; Wang, Z.; Yao, J.; Yin, X.; Shi, X.; Wu, J.; Zhang, H. MSAID: Automated detection of interference in multiple SDN applications. *Comput. Netw.* **2019**, *153*, 49–62. [CrossRef]
26. Durairajan, R.; Sommers, J.; Barford, P. Controller-agnostic SDN debugging. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, Sydney, Australia, 2–5 December 2014; pp. 227–234.
27. Barabash, O.; Kravchenko, Y.; Mukhin, V.; Kornaga, Y.; Leshchenko, O. Optimization of Parameters at SDN Technologie Networks. *Int. J. Intell. Syst. Appl.* **2017**, *9*, 1–9. [CrossRef]
28. Muhizi, S.; Shamshin, G.; Muthanna, A.; Kirichek, R.; Vladyko, A.; Koucheryavy, A. Analysis and performance evaluation of SDN queue model. In *International Conference on Wired/Wireless Internet Communication*; Springer: Cham, Switzerland, 2017; pp. 26–37.
29. Tan, X.; Zou, S.; Guo, H.; Tian, Y. POFOX: Towards controlling the protocol oblivious forwarding network. In *Advances in Parallel and Distributed Computing and Ubiquitous Services*; Springer: Singapore, 2016; pp. 21–30.
30. Wang, X.; Tian, Y.; Zhao, M.; Li, M.; Mei, L.; Zhang, X. PNPL: Simplifying programming for protocol-oblivious SDN networks. *Comput. Netw.* **2018**, *147*, 64–80. [CrossRef]
31. Li, S.; Hu, D.; Fang, W.; Ma, S.; Chen, C.; Huang, H.; Zhu, Z. Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability. *IEEE Netw.* **2017**, *31*, 58–66. [CrossRef]
32. Pham, M.; Hoang, D.B. SDN applications-The intent-based Northbound Interface realisation for extended applications. In Proceedings of the 2016 IEEE NetSoft Conference and Workshops (NetSoft), Seoul, Korea, 6–10 June 2016; pp. 372–377.