IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 16, 2020, accepted May 9, 2020, date of publication May 12, 2020, date of current version May 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2994268

Topic-Oriented Bucket-Based Fast Multicast Routing in SDN-Like Publish/Subscribe Middleware

YULONG SHI[1,2], JONATHON WONG[2], HANS-ARNO JACOBSEN[2], (Fellow, IEEE), YANG ZHANG[1], AND JUNLIANG CHEN[1]

[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
[2]Middleware Systems Research Group, University of Toronto, Toronto, ON M5S 1A1, Canada

Corresponding author: Yulong Shi (shiyulong2015@bupt.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003800.

ABSTRACT In traditional IP-based publish/subscribe middlewares, a detour to overlay network is demanded to match events with defined filters, which introduces more latency overhead for delivering events from publishers to subscribers. The emerging Software Defined Networking (SDN) creates boundless possibilities to improve the efficiency of event delivery because of its centralized control and customized programmability. In this paper, we propose a topic-oriented bucket-based fast multicast routing to improve the efficiency of delivering events in SDN-like publish/subscribe middlewares. First, we design an SDN-like publish/subscribe middleware hierarchical architecture with an implementation framework in SDN controller to deliver events. Topic encoding, topic prefix matching, and the subscription coverage relationships between topics are considered to reduce the number of flow entries and improve the matching abilities of SDN switches. Then, a topic-oriented multicast tree construction algorithm is proposed to build publish/subscribe overlay networks with the minimal total link delay of event transmission and minimal switches in SDN controllers to deliver events fast. A topic-oriented bucket-based multicast forwarding algorithm is designed to achieve efficient multicast forwarding in SDN switches. Finally, experiments are conducted to verify that our construction algorithm has the minimal total delay of event transmission and our bucket-based multicast forwarding algorithm is effective.

INDEX TERMS IoT, publish/subscribe, middleware, SDN, bucket, multicast.

I. INTRODUCTION

In Internet of Things (IoT) scenarios, the publish/subscribe (pub/sub) paradigm is commonly exploited to establish the communication infrastructure for multiple clients to access tremendous real-time sensor data [1]–[5]. Software Defined Networking (SDN) [6]–[11] is used to solve the difficult problem of Quality of Service (QoS) guarantees of delivering events from publishers to subscribers in IoT. For traditional IP-based pub/sub middlewares [12], [13], event matching with defined filters takes more latency owing to a detour to broker network. However, event matching and forwarding can become more efficient in SDN-based pub/sub middlewares [14], [15] because these processes can be executed directly and fast on SDN-enabled switches via OpenFlow specifications [16] and OF-Config protocol [17].

In SDN, the control layer of the network is decoupled from the forwarding layer. The centralized control function of SDN controller makes it convenient to acquire the global network information (i.e., the whole network topology), which can be adopted to calculate event routing, deploy global security strategies, and make global QoS decisions. We can also take full advantage of the programmability of SDN to encode event priorities, topics, and security strategies into the flow tables of SDN switches to achieve personalized QoS guarantees. In our previous works of [18], [19], topic priorities and authorization strategies are encoded into flow tables to provide differentiated IoT services and cross-layer access control for clients, respectively in our SDN-like [20] pub/sub middleware [21]. Therefore, SDN can be used to simplify

The associate editor coordinating the review of this manuscript and approving it for publication was Vyasa Sai.

the design and management of networks and enhance their flexibility and scalability. In this paper, we focus on designing an efficient fast multicast routing to improve the efficiency of event delivery in pub/sub middlewares over SDN networks.

One huge challenge in SDN-like pub/sub middlewares is how to minimize the cost of event transmission from publishers to subscribers for delivering events efficiently. Every event should be transmitted only once along each related link, which is a multiple source multicast communication. Pub/sub overlay networks, usually topic-oriented multicast trees need to be constructed to cover all publishers and subscribers in SDN controllers. There are three common methods to construct multicast trees for SDN-like pub/sub routing.

A. SHORTEST PATH TREE (SPT)

The SPT algorithm [22] can minimize the cost of each *(publisher, subscriber)* pair. However, it can only guarantee that the local cost is optimal and cannot guarantee that the global cost is optimal for all *(publisher, subscriber)* pairs. Moreover, this algorithm also restricts the reuse of common paths between multicast trees. When every new advertisement or subscription arrives, new paths or multicast trees are added for related pub/sub nodes, increasing the overhead of dynamic routing computations.

B. MINIMUM SPANNING TREE (MST)

The MST algorithm [23] can be used to construct a single spanning tree across all subscribers and publishers. The forwarding times of events decrease obviously because all pub/sub paths are generated in one tree. However, the critical links can be overloaded, and the delay may increase due to a few edge selections for new paths.

C. STEINER TREE

The Steiner tree [24] can minimize the total link cost of event transmission in network. It has a smaller total cost than the MST because it allows nodes that are not publishers or subscribers (extra nodes) to join the pub/sub transmission network. The Steiner tree problem is NP-hard [25]. In this paper, we improve the classic heuristic KMB algorithm [26] to get an approximate solution, which has a lower time complexity and a higher approximation rate.

Therefore, we can use the Steiner tree to construct topic-oriented pub/sub overlay with the minimal total link delay of event delivery, forming a fast multicast routing. Moreover, we improve the shortest path in the KMB algorithm to choose the paths with fewer extra nodes (switches) between the shortest path and second shortest path to reduce the number of flow entries in SDN switches, further enhancing the matching efficiency of flow tables in SDN-like pub/sub middlewares. The work of [27] proposed a minimum topic-connected overlay (Min-TCO) to minimize the number of links without considering weighted links. The work of [28] proved the feasibility of the Steiner tree routing for pub/sub networks by using the KMB algorithm directly to minimize the total delay of event delivery. However, we improve the algorithm by modifying the shortest path in it to minimize not only the delay but also the number of extra nodes, achieving fast event delivery and reducing the number of flow entries.

Another huge challenge in SDN-like pub/sub middlewares is how to implement an efficient topic-oriented multicast forwarding in SDN switches based on pub/sub overlay. Many pub/sub middlewares adopted IP multicast addresses [18], [19], [29] to realize network level multicast. However, IP multicast does not have good scalability for large groups, and it takes more latency for event matching due to a detour routing to brokers. The works of [30] and [27] described application level multicast for event notification services. However, event delivery at the application level is less efficient than at the network level [31].

In SDN, events are matched and forwarded directly, fast and controllably in SDN switches with OpenFlow, which supports the installation and modification of flow tables in SDN switches to reduce the delay of event delivery. A common way to realize multicast is writing multiple actions into the same action set for a matching flow entry, but it is not convenient for group management because no group concept is used. In order to address this issue, we propose a new way that uses the action buckets of group tables in OpenFlow to achieve efficient topic-oriented bucket-based multicast forwarding in SDN switches, improving the efficiency of event forwarding and facilitating group management.

Topic design is also an important issue in topic-oriented SDN-like pub/sub middlewares. In our work, topics are organized into a Lightweight Directory Access Protocol (LDAP) topic tree with the Huffman coding to prevent topic explosion. Topic prefix matching based on the parent-child relationships between topics is presented to enhance the matching efficiency of topic events. In order to map multiple 64-bit topics to 32-bit group IDs, the subscription coverage relationships between topics are considered to merge flow entries in SDN switches, further reducing the number of flow entries and improving the matching efficiency of flow tables.

In this paper, we propose a topic-oriented bucket-based fast multicast routing in SDN-like pub/sub middlewares, aiming at improving the efficiency of delivering events between the publishers and subscribers of messages in IoT.

The major contributions of this paper are as follows:

(1) We propose an SDN-like pub/sub middleware hierarchical architecture and an implementation framework in SDN controller with topics encoded into the flow entries of SDN switches for directly and fast matching to deliver events efficiently in IoT scenarios.

(2) A topic representation method is presented with an efficient topic encoding to present topic explosion. Topic prefix matching is designed to improve the matching efficiency of flow tables in SDN switches, and the subscription coverage relationships between topics are considered to merge flow entries in SDN switches to map multiple topics to limited group IDs, further reducing the number of flow entries and improving the matching abilities of SDN switches.

(3) We propose a topic-oriented bucket-based fast multicast routing algorithm to improve the efficiency of delivering events between publishers and subscribers in SDN-like pub/sub middlewares. On one hand, a topic-oriented Steiner tree multicast routing algorithm with improved shortest path algorithm is designed to construct pub/sub overlay networks about multiple topics in SDN controllers, which can minimize the total link delay of event transmission and have minimal switches for multiple event streams, realizing a real-time fast multicast routing to improve the efficiency of event delivery, and reducing the number of flow entries to save the storage space of flow tables in SDN switches. On the other hand, a topic-oriented bucket-based multicast forwarding algorithm with OpenFlow is also designed to improve the efficiency of event forwarding in SDN switches. The forwarding algorithm considers the subscription coverage relationships between topics to merge the flow entries of SDN switches, further reducing the number of flow entries and improving the matching efficiency of SDN switches. These two algorithms and the SDN-like design together compose our topic-oriented bucket-based fast multicast routing in SDN-like pub/sub middlewares.

(4) Experiments are conducted to verify our topic-oriented Steiner tree multicast routing algorithm by comparisons with the MST and SPT algorithms. We also perform some experiments to validate our topic-oriented bucket-based multicast forwarding algorithm by a contrast of group and no group. Experimental results show that our algorithms are effective.

The remainder of this paper is organized as follows. Section II introduces the related work. Section III proposes the system design of SDN-like pub/sub middlewares. Section IV presents the topic-oriented Steiner tree multicast routing algorithm. Section V proposes our topic-oriented bucket-based multicast forwarding algorithm. Section VI presents experimental evaluations. Finally, we conclude this paper with an outlook on future research in Section VII.

II. RELATED WORK

There are many significant works for pub/sub middlewares. They are separated into several types based on different subscription schemes. For instance, VCube-PS [32], PICADOR [33], and Poldercast [34] are popular topic-oriented pub/sub middlewares. PhSIH [35] and PADRES [36] are famous content-oriented pub/sub middlewares. Flexpath [37] is a classic typed-oriented pub/sub middlewares. The topic-oriented pub/sub middlewares cost less runtime overhead and are easy to develop, which are very appropriate for IoT services. However, these middlewares are built on overlay networks, underlying switches are difficult to control by publishers/subscribers and a detour to overlay network is needed for routing, therefore, the routing of delivering events is not efficient enough.

With the development of IoT, open source commercially supported Message-Oriented Middlewares (MOMs) are becoming more and more popular. The famous pub/sub bus RabbitMQ [38] is an open source widely deployed message broker, which is an implementation of the Advanced Message Queuing Protocol (AMQP). It is featured by reliability, high availability, clustering, and fault tolerance. RabbitMQ has a powerful routing function, supporting multiple exchange categories such as direct exchange, topic exchange, and fanout exchange. Many studies have verified its good performance. The work of [39] indicates that RabbitMQ is more stable than the Representational State Transfer (REST) API approach under substantial concurrent client requests about microservices. The work of [40] shows that RabbitMQ has a better throughput than Kafka [41]. However, RabbitMQ has distributed consistency issues. Redis (Remote Dictionary Service) [42] is an open source memory-based storage middleware, which is very suitable for frequent search scenarios and often adopted as message broker, database and cache. It can be used to build middleware cluster to improve the scalability of IoT middleware [43]. However, Redis does not have automatic fault tolerance and recovery functions.

In recent years, SDN are increasingly popular due to its customized programmability and flexibility. However, there are few works about SDN-like pub/sub middlewares. PLEROMA [44] is an SDN-oriented pub/sub middleware, which adopts the Ternary Content Addressable Memory (TCAM) of switches to implement the line-rate forwarding of events. However, the storage space of TCAM is scarce and the cost is very expensive. In the work of [45], authors proposed a data-centric SDN-based pub/sub middleware POSEIDON by proactive overlay to improve the capabilities of data delivery. A load balancing algorithm was presented in [46] to realize the minimal forwarding cost for topic overlay network in SDN-based pub/sub systems.

The routing problems of pub/sub middlewares contain routing selection and event forwarding. Routing selection means constructing a multicast tree for pub/sub networks to find paths from publishers to subscribers. LIPSIN [47] proposed a line speed pub/sub network which adopted the SPT [48] and Bloomed link identifiers to realize energy efficient forwarding. For each publisher, there is a SPT for multicast; For all publishers, there are multiple per-source SPTs. However, the SPT algorithm cannot acquire the whole optimal cost. PADRES [36] introduced the MST multicast routing to forward content-based events. A single spanning tree is created to reduce forwarding times. However, the key links may be overloaded and the construction delay of new paths may increase [49]. To overcome these shortcomings, the famous SDN-based middleware PLEROMA [44] presented the method of multiple spanning trees. For new advertisements, the middleware renews spanning trees and decides trees to which a publisher can forward messages for link load balancing. Moreover, for new subscriptions, it only uses edges associated with subscribers and publishers to reduce path length, saving the delay of event delivery. However, the total cost for event transmission is not optimal.

The Steiner tree can minimize the total cost of event transmission. There are two kinds of algorithms to construct the Steiner multicast tree. The first kind of algorithms is

to find the optimal solution. Hakimi proposed the spanning tree enumeration algorithm in [50], but it has a very high time complexity $O(2^{|V|-|S|})$, where V is the vertex set of network graph, S is the set of publishers and subscribers. Aho *et al.* presented a dynamic programming algorithm [51] with a high space complexity $O(|S| * 8^{|S|})$. The second kind of algorithms is to get approximate solutions. The basic principle is to reduce the complexity by sacrificing accuracy. Takashami and Matsuyama proposed the nearest participant first greedy algorithm in [52]. The time complexity is $O(|S| * |V|^2)$, and the approximation ratio is less than or equal to 2. Kou *et al.* presented the classic KMB algorithm [26]. The time complexity is $O(|S| * |V|^2)$, and the approximation ratio is $2-2/k$, where k is the leaf number in the optimal tree [53]. Mehlhorn [54] introduced another implementation of the KMB algorithm. It has a time complexity $O(|V| * log|V| + |E|)$ (E is the edge set of network graph), and the approximation ratio is 2. In this paper, we select the KMB algorithm because it has a lower time complexity and a higher approximation rate.

There are three approaches to achieve multicast forwarding about events in pub/sub middlewares. Namely, network level multicast, application level multicast and SDN-based multicast. In the work of [29], authors used IP multicast addresses to realize network level multicast in pub/sub communication schemes. However, IP multicast is difficult to scale and costs more time to match events because a detour to broker network. In the famous middleware SCRIBE [30], authors proposed application level multicast methods for event notification services. However, it is less efficient than in the network layer [31]. In SDN, events matching and forwarding can be executed directly in SDN switches, reducing the end-to-end delay for delivering events. The work of [55] described an SDN based multicast method in pub/sub networks, making full use of the matching abilities of SDN switches.

There are also some works similar to ours. The work of [28] verified the feasibility of the Steiner tree routing to deliver messages in real-time for pub/sub applications in the Future Internet by using the KMB algorithm directly. However, we improve the KMB algorithm by modifying the shortest path in it to minimize the total delay of event delivery and the number of switches in paths, realizing fast delivery of events and less flow entries in SDN switches. The work of [56] proposed the Branch-aware Steiner Tree (BST) to improve the scalability of pub/sub multicast in SDN by minimizing the total numbers of branch nodes and edges, which is different from our optimization goals. The work of [19] presented a policy-driven pub/sub topology construction about many topics by excluding unauthorized nodes to minimize the total delay of event delivery. In a word, our optimization objectives are different from them about the pub/sub overlay construction. In the work of [55], authors implemented an SDN-based multicast in pub/sub middlewares to reduce end-to-end delay and flow table size. However, they did not consider using the subscription coverage relationships between topics to reduce the flow entries of SDN switches.

In this paper, we consider the pub/sub routing problem comprehensively. Namely, the topic-oriented Steiner multicast tree construction for fast routing and the topic-oriented bucket-based multicast for efficient event forwarding are considered together to implement a topic-oriented bucket-based fast multicast routing in SDN-like pub/sub middlewares.

III. SYSTEM DESIGN

In IoT environments, pub/sub middleware is required to construct an IoT communication infrastructure aimed at seamlessly interconnecting heterogeneous networks with IoT applications. With the popularity of SDN, how to design the next generation pub/sub middleware becomes an urgent problem. In this section, we propose an SDN-like pub/sub middleware architecture with an implementation framework in SDN controller to design our SDN-like pub/sub middleware, and discuss topic design including topic encoding and matching in our topic-oriented SDN-like pub/sub middleware.

A. SDN-LIKE PUBLISH/SUBSCRIBE MIDDLEWARE ARCHITECTURE

We propose an SDN-like pub/sub middleware architecture, *as illustrated in* Fig. 1. The SDN network is separated into several clusters (brokers, partitions, domains or nodes). Each cluster is a relatively independent network area, which is composed of a local controller, several SDN-enabled switches, and clients (publishers or subscribers). Neighbour clusters are interconnected by a pair of border switches. Therefore, the topology is a hierarchical structure, which is very suitable for deploying distributed large-scale IoT services. One layer is intra-cluster topology, the other layer is inter-cluster topology. In this paper, we adopt the topic-oriented pub/sub interaction model [57]. An event (message) is identified by a topic exclusively, consisting of a topic name and a *(attribute, value)* pair. All topics constitute a topic tree.

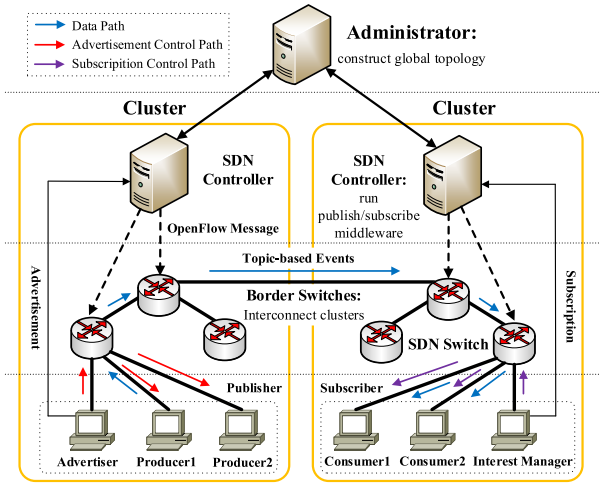


FIGURE 1. Our SDN-like publish/subscribe middleware architecture.

Our SDN-like pub/sub middleware is composed of four layers, *as illustrated in* Fig. 1.

(1) *Global Management Layer:* The global management layer only includes one global administrator, which is used to manage all SDN controllers. It runs on a server to build topic trees for events, the whole network topology, security strategies and QoS guarantee mechanisms.

(2) *Control Layer:* This layer is composed of SDN controllers. Its main function is to run the SDN-like pub/sub middleware. Each SDN controller manages the cluster in its domain, maintains link states, computes routings for event flows according to subscriptions, advertisements and the network topology and installs flow forwarding rules into SDN-enabled switches.

(3) *Data Layer:* The data layer contains SDN-enabled switches, which forward event flows in line with the flow matching rules installed in their flow tables. The matching rules for flow entries can be encoded into MAC addresses, IP addresses or VLAN tags [16] to match against the packet header fields of events, such as IPv4 or IPv6 addresses.

In this paper, we encode event types, priorities, security policies and topics into IPv6 multicast addresses as the matching fields of event flows, *as illustrated in* Fig. 2. Topics are encoded as 64-bit binary strings of IPv6 destination addresses. If the matching fields of flow entries can match with the IP multicast address of topic events, the events will be forwarded to the specific output ports of switches; otherwise they will be dropped.

(4) *Access Layer:* This layer consists of clients, that is to say, subscribers or publishers. The main function is to offer the local access interfaces for handling events, i.e., the interfaces to publish or receive events.

In traditional pub/sub middlewares [13], [35], clients are divided into publishers and subscribers. In our SDN-like pub/sub middleware, publishers are further decoupled into producers and advertisers; subscribers are further decoupled into consumers and interest managers. Advertisers and interest managers are responsible for advertisements and subscriptions (control plane), respectively. Producers and consumers are responsible for the productions and consumptions (data plane) of publications, respectively. This decoupling of the control and data planes in clients can provide more powerful functionality for pub/sub middlewares [20], i.e., we can configure dedicated access control mechanisms or security policies for publishers and subscribers.

B. SDN-LIKE PUBLISH/SUBSCRIBE IMPLEMENTATION FRAMEWORK

In this section, we propose our SDN-like pub/sub implementation framework with Ryu controller [58] based on the architecture presented in Section III-A, *as shown in* Algorithm 1. A complete publish/subscribe process is as follows:

Step 1: *Topology discovery.* SDN controllers capture switches, hosts and links in their network autonomous domains by the Link Layer Discovery Protocol (LLDP). The switch ID, port number, and the connection relationships

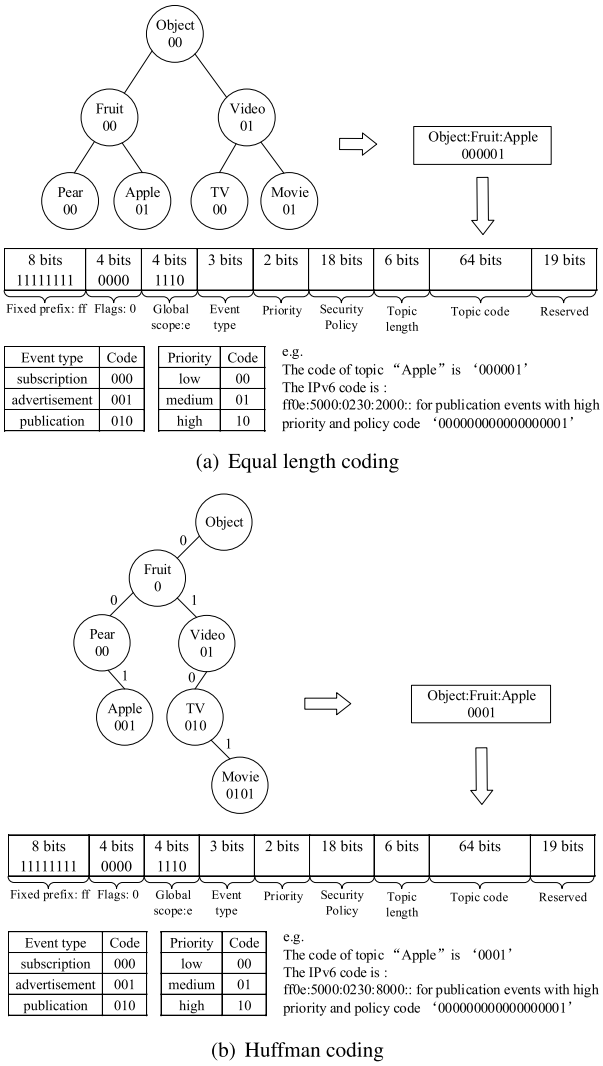


FIGURE 2. Topic encoding.

of switches are collected by SDN controllers. Then they exchange network topology information with each other and upload the information to the administrator of systems. In this way, SDN controllers can get the global network topology and know the whole network status information.

During this process, we should create network topology and start SDN controllers first. There are three important events, *as shown in* lines 2–8 of Algorithm 1. (1) *SwitchFeatures events.* A table-miss flow entry should be installed on switches by SDN controllers when events come to switches for the first time. (2) *StateChange events.* The states of switches have changed. It means that switches join or leave the network, and they should register with or logout SDN controllers. (3) *SwitchEnter events.* Switches enter the network. SDN controllers discover (capture) switches, links and connected ports in their autonomous domains, the local network topology is obtained.

Step 2: *PacketIn event processing.* After topology discovery, packets enter the network and are processed, *as shown*

Algorithm 1 SDN-Like Pub/Sub Implementation Framework

Input: Network topology, SDN controller Ryu, *Publishers*, *Subscribers*

Output: SDN controller adds group tables and flow tables to SDN switches, event flows are forwarded from *Publishers* to *Subscribers*

```
1: Initialize Create network topology, start SDN controller Ryu
2: @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
3: function switch_features_handler(event)
4:     add_flow(datapath, 0, match, output to controller)
5: @set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
6: state_change_handler(event)
7: @set_ev_cls(event.EventSwitchEnter)
8: get_topology_data(event)
9: @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
10: function packet_in_handler(event)
11:     Get IPv6, Ethernet packets by event (message) parsing
12:     Calculate message type, ip by packet parsing
13:     if message type is Advertisement then
14:         function advertisement_handler(event)
15:             Add publisher nodes to topology
16:             if find subscription matching event topic then
17:                 path ← find_path(publishers, topic, type)
18:                 add path about new advertisement to paths
19:     else if message type is Subscription then
20:         function subscription_handler(event)
21:             Add subscriber nodes to topology
22:             if find advertisement matching event topic then
23:                 path ← find_path(subscribers, topic, type)
24:                 add path about new subscription to paths
25:     else if message type is Publication then
26:         publication_handler(event)        ▷ Not run forever
27:     if len(paths) > 0 then        ▷ Bucket multicast along paths
28:         Bucket_based_multicast(paths, topic)
```

in lines 9–28. First, events are parsed into IPv6 packets, message type can be acquired by packet parsing. Then, events are processed according to their message types. When the message type is *Advertisement* or *Subscription*, pub/sub paths about the same topic is computed from publishers to subscribers by SDN controllers (Step 3). When the message type is *Publication*, events are delivered from publishers to subscribers (Step 5). At last, flow tables and group tables are installed on switches for event forwarding according to the paths (Step 4).

Step 3: *Routing computation.* After acquiring the network topology, we can use it to compute paths from publishers to subscribers. The commonly used routing algorithms are Flooding, IP multicast, the Shortest Path First (SPF) and the Minimum Spanning Tree (MST). When SDN controllers receive an advertisement message which announces a publisher will publish specific events to SDN-like pub/sub systems, they compute the related paths from the publisher to all subscribers who express their interests in the events beforehand in lines 13–18. Similarly, when SDN controllers receive a subscription message which announces a subscriber is interested in specific events, paths are computed from the subscriber to all publishers who can publish the events in lines 19–24. The key issue is how to compute efficient event transmission paths between publishers and subscribers by function *find_path* in lines 17 and 23, which will be discussed in detail in Section IV.

Step 4: *Flow table installation.* For each switch on forwarding paths, flow forwarding rules should be installed on it to decide the output ports of the switch for event forwarding. We propose a topic-oriented bucket-based multicast algorithm, which use topic-oriented action buckets to install group tables and flow tables into switches by SDN controllers in lines 27–28, *as described in* Section V.

Step 5: *Publication and subscription.* Publishers publish events, subscribers receive the events if they subscribe to them in advance. Switches are responsible for event forwarding by installed flow tables. SDN controllers will do nothing in this process in lines 25–26. All publication code will not run, because the code cannot reach SDN controllers, events are handled by switches directly. SDN controllers are only responsible for control and SDN switches become simple packet forwarding devices, which simplifies network design and management. This also reflects the idea of separation between the control and forwarding planes in SDN.

C. TOPIC DESIGN

In topic-oriented SDN-like pub/sub middlewares, one hard issue is how to organize topics into a topic tree with optimized topic encoding to prevent topic explosion, the other is how to match topic events against the flow tables of SDN switches efficiently. In order to solve these issues. We discuss the equal length coding and Huffman coding by the parent-child relationships between topics, and design an efficient topic prefix matching method based on the subscription coverage relationships between them.

In our topic-oriented SDN-like pub/sub middleware, events are identified uniquely by topics. Topics are represented as a Lightweight Directory Access Protocol (LDAP) topic tree. Parent-child relationship exists between topics in the adjacent layers of topic tree. If a parent topic is subscribed, all child topics are subscribed by default, which is called subscription coverage. We can use this subscription coverage relationships between topics to merge flow entries in SDN

Algorithm 2 Topic Encoding in Event Header by Publisher

Input: A LDAP topic tree T, a topic *Topic* in T

Output: Topic code *TopicCode*, IPv6 code corresponding to topic *TopicIPv6Code*

1: Compute the Huffman Coding of T
2: *TopicPath* ← *GetTopicPath*(*Topic*)
3: **for** *Node* in *TopicPath* **do**
4: *TopicCode* ← *TopicCode* + *NodeCode*
5: *TopicIPv6Code* ← "ff0e" + *EventType* + *Priority* + *SecurityPolicy* + *TopicLength* + *TopicCode* + *Reserved*

switches, reducing the number of flow entries and enhancing the matching efficiency of flow tables.

1) TOPIC ENCODING

In order to prevent topic explosion, a topic encoding method is needed to embed topic into the header of event by publishers. One way is the equal length coding, another way is the Huffman coding, *as illustrated in* Fig. 2. Event type, priority, security policy and topic are embedded into IPv6 multicast addresses. Topics are represented as 64-bit binary strings, which can be used to match the flow entries of SDN switches for event forwarding directly. For example, if we use the equal length coding, the topic code of *Apple* is 000001, and the IPv6 code will be ff0e:5000:0230:2000:: for publication events with high priority and policy code 0000000000000000001. If we use the Huffman coding, the topic code of *Apple* is 0001, and the IPv6 code will be ff0e:5000:0230:8000::. In fact, the Huffman coding can use fewer bits to encode topics than the equal length coding with a little more computation overhead. Therefore, we adopt the Huffman coding described in Fig. 2(b) to encode topic.

We propose our topic encoding algorithm executed in the topic code fields of event headers by publishers, *as illustrated in* Algorithm 2. First, we compute the Huffman coding of topic tree in line 1. Then, topic path is calculated by traversing the topic tree in level order in line 2. In the for loop of line 3–4, topic code is computed by joining the coding of each node in the topic path. At last, we get the IPv6 code of the topic by connecting the topic code with other fields in line 5.

2) TOPIC MATCHING

In our SDN-like pub/sub middleware, event topics are encoded into event headers and used to match the flow entries of SDN switches, solving the problem of routing selection in the data link layer directly. The more bits the topic matches, the better the matching accuracy will be, but the flow table size will be larger and the matching speed will be slower, and vice versa. Therefore, it is necessary to strike a balance between matching accuracy and speed, i.e., we can use all topic bits to match flow entries, but a new flow entry should be added for each topic. The matching accuracy is best, but the flow table size will be very large with a lower matching speed. However, the storage space in SDN switches is very limited,

Algorithm 3 Topic Matching in Flow Entry by SDN Controller and Switch

Input: Topic publication event *Pub*

Output: Topic matching successful, events are forwarded, or failed

1: *IPv6Addr* ← *PublicationEventParsing*(*Pub*)
2: *TopicCode* ← *TopicParsing*(*IPv6Addr*)
3: *mask* ← ff:ff:ff:ff ▷ 32 bit mask matching
4: **if** *TopicCode* ⊗ *mask* = *TopicMatingField* **then** ▷ ⊗ is bitwise AND
5: **return** Topic matching successful
6: **else**
7: **goto** Next flow entry
8: **return** Topic matching failed

exact topic matching with all bits is unrealistic in practise. According to the subscription coverage relationships between topics, we can use mask to define the bits of topic matching like the prefix match for IP addresses, called as topic prefix matching.

The algorithm of topic matching is proposed in Algorithm 3. First, topic publication event is parsed into IPv6 multicast address and topic code in lines 1–2. The mask of topic matching field is defined by SDN controller in line 3. In this paper, the mask is set to a 32-bit string ff:ff:ff:ff. The filed of topic matching in the flow entries of SDN switch is a bitwise AND operation of topic code and the mask in line 4. In this way, topics with subscription coverage relationships have the same topic matching filed, namely, the 32-bit prefix of topic codes, which can be mapped into the same group ID for multicast directly, reducing the number of groups and saving the space of group tables and flow tables.

IV. TOPIC-ORIENTED STEINER TREE MULTICAST ROUTING

In the SDN-like pub/sub implementation framework Algorithm 1, when the message type is advertisement or subscription, paths from publishers to subscribers should be found by SDN controllers, which is called routing (path) selection realized by function *find_path* in lines 17 and 23 of Algorithm 1. Routing problems include routing selection and event forwarding. In this section, we focus on routing selection, namely, how to select optimal paths from publishers to subscribers to deliver events efficiently in topic-oriented SDN-like pub/sub middlewares. This problem is also called as the construction of pub/sub overlay network.

The work of [27] proposed a minimum topic-connected overlay to trade off the scalability and forwarding overhead. The work of [19] presented a policy-driven pub/sub topology construction about many topics via bypassing the unauthorized nodes. In this paper, we focus on constructing the topic-oriented pub/sub overlay with the minimum cost of event transmission and the minimal extra nodes to reduce the overhead of flow entries in SDN switches, as shown below:

1) In order to transmit events fast and efficiently in SDN-like pub/sub middlewares, the total cost of event transmission should be minimized, the cost can be link delay, bandwidth and switch hops. Many IoT services are delay-sensitive, so we choose delay as the weight (cost) of each edge in pub/sub networks. Our goal is to find several minimum cost multicast trees covering all nodes of publishers and subscribers.

2) Many event streams can cross the same extra node, namely, non-publisher and non-subscriber node. Extra nodes and publisher/subscriber nodes consist of topic-oriented overlay topology. Our goal is to minimize extra nodes for multiple event streams to reduce the number of flow entries for SDN switches.

A. PROBLEM STATEMENT

We use an undirected weighted connected graph $G = (V, E)$ to describe the SDN-like pub/sub network, where $V = \{v_i | 1 \leq i \leq n\}$ is the node set, $E = \{e_j | 1 \leq j \leq m\}$ is the edge set with a link cost c_j about e_j. If $W = \{w_t | 1 \leq t \leq p\}$ is used to denote the event stream set about p topics subscribed by several nodes in G, $M_t = (S_t, D_t, w_t)$ can represent the multicast from the node set of publishers S_t to the node set of subscribers D_t about event stream w_t with a multicast cost $cost(M_t)$.

Definition 1 (MCMN-TC-SDN): The problem of publish/subscribe Topology Construction about multiple topics to Minimize the total Cost of event transmission and Minimize extra Nodes in Topic-oriented SDN-like publish/subscribe middlewares is called as MCMN-TC-SDN, which is defined as follows:

Given an SDN network $G(V, E)$ with multicast set $M = \{M_t | 1 \leq t \leq p\} = \{(S_t, D_t, w_t) | 1 \leq t \leq p\}$ from publisher set S_t to subscriber set D_t about event stream w_t for p topics. The goal of MCMN-TC-SDN is to find several sub-graphs $G_t = (V_t, E_t), V_t = \{v_{ti} | 1 \leq i \leq n_t\}, E_t = \{e_{tj} | 1 \leq j \leq m_t\}, 1 \leq t \leq p$ with link cost c_{tj} about e_{tj} to connect all publishers and subscribers, and each multicast M_t should satisfy the following formulas:

$$minimize(|\bigcup_{t=1}^{p}(V_t - S_t - D_t)|) \tag{1}$$

$$minimize(\sum_{t=1}^{p} cost(G_t) | cost(G_t) = \sum_{j=1}^{m_t} c_{tj}) \tag{2}$$

The goal of MCMN-TC-SDN is to connect all publishers and subscribers with the minimal total cost of event transmission and the minimal extra nodes for all multicast streams in SDN-like topic-oriented publish/subscribe middlewares.

Theorem 1: The problem of MCMN-TC-SDN is NP-hard.

Proof: In formula (1), our goal is to minimize the total extra nodes except publisher/subscriber nodes. If we remove this goal, the problem of MCMN-TC-SDN will be the classic Steiner tree problem [24] in the case of considering only formula (2). Therefore, our goal of MCMN-TC-SDN can be viewed as a special case of the Steiner tree. The Steiner tree problem is NP-hard [25], so the problem of MCMN-TC-SDN is also NP-hard. ■

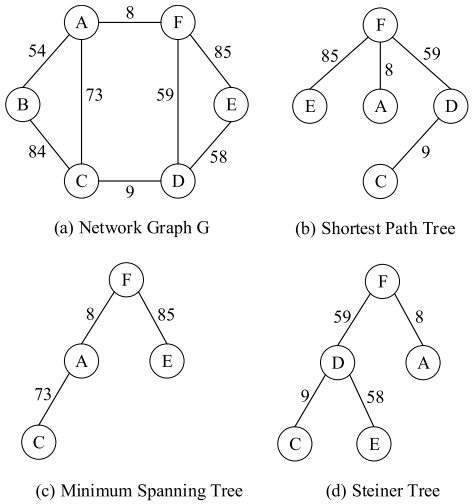


FIGURE 3. An example of multicast tree.

In graph theory, the Steiner tree [24] is a minimum cost tree, which can minimize the consumption of network resources. The Steiner tree can contain some Steiner (extra) nodes, so the minimum cost can be better than the MST and SPT, *as illustrated in* Fig. 3. Fig. 3(a) is the network graph G. The weights of edges are randomly generated, ranging from (0, 100). Node F is a publisher, nodes A, C and E are subscribers. Fig. 3(b) is a SPT with a total cost 161. Fig. 3(c) is a MST with a total cost 166. Fig. 3(d) is a Steiner tree with a total cost 134. The terminal nodes are F, A, C and E, namely, publishers and subscribers. The Steiner node is D, namely, non-terminal nodes. Obviously, compared with the SPT and MST, the Steiner tree has a minimum cost.

B. SOLVING MCMN-TC-SDN

For each multicast stream $M_t = (S_t, D_t, w_t)$ about $topic_t$ in SDN network $G(V, E)$, we first calculate the shortest paths and the second shortest paths between S_t and D_t ($S_t \subseteq V$, $D_t \subseteq V$), then we choose the path which has the smaller number of extra nodes as the path from each source node s ($s \in S_t$) to each destination node d ($d \in D_t$) to save the space of flow tables in SDN switches, *as shown in* the improved shortest path Algorithm 4.

In Algorithm 4, $dis[d][0]$ is the shortest path from node s to node d, $dis[d][1]$ is the second shortest path from s to d, $Dis[s][d]$ is the selected path cost from s to d. In lines 6–9, we get the current shortest or second shortest paths, then mark the optimal node and add connected edges to update paths in lines 10–16. At last, the paths $\{Dis[s][d] | s \in S, d \in D\}$ which have a smaller number of extra nodes (switches) are selected as the cost paths to reduce the number of flow entries in SDN switches in lines 17–21.

Algorithm 4 *ImprovedDijkstra*(G, S, D) From S to D

Input: $G = (V, E)$ with edge costs $\{cost[v_i][v_j]|v_i, v_j \in V\}$, multicast stream $M = (S, D, w)$, $S \subseteq V$, $D \subseteq V$

Output: The improved shortest paths $\{Dis[s][d]|s \in S, d \in D\}$

1: **Initialize** G, $\{cost[v_i][v_j]|v_i, v_j \in V\}$
2: **for** $s \in S$ **do**
3: $dis[s][0] \leftarrow 0$
4: **for** $i = 1 \rightarrow 2 * |V|$ **do**
5: $min \leftarrow \infty$
6: **for** $v \in V$ **do**
7: **for** $t = 0 \rightarrow 1$ **do**
8: **if** $visit[v][t]$ is false and $dis[v][t] < min$ **then**
9: $min \leftarrow dis[v][t], v_{min} \leftarrow v, k \leftarrow t$
10: $visit[v_{min}][k] \leftarrow$ true
11: **for** each edge connected to v_{min} **do**
12: $newDis \leftarrow dis[v_{min}][k] + cost[v_{min}][v_j]$
13: **if** $newDis < dis[v_j][0]$ **then**
14: $dis[v_j][1] \leftarrow dis[v_j][0], dis[v_j][0] \leftarrow newDis$
15: **else if** $newDis < dis[v_j][1]$ **then**
16: $dis[v_j][1] \leftarrow newDis$
17: **for** $d \in D$ **do**
18: **if** the node number in the path of $dis[d][0]$ > the node number in the path of $dis[d][1]$ **then**
19: $dis[d][0] \leftarrow dis[d][1]$
20: $Dis[s][d] \leftarrow dis[d][0]$
21: **return** The improved shortest paths $\{Dis[s][d]|s \in S, d \in D\}$

Algorithm 5 Topic-Oriented Steiner Tree Multicast Routing

Input: $G = (V, E)$ with edge costs $\{cost[v_i][v_j]|v_i, v_j \in V\}$, multicast stream $\{M_t = (S_t, D_t, w_t)|1 \le t \le p\}$, $S_t \subseteq V$, $D_t \subseteq V$

Output: A multicast forest F consisting of several Steiner trees

1: **Initialize** $S \leftarrow \bigcup_{t=1}^{p} S_t, D \leftarrow \bigcup_{t=1}^{p} D_t, \{cost[v_i][v_j]\}$
2: $\{Dis[s][d]|s \in S, d \in D\} \leftarrow ImprovedDijkstra(G, S, D)$
3: $G_1 = (S \cup D, E_1), \forall(s_i, s_j) \in E_1$, link cost is $Dis[s_i][s_j]$
4: $T_1 \leftarrow Prim(G_1)$
5: $G_2 \leftarrow PathRecovery(T_1, G)$
6: $T_2 \leftarrow Prim(G_2)$
7: **for** *node* in T_2 **do**
8: **if** *node* is leaf and $node \notin S \cup D$ **then**
9: Remove *node* and connected edges from T_2
10: Get a Steiner tree ST_1 or multiple Steiner trees $\{ST_i|1 \le i \le p_1 \le p\}$ if G or G_1 or G_2 is not a connected graph
11: **return** A multicast forest $F \leftarrow \{ST_i|1 \le i \le p_1\}$

The pub/sub topology for multiple topics are constructed by the classic KMB algorithm [26] about the Steiner tree with an improvement on the shortest path, *as shown in* Algorithm 5. In line 2, we get the optimal cost path from publishers S to subscribers D according to the improved shortest path Algorithm 4. Then, the complete cost graph connecting all publishers and subscribers $G_1 = (S \cup D, E_1)$ is computed in line 3, and the path of T_1 are recovered with the original path in G in line 5. At last, the cycles of G_2 are removed in lines 6–9. The time complexity of Algorithm 5 is $O(|S \cup D| * |V|^2)$. We use the Dijkstra algorithm to get the optimal cost paths for reducing complexity, not the Floyd algorithm, because most pub/sub network graphs are sparse graphs ($|S_t \cup D_t| \ll |V|$), the time complexity of Algorithm 5 can reach $O(|V|^3)$ if we use the Floyd algorithm.

After constructing the pub/sub topology for multiple topics, we can get paths for event routing conveniently by the multicast forest to deliver events fast. This construction algorithm uses the minimal total cost of event transmission and minimal extra nodes to save the space of flow tables in SDN switches in SDN-like topic-oriented pub/sub middlewares.

V. TOPIC-ORIENTED BUCKET-BASED MULTICAST FORWARDING

In this section, we focus on how to forward topic-based events efficiently in SDN switches according to paths constructed by the Steiner multicast trees in Section IV, which is called event forwarding. Namely, the design and implementation of function *Bucket_based_multicast* in line 28 of Algorithm 1. First, OpenFlow group table and an example of bucket-based multicast are introduced in detail. Then, we propose our topic-oriented bucket-based multicast forwarding algorithm to forward events efficiently from publishers to subscribers by flow tables and group tables installed on SDN switches along the paths in SDN-like pub/sub middlewares.

A. OpenFlow GROUP TABLE

Group table is an extension of flow table forwarding methods, which supports forwarding events to group, namely, multiple receivers. We can associate flow table with group by adding a flow entry to the group. Group table is composed of group entries. Each group entry contains a group ID, group type, counters and action buckets. The group ID is used to mark a group uniquely with a 32-bit unsigned integer. The group type *all* indicates that all action buckets should be performed in a group, which is very suitable for multicast, broadcast and flooding scenarios. The action buckets defines several OpenFlow buckets, each bucket is an action list.

In this paper, OpenFlow 1.3.0 [59] is adopted to implement group multicast. The group ID is a mapping of topics, i.e., the integer form of 32-bit topic prefix, if the topic prefix is 1.2.3.4, the group ID will be 16909060. All subscribers interested in the same topic (prefix) form one group. When publishers publish events with this topic prefix, the action buckets of the group are executed to forward topic events to the output ports of SDN switches, *as shown in* Fig. 4, when publication events with topic prefix 1.2.3.4 enter the port 0 of the SDN switch, all action buckets (Bucket 1 and Bucket 2) in group 16909060 are executed to forward events to the port 1

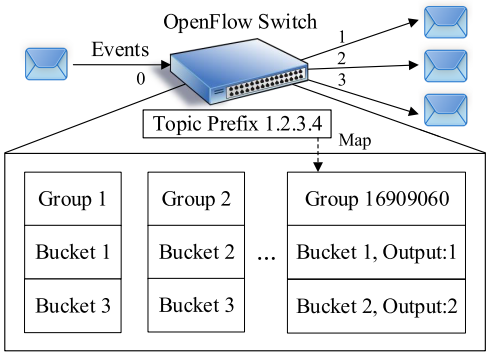


FIGURE 4. An example of bucket-based multicast.

and port 2 of the switch. In this way, one-to-many multicast is implemented in SDN-like pub/sub middlewares.

There are four advantages for our bucket-based multicast. First, subscribers interested in the same topic prefix are put into one group, which makes it easier to aggregate subscriptions. Moreover, our topic-oriented bucket-based multicast makes full use of the subscription coverage relationship between topics to merge flow entries, reducing the number of flow entries installed on SDN switches. False positives mean that subscribers receive topic events that they are not interested in. If a subscriber cancels a subscription, but the installed flow entry has not expired, false positives will occur. Therefore, our bucket-based multicast can reduce the false positive rate under dynamic subscriptions or unsubscriptions with fewer flow entries and subscription aggregations. Second, the additional cost is saved for the only multicast switch. Third, this method is suitable for all kinds of network protocols, which is more efficient and reliable, because traditional IP multicast uses the special class D multicast addresses that limits its use. Fourth, it facilitates group management by group ID (topic).

B. TOPIC-ORIENTED BUCKET-BASED MULTICAST FORWARDING ALGORITHM

We propose our topic-oriented bucket-based multicast forwarding algorithm, *as shown in* Algorithm 6. It is used to implement multicast forwarding from a publisher to multiple subscribers in topic-oriented SDN-like pub/sub middlewares. In order to map 64-bit topic to 32-bit group ID, the subscription coverage relationships between topics are considered to merge topics with the same 32-bit topic prefix into one group, reducing the number of group entries and flow entries and improving the matching abilities of SDN switches.

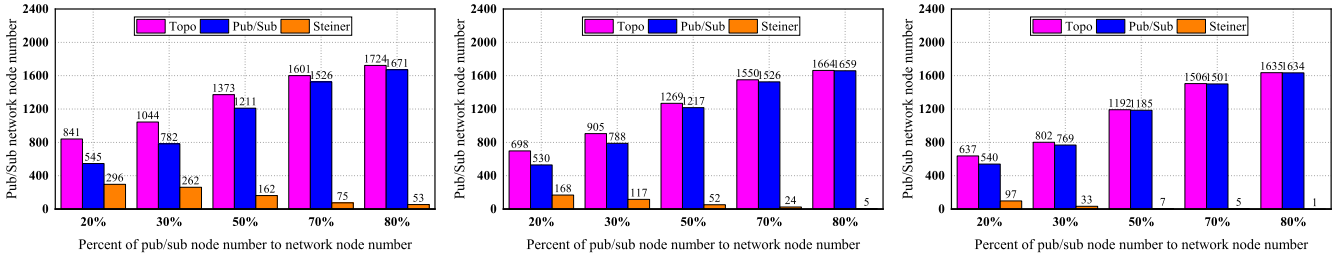
The algorithm contains two-layer for loops, *as shown in* lines 1–18. In lines 3–10, we merge the subscriptions of topic to an ancestor topic with the topic length less than or equal to 32 if the length of topic greater than 32. In this way, topics with the same 32-bit topic prefix are mapped to the same group, reducing the number of group greatly. If (*switch*, *topic*) pair is not in buckets, we should add a new group in lines 11–12. Otherwise, if the pair is in old group, we should modify the group, and wrap actions to bucket for forwarding in lines 13–18. At last, group table should be updated for each changed switch node in lines 19–26.

The function of group table update *is shown in* lines 27–31. Line 28 is the definition of group table. SDN controller sends the group table update message to switch to modify the group table in line 29. Line 30 defines the matching rules for topic events embedded into IPv6 (0×86dd) packets. Line 31 associates group ID to a flow entry.

Algorithm 6 Topic-Oriented Bucket-Based Multicast Forwarding

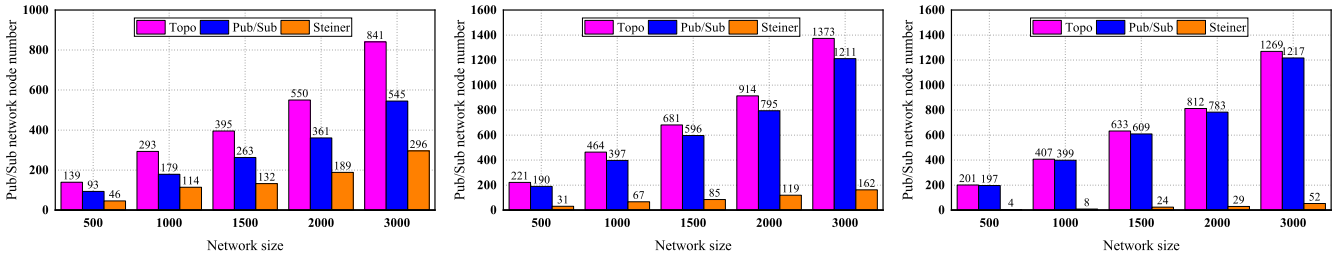
Input: publish/subscribe network topology, *paths*, *topics*
Output: multicast forwarding for topic events in switches

1: **for** *path* in *paths* **do** ▷ Traverse each publish/subscribe path
2: **for** *switch* in *path* **do**
3: **if** the length of topic code $>$ 32 **then**
4: **if** there is an ancestor topic of *topic* in *buckets* **then**
5: Merge *topic* to the ancestor topic
6: **continue**
7: **else**
8: find an ancestor topic with topic length $\leqslant$ 32 in topic tree
9: Merge *topic* to the ancestor topic
10: *topic* $\leftarrow$ *AncestorTopic*
11: **if** (*switch*, *topic*) is not in *buckets* **then**
12: *cmd*[*switch*][*topic*] $\leftarrow$ OFPGC_ADD
13: **else if** (*switch*, *topic*) not in new group **then**
14: *cmd*[*switch*][*topic*] $\leftarrow$ OFPGC_MODIFY
15: *actions* $\leftarrow$ output to *switch.outports*[*node*]
16: *bucket* $\leftarrow$ OFPBucket(0, *wp*, *wg*, *actions*)
17: Add *bucket* to *buckets*
18: Add *switch* to *changed_nodes*
19: **for** *switch* in *changed_nodes* **do** ▷ Update group table
20: *gid* $\leftarrow$ *Map*(*topic*)
21: **for** *topic* in *buckets* **do**
22: **if** *cmd*[*switch*][*topic*] is OFPGC_ADD **then**
23: *c* $\leftarrow$ OFPGC_ADD ▷ Add group
24: **else**
25: *c* $\leftarrow$ OFPGC_MODIFY ▷ Modify group
26: UpdateGroup(*switch*, *c*, all, *gid*, *buckets*, *topic*)
27: **function** UpdateGroup(*switch*, *c*, *type*, *gid*, *buckets*, *topic*)
28: *group_msg* $\leftarrow$OFPGroupMod(*switch*, *c*, *gid*, *buckets*)
29: *switch*.send_msg(*group_msg*)
30: *match* $\leftarrow$ OFPMatch(Publication, *topic*, 0×86 *dd*)
31: AddFlow(*switch*, *priority*, *match*, actions to group *gid*)



(a) 1≤degree≤5 (b) 3≤degree≤8 (c) 6≤degree≤12

FIGURE 5. Steiner nodes under different node degrees.



(a) percent=20% and 1≤degree≤5 (b) percent=50% and 1≤degree≤5 (c) percent=50% and 3≤degree≤8

FIGURE 6. Steiner nodes under different network sizes.

In this way, multiple topic events with the same 32-bit topic prefix corresponds to one group entry, which records all forwarding actions about topics with subscription coverage relationships, achieving topic-oriented bucket-based multicast, reducing the number of group entries and flow entries in SDN switches and improving the matching efficiency of flow tables in SDN-like pub/sub middlewares.

VI. EXPERIMENTAL EVALUATION

In this section, several sets of experiments are conducted to validate the effectiveness of our bucket-based fast multicast routing algorithm in SDN-like pub/sub middlewares. The first part is to verify the effectiveness of the Steiner tree multicast routing algorithm and test its performance. Specifically, (i) Steiner nodes under different node degrees and network sizes. (ii) Steiner multicast tree construction time. (iii) Multicast tree cost (delay) comparison. (iv) Multicast tree construction time comparison. The second part is to verify the effectiveness of our bucket-based multicast forwarding algorithm. Namely, (v) End-to-end delay comparison. (vi) Flow table size comparison.

A. STEINER TREE MULTICAST ROUTING ALGORITHM

In this part, four sets of experiments are performed to verify the effectiveness of our Steiner tree multicast routing algorithm, namely, experiments (i), (ii), (iii), and (iv). The first set of experiments is used to validate the rationality of our pub/sub topology construction method by the Steiner tree, and explore the impact of network size, node degree and the number of pub/sub nodes on the topology construction. The second set of experiments presents the time overhead of our Steiner multicast tree construction algorithm. The third set of experiments is used to verify that our Steiner tree has the minimum total cost compared with the SPT and MST. The fourth set of experiments verifies the rationality of the construction time of our Steiner multicast tree by comparison with the SPT and MST.

1) PUBLISH/SUBSCRIBE TOPOLOGY CONSTRUCTION

Two groups of experiments are performed for SDN-like pub/sub topology construction, Namely, Steiner nodes under different node degrees and network sizes. In the first group of experiments, the number of network nodes is 3000; the degree of nodes changes over [1, 5], [3, 8] and [6, 12], respectively; the pub/sub node percent varies among 20%, 30%, 50%, 70%, and 80%. In the second group of experiments, the network size changes over 500, 1000, 1500, 2000 and 3000; the degree of nodes is [1, 5] or [3, 8]; the pub/sub node percent is 20% or 50%.

The first set of experimental results *are illustrated in* Fig. 5. *Topo* represents the total number of nodes for SDN-like pub/sub topology construction. *Pub/Sub* denotes the pub/sub node (publisher/subscriber) number. *Steiner* means the Steiner node number. In each subfigure, the Steiner node number decreases as the pub/sub node percent increases. In different subfigures, as the degree of nodes increases, the Steiner node number decreases, and the number of nodes about topology construction also decreases.

Fig. 6 shows the second set of experimental results. In each subfigure, with the increase of network size, the topology node number, the pub/sub node number and the Steiner node number all increase. By comparing Fig. 6(a) and Fig. 6(b),

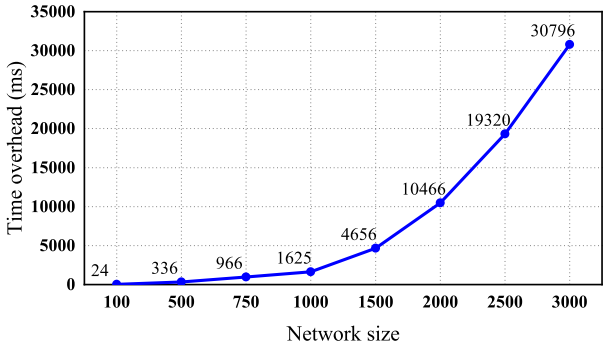


FIGURE 7. Steiner multicast tree construction time in different network sizes.

we can conclude that the topology node number increases and the Steiner node decreases as the pub/sub node percent increases for each network size. By comparing Fig. 6(b) and Fig. 6(c), we can conclude that the topology node number decreases and the Steiner node number also decreases as the degree of nodes increases for each network size.

These two sets of experiments indicate that in addition to network size, node degree and the percent of pub/sub node number also have an impact on the number of Steiner nodes and SDN-like pub/sub network topology construction, and our topology construction method is effective.

2) STEINER TREE CONSTRUCTION TIME OVERHEAD

Fig. 7 illustrates the time overhead of the Steiner tree construction in different network sizes. The size varies from 100 to 3000. The percent of pub/sub nodes is 20%, and the degree of nodes changes over [1, 5]. The experiment results indicate that our Steiner tree construction algorithm is efficient because the construction time is in the sub-second level when the network size is less than 750, it needs seconds when the network size is [1000, 2000), which is tolerable for a medium-sized network, and it costs tens of seconds when the network size is [2000, 3000]. The time overhead curve of our Steiner tree construction are also consistent with its time complexity $O(|S \cup D| * |V|^2)$.

3) MULTICAST TREE COST COMPARISON

The Steiner tree is the minimum cost multicast tree. We conduct two sets of experiments to verify this theory. One is multicast tree cost comparison among our Steiner tree, the MST and SPT under different node degrees, the other is multicast tree cost comparison under different network sizes. The parameters of the two set of experiments are the same as those in Fig. 5 and Fig. 6, respectively. For each experiment, we use link delay as the cost of edges, which is set to a random number between 0 and 100 (ms). *Steiner* denotes our Steiner multicast tree cost. *MST* represents the MST cost. *SPT* indicates the SPT cost.

The first set of experimental results *are illustrated in* Fig. 8. For each experiment, the Steiner multicast tree cost is less than the MST cost and the SPF cost. For each subfigure, the Steiner multicast tree cost increases with the increase of the percent of pub/sub node number, these results are consistent with the results of Fig. 5. When the percent of pub/sub node number increases, the pub/sub topology node number increases, so the Steiner multicast tree cost also increases in the same node degree. For different subfigures, the Steiner multicast tree cost decreases with the increase of node degree, these results are consistent with the results of Fig. 6. When the degree of nodes increases, the pub/sub topology node number decreases, so the Steiner multicast tree cost also decreases in the same pub/sub node percent.

The second set of experimental results *are illustrated in* Fig. 9. For each experiment, the Steiner tree cost is minimal. For each subfigure, the Steiner tree cost increases with the increase of network size. By comparing Fig. 9(a) and Fig. 9(b), we can conclude that the Steiner tree cost increases with the increase of the percent of pub/sub node number, these results are consistent with the results of Fig. 6. When the percent of pub/sub node number increases, the pub/sub topology node number increases, so the Steiner tree cost also increases in the same node degree. By comparing Fig. 9(b) and Fig.9(c), we can know that the Steiner tree cost decreases with the increase of the degree of nodes, these results are consistent with the results of Fig. 6. When the degree of node increases, the pub/sub topology node number decreases, so the Steiner tree cost also decreases in the same network size.

These two sets of experiments show that our Steiner multicast tree cost is minimal compared with common multicast tree (MST, SPT) algorithms for SDN-like pub/sub middlewares.

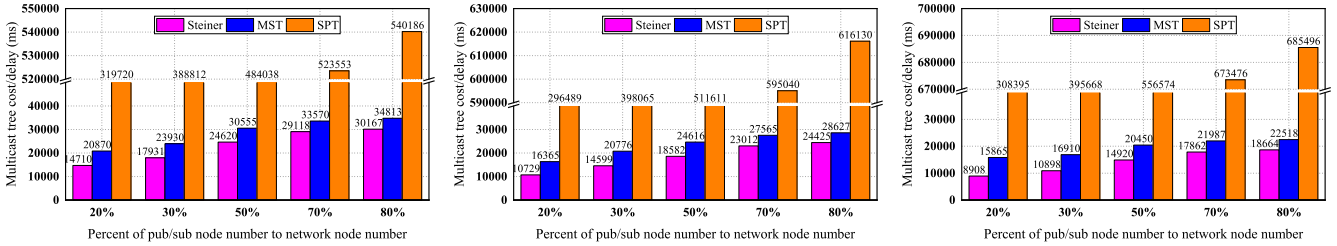
4) MULTICAST TREE CONSTRUCTION TIME COMPARISON

We perform one set of experiments to compare multicast tree construction time among our Steiner tree, the MST and SPT under different node degrees. For each experiment, the number of network nodes is 1000, and other parameters are the same with Fig. 5.

The experimental results *are illustrated in* Fig. 10. For each experiment, the multicast tree construction time is $Steiner > SPT > MST$. The reason is as follows: the time complexity of our Steiner tree construction is $O(|S \cup D| * |V|^2)$, *as shown in* Section IV-B. The time complexity of the SPT construction is $O(|S| * |V|^2)$. The time complexity of the MST construction is $O(|V|^2)$, which is much smaller than the SPT and our Steiner tree. This is the reason that many pub/sub systems select the MST as multicast tree, however, it cannot guarantee the minimum total cost (delay) of multicast tree. The time cost of constructing the Steiner tree and the SPT is in the same level. This set of experiments indicates that our Steiner multicast tree construction time is reasonable.

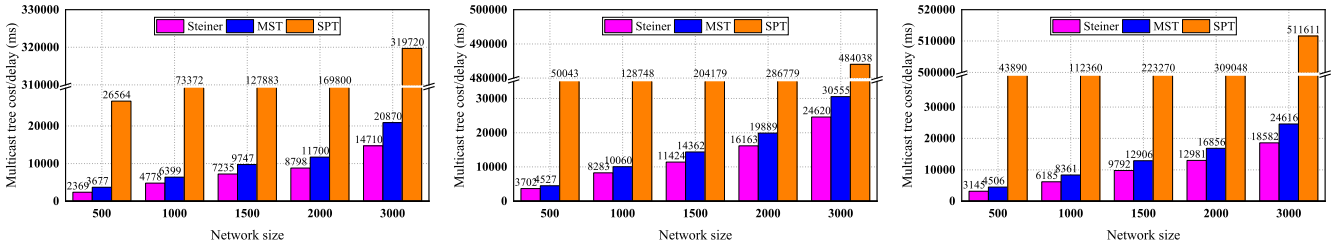
B. BUCKET-BASED MULTICAST FORWARDING ALGORITHM

We conduct some experiments about bucket-based multicast by the classic SDN simulation platform Mininet [60] and SDN controller Ryu, which both run on Ubuntu 14.04 LTS.



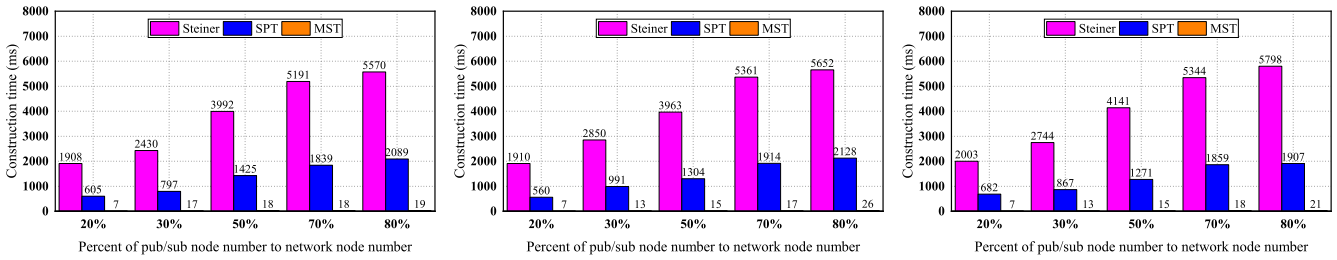
(a) 1≤degree≤5 (b) 3≤degree≤8 (c) 6≤degree≤12

FIGURE 8. Multicast tree cost comparison under different node degrees.



(a) percent=20% and 1≤degree≤5 (b) percent=50% and 1≤degree≤5 (c) percent=50% and 3≤degree≤8

FIGURE 9. Multicast tree cost comparison under different network sizes.



(a) 1≤degree≤5 (b) 3≤degree≤8 (c) 6≤degree≤12

FIGURE 10. Multicast tree construction time comparison under different node degrees.

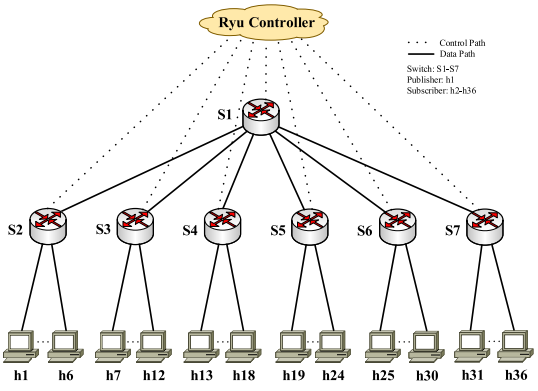


FIGURE 11. Experiment topology.

Fig. 11 shows the experiment topology with one Ryu controller, seven SDN-enabled switches, and 36 hosts. Switch S1 is connected to six switches, forming a two-layer tree structure. Every other switch has six hosts, host h1 is a publisher,

TABLE 1. Subscription Configuration.

Number of Subscribers	Subscribers
5	h7–h11
10	h7–h16
15	h7–h21
20	h7–h26
25	h7–h31
30	h7–h36
35	h2–h36

other hosts are subscribers. Java socket program is deployed on each host to send or receive packets. Publisher h1 publishes topic events which will be received by subscribers if they express their interests in these events beforehand. Several subscribers are put in the same multicast group, the subscription configuration *is shown in* Table 1, i.e., the first line in the table means five subscribers h7 to h11 subscribe a specific topic, so the corresponding group has five members.

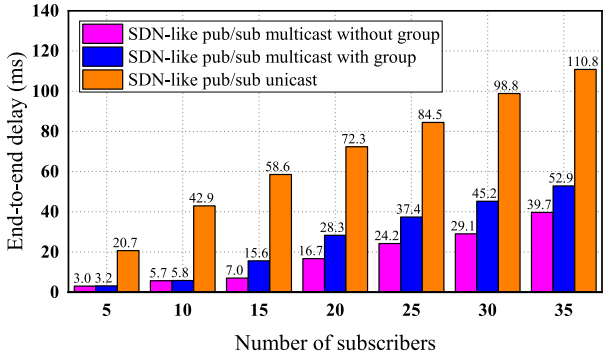


FIGURE 12. End-to-end delay comparison.

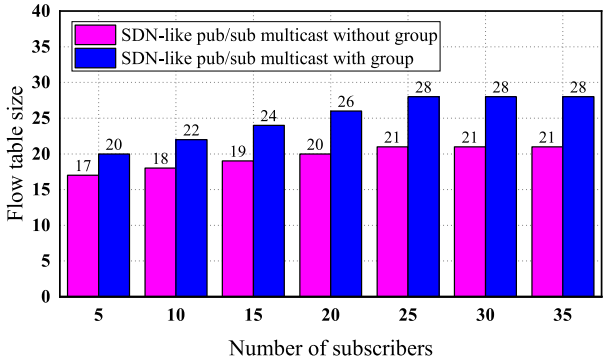


FIGURE 13. Flow table size comparison.

1) END-TO-END DELAY

Fig. 12 shows the experiment results of end-to-end delay comparison about three SDN-like pub/sub communication styles under different number of subscribers. The first one is the traditional multicast without group, multiple output actions in many ports of switches are placed into one action set. When switches receive publication messages about one topic, the packet header fields of messages first match the flow table of switches, and then the messages will be forwarded to the specified ports of switches which are written into the actions of the flow entry beforehand. The second one is the bucket based multicast with group, each output action is put into an OpenFlow bucket, multiple actions form a bucket list. Publication messages with specific topic first match the flow entry with a action to corresponding group ID, then the group entry with the same group ID is matched, the messages will be forwarded to the specified ports of switches which are written into the action buckets of the group entry in advance.

In Fig. 12, for each set of experiments, bucket-based multicast with group takes a little more time because it needs to matches two times for a pub/sub topic, one time in flow table, the other time in group table compared to the multicast without group, However, bucket-based multicast with group has some new advantages, it is easy for flow management and group management and easy to aggregate subscriptions. Security policies, multipath load balancing and fast failover can also be executed conveniently on specific groups.

As shown in Fig. 12, unicast takes more time than two multicast styles. If there are n subscribers, n flows should be sent for unicast communication, however, one flow is enough for multicast, packets are copied and forwarded at the branch of paths, so multicast can save a lot of end-to-end delay, which is very suitable for IoT scenarios with strict latency requirements. Therefore, these experiment results indicate that our bucket-based multicast method is efficient.

2) FLOW TABLE SIZE

Fig. 13 shows the experiment results of total flow table size comparison between SDN-like pub/sub multicast with group and without group. For each experiment, the flow table size of multicast with group is larger than multicast without group.

At first, each switch for two multicast styles has two default flow entries, so the total table size for all switches is 14. When the number of subscribers is 5, hosts h7-h12 subscribe topic, so pub/sub paths are formed from h1 to hi ($i = 7, 8, \ldots, 12$), three new flow entries are installed on switches S1, S2 and S3, so the flow table size for multicast without group is 17; For multicast with group, the size is 20 because each group table adds one group entry.

When the number of subscribers is 30 or 35, the flow table size no longer increases, because all flow entries and group entries are full for switches, no new flow entries or group entries can be added, only new output ports are added to the action lists of flow entries for multicast without group, or only new action buckets are added to the bucket lists of group entries. However, for unicast, when new subscribers join in pub/sub network, new flow entries are installed, increasing the burden of pub/sub systems. For multicast with group, only a small amount of group table space is added than multicast without group. These experiment results verify the effectiveness of our bucket-based multicast method.

VII. CONCLUSION

In this paper, we address the issue of how to adopt SDN to implement a topic-oriented bucket-based fast multicast routing for delivering events efficiently in SDN-like pub/sub middlewares. On one hand, we adopt the features of centralized control of SDN controllers to acquire the global network topology, then we use it and the Steiner tree to build the minimum cost multicast tree of event transmission for pub/sub topology construction in SDN controllers to deliver events fast. On the other hand, we utilize the programmability of SDN to install our customized topic-oriented flow tables and group tables, which use topic-oriented action buckets to implement the bucket-based multicast for efficient event forwarding in SDN switches. Moreover, the design of SDN-like pub/sub routing enables events to be matched directly and fast on SDN switches, avoiding the additional delay caused by a detour to overlay network in IP-based pub/sub middlewares. These three schemes together constitute our topic-oriented bucket-based fast multicast routing in SDN-like pub/sub middlewares. Experimental results indicate that our schemes are effective.

In the future, we can design an incremental Steiner tree construction algorithm to reduce the construction time overhead caused by frequent advertisements and subscriptions for further improving the efficiency of event delivery in topic-oriented SDN-like pub/sub middlewares.

REFERENCES

[1] A. Diro, H. Reda, N. Chilamkurti, A. Mahmood, N. Zaman, and Y. Nam, "Lightweight authenticated-encryption scheme for Internet of Things based on publish-subscribe communication," *IEEE Access*, vol. 8, pp. 60539–60551, 2020.

[2] J. Hasenburg and D. Bermbach, "GeoBroker: Leveraging geo-contexts for IoT data distribution," *Comput. Commun.*, vol. 151, pp. 473–484, Feb. 2020.

[3] R. S. Gargees and G. J. Scott, "Dynamically scalable distributed virtual framework based on agents and pub/sub pattern for IoT media data," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 599–613, Feb. 2019.

[4] A. E. C. Redondi, A. Arcia-Moret, and P. Manzoni, "Towards a scaled IoT pub/sub architecture for 5G networks: The case of multiaccess edge computing," in *Proc. IEEE 5th World Forum Internet Things (WF-IoT)*, Apr. 2019, pp. 436–441.

[5] P. Lv, L. Wang, H. Zhu, W. Deng, and L. Gu, "An IOT-oriented privacy-preserving publish/subscribe model over blockchains," *IEEE Access*, vol. 7, pp. 41309–41314, 2019.

[6] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, Q. Zhang, and K.-K.-R. Choo, "An energy-efficient SDN controller architecture for IoT networks with blockchain-based security," *IEEE Trans. Services Comput.*, early access, Jan. 15, 2020, doi: 10.1109/TSC.2020.2966970.

[7] A. Wang, Z. Zha, Y. Guo, and S. Chen, "Software-defined networking enhanced edge computing: A network-centric survey," *Proc. IEEE*, vol. 107, no. 8, pp. 1500–1519, Aug. 2019.

[8] S. Bera, S. Misra, and A. V. Vasilakos, "Software-defined networking for Internet of Things: A survey," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1994–2008, Dec. 2017.

[9] K. Smida, H. Tounsi, M. Frikha, and Y.-Q. Song, "Software defined Internet of Vehicles: A survey from QoS and scalability perspectives," in *Proc. 15th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Jun. 2019, pp. 1349–1354.

[10] K. Kalkan and S. Zeadally, "Securing Internet of Things with software defined networking," *IEEE Commun. Mag.*, vol. 56, no. 9, pp. 186–192, Sep. 2018.

[11] I. Farris, T. Taleb, Y. Khettab, and J. Song, "A survey on emerging SDN and NFV security mechanisms for IoT systems," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 1, pp. 812–837, 1st Quart., 2019.

[12] S. Balasubramanian, D. Ghosal, K. N. B. Sharath, E. Pouyoul, A. Sim, K. Wu, and B. Tierney, "Auto-tuned publisher in a pub/sub system: Design and performance evaluation," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Sep. 2018, pp. 21–30.

[13] S. Nishio, D. Amagata, and T. Hara, "Lamps: Location-aware moving top-K pub/sub," *IEEE Trans. Knowl. Data Eng.*, early access, Mar. 19, 2020, doi: 10.1109/TKDE.2020.2979176.

[14] P. F. Moraes and J. S. B. Martins, "A pub/sub SDN-integrated framework for IoT traffic orchestration," in *Proc. 3rd Int. Conf. Future Netw. Distrib. Syst. (ICFNDS)*, 2019, pp. 1–9.

[15] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Durr, T. Kohler, and K. Rothermel, "High performance publish/subscribe middleware in software-defined networks," *IEEE/ACM Trans. Netw.*, vol. 25, no. 3, pp. 1501–1516, Jun. 2017.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[17] R. Narisetty, L. Dane, A. Malishevskiy, D. Gurkan, S. Bailey, S. Narayan, and S. Mysore, "OpenFlow configuration protocol: Implementation for the of management plane," in *Proc. 2nd GENI Res. Educ. Exp. Workshop*, Mar. 2013, pp. 66–67.

[18] Y. Shi, Y. Zhang, H.-A. Jacobsen, L. Tang, G. Elliott, G. Zhang, X. Chen, and J. Chen, "Using machine learning to provide reliable differentiated services for IoT in SDN-like Publish/Subscribe middleware," *Sensors*, vol. 19, no. 6, p. 1449, 2019.

[19] Y. Zhang, H. Zhou, and J.-L. Chen, "Cross-layer access control in publish/subscribe middleware over software-defined networks," *Comput. Commun.*, vol. 134, pp. 1–13, Jan. 2019.

[20] K. Zhang and H.-A. Jacobsen, "SDN-like: The next generation of pub/sub," 2013, *arXiv:1308.0056*. [Online]. Available: http://arxiv.org/abs/1308.0056

[21] Y. Shi, Y. Zhang, and J. Chen, "Cross-layer QoS enabled SDN-like publish/subscribe communication infrastructure for IoT," *China Commun.*, vol. 17, no. 3, pp. 149–167, 2020.

[22] P. D. Thanh, H. T. T. Binh, and T. B. Trung, "An efficient strategy for using multifactorial optimization to solve the clustered shortest path tree problem," *Int. J. Speech Technol.*, vol. 50, no. 4, pp. 1233–1258, Apr. 2020.

[23] T. Zaarour and E. Curry, "Adaptive filtering of visual content in distributed publish/subscribe systems," in *Proc. IEEE 18th Int. Symp. Netw. Comput. Appl. (NCA)*, Sep. 2019, pp. 1–5.

[24] M. Siebert, S. Ahmed, and G. Nemhauser, "A linear programming based approach to the Steiner tree problem with a fixed number of terminals," *Networks*, vol. 75, no. 2, pp. 124–136, Mar. 2020.

[25] C.-Y. Chen and S.-Y. Hsieh, "An efficient approximation algorithm for the Steiner tree problem," in *Complexity and Approximation*. Cham, Switzerland: Springer, 2020, pp. 238–251.

[26] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for Steiner trees," *Acta Inf.*, vol. 15, no. 2, pp. 141–145, 1981.

[27] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "Constructing scalable overlays for pub-sub with many topics," in *Proc. 26th Annu. ACM Symp. Princ. Distrib. Comput. (PODC)*, 2007, pp. 109–118.

[28] C. Tsilopoulos, I. Gasparis, G. Xylomenos, and G. C. Polyzos, "Efficient real-time information delivery in future Internet publish-subscribe networks," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, 2013, pp. 856–860.

[29] S. Akkermans, R. Bachiller, N. Matthys, W. Joosen, D. Hughes, and M. Vucinic, "Towards efficient publish-subscribe middleware in the IoT with IPv6 multicast," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–6.

[30] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. T. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE J. Sel. Areas Commun.*, vol. 20, no. 8, pp. 1489–1499, Oct. 2002.

[31] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel, "The power of software-defined networking: Line-rate content-based routing using OpenFlow," in *Proc. 7th Workshop Middleware Next Gener. Internet Comput. (MW NG)*, 2012, p. 3.

[32] J. P. de Araujo, L. Arantes, E. P. Duarte, L. A. Rodrigues, and P. Sens, "VCube-PS: A causal broadcast topic-based publish/subscribe system," *J. Parallel Distrib. Comput.*, vol. 125, pp. 18–30, Mar. 2019.

[33] C. Borcea, Y. Polyakov, K. Rohloff, and G. Ryan, "PICADOR: End-to-end encrypted Publish–Subscribe information distribution with proxy re-encryption," *Future Gener. Comput. Syst.*, vol. 71, pp. 177–191, Jun. 2017.

[34] V. Setty, M. Van Steen, R. Vitenberg, and S. Voulgaris, "Poldercast: Fast, robust, and scalable architecture for p2p topic-based pub/sub," in *Proc. 13th Int. Middleware Conf.* New York, NY, USA: Springer-Verlag, 2012, pp. 271–291.

[35] Z. Liao, S. Qian, J. Cao, Y. Cao, G. Xue, J. Yu, Y. Zhu, and M. Li, "PhSIH: A lightweight parallelization of event matching in content-based pub/sub systems," in *Proc. 48th Int. Conf. Parallel Process.*, Aug. 2019, pp. 1–10.

[36] H.-A. Jacobsen, A. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh, "The PADRES publish/subscribe system," in *Principles and Applications of Distributed Event-Based Systems*. Hershey, PA, USA: IGI Global, 2010, pp. 164–205.

[37] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based Publish/Subscribe system for large-scale science analytics," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2014, pp. 246–255.

[38] M. Albano, L. L. Ferreira, L. M. Pinho, and A. R. Alkhawaja, "Message-oriented middleware for smart grids," *Comput. Standards Interface*, vol. 38, pp. 133–143, Feb. 2015.

[39] X. J. Hong, H. S. Yang, and Y. H. Kim, "Performance analysis of RESTful API and RabbitMQ for microservice Web application," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2018, pp. 257–259.

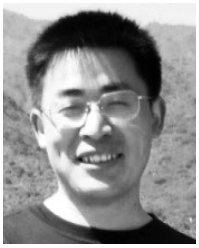
[40] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proc. 11th ACM Int. Conf. Distrib. Event Syst. (DEBS)*, 2017, pp. 227–238.

[41] Y. Lou, L. Chen, F. Ye, Y. Chen, and Z. Liu, "Research and implementation of an aquaculture monitoring system based on Flink, Mongodb and Kafka," in *Proc. Int. Conf. Comput. Sci.* Cham, Switzerland: Springer, 2019, pp. 648–657.
[42] P. Li, B. Luo, W. Zhu, and H. Xu, "Cluster-based distributed dynamic cuckoo filter system for Redis," *Int. J. Parallel, Emergent Distrib. Syst.*, pp. 1–14, Apr. 2019, doi: 10.1080/17445760.2019.1599889.
[43] E. S. Pramukantoro, J. Ratna Wulandari, W. Yahya, and H. Nurwarsito, "A cluster message broker in IoT middleware using Ioredis," in *Proc. Int. Conf. Sustain. Inf. Eng. Technol. (SIET)*, Nov. 2018, pp. 247–251.
[44] M. A. Tariq, B. Koldehofe, S. Bhowmik, and K. Rothermel, "PLEROMA: A SDN-based high performance publish/subscribe middleware," in *Proc. 15th Int. Middleware Conf. (Middleware)*, 2014, pp. 217–228.
[45] A. Hakiri and A. Gokhale, "Data-centric publish/subscribe routing middleware for realizing proactive overlay software-defined networking," in *Proc. 10th ACM Int. Conf. Distrib. Event-based Syst. (DEBS)*, 2016, pp. 246–257.
[46] Y. Wang, Y. Zhang, and J. Chen, "SDNPS: A load-balanced topic-based Publish/Subscribe system in software-defined networking," *Appl. Sci.*, vol. 6, no. 4, p. 91, 2016.
[47] P. Jokela, A. Zahemszky, C. E. Rothenberg, S. Arianfar, and P. Nikander, "LIPSIN: Line speed publish/subscribe inter-networking," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 195–206, Aug. 2009.
[48] H. T. T. Binh, P. D. Thanh, and T. B. Thang, "New approach to solving the clustered shortest-path tree problem based on reducing the search space of evolutionary algorithm," *Knowl.-Based Syst.*, vol. 180, pp. 12–25, Sep. 2019.
[49] M. A. Tariq, B. Koldehofe, and K. Rothermel, "Efficient content-based routing with network topology inference," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2013, pp. 51–62.
[50] S. L. Hakimi, "Steiner's problem in graphs and its implications," *Networks*, vol. 1, no. 2, pp. 113–133, 1971.
[51] A. V. Aho, M. R. Garey, and F. K. Hwang, "Rectilinear Steiner trees: Efficient special-case algorithms," *Networks*, vol. 7, no. 1, pp. 37–58, 1977.
[52] H. Takashami and A. Matsuyama, "An approximate solution for the Steiner tree problem in graphs," *Int. J. Math Educ. Sci. Technol*, vol. 14, no. 1, pp. 15–23, 1983.
[53] Y. Wenguo and G. Tiande, "An ant colony optimization algorithms for the minimum steiner tree problem and its convergence proof," *Acta Mathematicae Applicatae Sinica*, vol. 29, no. 2, pp. 352–361, 2006.
[54] K. Mehlhorn, "A faster approximation algorithm for the Steiner problem in graphs," *Inf. Process. Lett.*, vol. 27, no. 3, pp. 125–128, Mar. 1988.
[55] M. Hungyo and M. Pandey, "SDN based implementation of publish/subscribe paradigm using OpenFlow multicast," in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst. (ANTS)*, Nov. 2016, pp. 1–6.
[56] L.-H. Huang, H.-J. Hung, C.-C. Lin, and D.-N. Yang, "Scalable Steiner tree for multicast communications in software-defined networking," 2014, *arXiv:1404.3454*. [Online]. Available: http://arxiv.org/abs/1404.3454
[57] S. Nakamura, L. Ogiela, T. Enokido, and M. Takizawa, "An information flow control model in a topic-based publish/subscribe system," *J. High Speed Netw.*, vol. 24, no. 3, pp. 243–257, Jun. 2018.
[58] R. Project Team. *RYU SDN Framework, Release 1.0*. Accessed: Jun. 11, 2019. [Online]. Available: https://osrg.github.io/ryu-book/en/Ryubook.pdf
[59] B. Pfaff, B. Lantz, B. Heller, C. Barker, and C. Beckmann, "Openflow switch specification (version 1.3.0)," Open Netw. Found., Menlo Park, CA, USA, Tech. Rep. ONF TS-006, Jun. 2012. [Online]. Available: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf
[60] B. Rashma and G. Poornima, "Performance evaluation of multi controller software defined network architecture on mininet," in *Proc. Int. Conf. Remote Eng. Virtual Instrum.* Cham, Switzerland: Springer, 2019, pp. 442–455.

YULONG SHI is currently pursuing the Ph.D. degree with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications (BUPT), Beijing, China. He was a Visiting Ph.D. Student with the Middleware Systems Research Group, University of Toronto, Toronto, ON, Canada, from August 2017 to August 2018. His research interests include service computing, the Internet of Things (IoT), SDN, and publish/subscribe middleware.

JONATHON WONG received the M.A.Sc. and B.A.Sc. degrees from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada, in 2018 and 2016, respectively. He conducted research on efficient scaling techniques of microservice architectures in cloud data centres, distributed systems, and publish/subscribe systems with the Middleware Systems Research Group.

HANS-ARNO JACOBSEN (Fellow, IEEE) received the Ph.D. degree from Humboldt University of Berlin, Berlin, Germany, in 1999. He is currently a Professor of computer engineering and computer science and directs the activities of the Middleware Systems Research Group, University of Toronto, Toronto, ON, Canada. He engaged in Postdoctoral Research at Inria, Paris, France, before moving to the University of Toronto, in 2001. He conducts research at the intersection of distributed systems and data management, with a particular focus on middleware systems, event processing, and cyber-physical systems. In 2011, he received the Alexander von Humboldt-Professorship to engage in research at the Technical University of Munich, Munich, Germany.



YANG ZHANG received the Ph.D. degree in computer applied technology from the Institute of Software, Chinese Academy of Sciences, in 2007. He is currently an Associate Professor with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His research interests include service-oriented computing, the Internet of Things (IoT), and service security and privacy. He leads a research team on the Theoretic Foundation of EDSOA for IoT Services (National Natural Science Foundation of China under Grant 61372115).

JUNLIANG CHEN graduated from the Department of Telecommunications, Shanghai Jiao Tong University, in 1955. He received the Doctor of Engineering degree from the Moscow Institute of Electrical Telecommunications, former Soviet Union, in 1961. He was a Visiting Scholar with the University of California at Berkeley and also at Los Angeles, from 1979 to 1981. He is currently a Professor and the Academic Leader with the State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China. His current research interests include service-oriented computing and service generation systems. He is a member of the Chinese Academy of Science and the Chinese Academy of Engineering.