Data plane assisted state replication with Network Function Virtualization

Data plane assisted state replication with Network Function Virtualization

Iman Lotfimahyari, German Sviridov, Paolo Giaccone, Andrea Bianco
Dipartimento di Elettronica e Telecomunicazioni - Politecnico di Torino - Torino, Italy
e-mail: firstname.lastname@polito.it

Modern 5G networks are capable of providing ultra-low latency and highly scalable network services by employing modern networking paradigms such as Software Defined Networking (SDN) and Network Function Virtualization (NFV). The latter enables performance-critical network applications to be run in a *distributed fashion* directly inside the infrastructure. Being distributed, those applications rely on sophisticated state replication algorithms to synchronize states among each other. Nevertheless, current implementations of such algorithms do not fully exploit the potential of the modern infrastructures, thus leading to sub-optimal performance.

In this paper, we propose STARE, a novel state replication system tailored for 5G networks. At its core, STARE exploits stateful SDN to offload replication-related processes to the data plane, ultimately leading to reduced communication delays and processing overhead for VNFs. We provide a detailed description of the STARE architecture alongside a publicly-available P4-based implementation. Furthermore, our evaluation shows that STARE is capable of scaling to big networks while introducing low overhead in the network.

***Index Terms*—5G, NFV, Programmable data planes, State sharing, Publish-subscribe model**

I. Introduction

RAPID growth in demand for fast and reliable services has made of Network Function Virtualization (NFV) and Software Defined Network (SDN) the main pillar of modern 5G management infrastructures, as emphasized by ETSI [1]. SDN introduced the possibility of performing centralized management of the network thanks to the separation between the data plane, left in the switches and the control plane, delegated to a *network controller*, thus enabling fine-tuning of network operations. While SDN provides means of centrally orchestrating and managing the network operations, NFV poses itself as a real game-changer. Thanks to NFV, dedicated hardware devices, previously composing the network infrastructure, are substituted with general-purpose machines such as commodity servers with consequent *virtualization* of most of the legacy devices in the form of Virtual Network Functions (VNF). Network operators are now faced with a scenario in which general-purpose devices are widespread and easily programmable. This opens a new wide range of applications, ranging from IoT to real-time digital applications to be run close to the end customers. At the same time, it creates opportunities for deploying new control and management applications which permit gathering insights on fine-grained network statistics.

While substantial effort is being put into devising novel applications to be run on the newly added network devices, new challenges arise related to the necessity of richer coordination schemes. Most network applications are required to communicate with other applications and to take cooperative decisions either by design or to guarantee their correct functionality. Such is the case of VNFs running Distributed Deny-of-Service (DDoS) detection algorithms, stateful load balancers, or user mobility services. Although having a completely different purpose, all of the aforementioned applications depend on some *global states* which, by design, are shared across multiple VNFs.

At the same time, alongside speed and availability, application-transparency becomes fundamental in modern infrastructures. Developing custom algorithms for each network application to convey state updates to different VNFs becomes prohibitively tedious and expensive. This calls for a clean and versatile scheme capable of enabling data sharing across different VNFs with a bare minimum modification of their original structure.

In this paper, we propose STARE, a mechanism that mitigates the aforementioned issues by providing an application layer-transparent method for state sharing in 5G networks. The replication in STARE is complementary to eventual replication schemes running within clusters of distributed SDN controllers. STARE provides seamless sharing of application-level states by partially replicating those states across different VNFs. This is achieved by exploiting advances in the field of SDN, specifically in the field of programmable data planes. While in traditional SDN, switches are left with little to non-decision-making capabilities, recent advances in the field of SDN introduced the concept of *programmable data planes* by enhancing switches with powerful packet processing pipelines and support of stateful operations.

To this end, STARE exploits these advances by defining a custom *publish-subscribe* protocol run directly in the data plane, thus not requiring any additional hardware. By fully exploiting the potential of programmable switches, STARE can achieve high scalability and no additional latency while leading to memory-efficient *state replication* across multiple VNFs. Furthermore, STARE simplifies the development process of the VNFs by pushing the replication complexity down from each VNF to the network. This is done by

Corresponding author: Paolo Giaccone.

The work has been supported by the European Horizon 2020 Programme through the project 5G-EVE on "European 5G validation platform for extensive trials" (grant agreement n. 815074).

providing a middleware shared across all of the VNFs hosted in a single server which simplifies the design and provides a more transparent way for developing state sharing procedures.

The actual benefit of STARE is both toward the network and VNFs. Indeed, reducing the memory consumption for the matching tables allows us to better exploit the available matching tables. Indeed, those are typically implemented with expensive and limited-size TCAMs and extend the applicability of network applications running in P4 switches. Furthermore, STARE is beneficial for VNF developers since, by giving provided with standard STARE libraries, they are exempt from the burden of managing the real-time replication protocols.

Our main contributions are twofold: (1) validate the approach and show that STARE can run in a real P4-enabled network, and (2) evaluate the resource occupancy of the proposed solution to better understand the scalability of the approach.

A. State sharing in operational scenarios

Our work has been motivated by an operational use case in the context of the European 5G-EVE project[1] targeting the tracking of users' mobility in smart cities [2]. Mobility tracking enables a large variety of new applications: on-demand public transportation, crowd management, mobility planning, social distance monitoring, etc. All these applications fit very well the pervasive nature of mobile networks.

As shown in Fig. 1, the use case comprises a set of WiFi scanners deployed in an area around eNodeBs used in the testbed. The WiFi scanners capture the probe request messages periodically sent by the smartphones advertising the list of the WiFi access points to which they have been connected in the past. A tracking-mobility VNF runs in each edge cloud and processes the anonymized MAC addresses of the mobile devices observed by each WiFi scanner. This enables the possibility of tracking the device's mobility in a completely transparent way for the users. More details are available in [3]. To capture the mobility across an area spanning multiple eNodeBs, it is necessary to correlate the presence and the coverage time of any device across multiple VNFs. This requires sharing the internal states of each VNF. Indeed, whenever a previously detected MAC address by a given VNF is later detected by another VNF, it is possible to infer the spatial trajectory of the device and its speed. In the example scenario of Fig. 1, we have two VNFs (VNF_x and VNF_y) that exchange their internal states (S_x and S_y) with the timestamped list of all the observed MAC addresses. Thanks to this state sharing, each VNF can individually evaluate the direction of the path (either $x \rightarrow y$ or $y \rightarrow x$) and the corresponding average speed.

To support such an application, a centralized approach can be employed. Such an approach would require all the WiFi scanners to send their data to a single VNF which would aggregate the data from different eNodeB. This solution, although feasible in many scenarios, has limited scalability and does not exploit the natural spatial correlation of the mobile nodes. Trajectory and speed are intrinsically local properties from a spatial point of view and can be easily evaluated through a distributed approach. STARE finds excellent applicability in such a scenario as it enables direct state sharing across multiple VNFs through a programmable data plane (e.g., based on P4 switches) connecting the different eNodeBs. Contrary to the centralized solution, STARE is capable of achieving this goal without incurring scalability issues, even whenever the monitored area becomes very large.

[1]https://www.5g-eve.eu/

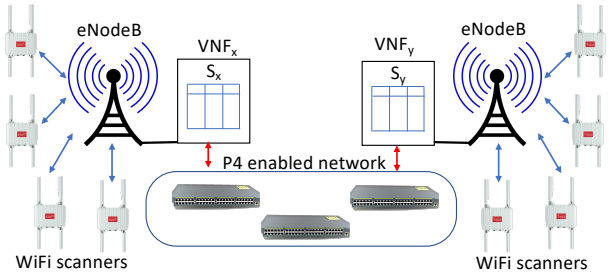


Fig. 1: Urban mobility tracking application with multiple WiFi scanners, leveraging a programmable data plane connecting the different eNodeB for replication of global states between VNFs.

B. Organization of Paper

The rest of the paper is organized as follows. In Sec. II we discuss the main challenges behind the implementation of STARE and the potential benefits it can bring, while in Sec. III we provide implementation details of STARE and its operation modes. In Sec. IV we evaluate STARE in an emulated test-bed to validate the approach, showing that it is feasible to be implemented and run in a real P4-based network. Alongside the evaluation, we quantify the resource occupancy to understand the scalability of the approach. Then we compare it against alternative solutions, either centralized or distributed ones. In Sec. V we provide an overview of how STARE relates to existing technologies based on stateful data planes and highlight possible future directions. Finally in Sec. VI we draw our conclusions.

II. Accelerating State Replication with STARE

The possibility of replicating local states on remote devices is the main pillar of most modern distributed applications. It allows systems to efficiently scale to large sizes, thus providing higher service availability and resilience.

The replication procedure typically involves a set of *states*, defined as generic data structures that are to be shared among different devices. Noteworthy, an efficient replication algorithm must guarantee that all shared states are kept "fresh" with respect to one another, i.e., all replicas should have the same value. An important example of such necessity comes from distributed database systems in which data is located in different regions. In the case of real-time services, the freshness becomes of paramount importance, as otherwise it

may introduce ambiguity or preclude the correct functionality of the service. To this end, multiple state replication schemes, tailored for different applications, have been proposed.

A. State sharing

Traditionally simple gossip protocols have been widely employed to disseminate information among different sites by optimistically propagating information in the network. Although being easy to implement and requiring low computational and storage overhead, this family of algorithms inevitably incurs practical limitations related to the data freshness and resilience to faults. To solve these issues, modern services rely on a broad variety of *replication algorithms* with specific algorithms being chosen to satisfy particular service requirements. Such requirements are summarized by the CAP theorem [4] which states that for a replication algorithm, out of Consistency, Availability, and Partition-tolerance only two properties can be chosen at the same time. More formally the CAP theorem defines three fundamental properties a replication algorithm should satisfy:

1) *Consistency*: Any read performed on any replica of the state will always return the most (globally) recent value of such state or eventually an error. It follows that all non-faulty replicas will always contain the same value for a given state.
2) *Availability*: The availability property guarantees that any read performed on any replica will yield a non-error result independently from the freshness of the state value. It follows that no service disruption is possible in the case of faulty replicas. Yet, if consistency property is not satisfied, the system must be designed in such a way to guarantee correct functionality in the presence of out-of-date state values.
3) *Partition-tolerance*: Partition-tolerance defines the reliability property of the replication algorithm. If this property is satisfied the replication system is guaranteed to operate correctly even in the case of an arbitrary number of state-update messages being dropped or delayed by the network.

Being the reliability a crucial factor in modern infrastructures, it follows that no practical replication algorithm can operate without satisfying the partition-tolerance property. This reduces the freedom of picking the possible property combinations down to Availability-Partition-tolerance and Consistency-Partition-tolerance, which leads to *eventual consistency* and *strong consistency* models for replication algorithms, respectively.

Recently different proposals have been emerging as substitutes to classic replication algorithms. Motivated by the ever-rising number of connected devices and an ever-growing number of different applications common to many devices, schemes such as *publish-subscribe* have been proposed.

B. Publish-Subscribe for VNF state synchronization

Low latency is among the main requirements for modern real-time network applications as it affects systems reactivity and dramatically impacts user satisfaction. The latency requirement is made even more crucial for sharing application-critical states across different applications. From the point of view of a replication scheme, this translates into the necessity of providing availability. Furthermore, since the majority of modern distributed systems such as IoT or cellular networks operate in a loosely-coupled or fully autonomous way, supporting also partition tolerance becomes fundamental. Indeed, no network is immune to faults, thus algorithms that implement some degree of resilience to network failures are required. As a consequence, the Consistency-Partition-tolerance model is typically deployed.

Publish-subscribe has been introduced as a possible solution for such kinds of environments by closely mimicking the eventual-consistency models, thus providing fast and reliable state replication. Publish-subscribe protocols provide a message exchange mechanism between two main actors involved in the algorithm, namely the publishers and the subscribers. The peculiarity of such schemes is that, differently from the traditional replication algorithm, most of the complexity is moved out of the endpoints and pushed inside a central entity, namely the *broker*, which is responsible to collect the data from the publishers and distribute to all the relevant subscribers.

Traditional publish-subscribe algorithms work in three phases. i) A given subscriber can express its interest in a particular topic by specifying it to the broker. ii) The broker builds and keeps track of a map between specific topics and the subscribers interested in those topics. iii) Each time a publisher sends an update on a specific topic to the broker, the broker forwards the update to all of the subscribers of that topic. This scheme effectively enables one-to-many communication, while decoupling in time and space the publishers from the subscribers.

Employing a publish-subscribe scheme is well-tailored for environments comprised of multiple heterogeneous devices as it requires little to no integration inside the devices while offering the flexibility in managing different states and implementing safeness properties of traditional replication algorithms based on eventual consistency. Nevertheless, differently from traditional gossiping algorithms, publish-subscribe schemes may suffer considerably from increased latency in the case of excessive overload of the broker. This in turn limits the applicability of such mechanisms for ultra-low latency applications such as in the case of most of the applications targeted by 5G networks.

C. Programmable data planes in next generation networks

Recent advances in the field of SDN led to considerable improvements not only in the scalability and performance of SDN controllers but most importantly in the architecture of SDN switches.

Programmable data planes emerged as a novel paradigm for the next generation SDN switches. Differently from traditional data planes, programmable data planes [5], [6] introduce the possibility of embedding user-defined programs directly inside switches, thus enabling the possibility of

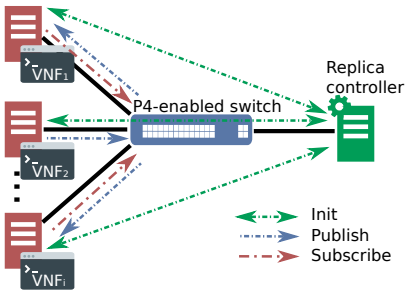


Fig. 2: Overview of STARE communication modes.

executing custom code during the packet processing pipeline at zero increased latency cost. Although commercial products employing programmable data planes still have numerous limitations in terms of resources and programming semantics, the degree of programmability offered by programmable switches remains sufficiently high. This flexibility is mainly because, alongside programmability, programmable switches offer the possibility of keeping *persistent states* inside switches thanks to the presence of stateful elements such as registers and counters, hence they are also usually referred to as *stateful data planes*. At the same time, numerous programmable switch architectures offer the opportunity of defining custom packet headers by directly programming the packet parser which was widely employed to develop novel traffic management schemes [7] and monitoring applications [8].

The combination of all of the novel features introduced by programmable data planes makes them a perfect foundation to build upon for novel algorithms targeting the in-network acceleration of application-layer services.

D. Dataplane-assisted publish-subscribe acceleration

Motivated by the flexibility of programmable data planes and the potential of achieving ultra-low latency, we propose STARE, a state sharing protocol to replicate the states between VNFs, able to fill the performance gap of traditional publish-subscribe protocols by performing acceleration of the protocol via data-plane operations.

The high-level communication pattern of the proposed approach is depicted in Fig. 2. STARE removes the inevitable drawbacks which come from employing a software broker by delegating its functionalities to the data plane. Switches act as a distributed publish-subscribe brokers for inter-VNFs state replication while relying on a centralized controller only for the initialization phase and in case of critical events. All publish/subscribe messages are processed directly by programmable switches, thus allowing packets to always follow the shortest path to their destination without any need of being detoured to a middle-box first. At the same time, such an approach does not introduce any processing latency, since, as previously discussed, programmable switches can process packets at the line rate.

While reducing the communication latency, STARE also provides means for effortless integration in existing and new VNFs. In the following section, we will discuss in detail the implementation and the operation of STARE.

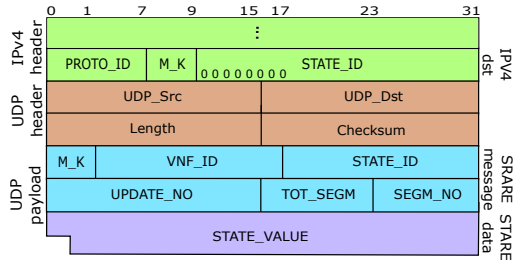


Fig. 3: STARE message format

TABLE I: Coding and definition of different *Message_kind*s.

Message_kind	Definition	Message_kind	Definition
00	Publish	01	SUB-ack
10	SUB-remove	11	SUB-register

III. STARE IMPLEMENTATION

At its core, STARE exploits three main components to provide fast and application layer-transparent publish-subscribe service: i) a custom communication protocol that is easily interpretable both by the VNFs and by the switches, ii) a middleware running at each machine hosting VNFs which exposes clean and easily accessible communication hooks to each VNF and iii) a data plane algorithm running inside the programmable switches responsible of performing broker's tasks. In the following section, we closely analyze the implementation and design choices behind each of the three components.

A. STARE communication protocol

STARE heavily exploits features provided by modern programmable data planes. Notably, it relies on the possibility of defining custom protocol headers and the possibility of taking switch-local decisions based on them.

STARE uses the IPv4 *Administratively Scoped* IP Multicast range (239.0.0.0 - 239.255.255.255) as destination IP address in order to disseminate the publish/subscribe messages. STARE exploits this 32-bits length field by including within the essential data required to guarantee correct message identification and forwarding. Specifically, three custom fields are encoded by STARE within the IPv4 destination field that is summarized in Fig. 3: i) a constant 8 bit-long protocol_ID (`PROTO_ID`) equal to (`11101111`), which is required to correctly identify STARE packets, ii) a 2 bit-long Message_kind (`M_K`), which permits to disambiguate among different kind of messages present in STARE whose values are summarized in Table I, and iii) a 22 bit-long field for state_ID (`STATE_ID`) to uniquely identify the ID of the state for which the message has been generated, corresponding to about $4\ 10^6$ maximum number of states. All of the above information, combined with a dedicated destination UDP port (number 65432 in our evaluation), ensures correct identification of STARE packets inside switches and endpoints.

By construction, independently from the size of the network and the number of states, the message overhead is fixed (i.e., 36 bytes for each subscription and publish) and the number

```
apply {
  if (!hdr.ipv4.isValid()){ // Drop non-IPv4 packets
    drop(); return;}
  // Process the IPv4 destination field and
  // check if a STARE message has been received
  ipDstCheck();
  if (loc_metadata.ipDstProtoId == IP_STARE
      && hdr.udp.isValid()
      && hdr.udp.dstPrt == PROTO_STARE){
    checkMsgKindAndInputPortMask();
    if (local_metadata.STARE_msgKind == 0){
      setMcastGrp(); publish();}
    else if (local_metadata.STARE_msgKind == 1)
      sendToCpuPort();// Embedded controller-based
      Drop();// Register-based
    else if (local_metadata.STARE_msgKind == 2)
      updateUnsubscribe();
    else if (local_metadata.STARE_msgKind == 3)
      updateSubscribe();}
  else // process non-STARE packets
    ipv4_lpm.apply();}

```

Listing 1: STARE message processing in P4

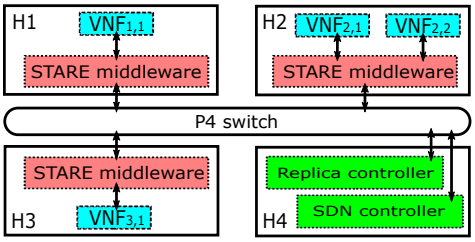


Fig. 4: VNFs, middleware and replica controller running in a STARE scenario.

of exchanged messages is minimum, thanks to the use of spanning trees connecting the subscriber and publisher nodes.

Notably, the use of UDP poses hard constraints on the maximum size of the states to be replicated. For this reason, STARE exploits some semi-application layer segmentation to support large states, thus appearing in a completely transparent way to the data plane.

The general implementation of the switch logic for STARE, alongside the definition of the IPv4 destination type in P4, is depicted in the Listing 1. Since the IPv4 header is discarded at the endpoints once it is received, an application layer message is placed on top of the UDP header including: i) message_kind (`M_K`), ii) Virtual Network Function (VNF)_ID (`VNF_ID`), iii) state_ID (`STATE_ID`), iv) update_number (`UPDATE_NO`), v) the update total_segments (`TOT_SEGM`), and vi) segment_number (`SEGM_NO`). Although the definition of the format for each field can be arbitrarily decided by the programmer, in our implementation we employed a format represented in Fig. 3.

B. STARE Middleware

As previously anticipated, STARE is transparent to the actual implementation of the core logic of each VNF. This is achieved thanks to the fact that STARE incorporates flexible middleware which is responsible for managing the overall process of state replication. Thus, as the replication protocol runs in the network, the VNFs are not directly involved in

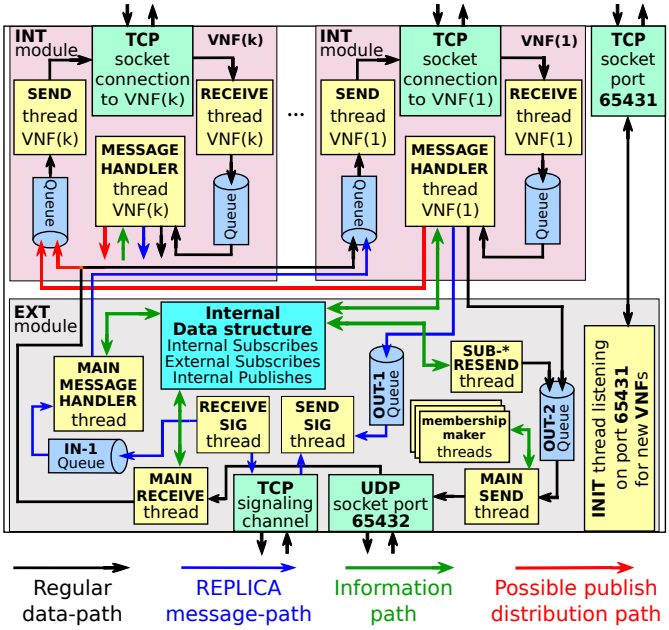


Fig. 5: The software architecture for STARE middleware

it. This approach permits the reduction of the computation overhead incurred by the management of complex replication algorithms at the VNF level and further simplifies both the development of new VNF and integration of STARE in legacy ones. The proposed middleware acts as a single endpoint of the replication scheme within each server, thus it is agnostic to the amount of VMs and VNFs running on each server. The main role of the middleware is de facto to act as a proxy between local VNFs and other servers comprising the network, as shown in the example of 4 servers depicted in Fig. 4.

Fig. 5 depicts an overview of the main components of the sample middleware we implemented. The black lines are the *regular data-paths* carrying the STARE data, the red lines are the possible *Publish distribution paths* which can be either intra-server or inter-server, the green lines define the *Information paths* used to keep the control information updated inside the middleware, and the blue line defines the *replica controller message path*.

To achieve our final goal, we implemented a simple protocol to permit the interaction between the VNF and the STARE middleware. The architecture is made clearer by mimicking all messages exchanged inside the middleware to the behavior of the publish-subscribe protocol adopted at the network level by the programmable switches.

The middleware is composed of two macro elements: i) an external module *EXT module* which is responsible for managing communication between the server hosting the VNFs and the external network and ii) a dedicated internal module *INT module* which is responsible for managing communications between individual VNFs and the EXT module.

The main components of the EXT module can be summarized as follows:

1) **INIT thread:** The main entry point to the middleware is defined by the *INIT thread* which is responsible for

opening a new socket and spawning a listener on a specific TCP port (65431 in our case). This listener provides the main means of communication between the VNFs and the STARE middleware. Whenever a new VNF is instantiated it will try to reach the middleware by binding on this particular port (we assume the port number to be known in each VNF).

2) **SIG SEND and SIG RECEIVE threads:** These processes implement a TCP connection with the replica controller and are responsible for managing all of the control information between the replica controller and the VNFs. This control information, implemented via a set of replica controller messages, includes connection initialization of each VNF and the management of the publish messages losses.
3) **MAIN MESSAGE-HANDLER thread:** *MAIN MESSAGE-HANDLER*, is responsible for reading and rebuilding the signaling messages queued in the incoming *IN-1 Queue*. Whenever a message from the replica controller is received, it will update the *Internal data structure* and will forward the message to the input queue of the proper VNF.
4) **INTERNAL DATA-STRUCTURE:** The internal data structure is responsible for keeping track of the publish and SUB-register requests made by the VNFs connected to the middleware. This component helps the EXT module with the distribution of the publish packets received from the network to the proper VNFs. Furthermore, whenever required, the *MESSAGE-HANDLER thread* of the INT modules can access this component for the information about the local subscribers, if any, for their publishes, so that avoid extra circulation of their published data using the possible *Publish distribution path*s.
5) **MAIN SEND thread:** All the outgoing messages generated by the VNFs, except the replica controller messages, are managed by the *MAIN SEND thread*. This process provides connectivity to the external network by writing data present in the *OUT-2 Queue* on a dedicated UDP socket. Depending on the message kind it will take care of specifying a proper destination for the IP layer, set the UDP layer destination port to another special port number (65432 in our case), and will send the message to the network. Whenever a *SUB-register* message is sent to the network, the employed IPv4 address is recorded in the *IP-multicast membership* module of the OS. This ensures that the OS will be able to deliver the published messages related to the sub-register to the STARE middleware. A similar procedure will be done to remove the registration in the membership module of the OS in case of a SUB-remove message. To address the current challenges, we integrate information from all of the DC network.
6) **MEMBERSHIP-MAKER threads:** These threads permit to overcome the limitation of the UNIX-based OSs for multicast-group memberships which have a limited membership size. Following an attempt in sending a new message, an instance of such thread is spawned whenever then IP-multicast registration in the IP-multicast membership module of the OS fails.
7) **SUB-* RESEND thread:** As a primitive practice for ensuring the delivery of the SUB-register and SUB-remove messages to the network, this process emulates an ARQ protocol. It will periodically check the *Internal data structure* for the SUB-register messages that have been sent but which did not receive a *SUB-ack* message from the network. It will then add them to the *OUT-2 Queue* to be resent. It loosely ensures the delivery of the subscription messages to at least one of the switches participating in our protocol.
8) **MAIN RECEIVE thread:** Similarly to the *MAIN SEND thread*, the *MAIN RECEIVE thread* is responsible for managing all of the incoming messages with the exception for the *replica controller* messages. Upon message reception, this thread will forward the message to the proper *INT module* according to the information stored in the *Internal Data Structure*. Additionally, if the processed message is an *SUB-ack message*, the thread will update the information associated with the waiting list of the non-acknowledged *SUB-remove* or *SUB-register* messages. The list will then be checked by the *SUB-* RESEND thread* periodically.

For each VNF the STARE middleware creates a dedicated instance of an INT module that provides message connectivity to the rest of the middleware. The architecture of such a module closely mimics that of the EXT module: two different threads (SEND and RECEIVE threads) are responsible for reading and writing from/on a dedicated socket which is continuously listened by the VNF, thus providing the last hop to the actual implementation of the VNFs. Analogously to the EXT module, input and output queues are used to buffer messages with the latter being read by a dedicated message handler which is responsible for forwarding the message to the EXT output queue.

C. The role of the replica controller

Although being highly decoupled from centralized entities, STARE still requires minor intervention from a central controller. In particular, during network setup, a *replica controller* provides unique identifiers for each VNF, which will be later exploited by STARE to avoid ambiguity in the communication protocol. Furthermore, globally unique IDs are assigned to each state in an analogous way to what already happens in classic publish-subscribe schemes. Those IDs permit discrimination among different publish/subscribe messages and, as we will show later, are exploited to perform message forwarding in the network. Finally, there is the need of keeping a backlog of published messages in the network which must be used to recover from message losses. The replica controller may be implemented as a dedicated server or as an integrated process running inside the SDN controller.

D. Message loss recovery

In the case of a loss of a publish message, the event can be easily detected inside the subscriber VNFs. This

TABLE II: Example of a state forwarding table and the corresponding match-action rules for messages forwarding

State forwarding table		Port forwarding table	
State_ID — i	register[i]	**Match** Dst. bitmask	**Action** Forward on ports
0	00011110	00011110	{4,5,6,7}
1	10001010	10001010	{1,5,7}
2	10000000	10000000	{1}

is performed by checking the application layer message information regarding the continuity of the segment number in relevance with the state update sequence number and the total segments of the update. Whenever a VNF detects a loss, it will notify the replica controller through a RECOVER message containing the related information from the application message layer, in turn, and the replica controller will provide the lost message by looking for it in the stored backlog. The use of the replica controller for such tasks significantly reduces the resource overhead on each server running STARE. Indeed, all of the backlogs of the published messages are stored at the controller instead of being distributed on every single server. This approach reduces the overall memory requirement for the deployed servers and considerably simplifies the design of the STARE middleware. To keep the memory utilization low, the backlog must be periodically truncated. This can be achieved by actively querying single VNFs for the maximum segment numbers received so far for any given state. This information can then be easily exploited to perform backlog truncation by considering the minimum among the received segment numbers and by truncating the backlog up to that point.

We address the recovery of the SUB-remove and SUB-register messages on both the: i) VNF to switch / switch to VNF and ii) switch to switch segments. As explained in Sec. III-B, we delegate the reliable delivery of these messages to the first P4 switch (VNF to switch segment) through a simple ARQ algorithm and by employing the *SUB-* RESEND* process for both solutions. Employing such a simple ARQ protocol implementation inside a light-weighted embedded-controller on each P4 switch effectively permits to support reliable delivery of these messages between switches.

E. Dataplane Implementation

Concerning the data plane implementation, we consider P4-enabled devices [5] as the main candidate for the implementation of STARE functionalities. This, however, does not limit the generality of our approach since most of the modern programmable switches architectures, as we discussed in Sec. II-C, offer similar capabilities. Nevertheless, in the case of computationally or resource-limited devices, some of the requirements of the STARE data plane implementation may be unfeasible to implement. For this reason, we propose two alternative solutions, the first (denoted as "Register-based") being a pure data plane implementation, while the second (denoted as "Embedded controller-based") being a hybrid data plane/CPU implementation.

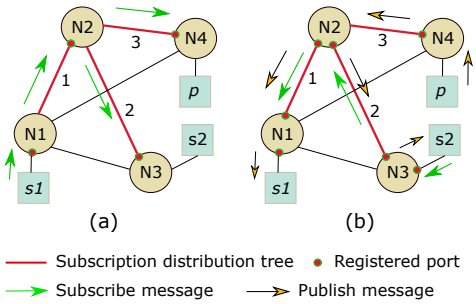


Fig. 6: The general subscription distribution tree and the possible paths traversed by the subscription messages.

For further discussion, we assume that the SDN controller has already set up a subscription distribution tree spanning all the switches to which the servers running the VNFs are connected. An example of this tree is highlighted with the red lines in Figs. 6(a) and 6(b). The subscribe (i.e., SUB-register and SUB-remove) and publish messages will only be forwarded on this tree or a subset of if. Details will be provided in the following sections.

1) Register-based with P4 data plane implementation

A pure data plane implementation requires a considerable amount of stateful resources inside the switch to support the broker functionalities. Analogously to a traditional software broker, the switch must be able to correctly process three main kinds of messages: the subscribe, the unsubscribe, and the publish messages. To do so, switches must keep track of a data structure mapping particular state_IDs specified in the Sub-register messages to a set of output ports on which VNFs requiring updates of that particular state are reachable. This is achieved thanks to a dedicated data structure, namely the *state forwarding table* implemented with an array of registers. Analogously to a hashmap, given a state_ID x, the state forwarding table stores a bitmask B_x of output ports inside the register indexed by index x. Noteworthy, such implementation implies that the number of bits to store each bitmask is equal to the number of ports in the switch. Table II depicts an example of a state forwarding table for an 8 ports switch storing 3 state_IDs alongside the corresponding match-action port forwarding table responsible for multicasting the message on given ports.

Notably, although the distribution tree spans all of the available endpoints, to reduce the total data overhead in the network, only a subset of the links is used each time. Each switch forwards publish messages on a given port only if an active subscriber is reachable through that port (yellow line in Fig. 6a). Similarly, whenever a subscribe message floods the distribution tree, all ports leading to an active publisher are ignored (green line in Fig. 6a) since the publish connectivity is already guaranteed up to that point.

A partial implementation of this solution with P4 is available on `github` in [9].

2) Embedded controller-based with P4 data plane implementation

This second solution is motivated by the fact that the scarce availability of registers can pose hard constraints

on the feasibility of the register-based solution due to the impossibility of storing the state forwarding table. Yet, the match-action tables are plentiful even in low-end devices which provides an alternative for building the state forwarding table. Nevertheless, commercial switches are typically equipped with an onboard CPU that can operate as a local embedded controller, although without the global visibility of the SDN controller, but at a much-reduced latency. To build the state forwarding table, The switch exploits the arrival port ID of the Sub-register messages and uses the state_ID as a key in the state forwarding table to retrieve and eventually update the current destination bitmask. Unfortunately, the current implementation of the P4 programming abstraction does not provide means of direct match-action table manipulation, thus relying on the controller for such a task.

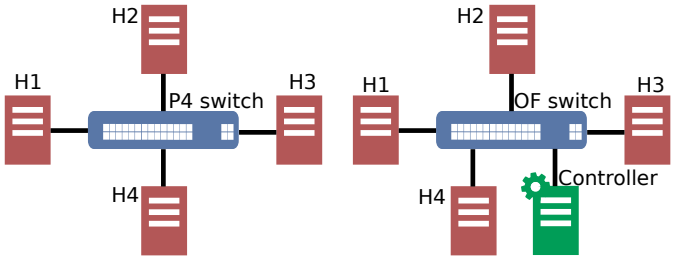
3) Comparison among the two implementations

The register-based implementation relies on a static number of registers, defined at compilation time. This implies that anytime the amount of different states present in the publish-subscribe scheme exceeds the available ones, it is necessary to perform switch reconfiguration. While, in theory, this reconfiguration can be operated live on specialized hardware, in practice it may introduce transient service outage. On the other hand, the embedded controller-based solution can overcome this issue, as match/action tables are plentiful in modern switches, but at the cost of increased complexity and new rule installation latency [10]. Indeed, the processing time of STARE subscribe messages for the *Embedded controller-based* solution can be slightly higher than the *Register-based* solution due to the time required to modify the match-action tables. More on that, the interaction required with the embedded controller can introduce a very small latency, which such latency is still considerably lower in comparison with the interaction to a remote-controller. Due to its general-purpose nature, the embedded controller can implement additional capabilities such as secure communications, e.g., by supporting MACsec to protect network links between P4-based SDN switches as described in [11]. It is worth highlighting that both schemes do not require the interaction of the switch and the VNF with the replica controller or with the SDN controller, since all the related operations are offloaded directly to the data plane.

IV. Experimental evaluation

We implemented the STARE framework and performed a set of experiments to compare it with alternative solutions. We used Mininet [12], a network emulator that provides software models for vanilla SDN switches and P4 switches, and BMV2 [13] software switch to run the experiments based on P4 switches.

First, we tailored our experiments to the 5G use case described in Sec. I-A, where a VNF for the mobility tracking is available at each edge cloud. Fig. 7a depicts the testbed topology, which includes 4 VNFs and one P4 switch. The replica controller runs in H4, and STARE middleware runs in H1, H2, and H3 while to mimic a more heterogeneous



(a) STARE scenario (b) Pure OpenFlow scenario

Fig. 7: Topologies employed in the experimental testbed.

scenario, we consider one VNF running on H1 and H3 and two VNFs running on H2. The VNFs track the users' mobility according to the considered 5G use case. To measure the resources required in the P4 switches and compare the different solutions, we evaluated the memory occupancy in terms of Resident Set Size (RSS) of the process running the virtual switch.

We run an alternative scenario based on a pure OpenFlow approach. Ryu [14] is the SDN controller interacting with the OpenFlow switches and P4 switches, chosen thanks to its simplicity of deployment and its wide support of OpenFlow standards. The STARE middleware has been developed in Python and implemented through the standard Python socket library. For the register-based solution, we have pre-allocated the switch resources by defining 2048 registers inside the P4 program at the compilation time.

To emulate accurately the arrival process of messages generated by the WiFi scanners, every 20 seconds a report message is generated by each scanner. Each message carries a list of MAC addresses (corresponding to the detected mobile devices) with the corresponding timestamps, both encoded in JSON format. The number of detected MAC addresses varies randomly between 10 and 50 to mimic the experimental conditions observed in the 5G EVE use case during rush hours in the area between our university and the main train station of our city.

As a term of comparison, we considered a pure OpenFlow implementation for the adopted publish/subscribe protocol, based on OpenFlow 1.3. In particular, we use Open vSwitch [15] software switch for the experiments based on standard OpenFlow switches. A simple topology for this scenario is shown in Fig. 7b.

In such a scenario the SDN controller takes the responsibility of storing the state forwarding table, which forces the switch to interact with the SDN controller anytime a new subscription occurs. Thus, this scenario is representative of a centralized solution alternative to STARE. Also, the initialization phase, required to assign the VNFs with their corresponding unique IDs and to know the state IDs used to publish and subscribe, is managed centrally by the SDN controller. All the subscription messages received by a switch are sent to the SDN controller, which in turn programs the switch flow tables to associate the corresponding incoming port of the switch as the forwarding port for all the corresponding publish messages. This permits avoiding

TABLE III: Memory occupancy of the three scenarios.

Scenario	Average memory per rule
Register-based P4 solution	119 bytes
Embedded-controller P4 solution	184 bytes
OpenFlow-based solution	586 bytes

additional interaction with the SDN controller whenever an already seen publish message arrives since the packet will be directly sent to all the switch ports corresponding to subscribing VNFs.

We do not report any results regarding latency measurements, since Mininet is an emulated network environment and does not allow us to evaluate meaningful statistics regarding the delays. Nevertheless, it is worth recalling that, by design, P4 switches perform packet processing at the line rate, thus we expect minimum processing delays (which depends on the internal hardware pipeline and the organization of the programmable matching tables of the P4 switch). Thus, by eliminating the necessity of the switch to interact with a remote controller (replica or SDN) anytime a publish message is received, we expect STARE to achieve lower latency in respect to traditional solutions.

A. Resource consumption

We compare the performance of the STARE with the solution based on OpenFlow switches in terms of internal resource consumption within the switch.

1) Registered-based with P4 solution

Fig. 8a reports the average RSS memory, including the 95% confidence intervals and a linear regression over the average values. The achieved accuracy is very high since the relative width of the confidence interval is around 0.2%. From the figure, the average amount of memory is 119.0 bytes for each installed rule.

2) Embedded controller-based with P4 solution

In the adopted match-action table, we evaluate the width of the key. In the flow match-action table, we use an exact matching on the IP address, so 32 bits are needed, and the corresponding action is coded as a 32-bit number representing the multicast group, thus we can expect a minimum of 64 bits for each rule.

The result of the measurement is shown in Fig. 8b, which shows that the memory occupancy increases linearly with the number of installed rules and that the average memory occupancy is 184 bytes per installed rule.

3) OpenFlow-based solution

We evaluated the RSS of the OVSK process and the measurements were collected by the remote SDN controller. The result of the measurements is depicted in Fig. 8c, which shows a step-wise increasing function. Such behavior is due to the internal memory allocation scheme, which employs a batch allocation process in the memory. The average occupancy is 586 bytes per installed rule.

As a summary, the memory occupancy for the three above scenarios is shown in Table III.

TABLE IV: Network messages to deliver a publish message.

Approach	In-net(B)	Out-net(B)	In-net(W)	Out-net(W)
Unicast	32	17	96	17
Broadcast	84	85	84	85
ALM	20	59	26	67
OFM	20	17	26	17
P4 source routing	20	17	26	17
STARE	20	17	26	17

B: Best case W: Worst case

The most efficient solution in terms of memory is based on register-based STARE implementation, whereas the least efficient solution is the one based on a standard OpenFlow switch. The differences are due to the internal memory management process internal to the BMV2 software switch (for both P4-based solutions) and the Open vSwitch software switch (for the OpenFlow-based solution).

B. Traffic overhead

We compare the register-based solution of STARE in terms of the network traffic and protocol overhead with the following five alternative approaches.

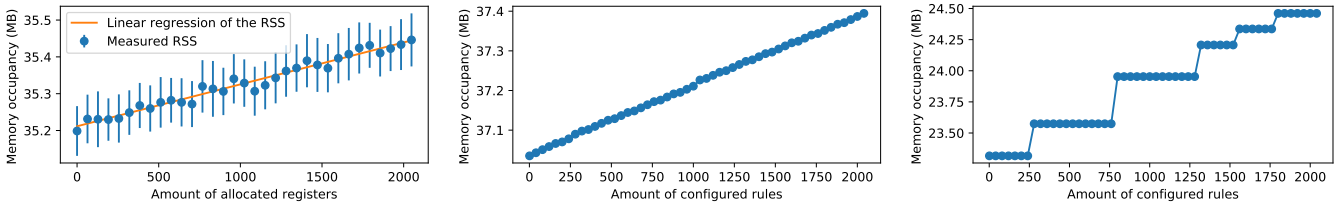
In the *Unicast* approach, the publisher sends one publish message individually to each subscriber. In the *Broadcast* approach, the publisher sends one publish message in the whole distribution tree.

Application-Layer Multicast (ALM) is based on a broker for each switch, which is aware of the switch ports that are on the distribution tree towards the subscribers. This broker will receive the publish packet of the switch and will return the packet to the switch with a header containing the corresponding multicast group.

OpenFlow Multicast (OFM) [16] is an OpenFlow-based approach that uses an IP address and a UDP port number for addressing a multicast-tree stored on the switches. We assume that all these OFM trees have been already configured through related flow rules from the SDN controller.

The *P4 source routing* approach that was recently proposed in [17] as a centralized approach for publish-subscribe based on source-routing multicast implemented in P4. It uses a stack of headers added to the MAC header containing the switch identifiers of the path to the subscribers and the corresponding multicast address for each switch. The SDN controller is responsible for receiving the subscription and informing the publisher on how to generate this header stack.

The traffic overhead depends on the network topology. In our evaluation, we considered a symmetric tree topology connecting the switches with 63 leaves, distributed in 4 layers (comprising root and leaves). All nodes (except the leaves) have 4 children. The total number of links is 84. The subscribers are connected to the leaf switches while the publisher is connected to an arbitrary switch through a dedicated link. We calculated the best and the worst case with respect to selecting any set of leaves for the subscribers and the position of the publisher. Since subscriptions are transient and are defined only in publish-subscribe schemes, for a fair comparison, we consider only the publish phase since it is well defined for all of the considered approaches. Thus, we evaluate



(a) Register-based solution. (b) Embedded controller-based solution. (c) Remote Controller-based solution.

Fig. 8: Comparison of the internal memory occupancy

the traffic in terms of "in-net" traffic, i.e., across links in the considered topology, and in terms of "out-net" traffic, i.e., across the links that will connect the publisher and subscribers VNFs (not considered in the above topology).

Table IV depicts the in-net and out-net messages needed for delivering a publish message to 16 subscribers in the network for the different approaches. The minimum number of in-net traffic is achieved by STARE, P4 source-routing, and OFM since each link in the distribution tree is used just once for each publish message. The results of ALM are worse since it requires each switch to forward the publish messages to the software broker and to receive the corresponding multicast group from it. ALM is similar to STARE, except for the fact that in STARE the switch does not need an external interaction. Unicast is inefficient since the same publish message may be sent multiple times on the same link, and it would be a reasonable solution for few subscribers. Instead, broadcast results to be the least inefficient since each publish message is flooded across the whole distribution tree, independently from the subscribers. Although it is not depicted in the table, the amount of traffic generated by STARE, P4 source-routing, and OFM converges to the amount of traffic generated by broadcast whenever all nodes act as subscribers and it converges to the unicast case whenever there is only one subscriber.

After considering the traffic overhead in terms of the number of packets, we consider the traffic overhead in terms of bytes, for each protocol message. All of the considered approaches adopt standard protocol headers at layers 2, 3, and 4, except for the P4 source routing. [17] adopts a stack of sorted labels that enable simple processing at each switch, yet is it required to be transmitted across links even if not required. Each label is 3 bytes-long and includes the switch identifier and the bitmask with the local destination ports. In the considered scenario with 16 subscribers at the leaves of the topology, it can be shown that, in the best case, the stack comprises between 16 and 20 labels, depending on the level in the topology, with an average of 18 labels per publish message (54 bytes total). In the worst case, the stack comprises between 16 and 24 labels, depending on the level in the topology, with an average of 18.75 labels (56.25 bytes). Note that such overhead may not be negligible in the case of updates corresponding to "small" states, e.g., integer counters.

V. Related works

The work in E-State [18] presents a state management framework to share the states between VNFs by creating a logically distributed state memory. It provides fast reads and writes to flow-related states on the local VNF instance and, when global reads are used, it provides an eventual consistency model, exactly as STARE assumes. In E-State replication is managed directly within the application layer, thus reducing the net resources devoted to the VNF, especially when the replication is directed to a very large number of VNFs. On the contrary, in STARE the replication is offloaded into P4 switches through a middleware running outside the VNF application. This reduces the resources required within each VNF, which is not involved anymore in the replication process. Furthermore, replication latency in STARE is mainly affected by the network congestion and communication details, and not by the CPU load on the VM/container running the VNF. Moreover, STARE is providing a simple interface to the middleware to facilitate the VNF development.

The work [19] addresses the problem of live-migrating VNFs. The migration requires to transfer of the internal state of the original VNF to the new one in real-time. Thus, the problem is similar to our state sharing problem between VNFs, but with the peculiarity that the replication process occurs just once and in one direction. The proposed framework SHarP separates the state migration problem from the traffic steering scheme that reroutes the flows from the old VNF to the new one. By decoupling the state replication by the traffic management, the framework is compatible with any replication scheme, and in particular, it is compatible with our STARE scheme, offloading the state replication into the data plane.

There is a substantial ongoing effort in investigating application-level acceleration and state replication via programmable data planes. This is enabled by multiple stateful switch architectures capable of holding persistent states and performing custom packet processing being proposed in the last years. OpenState [20], which introduces a minor architectural extension to the OpenFlow data plane and control plane, is capable of supporting custom persistent states inside the switch which can be interacted with by custom routines upon packet arrival, internal switch signals, or timers. Open Packet Processor (OPP) [21] extended OpenState by adding additional features that allow the executions of Extended Finite State Machines (EFSM) directly in the data plane. Similarly, P4-enabled switches [5] switches are capable of performing the same tasks as OPP-based switches, yet also, they provide a comprehensive high-level programming language for the definition of custom packet processing routines.

In NetPaxos [22] the authors propose to move part of the

traditional Paxos consensus protocol into the network in an effort of accelerating its application-layer performance. Being one of the most deployed protocols in distributed systems and a fundamental building block to several distributed applications [23], [24], [25], NetPaxos showed the potential benefit programmable data planes can bring to the application level. On the contrary, in SwingState [26] a first attempt has been made of performing state migration entirely in the data plane. The authors were able to dynamically migrate an in-switch internal state across different switches but assumed a single copy of the state that is on-demand migrated across the network. A step forward towards full data plane solutions has been made in [27], which proposed LOADER, a programming abstraction for defining distributed network applications based on replicated states. Yet, similarly to [26], the authors of LOADER focused just on internal states available at the switches, thus, without providing any interaction with other elements of the network such as servers or VNFs. STARE, instead, closes this gap by providing a comprehensible middleware between the application layer and the network. The consistency model for the state replication adopted in STARE is the same as the one considered in [27], for which the P4 implementation on the data plane provides a prototypal idea for the implementation of the publish-subscribe scheme in STARE.

More related to our work, the work in [28] proposed to use the concept of group table introduced in OpenFlow 1.3 to implement a publish-subscribe scheme to increase the performance in terms of data delivery and notification process. While it takes a step towards providing application-layer acceleration using SDN, it remains highly dependent on the remote controller for any change in the subscriptions, while STARE manages the subscription directly on the data plane. The idea of [17] which is source-routing pubsub with P4, is that the publisher sends the notification packet containing a stack of headers similar to MPLS after the Ethernet header. Each stack has two fields containing the switch ID and a bit-mask of the ports of the mentioned switch that the copies of the publish packet (notification) should be sent out of them.

The authors of [17] propose a content-based publish/subscribe based on source-routing multicast which exploits P4 programmable data planes. They show that their approach is better than competing schemes, including different implementations of Application Layer Multicast (ALM) with software brokers, unicast, and broadcast, while having a similar structure to OpenFlow Multicast (OFM) [16]. In our paper, we compared STARE with the same approach. We show that our solution behaves similarly to [17] in terms of the number of packets and network overhead during the notification publish. Furthermore, apart from lacking scalability due to the absence of a complete decoupling of publishers and subscribers, the authors of [17] do not clarify how the subscription problem is handled in dynamically changing scenarios. Indeed, dynamic subscriptions and publishes are completely supported by construction in STARE and it is handled without interaction with the SDN controller.

The work in [29] has addressed the problem of how to optimally place replicated states within a programmable data plane-enabled network. This has been performed by taking into consideration both the data traffic and the replication traffic overheads. The work has proposed a framework to optimize the number of replicas and their placement within the network, taking into account the main trade-off between data traffic and replication traffic. Differently from [29], STARE does not optimize the placement of the VNFs within the network. Yet, we foresee optimizing VNF placement as a potential future work.

VI. Conclusion

In this paper, we propose STARE, a low-latency publish-subscribe architecture for NFV and SDN-enabled networks. STARE aims at achieving fast state replication among different VNFs in 5G networks. To do so, it exploits recent advances in the field of programmable data planes by offloading the functionalities of a traditional publish-subscribe broker to the programmable switches. At the same time, STARE provides an easily deployable middleware that permits rapid integration of new and existing VNFs by exposing simple APIs which can be accessed easily inside the source code of each VNF. We validate our approach by employing an emulated, yet realistic, testbed network with both P4-enabled switches and vanilla SDN based on OpenFlow switches. We also compared our solution with some other approaches for the distribution of the published notifications with different technologies.

Our experiments show that using STARE, in combination with P4-enabled switches, leads to completely homogeneous traffic in the network by using fixed-length headers that use the least possible number of packets for delivering the published notifications to the subscribers. It also remarks that as a trade-off by sending in total a few larger amount of bytes than one or two approaches, the approach will not need to interact with a software broker nor mapping the total path between publishers and subscribers inside the packet, which leads to completely decoupling of the publishers and subscribers. . At last, it incurs a smaller overhead in terms of memory in comparison with the traditional approaches based on vanilla SDN. Furthermore, thanks to the limited interaction with a centralized controller and removing software brokers, STARE is expected to lead to significantly lower state replication latency compared to the traditional publish-subscribe schemes.

References

[1] "ETSI," https://www.etsi.org/.

[2] F. Moggio, M. Boldi, S. Canale, V. Suraci, C. Casetti, G. Bernini, G. Landi, and P. Giaccone, "5G EVE a European platform for 5G application deployment," in *ACM WinTECH*, 2020.

[3] K. Gebru, C. Casetti, C. F. Chiasserini, and P. Giaccone, "IoT-based mobility tracking for smart city applications," in *EuCNC*, 2020.

[4] E. Brewer, "CAP twelve years later: How the rules have changed," *IEEE Computer Magazine*, 2012.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, 2014.

[6] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *NSDI*, 2019.

[7] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2016.

[8] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[9] "pub_sub.p4." [Online]. Available: https://github.com/imanlotfimahyari/State-Sharing-p4-python/blob/master/pubsub/pubsub_register/pub_sub.p4

[10] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in SDN-enabled switches," in *ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.

[11] H. Frederik, S. Mark, H. Marco, and M. Michael, "P4-MACsec: Dynamic topology monitoring and data layer protection with MACsec in P4-based SDN," *IEEE Access*, 2020.

[12] "MININET An instant virtual network on your laptop (or other PC)," http://mininet.org/.

[13] "BMV2 Behavioral Model Version 2," https://github.com/p4lang/behavioral-model.

[14] "Build SDN agilely." [Online]. Available: https://osrg.github.io/ryu/

[15] "OVSK Open Virtual Switch," https://www.openvswitch.org/.

[16] T. Akiyama, Y. Teranishi, R. Banno, K. Iida, and Y. Kawai, "Scalable pub/sub system using OpenFlow control," *Journal of Information Processing*, 2016.

[17] C. Wernecke, H. Parzyjegla, G. Muhl, P. Danielis, and D. Timmermann, "Realizing content-based publish/subscribe with P4." IEEE NFV-SDN, 2018.

[18] M. Peuster and H. Karl, "E-state: Distributed state management in elastic network function deployments." IEEE Conference on Network Softwarization (NetSoft), 2016.

[19] M. Peuster, H. Kuttner, and H. Karl, "A flow handover protocol to support state migration in softwarized networks." Wiley Online Library, 2019.

[20] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "OpenState: programming platform-independent stateful OpenFlow applications inside the switch." ACM SIGCOMM, 2016.

[21] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi, "Implementing advanced network functions for datacenters with stateful programmable data planes," in *IEEE LANMAN*, 2017.

[22] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soule, "NetPaxos: Consensus at network speed." ACM SIGCOMM, 2015.

[23] M. Burrows, "The chubby lock service for loosely-coupled distributed systems." USENIX OSDI, 2006.

[24] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter." ACM SOSP, 2011.

[25] J. C. Corbett and al., "Google's globally-distributed database." USENIX OSDI, 2012.

[26] M. Bonola, R. Bifulco, L. Petrucci, S. Pontarelli, A. Tulumello, and G. Bianchi, "Swing State: consistent updates for stateful and programmable data planes." ACM SOSR, 2017.

[27] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, and G. Bianchi, "LOcAl DEcisions on Replicated States (LOADER) in programmable dataplanes: Programming abstraction and experimental evaluation," *Computer Networks (Elsevier)*, 2020.

[28] M. Hungyo and M. Pandeyl, "SDN based implementation of publish/subscribe paradigm using OpenFlow multicast." IEEE ANTS, 2016.

[29] A. S. Muqaddas, G. Sviridov, P. Giaccone, and A. Bianco, "Optimal state replication in stateful data planes," *IEEE Journal on Selected Areas in Communications*, 2020.

Iman Lotfimahyari received his BSc and MSc in Electronics Engineering from IAU University in 2003 and 2007 respectively. In March 2020, he received his second MSc in Telecommunication Engineering from Politecnico di Torino, Italy, and joined the Telecommunication Networks Group at Politecnico di Torino as a Ph.D. student. His current research interests involve programmable data planes for SDN and Blockchain.

German Sviridov received his BSc in Computer Engineering and MSc in Telecommunication Engineering, both from Politecnico di Torino, Italy. In 2017 he joined the Telecommunication Networks Group at Politecnico di Torino as a Ph.D. student. He worked as a visiting researcher at Huawei Technologies France between 2019 and 2020. His current research interests involve programmable data planes for SDN and scheduling mechanisms for data center networks.

Paolo Giaccone received the Dr.Ing. and Ph.D. degrees in telecommunications engineering from the Politecnico di Torino, Italy, in 1998 and 2001, respectively. He is currently an Associate Professor in the Department of Electronics, Politecnico di Torino. During 2000-2001 and in 2002 he was with the Information Systems Networking Lab, Electrical Engineering Dept., Stanford University, Stanford, CA. His main area of interest is the design of network control and optimization algorithms.

Andrea Bianco is Full Professor and Department Head of the Dipartimento di Elettronica e Telecomunicazioni of Politecnico di Torino, Italy. He has co-authored over 200 papers published in international journals and presented in leading international conferences in the area of telecommunication networks. His current research interests are in the fields of protocols and architectures of all-optical networks, switch architectures for high-speed networks, SDN networks, and software routers.