SNAPS: Seamless Network-Assisted Publish-Subscribe

Hyunseok Chang Fang Hao Murali Kodialam T.V. Lakshman Sarit Mukherjee Matteo Varvello
Nokia Bell Labs
Murray Hill, NJ 07974, USA

***Abstract*—The Internet is poised to support new classes of industrial applications driven by large numbers of sensors and actuators generating and consuming sensor data streams. While the publish-subscribe (PubSub) architecture is well-suited for such large-scale data dissemination, existing over-the-top PubSub systems are not only vulnerable to network-induced performance degradation, but also unable to orchestrate the underlying network for better usage. This paper proposes a network-assisted PubSub architecture called SNAPS which enables seamless control of distributed PubSub brokers at the edge, as well as the network that interconnects them. SNAPS leverages SRv6-based network programmability to build and maintain network-efficient, shareable data distribution trees for brokers. The architecture is independent of broker implementation, making it suitable for "bringing your own brokers" at available edge locations. We describe how the network interconnecting the brokers can be programmed for efficient data distribution. Through simulation and prototype evaluation, we show the efficacy of SNAPS in providing better service for PubSub data delivery and better usage of the network resources.**

***Index Terms*—publish-subscribe, network architecture, network management, programmable networks, SRv6, IoT**

I. Introduction

With the adoption of Industry 4.0 and deployment of 5G networks, the industrial/enterprise automation market is going through massive growth and restructuring. Emerging in this landscape is a new class of distributed applications and their use cases, such as AR/VR, remote control of everything, massively multiparty collaborative services, teleoperated industrial robots, etc. Due to the inherent (geographically) distributed nature of on-premise sensors that drive these applications, there is an industry-wide move to utilize distributed edge cloud infrastructures [1]. Large scale deployments of industrial Internet applications across distributed edge clouds will encompass large numbers of (physical or virtual) sensors and actuators with each entity generating and consuming its own data stream with its own unique traffic pattern, fan-out/fan-in characteristics, bandwidth and delay needs.

To address the needs, the PubSub architecture has been making its way into the industrial Internet as a primary communication model. Today, this architecture is commonly realized as over-the-top (OTT) solutions such as application-level PubSub due to their easy adoption. In a standalone PubSub system, data streams from publishers are transported to a centralized broker which then disseminates them to subscribers based on their interest. This centralized broker, while suitable for dynamic data routing, does not take into account the massively distributed nature of data sources and sinks at the edge, and cannot make use of the resources available at edge clouds. Therefore, existing PubSub systems allow a distributed broker deployment via federation [2], where multiple brokers serve clients in a distributed fashion. The brokers communicate with one another over WAN to route messages from publishers to appropriate (local or non-local) subscribers. Brokers can be placed and federated from strategic edge cloud locations. Like a centralized approach, the federated brokers still use OTT connections to distribute messages among themselves.

Completely OTT approaches as described above, while easily deployable, are not designed to meet the heterogeneous delivery requirements of different sensor data streams, or to optimally utilize underlying network resources under heavy network usage. This motivates a *network-integrated* PubSub architecture, where data is replicated and disseminated from one edge location to others through *network-level mechanisms*, as opposed to point-to-point OTT transports. Such an architecture would replicate and route messages in a network-efficient and network-aware fashion by accounting for the geographical distribution of data sources and sinks, as well as underlying network capabilities. The realization of a new network programming paradigm via the SRv6 standard [3] and its ongoing adoption in production networks [4] make it possible to enable such network primitives without introducing disruptive changes in existing network infrastructure.

However, there are several challenges in adopting such network-integrated approach. For one, PubSub applications need to be closely coordinated with an underlying programmable network. This requires heavily adapting individual PubSub brokers to support the coordination. In terms of network management, the network needs to be programmed to support efficient data delivery with minimal control plane overhead. To address these challenges, we present in this paper SNAPS, an SRv6-assisted PubSub architecture. SNAPS enables *seamless* control of distributed brokers deployed at the edge, as well as the SRv6 network that interconnects them. To this end, SNAPS tightly integrates application-level PubSub coordination and SRv6 network programming via well-defined control plane interfaces. SNAPS leverages SRv6 to program network-efficient data distribution trees within a network. These trees are programmed to be *shareable* among different distribution groups, so that network programming introduces minimal control plane overhead. On the data plane, translation

ISBN 978-3-903176-48-5© 2022 IFIP

between application-level messages and SRv6 packets occurs in an *implementation-agnostic* fashion, allowing any existing broker implementation to be plugged into SNAPS.

The main contribution of this paper is as follows. (i) We design an SRv6-assisted PubSub architecture that seamlessly coordinates network programming with application-level Pub-Sub without changing existing interfaces; (ii) We develop a novel SRv6 use case that optimizes application-level PubSub via network-efficient and shareable distribution trees; (iii) We implement a proof-of-concept prototype based on the state-of-the-art SRv6 data plane and control plane solutions and evaluate its performance.

II. Publish-Subscribe Architecture

In this section, we present SNAPS. We first go over its design principles, and then describe the architecture in detail.

A. Design Principles

Our design of a network-assisted PubSub architecture is guided by the following principles:

- **Client compatibility**: Numerous client devices already use PubSub services via well-known interfaces. Ideally, the client-side interfaces should be preserved so that the client devices need not be modified.
- **Broker independence**: The architecture should not be tied to a particular PubSub broker implementation, but flexible to allow service providers to use a broker of their choice.
- **Flexible broker deployment**: Brokers should be flexibly placed at any available edge location and interconnected via either network-oblivious overlay (for ease of connectivity) or network-assisted underlay (for better service awareness).
- **Independent network optimization**: Even with network-assistance, ongoing PubSub operations should not be disrupted by any type of network-level optimization.

B. Overall Architecture

SNAPS is a layered architecture as depicted in Fig. 1. *Edge Brokers* are placed at edge locations close to the publishers and subscribers of data streams, and interconnected by a network. We envision (and assume in this paper) the interconnecting network to be programmable to handle the demands and the constraints of PubSub applications. The network is programmed by a *Network Controller* which is responsible for all network-related controls on the routers and switches in the network. In order to coordinate network programming with application-level PubSub operations, SNAPS operates a *PubSub Coordinator* on top of the Network Controller. The coordinator interacts with the Edge Brokers for all PubSub related activities and instructs the Network Controller to take appropriate network-related actions based on these activities.

Compared to traditional OTT brokers, the extended function of Edge Brokers is to *seamlessly interconnect them via the programmable network*. In federated broker setup [2], native OTT brokers use the network as an underlay only, and use overlaid point-to-point connections to route messages among themselves. To leverage network programmability, we want

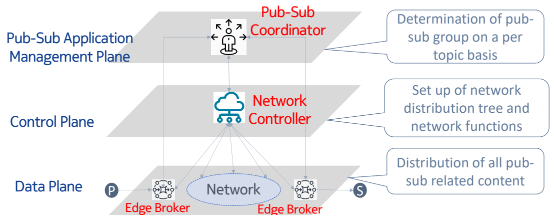

Pub-Sub Application Management Plane
Pub-Sub Coordinator
Determination of pub-sub group on a per topic basis
Control Plane
Network Controller
Set up of network distribution tree and network functions
Data Plane
P
Edge Broker
Network
Edge Broker
S
Distribution of all pub-sub related content

Fig. 1: Three layer architecture of SNAPS.

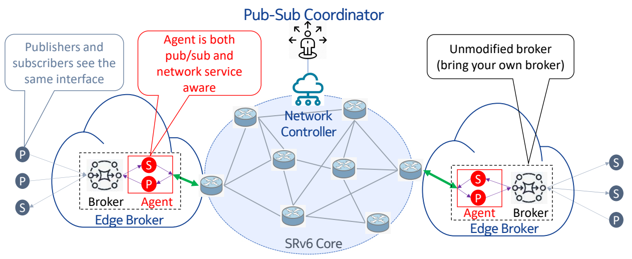

Pub-Sub Coordinator
Publishers and subscribers see the same interface
Agent is both pub/sub and network service aware
Unmodified broker (bring your own broker)
Network Controller
P
P
S
S
P
Broker
Agent
Edge Broker
SRv6 Core
S
P
Agent
Broker
Edge Broker
S
S
P

Fig. 2: Placement of the Edge Broker.

to replace point-to-point data delivery with network-assisted point-to-multipoint delivery. One challenge in doing so is that the network is not aware of application-level semantic. We address this challenge by transferring application-level semantic into network packet headers. Specifically, we encode publisher-related information (e.g., message topic) into a destination multicast address (Section II-C) and additional content labels into TLV fields. Reliable delivery, if required, can be provided by having a separate loss recovery mechanism.

Incorporating this new network interface within an existing broker implementation requires modifying each broker of choice. Instead, we introduce an *Agent* in between the broker and the programmable network to realize this functionality. An Edge Broker is essentially an unmodified broker bundled with the Agent (Fig. 2). To keep the broker interface intact, we use the same publish-subscribe messaging protocol (e.g., AMQP) for Broker-Agent communication. That is, the Agent acts as a subscriber (referred to as *Agent-Sub*) to the streams published at the broker and is responsible for transmitting them to the brokers that have subscribers for them. In the reverse direction, the Agent receives the streams from the network, and acts as a publisher to the broker (referred to as *Agent-Pub*), publishing them to the broker. The architecture itself is independent of broker implementations and messaging protocols being used.

C. Application-Layer Operations

In the following, we describe how SNAPS supports existing publish-subscribe clients seamlessly at the application-layer. We base the description on the AMQP messaging protocol [5]. Fig. 3 shows the overall interaction between clients, Edge Brokers, and the PubSub Coordinator. Similar to the federated broker setup, the clients join and leave at nearby Edge Brokers as publishers or subscribers, and create/delete exchanges and bind/unbind queues to/from exchanges. The PubSub Coordinator monitors all these events, and instructs the Agents at the corresponding Edge Broker to perform necessary control/data plane operations.

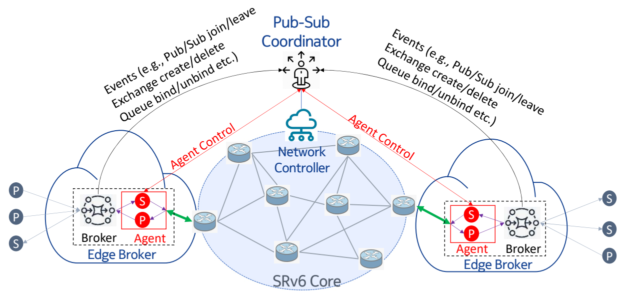

Pub-Sub Coordinator
Events (e.g., Pub/Sub join/leave Exchange create/delete Queue bind/unbind etc.)
Events (e.g., Pub/Sub join/leave Exchange create/delete Queue bind/unbind etc.)
Agent Control
Agent Control
Network Controller
P
P
S
S
Broker
Agent
Edge Broker
SRv6 Core
Agent
Broker
Edge Broker
S
S
P

Fig. 3: Functional architecture of the Edge Broker.

The following example shows the steps when a publisher P and a subscriber S join a centralized broker system.

1. Publisher P opens a connection to a well-known broker name, which is resolved (via DNS) to an address B.
2. P joins a broker at B and creates an exchange E.
3. Subscriber S opens a connection to a well-known broker name, which is resolved to an address B.
4. S joins a broker at B and binds its queue to topic T.
5. P sends a message on topic T to an exchange E at B.
6. Message is received by B and put into the queue of S.
7. S receives the message.

In SNAPS with distributed Edge Brokers, this message flow is extended as follows. We only highlight the changes made in the original steps using the same numeric labels.

1. A well-known broker name is resolved (via DNS) to an address B of a broker near the publisher P.

2a. The events of <P joined at B> and <E created at B> are reported to the PubSub Coordinator C.

2b. C inserts the events in its database and instructs the Agents at all the other brokers to create an exchange with the same name E. This ensures that any subscriber connecting to any broker finds an exchange named E.

3. A well-known broker name is resolved (via DNS) to an address B′ of a broker near the subscriber S.

4a. The events of <S joined at B′> and <Q with T created at B′> are reported to C.

4b. C inserts the events in its database, and instructs the Agents at all the brokers to create a subscription queue binding with topic T.

5a. When P sends a message on topic T to an exchange E at B, the Agent-Sub gets the message from B.

5b. Agent-Sub checks its local cache for a destination multicast address for the message. In case of cache miss, it queries C for it. C obtains a multicast address from the Network Controller. Agent-Sub multicasts the message using the obtained address.

6. Message is received by Agent-Pub associated with B′, and published to the queue of S at B′.

7. S receives the message from B′.

As shown above, SNAPS can support existing publish-subscribe clients without affecting their interface, and preserve the logically centralized broker semantic. In the next few subsections, we describe how we can bring in network assistance to support the application-layer operations.

D. Network Assistance: Design Choices

Due to the fan-out nature of sensor data streams, it is clear that network-supported multicasting of data streams will be more beneficial than application-level multicast. For the former, a network can use native IP multicast or the more recent Bit Indexed Explicit Replication (BIER) protocol [6]. With native IP multicast, an entire multicast tree is constructed by multicast-enabled routers with a special signaling protocol to generate and maintain states at those routers. With BIER, the list of recipients of a packet is maintained in the packet header, and each router replicates (when needed) and forwards the packet to downstream. Both approaches use the shortest path routing to construct an explicit or implicit multicast distribution tree within the network. These protocols are strict and are not amenable for easy traffic engineering (as needed to bring in application awareness).

In order to avoid the aforementioned drawbacks, we decide to use the recent standard SRv6 for in-network multicast among Edge Brokers. Its policy-based routing and network programming features help implement network assistance that is useful for publish-subscribe applications. For example, regular SR segments and replication segments can enable point-to-point and point-to-multipoint distributions, respectively. Any complex topology can be constructed using a combination of these SR segments. In addition, SRv6-based network programming supports including custom NF modules into SR segments. Such NF modules can enable additional in-network processing such as content filtering. Finally, SRv6 is incremental-deployment-friendly [7]. If routers do not support SRv6 or replication segments, traffic can be "sprayed" to individual destinations.

Since the multicast trees built using SRv6 can take network resources into consideration, packet loss can be significantly reduced, if not eliminated, compared to the conventional IP multicast. To further ensure reliable multicast among Edge Brokers, it is possible to add a reliable transport layer (e.g., Pragmatic General Multicast (PGM) [8]) on top of SRv6 for rate limiting and per-packet loss recovery.

E. SRv6 Multicast Tree Setup

In the application layer message flow shown previously, we assumed the Brokers utilize the SRv6 multicast service to deliver messages among the group. The PubSub Coordinator is responsible for requesting the Network Controller to create the multicast group and set up the tree. Algorithms to compute a multicast tree from a given root to a given set of receivers have been extensively studied in the past. Hence we do not intend to propose yet another multicast routing algorithm. In our prototype implementation and evaluation, we use the shortest path tree due to its simplicity, but the architecture design is not restricted to any specific multicast tree algorithms.

As we pointed out before, one advantage of using SRv6 to implement multicast is that replication segments can potentially be shared across different trees. Fig. 4 illustrates how replication segments and subtrees can be shared. The solid blue circles are SRv6-capable routers interconnected via green

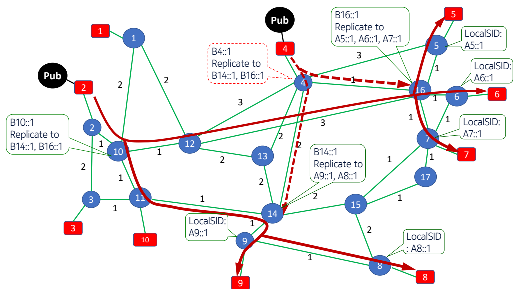


Fig. 4: Sharing replication segments across trees. Circles and squares represent SRv6 routers and hosts (where publishers and subscribers run), respectively.

network links. Solid red rectangles are Edge Brokers. Let's assume there are two publishers (at hosts `2` & `4`), each having the same set of subscribers (at hosts `5`, `6`, `7`, `8` & `9`). In this scenario, the replication segment `B14` located at router `14` and the replication segment `B16` at router `16` can be shared by both trees rooted at routers `2` and `4`, where publishers reside. Note that a replication segment can be shared only if both subtrees starting from this segment completely overlap. For example in Fig. 4, both trees for publishers `2` and `4` share the same replication segment `B16` because their subtrees rooted at router `16` are identical.

Given a set of links that represent a new multicast tree, and a set of replication segments that already exist in the network, we can use the following procedure to calculate the set of replication segments that can be used for the new multicast tree. Note that, as we pointed out previously, an existing replication segment can be reused by the new multicast tree if and only if the new multicast tree has a subtree that completely overlaps with the tree rooted at this existing replication segment.

Step 1. Generate a reverse tree from the new multicast tree. The reverse tree is the same as the original tree, except that the direction of the links are reversed. The reverse tree allows one to traverse the tree from each leaf to the root.

Step 2. Build a set of descendant leaves for every tree node. For a given tree node, the set of descendant leaves is defined as the set of all leaves that are reachable from the node on the original multicast tree. This step can be completed by traversing the reverse tree starting from each leaf to the root, and adding the current leaf to the set of descendant leaves of each intermediate tree node encountered along the way.

Step 3. Finally, generate a set of replication segments that can used for the multicast tree. The detailed steps are shown in Algorithm 1. Intuitively, the algorithm starts from each leaf and traverses the reverse tree up to the root. At each tree branching node, match the descendant leaf set of this node with the descendant leaf set of each existing replication segment to find out if any existing replication segment can be reused. A new replication segment is created only when no existing replication segment is available. The algorithm iterates through all leaves that are not covered by existing replication segments, so that the new replication segment can be constructed properly.

Algorithm 1: Computing replication segments

```
Result: $RS[n]$: replication segments for the new tree
Input:
  • $ListRS[n][i]$: list of existing replication segments at node $n$;
  • $ListRS[n][i].DLS$: set of descendant leaves for each existing
    replication segment $i$ of node $n$;
  • $RT$: the reverse tree of the original multicast tree;
  • $LocalSid[n]$: list of local segment IDs at node $n$;
  • $DL[n]$: set of descendant leaves of node $n$;
for each node n in RT.Leaves do
    $lastRS \leftarrow LocalSid[n]$;
    $n \leftarrow$ next upstream node in $RT$;
    while $n \neq$ RT.root do
        if node n has more than 1 child then
            for each replication segment i of node n do
                if ListRS[n][i].DLS == DL[n] then
                    $RS[n] \leftarrow ListRS[n][i]$;
                    $RT.Leaves \leftarrow RT.Leaves - DL[n]$;
                    $found \leftarrow$ TRUE;
                end
            end
            if $found$ == FALSE then
                add $lastRS$ to $RS[n]$;
            end
            $lastRS \leftarrow RS[n]$;
        end
    end
end
```

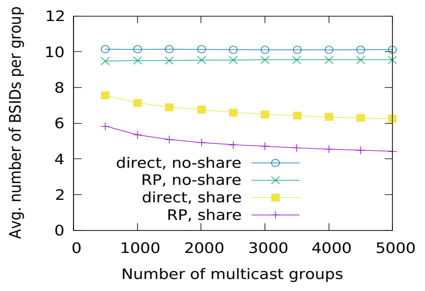
Fig. 4 illustrates how the trees are computed and set up. Assume that the publisher at node `2` is the first to join. We create a distribution tree for it as shown by the solid red line using the algorithm described above. At each leaf node `5`, `6`, `7`, `8` & `9`, we create one local segment ID to deliver the packet to the Edge Broker. Assuming no replication segment has been created in the network before, when we traverse the tree from the leaf nodes to the root node `2`, three new replication segments `B14::1`, `B16::1` and `B10::1` will be created at nodes `14`, `16` and `10` respectively, as shown in the figure. Each replication segment is responsible for replicating packets to its downstream leaves or branching nodes.

Next, the publisher at node `4` intends to send messages to the subscribers at nodes `5`, `6`, `7`, `8` & `9`. The same algorithm is invoked to traverse the reverse tree from the leaf nodes to the root node `4`. When it reaches node `14` and `16`, the algorithm discovers existing replication segments `B14::1` and `B16::1` can be reused. Then it will only create a new replication segment at node `4` to replicate packets to reach its downstream branching nodes `14` and `16`.

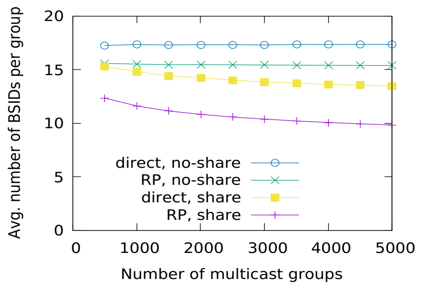
III. System Implementation

As a proof-of-concept we implement the SNAPS architecture based on off-the-shelf open-source software.

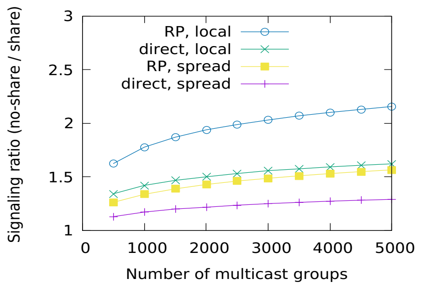
SRv6 data plane. We need SRv6 support in router instances as well as the Agent running alongside Edge Brokers on end hosts. There exist several open-source SRv6 implementations, e.g., Linux kernel implementations [9] and VPP [10]. We adopt the VPP implementation for router instances since it is more up to date with the evolving SRv6 standards, and also have better support for integration with SDN controllers. We make a few changes to VPP SRv6 implementation to be more inline with the more recent SRv6 standards. For the Agent, we use the in-kernel SRv6 module.



(a) Locality-based group



(b) Random spread group



(c) Saving ratio with tree sharing

Fig. 5: Impact of the number of multicast groups on the number of BSIDs.

Network Controller. The role of the Network Controller is to program the VPP-based SRv6 data plane to form necessary data distribution trees among Edge Brokers. However, existing open-source SDN controllers such as ONOS or OpenDayLight lack the capability to program the state-of-the-art SRv6 data plane implemented in VPP. Existing northbound control plane interfaces for VPP (e.g., `hc2vpp`, Ligato) are either incomplete in terms of SRv6 programmability or incompatible with available SDN controllers. Faced with these limitations, we implement a VPP plugin ($\sim$2K SLOC in Java) for ONOS, which extends ONOS to install SRv6 policies in VPP nodes via gRPC interface exposed by Ligato agent (v3.1.0). The ONOS-based Network Controller exposes northbound REST APIs to receive multicast group updates from the PubSub Coordinator. As multicast groups are updated, the ONOS controller computes optimal distribution trees, translates the trees into SRv6 policies, and push the policies to VPP nodes via the southbound gRPC interface. The tree management functionality in ONOS (described in Section II-E) is implemented with $\sim$3K SLOC in Java.

PubSub Coordinator. It is implemented as a Java program that interacts with the ONOS-based Network Controller and Edge Brokers. It records the state of each publish-subscribe group it receives from Edge Brokers in MySQL database. It invokes ONOS via its REST APIs to set up and optimize the multicast tree for each group, and configures the Agent-Sub/Pub according to policies.

Edge Broker. It consists of an unmodified RabbitMQ server and the Agent. We enable `exchange-event` plugin in RabbitMQ, so that it reports all publish-subscribe-related events (e.g., exchange creation, subscriber joining, etc.) to the PubSub Coordinator. The Agent is implemented as a PubSub client for RabbitMQ. We develop the Agent in C ($\sim$1K SLOC) for performance and easy integration with SRv6. The Agent-Sub/Pub modules within the Agent exchange messages with the RabbitMQ server via its unmodified publish/subscribe APIs and invoke `OpenPGM` transport-layer APIs to send/receive the messages to/from the SRv6 network. We extend `OpenPGM` to support SRv6, including packet replication and content labeling via SRv6 TLV fields.

IV. Evaluation

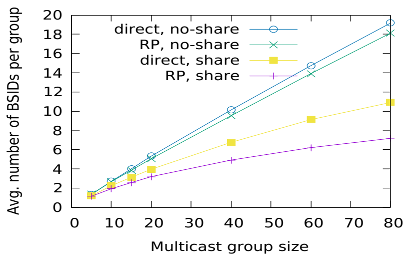
In our evaluation of SNAPS, we would like to answer the following questions. First, how would its control plane scale with a large number of PubSub groups? Second, what is the performance of SNAPS in realistic network settings, when compared to alternative schemes? We leverage simulations to investigate the scaling of the system (Section IV-A) and in-lab experiments with the SNAPS prototype to evaluate its performance (Section IV-B).

A. *Control Plane Scalability*

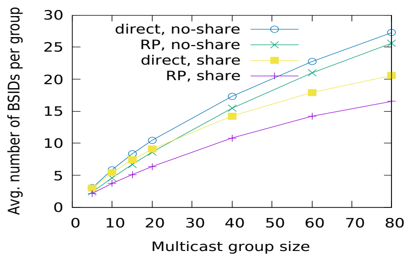
In order to set up multicast packet delivery, the Network Controller computes the shortest path tree from the root to all leaves, and generates a replication segment, *i.e.,* binding segment ID (BSID) at each branching point to replicate packets. One advantage of SNAPS multicast is the potential tree sharing across multicast groups (Section II-E), which can reduce the resource consumption in routers, as well as signaling messages for replication policy setup. Since both of these overheads are proportional to the number of BSIDs needed for each tree, we use the average number of BSIDs for each tree as the metric for tree setup overhead.

Methodology – In order to evaluate the tree setup overhead in larger networks, we perform the following simulation. We use real-world network topologies [11], [12] with sizes ranging from 53 nodes to 315 nodes. Multicast group members are generated in two ways: 1) *random spread*: each member is randomly selected using uniform distribution across all border nodes; and 2) *locality*: the first member is randomly selected, and then the rest of the members are selected with probability $e^{(2d)}$, where d is the distance (hop count) of that node to the previous member. The *locality* option intends to simulate a case where multicast groups are formed by geographically close by members, e.g., students in the same school district joining a remote learning session.

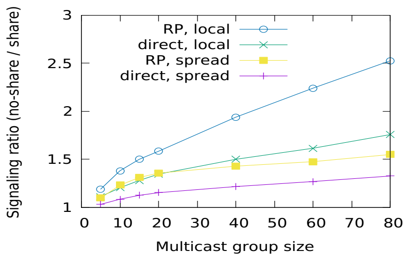
We use a simple shortest path tree computation algorithm, and experiment with two strategies: 1) compute the shortest path directly from the root to each leaf, referred as *direct*; and 2) compute the shortest path from the root to a pre-selected Rendezvous Point (RP), and then from the RP to each leaf, referred as *RP*. In our experiments, the RP is selected as the node with the shortest average distance to all border nodes. In



(a) Locality-based group



(b) Random spread group



(c) Saving ratio with tree sharing

Fig. 6: Impact of multicast group size on the number of BSIDs.

order to investigate the potential overhead reduction enabled by SNAPS, we compare two cases: 1) compute each tree independently, without sharing BSIDs across trees; 2) share the BSIDs across trees whenever such BSIDs cover the same set of leaves. During each experiment, we generate up to 5000 groups, and show the average number of BSIDs per group during the process.

Results – Figs. 5 (a) and (b) show how the average number of BSIDs change as more groups are generated using random spread and locality patterns. The network contains 197 router nodes. Each group contains 40 leaves. We observe that BSID sharing generally leads to a smaller number of BSIDs required for each tree. Without BSID sharing, the average number of BSIDs per group remains almost unchanged as the number of groups increases, since each tree is independently computed. With BSID sharing, on the other hand, the overhead decreases further as the number of groups increases, due to more potential sharing across a larger number of groups. Fig. 5(c) shows the ratio of the average number of BSIDs with no sharing vs. sharing. We observe that BSID sharing leads to resource savings by a factor of 1.2 to 2.2 across different cases. In general, locality-based groups lead to higher savings since small clusters of leaves lead to more BSID sharing. Likewise, using RP leads to higher savings since trees with the same root tend to have a better chance of sharing more nodes.

Fig. 6 shows the impact of multicast group size on the average number of BSIDs. We again use the 197-node network as an example. The number of groups is set to 2000. We find that in general a larger group size leads to significantly more reduction in the number of BSIDs required when BSID sharing is enabled. This is because it is generally easier to find overlapping subtrees when the trees are larger. Similar to results in Fig. 5, locality-based groups and using RP generally lead to higher savings.

We repeat the experiments with other topologies and observe similar results. Fig. 7 shows results for seven different topologies, including the 197-node network (Congentco) used previously. Here the number of groups is set to 3000, and group size to 20. We observe that, in general, under the same group size and the same number of groups, smaller networks tend to have more significant savings with BSID sharing, since each router may "see" more trees and hence it may be easier to find overlap across different trees.

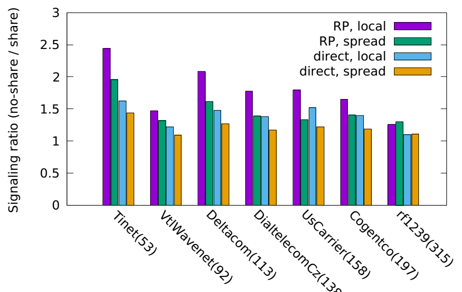


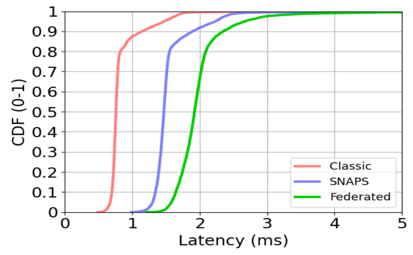
Fig. 7: BSID saving ratios for different networks. Topology names and sizes are shown on the x-axis.

B. Prototype Data Plane Performance

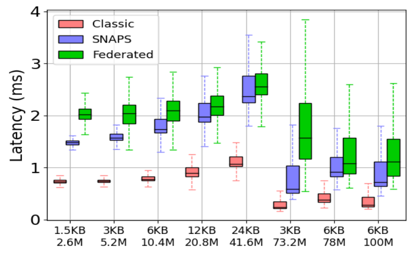
Next, we evaluate the performance of the SNAPS prototype using an emulated testbed. In this testbed, we emulate an SRv6 network with a set of containerized VPP routers interconnected via native Docker networks with predefined bandwidth/latency properties. We deploy Edge Brokers and clients in dedicated containers as well. We compare SNAPS against the centralized and federated RabbitMQ broker deployments (called "classic" and "federated" respectively). We enable federated mode with RabbitMQ's federation plugin.

1) Microbenchmarking: The goal of the next set of experiments is to microbenchmark the performance of the SNAPS prototype, without the impact of the network topology. Accordingly, we set up a minimal SNAPS deployment composed of two Edge Brokers interconnected via a single VPP router over $100\,\mathrm{Mbps}$ and zero-latency links. One publisher and one subscriber join at these brokers respectively. We compare this setup against minimal classic and federated cases. In classic, both the publisher and the subscriber connect to the RabbitMQ broker at the publisher side. The federated mode is similar to SNAPS, but without the Agent.

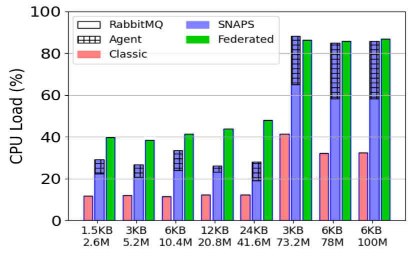
Results – We start by benchmarking per-message latency, computed as the time difference between when a message is sent by the publisher and received by the subscriber. With no network delay introduced, this latency represents the overhead of the logic implemented in each mode (e.g., in case of SNAPS, the Agents and multiple RabbitMQ instances). Fig. 8(a) shows the cumulative distribution function (CDF) of per-message latency for each mode with $2.6\,\mathrm{Mbps}$ transmission rate. According to the figure, SNAPS only adds about $750\,\mu\mathrm{s}$ to classic mode (a median latency of $1.500\,\mu\mathrm{s}$ vs.



(a) CDF of per-message latency

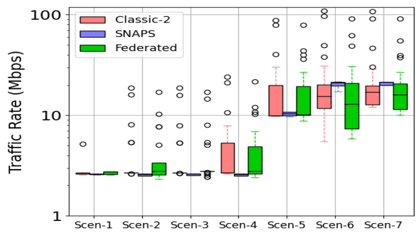


(b) Per-message latency vs. traffic rate (Mbps)

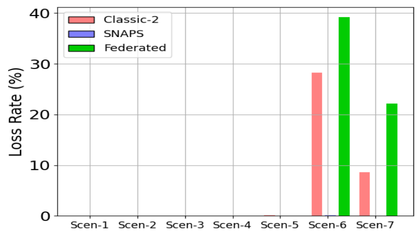


(c) CPU usage

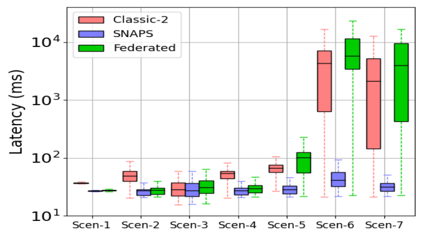
Fig. 8: Benchmark comparison: classic vs. federated vs. SNAPS.



(a) Per-link traffic utilization



(b) Message loss rate



(c) Per-message latency

Fig. 9: Performance comparison under realistic network settings.

750 µs) and saves about 400 µs when compared with federated mode. All three modes exhibit the same tail latency of 1–5 ms, which is caused by RabbitMQ for 10% of the traffic.

Next, we re-examine the latency under different traffic rate conditions. Fig. 8(b) shows boxplots (*i.e.,* minimum, first quartile, median, third quartile, and maximum) of per-message latency as a function of the publisher's transmission rate controlled by <*message size*, *publishing interval*>. Outliers are omitted for readability. Given a fixed publishing interval at 4.5 ms, increasing the message size from 1.5 KB up to 24 KB (and hence the traffic rate from 2.6 Mbps to 41.6 Mbps) causes the latency to increase in all three modes. While SNAPS's median latency is still better than federated RabbitMQ, the latency deterioration (amount of increase with growing traffic rates) is more pronounced for SNAPS than for federated. The variation in latency also tends to become higher for SNAPS. This is due to the `OpenPGM` implementation adopted in the Agent, whose large message handling (fragmentation and re-assembly) seems less optimized.

The right part of Fig. 8(b) shows several examples of faster traffic rate (70–100 Mbps) obtained by decreasing the publishing interval. It shows that increasing traffic rate by shortening the publishing interval reduces per-message latency. For example, latency in the classic case closely follows the publishing interval. This is due to message batching in RabbitMQ, which causes the higher latency with longer publishing intervals. SNAPS's median latency overhead compared to classic is within 750 µs, but with more variation. Overall, SNAPS's latency is still lower than federated.

As a takeaway, it is no surprise that SNAPS and federated introduce more processing latency than a single RabbitMQ in classic. However, the additional overhead ($\sim$1 ms) is small enough to be easily outweighed by the savings in network delays in SNAPS (see Section IV-B2). Compared to federated,

TABLE I: Scenario description.

Scenario	Publisher	Subscribers	Message size (KB)	Message rate (Mbps)
Scen-1	1	3	1.5	2.6
Scen-2	1	2, 3, 5, 7, 10, 8, 4	1.5	2.6
Scen-3	2	1, 3, 5, 7, 10, 8, 4	1.5	2.6
Scen-4	1	2, 3, 4, 5, 6, 7, 8, 9, 10	1.5	2.6
Scen-5	1	2, 3, 4, 5, 6, 7, 8, 9, 10	6	10.4
Scen-6	1	2, 3, 4, 5, 6, 7, 8, 9, 10	12	20.8
Scen-7	1	2, 3, 5, 7, 10, 8, 4	12	20.8

SNAPS is shown to outperform in terms of processing latency, even with the additional Agent.

Next, we evaluate SNAPS in terms of CPU usage. In particular, we focus on CPU overhead of Edge Brokers as they will be deployed at edge cloud locations, where compute resources are limited unlike in data center clouds. In Fig. 8(c), we compare aggregate CPU usage of Edge Brokers against that of federated native RabbitMQ brokers, as both of them will use edge cloud resources. The classic deployment is included as a reference. It shows that, across in all traffic conditions, Edge Brokers consume less CPU than federated brokers. In case of federated, broker-to-broker communication via RabbitMQ's federated plugin incurs much higher CPU cost in RabbitMQ. Even with the additional CPU overhead from the Agent, Edge Brokers still consume overall less CPU than federated brokers. The reason for lower CPU usage of classic is simply because a single RabbitMQ broker handles all publish-subscribe communication.

2) Performance under Realistic Network Settings: In the next set of experiments, we study SNAPS's performance in emulated network environments. We use the real ISP network topology depicted in Fig. 4 with bandwidth and latency per link set to 100 Mbps and 10 ms, respectively. We consider seven *scenarios* summarized in Table I. Each scenario is defined by where (*i.e.,* on which hosts) a publisher and a set of subscribers are running. Furthermore, we introduce several message sizes and transmission rates. In classic, we assume that a single RabbitMQ broker is located at host `2`. Thus, Scen-3 is the best case scenario for classic as the broker is co-located with the publisher at host `2`. We have investigated a larger set of scenarios than the ones reported here. The scenarios chosen in Table I are representative of the broader set of results.

Results – We start by analyzing the traffic carried by different links in each scenario. Fig. 9(a) shows boxplots of per-link traffic rate across different scenarios in each mode. Note that "Classic-2" means classic RabbitMQ with a centralized broker at host `2`. The key observation is that, on SNAPS, traffic rate is almost the same across all active links in the network (hence no variation in the boxplots). In fact, the observed link utilizations roughly match the publisher's sending rate, e.g., 2.6 Mbps for Scen-1 to Scen-4. This is only possible thanks to in-network multicast. In comparison, classic and federated RabbitMQ introduce traffic hotspots in specific links, depending on where the subscribers are located in each scenario. With high sending traffic (10.4 Mbps in Scen-5 and 20.8 Mbps for Scen-6 and Scen-7), some links are actually saturated at their current 100 Mbps capacity. As a consequence, they result in significant loss rates (10–40% as shown in Fig. 9(b)) for both classic and federated modes in Scen-6 and Scen-7.

Fig. 9(c) shows boxplots of per-message latency across different scenarios and modes. As discussed above, classic RabbitMQ has overall the worst latency due to the path inflation caused by one centralized server. For example, in Scen-2, all traffic published by a client at host `1` needs to be first transferred to host `2`. In Scen-3, classic RabbitMQ achieves a latency comparable with SNAPS and federated RabbitMQ because the publisher is co-located with the broker. In scenarios from 1 to 4, the federated mode achieves similar latency to SNAPS, minus some small extra latency discussed previously. But in Scen-6 and 7, federated suffers from dramatic latency increase (up to multiple seconds) due to the link saturation (and packet losses) with increasing traffic rates.

3) Traffic Engineering (TE): As explained earlier, one advantage of SRv6-based multicast in SNAPS compared to IP multicast and BIER is its TE capability. We perform the following experiment with the same network in Fig. 4 to demonstrate its TE capability. One publisher at host `2` sends a data stream at 17 Mbps to three subscribers at hosts `5`, `7` and `8`, respectively. Two BSIDs are used for replication at routers `10` and `16`. To introduce congestion, we use `iperf` to generate UDP background traffic on link `12–16`. The SR policy is configured to shift multicast traffic from link `12–16` to links `12–4` and `4–16` when the bandwidth usage exceeds 65%. Latency of links `4–12` and `4–16` are set to 5 ms, while other links remain at 10 ms; this ensures both the default shortest path and TE path have the same propagation delay.

Fig. 10(a) shows per-message latency experienced by the publish-subscribe clients across different background traffic rates, with and without TE enabled. It shows that the median latency for TE case is slightly higher than that for non-TE when the background traffic rate is relatively low. This is because the TE path has one additional router that introduces processing delay. However, as traffic rate grows higher and congestion delay becomes more significant, median latency for TE becomes lower than that for non-TE. Messages are slowed down due to congestion in non-TE case; while the TE-path is congestion-free. Fig. 10(b) shows the corresponding message loss rate in both cases. Not surprisingly, loss rate for non-TE case increases as background traffic rate increases; while the loss rate for TE almost always remains at zero.

V. Related Work

SDN-aware publish-subscribe. Prior work has explored a new design of publish-subscribe systems that leverage OpenFlow SDN [13]. Although OpenFlow-based multicast delivery can achieve better performance than the OTT approach, it requires users to adopt a new application interface, which is often not practical as many existing publish-subscribe applications are already deployed.

New publish-subscribe data plane. There have been efforts to realize content-based publish-subscribe directly over P4-programmable hardware. These works include encoding either distribution trees [14] or published content [15] in P4 packet headers, or proposing a generic packet subscription abstraction [16]. However, they require more disruptive change in the network infrastructure. Their packet-based interfaces suffer from the same compatibility problem of the SDN-based approach. In-network content inspection would not even be possible for encrypted content. Finally, their scalability is not known in terms of distribution tree size and content diversity.

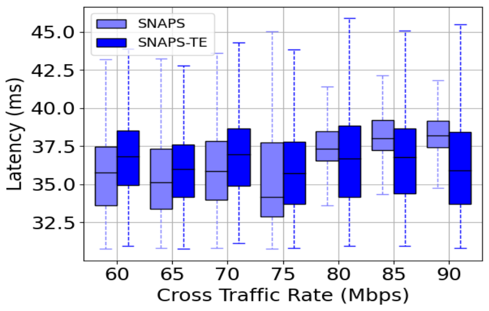
Multicast scalability. Elmo [17] attempts to address scalability of one-to-many communications in virtualized data center environments. It encodes multicast forwarding state inside a packet header and relies on P4 programmability of network and hypervisor switches. However, as its scalability is achieved by exploiting tiered data center topologies and data center's ability to handle non-standard P4 packets, Elmo is not suitable for wide-area networks.

QoS support in publish-subscribe. There have been proposals to improve OTT publish-subscribe systems from QoS perspectives. These efforts include dynamic load re-balancing among available brokers for timely message delivery [18], [19], intelligent data transformation at the edge for latency and bandwidth requirements [20], [21], and transparent client migration across brokers for optimal QoS [22]. They are based on network-oblivious optimizations, and hence are orthogonal to our network-assisted approach.

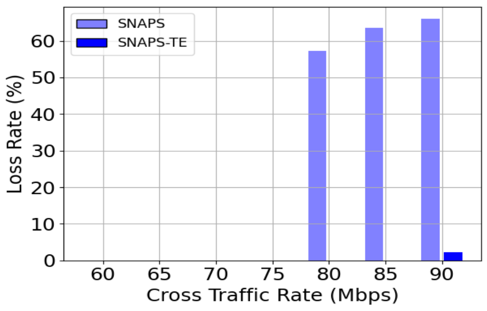
VI. Discussion

In the following we discuss possible extensions of SNAPS and future works we plan to explore.

SRv6-assisted multicast. In SNAPS, NFs could be used to bring additional flexibility to SRv6-assisted multicast. For example, NFs can be deployed at strategic locations in a multicast tree for in-network content filtering based on TLV fields carried by SRv6 packets, especially when receivers have



(a) Per-message latency vs. cross traffic rate (Mbps)



(b) Message loss rate

Fig. 10: Benefits of traffic engineering on SNAPS.

different subscription criteria. This motivates a *unified* multicast tree construction algorithm that systematically determines NF placements and replication segments to minimize router overheads while supporting necessary multicast.

Multi-provider support. Our current design assumes a *single-provider network*, since it requires centralized control and coordination. To extend the solution to a *multi-provider environment*, we need to tackle two main challenges. First, building and maintaining multicast trees Internet-wide is not scalable. A more feasible approach would be to allow each provider or network domain to build its own multicast trees, and provide a mechanism to allow such trees to be combined and shared by different applications. Second, this requires a signaling protocol to set up a multicast tree across all providers involved, and tunnel through legacy networks that do not support SRv6 (e.g., AMT [23], [24]).

VII. Conclusion

This paper has presented SNAPS, an SRv6-assisted publish-subscribe architecture. SNAPS is a case of cross-layer design, leveraging tight integration between a publish-subscribe application system and the network, but without introducing any non-sustainable disruptive changes in either interface. With inherent flexibility of SRv6 network programming and ongoing adoption of SRv6 in production networks as well as in end-host stack, we envision that network-integrated application design approaches will gain traction. SNAPS is a case in point to demonstrate the possibility.

Acknowledgements. We would like to thank Adiseshu Hari and Mostafa Uddin for their contribution at an earlier stage of the work.

References

[1] G. V. Research, "Edge Computing Market Share & Trends Report, 2021–2028," May 2021.

[2] "RabbitMQ Federation Plugin," https://www.rabbitmq.com/federation.html.

[3] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," https://datatracker.ietf.org/doc/rfc8986/, 2021, RFC 8986.

[4] "State of SRv6 – October 2019," https://www.segment-routing.net/images/20191029-01-State-of-SRv6-October-2019.pdf, 2019.

[5] "AMQP 0-9-1 Protocol Specification," https://www.rabbitmq.com/protocol.html.

[6] I. Wijnands, E. C. Rosen, A. Dolganow, T. Przygienda, and S. K. Aldrin, "Multicast Using Bit Index Explicit Replication (BIER)," https://datatracker.ietf.org/doc/rfc8279/, 2018, RFC 8279.

[7] H. Tian, F. Zhao, C. Xie, T. Li, J. Ma, R. Mwehaire, E. Chingwena, S. Peng, Z. Li, and Y. Xiao, "SRv6 Deployment Consideration," https://datatracker.ietf.org/doc/draft-tian-spring-srv6-deployment-consideration/, January 2021.

[8] T. Speakman *et al.*, "PGM Reliable Transport Protocol Specification," 2001, RFC 3208.

[9] D. Lebrun and O. Bonaventure, "Implementing IPv6 Segment Routing in the Linux Kernel," in *Proc. Applied Networking Research Workshop*, 2017.

[10] "VPP/Segmented Routing for IPv6," https://wiki.fd.io/view/VPP/Segment_Routing_for_IPv6.

[11] "The Internet Topology Zoo," http://www.topology-zoo.org.

[12] "REPETITA: Repeatable Experiments for Performance Evaluation of Traffic-Engineering Algorithms," https://github.com/svissicchio/Repetita.

[13] T. Akiyama, Y. Kawai, Y. Teranishi, R. Banno, and K. Iida, "SAPS: Software Defined Network Aware Pub/Sub – A Design of the Hybrid Architecture Utilizing Distributed and Centralized Multicast," in *Proc. IEEE 39th Annual Computer Software and Applications Conference*, 2015.

[14] C. Wernecke, H. Parzyjegla, G. Mühl, P. Danielis, and D. Timmermann, "Realizing Content-Based Publish/Subscribe with P4," in *Proc. IEEE NFV-SDN*, 2018.

[15] R. Kundel, C. Gärtner, M. Luthra, S. Bhowmik, and B. Koldehofe, "Flexible Content-based Publish/Subscribe over Programmable Data Planes," in *Proc. IEEE/IFIP Network Operations and Management Symposium*, 2020.

[16] T. Jepsen, A. Fattaholmanan, M. Moshref, N. Foster, A. Carzaniga, and R. Soulé, "Forwarding and Routing with Packet Subscriptions," in *Proc. ACM CoNEXT*, 2020.

[17] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source Routed Multicast for Public Clouds," in *Proc. ACM SIGCOMM*, 2019.

[18] J. Gascon-Samson, F.-P. Garcia, B. Kemme, and J. Kienzle, "Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud," in *Proc. IEEE ICDCS*, 2015.

[19] D. Dedousis, N. Zacheilas, and V. Kalogeraki, "On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems," in *Proc. IEEE ICDCS*, 2018.

[20] Z. Wen, D. L. Quoc, P. Bhatotia, R. Chen, and M. Lee, "ApproxIoT: Approximate Analytics for Edge Computing," in *Proc. IEEE ICDCS*, 2018.

[21] A. George, A. Ravindran, M. Mendieta, and H. Tabkhi, "Mez: An Adaptive Messaging System for Latency-Sensitive Multi-Camera Machine Vision at the IoT Edge," *IEEE Access*, vol. 9, 2021.

[22] T. Rausch, S. Nastic, and S. Dustdar, "EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications," in *Proc. IEEE International Conference on Cloud Engineering*, 2018.

[23] G. Bumgardner, "Automatic Multicast Tunneling," https://datatracker.ietf.org/doc/rfc7450/, 2018, RFC 7450.

[24] "Akamai Multicast Interconnect for Carriers (AMIC) - Proof of Concept," https://community.akamai.com/customers/s/article/Akamai-Multicast-Interconnect-for-Carriers-AMIC-Proof-of-Concept.