A Low-cost, High-efficiency SDN Framework to Diminish Redundant ARP and IGMP Traffics in Large-scale LANs

You-Chiun Wang
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, 80424, Taiwan
Email: ycwang@cse.nsysu.edu.tw

Han Hu
Department of Computer Science and Engineering
National Sun Yat-sen University
Kaohsiung, 80424, Taiwan
Email: m033040090@student.nsysu.edu.tw

***Abstract*—Ethernet is the predominant protocol in the data-link layer of wired networks. It works based on the concept of *broadcast domain* by using switches to connect hosts, which means that every host in a *local area network (LAN)* will receive packets sent from other nodes. Thus, Ethernet's performance inevitably degrades when LAN scale grows, as the LAN will be congested by superfluous packets due to broadcast. Moreover, Ethernet switches cannot support multicast protocols but realize them by also broadcasting packets, which worsens the situation. To solve the broadcast problem, we develop a low-cost, high-efficiency framework based on *software defined network (SDN)*, which aims to diminish redundant traffics produced by the *address resolution protocol (ARP)* and the *Internet group management protocol (IGMP)*. Through simulations by Mininet, we show that our SDN framework not only greatly reduces broadcast packets in Ethernet but also significantly saves the controller's cost as comparing with other SDN-based methods. In addition, the proposed framework is also implemented in our campus network to show its effectiveness.**

I. Introduction

Ethernet is regulated by the IEEE 802.3 standard [1], and it is extensively used in many huge LANs (e.g., campus, data center, and enterpise networks). What lies behind Ethernet's achievements is its low overhead and ease of use. Hosts can fast join an Ethernet network with simple configuration. Nowadays, Ethernet interface cards are necessary to most hosts for wired communications.

In Ethernet, hosts are linked together via switches and form a *broadcast domain* accordingly. Consequently, most protocols above Ethernet should use the broadcasting operation for service or resource discovery. Let us take ARP [2] as an example. It helps a host to find the mapping of MAC (medium access control) address and IP (Internet protocol) address of another host in the same domain by flooding its query. Another example is IGMP [3]. Because an Ethernet switch does not understand the multicast protocol defined in the network layer, it simply transmits packets to every host in the broadcast domain.

When the scale of a LAN is small, the broadcast mechanism of Ethernet works well. However, when the number of hosts increases, switches have to be organized hierarchically to form a large broadcast domain to connect these hosts, as illustrated in Fig. 1. In this case, the LAN will be inevitably full of broadcasting packets. To verify this argument, we gather statistics of packets received by each host in our campus LAN, whose network topology is similar to Fig. 1 and contains 9 class-C subnets. Fig. 2 presents the ratios of different types of packets averagely received by a host in one day. We can observe that ARP and IGMP traffics contribute 67.0% and 15.1% packets, respectively. In fact, most ARP and IGMP packets are irrelevant to their capturing hosts (in other words, these packets are redundant). The experimental result shows that Ethernet becomes inefficient in a large-scale LAN, because each host in fact requires a pretty small portion of its receiving data.

Common solutions to the above broadcast problem are to divide the LAN into multiple broadcast domains (e.g., by

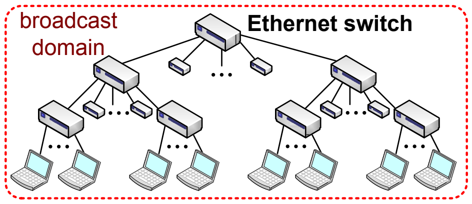


Fig. 1. A large-scale LAN formed by one single broadcast domain.

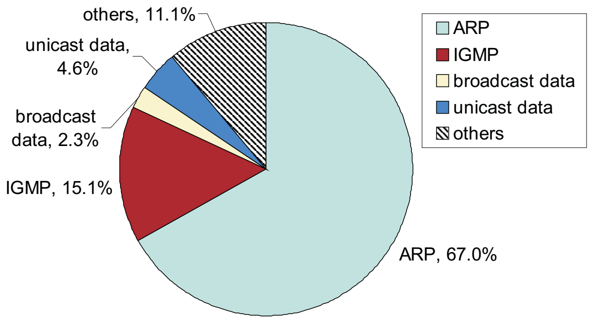


Fig. 2. Ratios of different packets received by a host in one day.

routers or 802.1Q switches) to confine broadcasting packets to each small domain. Unfortunately, these solutions have some shortcomings. First of all, they usually incur a high cost in terms of hardware (i.e., using routers) or manual configuration (i.e., using 802.1Q switches). Second, if we do not use mobile IP, the mobility of hosts or the migration of virtual machines (in data center networks) across different domains will become difficult. Third, the packets produced by some protocols such as NetBIOS [4] cannot be disseminated to other broadcast domains managed by routers.

Recently, the SDN technique is proposed to provide a novel paradigm for network management [5]. It logically divides a LAN into control and data planes. The core of the control plane, namely the *controller*, coordinates switches and conducts management jobs such as how to interpret packet headers and where to forward them. The data plane is distributed over switches to handle packet transmissions. Through SDN, network administrators can easily manage traffic flows by installing transmission rules to OpenFlow switches with the help of the controller. SDN, together with the techniques of wireless sensor network [6], RFID (radio frequency identification) [7], cloud computing [8], mobile network [9], and intelligent vehicles [10], have been shaping the development of the future Internet.

The objective of this paper is to develop an efficient framework by taking advantage of SDN to *automatically* set up transmission rules to diminish redundant traffics to improve Ethernet's performance in large-scale LANs. In our proposed framework, the controller analyzes incoming packets and learns the status of the ongoing protocol. Then, it spontaneously generates rules to prevent switches from broadcasting superfluous packets produced by that protocol. We use both ARP and IGMP to demonstrate how our framework operates, which are two essential auxiliary protocols for IP but produce numerous redundant packets (as shown by our experiment in Fig. 2).

Our contributions are threefold. First of all, unlike most SDN-based methods, the proposed framework helps the controller smartly translate ARP addresses rather than forcing it to serve as a proxy to process every ARP packet. In this way, our SDN framework can significantly save the message overhead and computation cost of the controller. Second, the design of our SDN framework considers the issues of backward compatibility and multicast, which are not addressed by existing methods. Simulation results verify that our SDN framework still performs well in a LAN which has Ethernet switches. Finally, we implement the proposed framework in our campus LAN, and the experimental result shows that it can greatly improve Ethernet's performance.

This paper is outlined as follows: Section II briefly introduces the SDN technique, and Section III presents related work. In Section IV, we discuss the detailed design of our SDN framework. Afterwards, Section V measures LAN performance by simulations, followed by the discussion of our implementation in Section VI. Finally, Section VII concludes this paper and gives some future research directions.

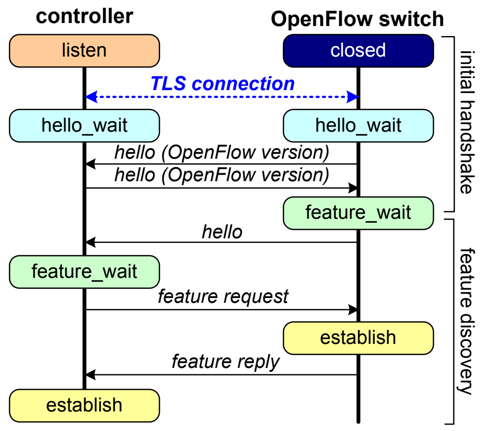


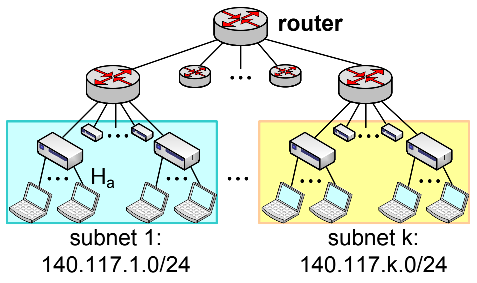
Fig. 3. Message flowchart to negotiate with an OpenFlow switch.

II. SDN Technique

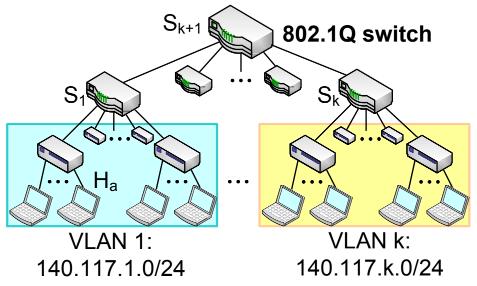
To carry out the concept of SDN, the OpenFlow protocol [11] replaces Ethernet switches by *OpenFlow switches*, which are capable of executing commands sent from the controller. The protocol defines not only the operations of an OpenFlow switch but also how it communicates with the controller. To do so, each OpenFlow switch refers to a flow table to process incoming packets, which contains *flow entries* that give rules and actions. On receiving a packet, the OpenFlow switch searches for the first flow entry whose rules are matched. It then transmits (or discards) the packet according to the action indicated by that entry. In case that no entry can be found, the OpenFlow switch triggers a table-miss event, which makes it transmit a *Packet_In message* along with the packet's information to the controller. Then, the controller replies a flow entry to guide the OpenFlow switch how to process the packet by sending a *Packet_Out message*. In this way, the controller can manage OpenFlow switches and master their packet transmissions.

In our work, we adopt the Ryu platform [12] to implement the controller, which is a popular open-source SDK (software development kit) for SDN controllers. Ryu supports the OpenFlow protocol and offers software components with well-defined API (application program interface) in Python to help programmers create network management and control applications. A programmer can implement the SDN application by registering input events together with the handling functions in Ryu. These events are stored in a queue and dispatched to their functions in a first-in, first-served manner. Besides, Ryu provides a packet-handling mechanism for the controller to get the state of the ongoing protocol by analyzing the headers of capturing packets.

Fig. 3 gives the message flowchart for the controller to negotiate with an OpenFlow switch. In particular, when an OpenFlow switch joins the LAN (e.g., start operating), it establishes a connection with the controller through transport layer security (TLS) to trigger initial handshake. This connection makes both the OpenFlow switch and the controller change to the *hello_wait state* and exchange hello messages with



(a) Ethernet-based method by using routers



(b) Ethernet-based method by using 802.1Q switches

Fig. 4. Dividing a LAN into multiple broadcast domains.

their supported versions of OpenFlow. They will agree using the lowest version and finish the initial handshake procedure. Afterwards, the OpenFlow switch changes to the *feature_wait state* to invoke the feature discovery procedure, which also sends a hello message to the controller to make it change to the same state. Then, the controller sends a feature request to the OpenFlow switch to ask for its parameters. After returning the feature reply, both the OpenFlow switch and the controller change to the *establish state* and finish the feature discovery procedure. In this way, the controller can get the information of each OpenFlow switch.

III. Related Work

We survey existing methods in the literature to cope with the broadcast problem in a large-scale LAN, which can be categorized into Ethernet-based and SDN-based groups.

A. Ethernet-based Methods

Past methods to solve the broadcast problem is to divide the LAN into many small broadcast domains. One method is to use more powerful routers, where each router deals with packet transmissions in a broadcast domain (usually called *subnet*), as illustrated in Fig. 4(a). Nevertheless, this method puts constraints on the allocation of IP addresses, because a router will check the legality of IP address of each host in its subnet through the subnet mask. In addition, it is infeasible to support host mobility across different subnets (unless we use mobile IP). Fig. 4(a) gives an example, where a host H_a with IP address of 140.117.1.2 cannot move into subnet k with network segment of 140.117.k.0/24, since the router in subnet k will discard all of its packets.

Using 802.1Q switches is another method, where each switch consults a four-byte label in the Ethernet header to forward the packet to the right port [13]. In this way, each 802.1Q switch can divide its child switches into *virtual LANs (VLANs)*, each mapping into a broadcast domain, as Fig. 4(b) shows. Comparing with the router-based method, 802.1Q switches is able to support host mobility without mobile IP. Fig. 4(b) gives an example, where host H_a wants to move to VLAN k. In this case, switches S_1, S_k, and S_{k+1} should be manually reconfigured to let host H_a become a member of VLAN k. Apparently, this method is inefficient and uneconomic, as the network administrator has to configure multiple switches to support host mobility.

There are also a number of Ethernet-based methods developed. The work of [14] adopts specially made hardware to help Ethernet switches know upper-layer protocols and transform broadcast traffics into unicast ones. This method incurs extra costs of using special hardware. Besides, each switch has to record the information of all hosts. To address this issue, a hash table is used in [15] to share host information among different switches, so as to save their memory space. In [16], a gateway is placed on the entrance of every broadcast domain to monitor passing packets. Then, each broadcast packet is replaced by multiple unicast packets to avoid wasting bandwidth. Similar to NAT (network address translation), [17] divides a broadcast domain into internal and exterior parts, and broadcast traffics are limited in the internal part. However, this method will encounter the same problem incurred by the router-based method.

B. SDN-based Methods

Several studies exploit SDN to improve Ethernet's efficiency. To facilitate the allocation process of IP addresses by DHCP (dynamic host configuration protocol) [18], both [19] and [20] ask the SDN controller to serve as a DHCP server. When a host joins the LAN, the OpenFlow switch forwards its DHCP discovery packet to the controller to get an unused IP address. In [21], the SDN controller plays the role of ARP proxy to reduce ARP traffics in a data center network. Both IP and MAC addresses of each server should be set in the controller beforehand. OpenFlow switches then relay all ARP requests to the controller, and the controller sends an ARP reply to each target server via unicast. In the SEASDN (scalable Ethernet architecture using software defined networking) framework [22], an independent DHCP server is used to deal with the allocation of IP addresses. SEASDN makes the controller do the job of ARP proxy, and asks OpenFlow switches to forward DHCP packets to the controller (to let it record the information of hosts). Nevertheless, the above studies do not make good use of the superiority of SDN to adaptively reroute packets, but just make the controller act as a DHCP server or an ARP proxy to deal with these traffics. Therefore, they will inevitably impose a heavy load on the controller.

Kataoka et al. [23] develop an ETF (extensible transparent filter) mechanism to diminish ARP and DHCP traffics. It asks each OpenFlow switch to send ARP and DHCP packets to two destinations: the controller and the target (e.g., DHCP server,

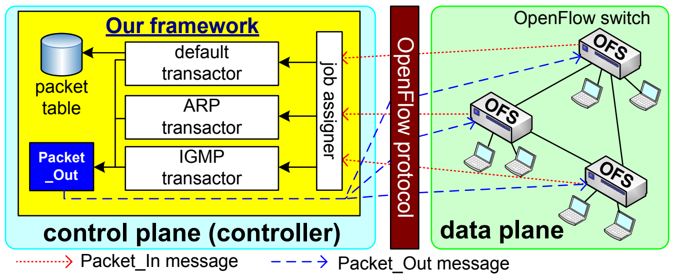


Fig. 5. Architecture of our proposed SDN framework.

a host, or broadcast address). In this way, the controller can know all hosts and the DHCP server. Then, it can instruct OpenFlow switches to forward packets on the designate ports accordingly. Since OpenFlow switches need to send a copy of every broadcast packet to the controller, ETF will increase the controller's overhead. Besides, the issue of backward compatibility is not addressed in ETF. In case that there are Ethernet switches in the LAN, ETF will make them use broadcast to deal with ARP and DHCP packets, which degrades network performance.

Comparing with these methods, our proposed framework not only substantially reduces the controller's overhead by sending only required packets to it, but also handles IGMP multicast traffics that produce lots of superfluous packets in the LAN. In addition, we will show that our SDN framework supports good backward compatibility with Ethernet through the experimental results discussed in Section V, where it can significantly reduce redundant packets in a LAN that consists of both Ethernet and OpenFlow switches.

IV. The Proposed SDN Framework

Our SDN framework works based on the OpenFlow protocol, whose architecture is given in Fig. 5. The controller communicates with OpenFlow switches through Packet_In and Packet_Out messages. As we do not modify the OpenFlow protocol, our framework will focus on the design of the SDN controller. Specifically, it contains four major modules to help the controller learn the status of ongoing protocols and generate flow entries to guide OpenFlow switches how to process their packets. In particular, the *job assigner* takes charge of dispatching each Packet_In message to the right transactor for further processing. If the message is related to ARP or IGMP traffics, it is dispatched to the *ARP transactor* or *IGMP transactor*, respectively, which helps the controller install suitable flow entries in the OpenFlow switch. Otherwise, the message is dispatched to the *default transactor* to let the controller store the information of packets. Below, we present the detailed design of each module.

A. Job Assigner

To make OpenFlow switches send necessary Packet_In messages to the job assigner, the controller installs three flow entries in every OpenFlow switch as follows:

(1) **Rules:** dl_dst=ff:ff:ff:ff:ff:ff, arp, arp_op=1
Actions: actions=controller:6633
(2) **Rules:** igmp, nw_dst=224.0.0.1
Actions: actions=controller:6633, flood
(3) **Rules:** igmp, nw_dst=224.0.0.0/3
Actions: actions=controller:6633

The 1st flow entry asks the OpenFlow switch to send each *new* ARP request whose MAC address is "ff:ff:ff:ff:ff:ff" (i.e., broadcast address) to the controller whose default port is 6633. Since the ARP request contains both the IP and MAC addresses of the source host, the controller can store this information to facilitate the process in the ARP transactor. Then, the 2nd flow entry instructs the OpenFlow switch to send IGMP membership-query packets (with IP address of 224.0.0.1) to not only the controller but also all other hosts (i.e. flood). In this way, the controller can get the information of a multicast group. Besides, hosts can also know the multicast group through the membership-query packets. The 3rd flow entry helps the job assigner get other types of IGMP packets such as membership report and group leave. Unlike membership-query packets, not every host requires these IGMP control packets, so we do not add the flood command in the action of the 3rd flow entry.

B. ARP Transactor

The objective of ARP is to allow a host to learn the relationship between each IP address and its MAC address. To get this information, there are four address fields indicated in an ARP packet:

- *Sender hardware address (SHA):* The MAC address of the host which sent the ARP request.
- *Sender protocol address (SPA):* The IP address of the host which sent the ARP request.
- *Target hardware address (THA):* The MAC address of the host that will send the ARP reply.
- *Target protocol address (TPA):* The IP address of the host that will send the ARP reply.

Let us consider an example where one Ethernet switch connects three hosts H_a, H_b, and H_c, whose addresses and connecting ports are given in Table I. Consider that host H_a wants to send a packet to host H_c. It knows that H_c has IP address of 140.117.0.3 (e.g., through the domain name server). To send out the packet, H_a must get H_c's MAC address. In this case, H_a consults its ARP table to search for any cached record of H_c's MAC address with respect to IP address of 140.117.0.3. If the ARP table returns no result, H_a sends an ARP request with SHA = 20:18:07:23:27:01, SPA = 140.117.0.1, THA = ff:ff:ff:ff:ff:ff, and TPA = 140.117.0.3. When the switch receives this ARP request, it then forwards the request to all its ports (i.e., broadcast). After H_c gets the ARP request, it transmits an ARP reply with SHA = 20:18:07:23:27:03, SPA = 140.117.0.3, THA = 20:18:07:23:27:01, and TPA = 140.117.0.1. Therefore, the switch forwards the ARP reply to H_a via its port 1. Then, H_a caches this information in its ARP table. Next time when it wants to send a packet to H_c again,

TABLE I
PORTS AND ADDRESSES OF THREE HOSTS.

host	port	IP address	MAC address
H_a	1	140.117.0.1	20:18:07:23:27:01
H_b	2	140.117.0.2	20:18:07:23:27:02
H_c	3	140.117.0.3	20:18:07:23:27:03

H_a will send an Ethernet frame with the destination MAC address of 20:18:07:23:27:03 to the LAN.

Although ARP is simple, it unavoidably produces numerous ARP requests when many hosts query their target MAC addresses. Even when a host knows the MAC address of its target (by consulting the ARP table), it still adopts broadcast to send out the packet. Consequently, the LAN will be congested by superfluous packets due to the ARP mechanism. To cope with the above problem, our solution is to let the controller learn the ARP information (including IP addresses, MAC addresses, and ports) by analyzing ARP packets. Then, it can install flow entries in OpenFlow switches to translate ARP broadcast packets into unicast ones, which reduces the amount of ARP data in the LAN. In our framework, when one OpenFlow switch gets an ARP request, it consults the flow table to seek to send the packet to the right port. If there is no flow entry that has the IP address given in the ARP request, the OpenFlow switch broadcasts the ARP request and also sends a copy to the controller. On the other hand, the controller adaptively learns a host's IP and MAC addresses from its *first* ARP request and also the target host's IP and MAC addresses from the corresponding ARP reply. It can be done by referring to both the SPA and SHA fields of each ARP packet. In this way, the controller can quickly learn the IP and MAC addresses of all hosts in the LAN.

Table I gives an example to show how the ARP transactor sets up flow entries for OpenFlow switches to process ARP traffics. Assume that the controller has gotten the ARP request from host H_a that queried the MAC address of host H_c. In this case, the controller can know the IP and MAC addresses of H_a. Therefore, it installs two flow entries in the corresponding OpenFlow switch as follows:

(1) **Rules:** arp, tpa=140.117.0.1
Actions: actions=set_field:20:18:07:23:27:01
->the_dst, output:1
(2) **Rules:** arp, op=2, spa=140.117.0.3, tpa=140.117.0.1
Actions: actions=controller:6633, output:1

Here, the 1st flow entry asks the OpenFlow switch to unicast each ARP packet whose TPA is 140.117.0.1 to H_a via its port 1. Based on the ARP mechanism, H_c will send an ARP reply to H_a. So, the 2nd flow entry instructs the OpenFlow switch to forward the ARP reply to the controller. In this way, the controller can also know H_c's MAC address. Note that the 2nd flow entry is not permanent, as it becomes useless once the controller gets the ARP reply. Thus, we can set a short lifetime (e.g., 10 seconds) for this flow entry. On the other hand, when the controller receives the ARP reply, it will set up a flow entry as follows:

- **Rules:** arp, tpa=140.117.0.3
Actions: actions=set_field:20:18:07:23:27:03
->eth_dst, output:3

The flow entry will make the OpenFlow switch to forward all ARP (broadcast) packets whose TPA is 140.117.0.3 (i.e., H_c's IP address) to only its port 3. In this way, we can translate ARP broadcast packets into unicast packets and save network bandwidth accordingly.

We remark that there are two features in the ARP transactor. First of all, it deals with only *one pair* of ARP request and ARP reply for every unknown host. Consequently, unlike most SDN-based methods mentioned in Section III-B, the controller will not serve as an ARP proxy to process each ARP packet, and its processing load and message overhead can be substantially reduced. Second, when the ARP transactor obtains the IP and MAC addresses of a host, it will install a flow entry to allow the OpenFlow switch translating the broadcast address into a unicast address. This design considers the backward compatibility to Ethernet switches. In particular, since the broadcast address has been translated into the unicast address, the Ethernet switch will not send the ARP packet to all its ports but only forward the packet to the port connected with the target host. Therefore, we can significantly diminish the number of superfluous packets caused the ARP mechanism. Note that when IPv6 is adopted, we can replace the IPv4 addresses (i.e., SPA and TPA) by IPv6 addresses in the aforementioned flow entries.

C. IGMP Transactor

In a LAN, IGMP allows hosts managing dynamic membership of their multicast groups. To do so, IGMP defines a special role, called *querier*, whose job is to keep sending *membership-query* packets to maintain a multicast group. The querier H_q has three actions, depending on the packet received by it:

- When H_q gets a membership-query packet sent from another host that has a smaller IP address, H_q gives up the role of querier. In this way, each multicast group can have just one querier to send membership-query packets.
- If H_q receives a *membership-report* packet before timeout, it forwards multicast data packets to the members in the group. Otherwise, H_q will stop sending data packets (since there are no other members in the group).
- When H_q acquires a *group-leave* packet, it terminates the transmission of multicast data packets.

When a host H_i in the multicast group gets a membership-query packet, H_i has to reply a membership-report packet to the querier. However, if H_i has ever heard membership-report packet(s) sent from others in the same group, it need not reply the membership-report packet. On the other hand, if a new host wants to join the multicast group, it actively sends a membership-report packet to notify the querier. In addition, a host can leave the multicast group without sending any packet to the querier. However, if the host is the last one that sent the membership-report packet, it has to send a group-leave packet to notify the querier to disband the multicast group.

Unfortunately, an Ethernet switch cannot well support IGMP but simply sends multicast data packets to all of its ports (i.e., broadcast). It is obvious that not every host wants to receive such packets. However, these hosts still need to process the irrelevant packets by their network-layer protocols, which results in extra processing loads. The IGMP snooping mechanism [24] uses a table for the Ethernet switch to map between ports and multicast traffics, so as to filter out irrelevant multicast data packets. However, it assumes that there is a multicast router to keep on generating membership-query packets, and these packets must be forwarded by all switches. In addition, not every Ethernet switch can offer the functions of IGMP snooping, as it is an optional mechanism.

In IGMP snooping, each switch should decide the relationship between ports and multicast packets on its own. Comparing with IGMP snooping, our IGMP transactor provides a more efficient mechanism to let the controller install flow entries learned from IGMP packets in OpenFlow switches to filter multicast traffics. Below, we discuss how the IGMP transactor process each type of IGMP packets.

1) Membership Query: Since this packet is transmitted by a querier H_q, the IGMP transactor can obtain its IP and MAC addresses, along with the port that H_q connects to the OpenFlow switch. Therefore, the IGMP transactor installs a flow entry to the OpenFlow switch to ask it to send subsequent IGMP packets to the controller and H_q:

- **Rules:** igmp, nw_dst=224.0.0.0/3
  Actions: actions=controller:6633, output:3

Specifically, this flow entry instructs the OpenFlow switch to send all IGMP packets with IP address of 224.0.0.0 to not only the controller but also the querier H_q which connects to the switch's 3rd port. Consequently, the controller can also get the information of the multicast group based on the subsequent IGMP packets. Moreover, the OpenFlow switch can relay IGMP packets such as membership reports to just H_q and the controller, which further reduces superfluous IGMP packets in the LAN.

2) Membership Report: Thanks to the flow entry mentioned in Section IV-C1, the controller can also acquire the membership-report packet transmitted from a member in the multicast group. However, to allow the controller getting the information of all members in the group, each member must reply the membership-report packet for the *first time* that it receives the membership-query packet. Then, the controller adopts a *group table* for the OpenFlow switch to record every member that the switch connects. Each group is assigned with one unique group identification (ID). In OpenFlow, the group ID can be calculated by converting the multicast IP address of the group into a decimal number. For example, if a multicast group is associated with IP address of 233.0.0.2, its group ID will be 3909091330. Suppose that the group has three hosts H_a, H_b, and H_c whose MAC addresses are 87:02:16:88:03:02, 87:02:16:88:03:04, and 87:02:16:88:03:06, respectively. Let H_a, H_b, and H_c connect to ports 2, 3, and 5 of the OpenFlow switch. Then, the controller adds an entry in the group table for the switch:

- **Group ID:** group_id=3909091330
  Bucket:
  bucket=actions=set_field:87:02:16:88:03:02
  ->eth_dst,output:2
  bucket=actions=set_field:87:02:16:88:03:04
  ->eth_dst,output:3
  bucket=actions=set_field:87:02:16:88:03:06
  ->eth_dst,output:5

The above entry will guide the OpenFlow switch to transmit multicast data packets to only the members in the multicast group (i.e., via its ports 2, 3, and 5).

3) Multicast Data: To avoid sending multicast data packets to the hosts that do not join the multicast group, the controller installs the following flow entry to ask the OpenFlow switch referring to the group table to transmit these packets:

- **Rules:** ip, nw_dst=233.0.0.2
  Actions: actions=group:3909091330

Since a multicast address can be adaptively translated into the unicast address of each member in the multicast group, the OpenFlow switch can forward multicast data packets to merely the ports that connect with member hosts. In this way, we can eliminate redundant multicast traffics. On the other hand, when the OpenFlow switch has *child* Ethernet switches, these Ethernet switches will get data packets with unicast IP addresses. Therefore, our IGMP transactor can also support backward compatibility with Ethernet.

4) Group leave: As discussed earlier, the group-leave packet is transmitted by the last member of the multicast group. Thus, when the controller gets the packet, the IGMP transactor removes the corresponding record in the group table (in other words, the multicast group is disbanded).

D. Default Transactor

The major job of the default transactor is to record the information of packet transmissions in the LAN. In particular, when an OpenFlow switch receives a packet that its flow table has no rules to be matched, an event of table miss will occur. In this case, the OpenFlow switch sends a Packet_In message to the controller. Then, the default transactor uses a *packet table* to store the information of that packet, including its data path identification (DPID), MAC address, and port number. Here, each device is associated with one unique DPID by the OpenFlow protocol to help the controller distinguish different devices in the LAN. The packet table provides statistics for network administrators to monitor traffic flows in the LAN. Besides, it also helps us develop other transactors to deal with different types of traffics in the future.

After recording the packet's information, the default transactor will reply a Packet_Out message to the OpenFlow switch to tell it how to process that packet. For example, suppose that the packet's destination is host H_i that has MAC address of 12:20:01:10:87:90 and connects to port 2 of the OpenFlow switch. Then, a flow entry will be installed as follows:

- **Rules:** dl_dst=12:20:01:10:87:90
  Actions: actions=output:2

Therefore, the OpenFlow switch will forward the packet to H_i via its port 2.

V. Simulation Study

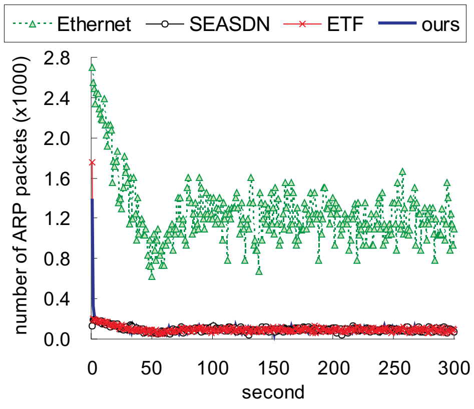
In this section, we measure LAN performance of our SDN framework through the Mininet simulator [25], which supports both the OpenFlow protocol and the Ryu platform. Our simulations consider a LAN with 50 hosts, where each host uses an ARP table to record the relationship between IP and MAC addresses that it obtains from ARP packets. According to Linux's configuration, each record in the ARP table will be removed after 1 minute. On the other hand, we consider two network topologies in our simulations:

- *Flat topology:* We connect all hosts by one switch. The switch is an Ethernet switch if we do not adopt SDN; otherwise, it is an OpenFlow switch. The flat topology is used to imitate large switches in certain LANs such as data center networks.
- *Tree topology:* We use three switches in this topology, which form a tree structure. The root switch has two child Ethernet switches, where each one connects 25 hosts. Similarly, the root switch is an Ethernet switch if we do not adopt SDN; otherwise, it is an OpenFlow switch. The tree topology is used to imitate a scenario where the network administrator wants to add some OpenFlow switches in a LAN. It can also help evaluate the degree of backward compatibility by different SDN-based methods.

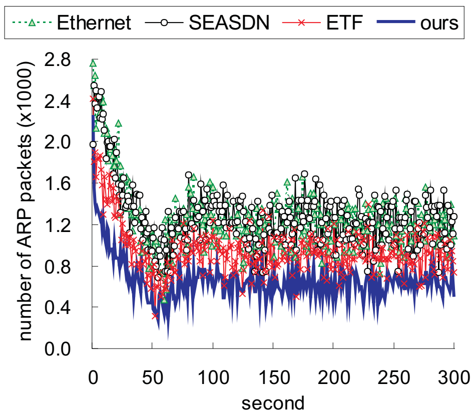
Except for Ethernet, we also compare our SDN framework with two SDN-based methods, SEASDN and ETF, discussed in Section III-B. The simulation time is set to 300 seconds.

We first measure the number of ARP packets produced per second in the flat topology, whose simulation results are given in Fig. 6(a). Because the expiration timer of each record in the ARP table is set to 1 minute, the number of ARP packets in Ethernet gradually decreases before the first 60 seconds and then keeps bumpy. For SEASDN, ETF, and our SDN framework, there is a small impulse in the beginning. The reason is that the controller has no host information initially, so it needs to receive ARP packets to get the information. After the controller collects each host's information (which spends no more than 2 seconds), it can ask the OpenFlow switch to diminish most ARP traffics (as they are redundant) to improve LAN performance.

Then, Fig. 6(b) presents the number of ARP packets produced per second in the tree topology. Since the two child Ethernet switches do not understand the flow entries set by the controller, they still broadcast each ARP packets. Consequently, the number of ARP packets sent by SDN-based methods increase accordingly. Since SEASDN asks the controller to serve as the ARP proxy, its performance is quite similar to that of Ethernet. ETF allows switches forwarding ARP packets to only the ports that link to target hosts. However, it does not convert the MAC address of these packets. Thus, the Ethernet switches still broadcast these ARP packets as usual. Comparing with both SEASDN and ETF, our SDN framework adaptively translates the broadcast address of



(a) flat topology



(b) tree topology

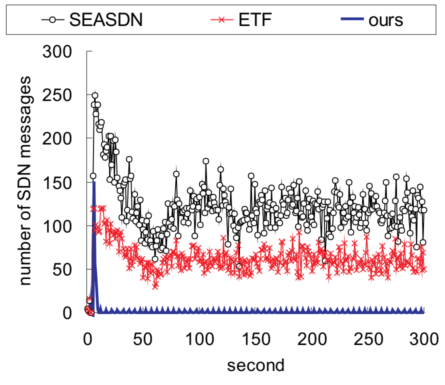
topology	SEASDN	ETF	ours
flat	92.03%	91.98%	92.37%
tree	-1.15%	24.80%	47.45%

(c) saving ratios of total ARP packets

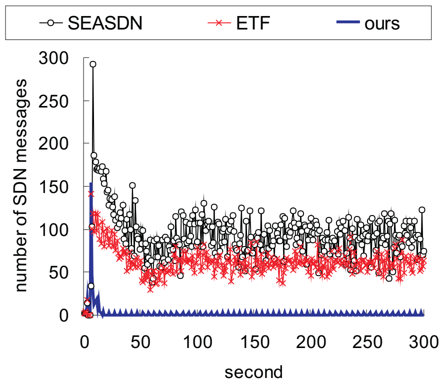
Fig. 6. Comparison on ARP packets produced by different methods.

each ARP packet into the unicast address of the target host. In this way, the Ethernet switches can forward these ARP packets to the right ports, which further reduces unnecessary ARP broadcasting traffics.

In Fig. 6(c), we compare the saving ratio of ARP packets by each SDN-based method on the basis of Ethernet. Obviously, a higher ratio means that the method can diminish more redundant ARP traffics and improve Ethernet's performance accordingly. In the flat topology, since the only switch can support the OpenFlow protocol, all of the three SDN-based methods can save more than 90% ARP packets, which demonstrates the superiority of using SDN. However, when there exist two Ethernet switches in the tree topology, SEASDN performs even worse than Ethernet. The reason is that these Ethernet switches simply forward the ARP packets produced by the controller to all of their ports. With the help of the address translation mechanism by the ARP transactor, our SDN framework can ask Ethernet switches to forward ARP packets (produced by the OpenFlow switch) to only the ports linking to target hosts. Therefore, it can save nearly 50%



(a) flat topology



(b) tree topology

topology	SEASDN	ETF
flat	98.74%	97.58%
tree	98.20%	97.30%

(c) saving ratios by our framework

Fig. 7. Comparison on the message cost of the SDN controller.

redundant ARP traffics in the tree topology, which verifies that our SDN framework supports much better backward compatibility than both SEASDN and ETF.

We then evaluate the cost of the controller imposed by SEASDN, ETF, and our SDN framework, which is measured by the number of SDN messages (i.e., Packet_In and Packet_Out) produced by these methods. Apparently, a large number of SDN messages implies that the SDN method gives a large burden to the controller. Fig. 7 gives the experimental results. In particular, SEASDN forces the controller to play the role of an ARP proxy, so it uses Packet_In messages to forward ARP requests to the controller and Packet_Out messages to forward ARP replies to the target hosts. In other words, each ARP procedure will produce 2 Packet_In messages and 2 Packet_Out messages in SEASDN, which imposes a heavy load on the controller. On the other hand, ETF also uses Packet_In messages to forward ARP packets to the controller, so it produces 2 Packet_In messages for each ARP procedure. In this case, ETF can save the message cost as comparing with SEASDN. In our SDN framework, the controller uses Packet_In messages to learn all hosts in the LAN and then adopts Packet_Out messages to set flow entries in OpenFlow switches in the beginning. That is why there exists an impulse (during the 6th to 9th seconds) in Fig. 7(a) and (b). Afterwards, our SDN framework produces very few SDN messages, because the controller has learned the mapping of IP and MAC addresses of each host in the LAN. Consequently, our SDN framework can save more than 98% and 97% message cost than SEASDN and ETF, respectively. Through the experimental results from both Fig. 6 and Fig. 7, we show that our SDN framework indeed provides a low-cost, high-efficiency approach to solve the broadcast problem caused by ARP.

TABLE II
COMPARISON ON IGMP TRAFFICS (IN PACKETS).

host	**IGMP control**		**multicast data**	
ID	Ethernet	ours	Ethernet	ours
H_1	305	3	300	300
H_2	305	220	300	0
H_3	305	0	300	0
H_4	305	3	300	300
H_5	305	219	300	0
H_6	305	0	300	0

Next, we investigate the effect of IGMP traffics on LAN performance. Since SEASDN and ETF do not address multicast traffics, we compare our SDN framework with only Ethernet. In the IGMP simulations, we use a multicast server and 6 hosts linked together by an OpenFlow switch. The server transmits multicast packets to the members of a multicast group G_1 in each second. Besides, we divide these hosts (denoted by $H_1 \sim H_6$) into 3 clusters:

- H_1 and H_4 join group G_1 at the 0th second and will not leave the group until the end of simulation (at the 300th second).
- H_2 and H_5 repeat the following operations in every 3-second period: 1) randomly join a multicast group other than G_1 for 2 seconds, 2) leave that group, and 3) keep idle for 1 second.
- H_3 and H_6 will not join any multicast group during the simulation period.

Table II presents the number of IGMP control and multicast data packets received by each host. According to the mechanism of Ethernet, the switch simply broadcasts each packet to all hosts. In this case, even though H_3 and H_6 do not join any multicast group, they still need to receive and process these irrelevant packets (since IGMP packets are defined in the network layer). In contrast to Ethernet, our SDN framework can help the controller adaptively learn the members of a multicast group and translate the multicast address into unicast address(es) accordingly. In particular, these hosts have different behaviors in our SDN framework as follows:

- Since H_1 and H_4 always join multicast group G_1, the OpenFlow switch transmits just 3 IGMP control packets (e.g., membership query) to them in the beginning. Then, each multicast packet will be sent to H_1 and H_4 via unicast.

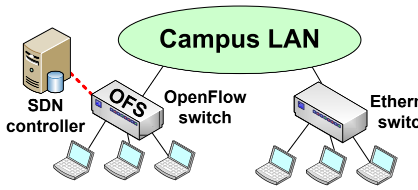


Fig. 8. Implementation of our SDN framework on the campus network.

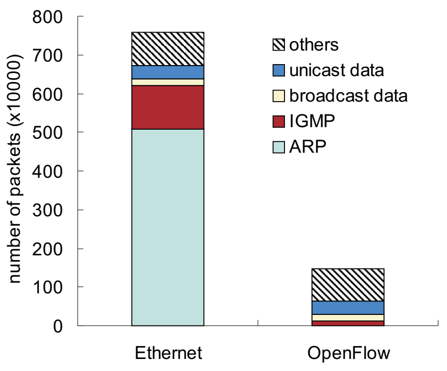


Fig. 9. Number of packets received by a host in each broadcast domain.

- As H_2 and H_5 randomly join a multicast group other than G_1 in every 3 seconds, the OpenFlow switch will send them more IGMP control packets but no multicast data packets. Note that our SDN framework can use fewer IGMP control packets than Ethernet to keep membership of both H_2 and H_5.
- Because H_3 and H_6 do not join any multicast group, there is no need for the OpenFlow switch to transmit IGMP control or multicast data packets to them.

From this experiment, we show that the IGMP transactor of our SDN framework can send multicast data packets to only the hosts which need them. Moreover, a host will not be bothered by the OpenFlow switch if it does not join any multicast group. In this way, our SDN framework can substantially reduce redundant multicast traffics, thereby saving network bandwidth and improving LAN performance.

VI. Framework Implementation

To verify the effectiveness of our SDN framework in a practical LAN, we implement it on our campus network discussed in Section I, whose network topology is given in Fig. 8. Specifically, we update the firmware of a TP-LINK WR1043NR switch by the OpenWrt module [26] to make it support the OpenFlow protocol. Then, we monitor network traffics in the two broadcast domains formed by the OpenFlow switch and the Ethernet switch in Fig. 8.

We measure the number of packets averagely received by one host in each domain during one day, whose results are presented in Fig. 9. It can be observed that both ARP and IGMP traffics produce a lot of packets in the broadcast domain formed by the Ethernet switch. On the contrary, the OpenFlow switch removes 99.93% and 89.74% of ARP and IGMP packets in its broadcast domain, respectively. In this experiment, we demonstrate that our SDN framework can reduce around 80.44% of redundant packets in a broadcast domain, which proves its effectiveness in a practical LAN.

VII. Conclusion and Future Work

Ethernet is popularly used for wired communications but produces many superfluous packets due to its broadcast nature. In this paper, we show that a large-scale Ethernet-based LAN will be inevitably flooded with ARP and IGMP traffics, and thus propose an SDN framework to remove redundant packets produced by these traffics. Through Mininet simulations, we verify that our SDN framework significantly saves the controller's cost and provides better backward compatibility than previous SDN-based methods including SEASDN and ETF. Moreover, the effectiveness and practicability of the proposed framework is also demonstrated by real implementation in our campus network.

We then discuss some directions for future work. First, it is also critical to provide fair transmissions in a LAN [27], where we should avoid some flows occupying the bandwidth so that other flows will not be starved. Thus, how to guarantee user fairness by SDN is a challenging issue, especially in a multi-rate environment [28]. Second, the DVB-H (digital video broadcasting-handheld) service becomes more popular to provide multimedia applications like mobile television [29]. It is interesting to apply the SDN technique to a DVB-H network to improve performance, where we have to quickly retransmit the lost DVB-H packets to support QoS (quality of service) [30]. Third, it deserves further investigation on how to adopt SDN to facilitate the management jobs of the core network in an LTE-A (long term evolution-advanced) system, including the issues of user billing [31], security management [32], and resource allocation [33]. Finally, since Wi-Fi systems have been widely deployed to form wireless LANs and provide various applications such as interactive shopping [34] and smart home [35], we will study how to use the SDN technique to improve their performance in the future.

Acknowledgement

Y.-C. Wang's research is co-sponsored by the Ministry of Science and Technology under Grant No. MOST 106-2221-E-110-022-MY2, Taiwan.

References

[1] IEEE, "IEEE Draft Standard for Ethernet," IEEE P802.3/D3.2, 2018.
[2] IETF RFC 826, "An Ethernet address resolution protocol," 1982.
[3] IETF RFC 3376, "Internet group management protocol," 2002.
[4] Architecture Technology Corpor, *NETBIOS Report and Reference*. Elsevier Science, 1991.
[5] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: a comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
[6] Y. C. Wang, F. J. Wu, and Y. C. Tseng, "Mobility management algorithms and applications for mobile sensor networks," *Wireless Communications and Mobile Computing*, vol. 12, no. 1, pp. 7–21, 2012.
[7] Y. C. Wang and S. J. Liu, "Minimum-cost deployment of adjustable readers to provide complete coverage of tags in RFID systems," *Journal of Systems and Software*, vol. 134, pp. 228–241, 2017.

[8] A. R. Khan, M. Othman, S. A. Madani, and S. U. Khan, “A survey of mobile cloud computing application models,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 393–413, 2014.

[9] Y. C. Wang and C. A. Chuang, “Efficient eNB deployment strategy for heterogeneous cells in 4G LTE systems,” *Computer Networks*, vol. 79, pp. 297–312, 2015.

[10] Y. C. Wang, “Mobile sensor networks: system hardware and dispatch software,” *ACM Computing Surveys*, vol. 47, no. 1, pp. 12:1–12:36, 2014.

[11] A. Lara, A. Kolasani, and B. Ramamurthy, “Network innovation using OpenFlow: a survey,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 493–512, 2014.

[12] Ryu. [Online]. Available: http://osrg.github.io/ryu/

[13] G. Parsons, “Ethernet bridging architecture,” *IEEE Communications Magazine*, vol. 45, no. 12, pp. 112–119, 2007.

[14] A. Myers, E. Ng, and H. Zhang, “Rethinking the service model: scaling Ethernet to a million nodes,” in *ACM SIGCOMM Workshop on Hot Topics in Networking*, 2004, pp. 1–6.

[15] C. Kim, M. Caesar, and J. Rexford, “Floodless in SEATTLE: a scalable Ethernet architecture for large enterprises,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 3–14, 2008.

[16] K. Elmeleegy and A. L. Cox, “EtherProxy: scaling Ethernet by suppressing broadcast traffic,” in *IEEE INFOCOM*, 2009, pp. 1584–1592.

[17] C. H. Chiu and C. L. Lei, “Etheragent: scaling ethernet for enterprise and campus networks,” *International Journal of Innovative Computing, Information and Control*, vol. 9, no. 6, pp. 2465–2483, 2013.

[18] B. Kercheval, *DHCP: A Guide to Dynamic TCP/IP Network Configuration Hardcover*. Prentice Hall, 1999.

[19] P. W. Chi, Y. C. Huang, J. W. Guo, and C. L. Lei, “Give me a broadcast-free network,” in *IEEE Global Communications Conference*, 2014, pp. 1968–1973.

[20] J. Wang, T. Huang, J. Liu, and Y. Liu, “A novel floodless service discovery mechanism designed for software-defined networking,” *China Communications*, vol. 11, no. 2, pp. 12–25, 2014.

[21] H. Cho, S. Kang, and Y. Lee, “Centralized ARP proxy server over SDN controller to cut down ARP broadcast in large-scale data center networks,” in *IEEE International Conference on Information Networking*, 2015, pp. 301–306.

[22] N. Jehan and A. M. Haneef, “Scalable Ethernet architecture using SDN by suppressing broadcast traffic,” in *International Conference on Advances in Computing and Communications*, 2015, pp. 24–27.

[23] K. Kataoka, N. Agarwal, and A. V. Kamath, “Scaling a broadcast domain of Ethernet: extensible transparent filter using SDN,” in *International Conference on Computer Communication and Networks*, 2014, pp. 1–8.

[24] IETF RFC 4541, “Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) snooping switches,” 2006.

[25] Mininet. [Online]. Available: http://mininet.org/

[26] OpenWrt. [Online]. Available: https://openwrt.org/

[27] Y. C. Wang, S. R. Ye, and Y. C. Tseng, “A fair scheduling algorithm with traffic classification in wireless networks,” *Computer Communications*, vol. 28, no. 10, pp. 1225–1239, 2005.

[28] Y. C. Wang, Y. C. Tseng, and W. T. Chen, “MR-FQ: a fair scheduling algorithm for wireless networks with variable transmission rates,” *Simulation*, vol. 81, no. 8, pp. 587–608, 2005.

[29] W. H. Yang, Y. C. Wang, Y. C. Tseng, and B. S. P. Lin, “A request control scheme for data recovery in DVB-IPDC systems with spatial and temporal packet loss,” *Wireless Communications and Mobile Computing*, vol. 13, no. 10, pp. 935–950, 2013.

[30] Y. C. Wang, “Profit-based exclusive-or coding algorithm for data retransmission in DVB-H with a recovery network,” *International Journal of Communication Systems*, vol. 28, no. 9, pp. 1580–1597, 2015.

[31] Y. C. Wang and T. Y. Tsai, “A pricing-aware resource scheduling framework for LTE networks,” *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1445–1458, 2017.

[32] J. Cao, M. Ma, H. Li, Y. Zhang, and Z. Luo, “A survey on security aspects for LTE and LTE-A networks,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 283–302, 2014.

[33] Y. C. Wang and S. Y. Hsieh, “Service-differentiated downlink flow scheduling to support QoS in long term evolution,” *Computer Networks*, vol. 94, pp. 344–359, 2016.

[34] Y. C. Wang and C. C. Yang, “3S-cart: a lightweight, interactive sensor-based cart for smart shopping in supermarkets,” *IEEE Sensors Journal*, vol. 16, no. 17, pp. 6774–6781, 2016.

[35] H. Jiang, C. Cai, X. Ma, Y. Yang, and J. Liu, “Smart home based on WiFi sensing: a survey,” *IEEE Access*, vol. 6, pp. 13 317–13 325, 2018.