SMART: Dual-channel Southbound Message Delivery in Clouds with Rate Estimation

Luyao Luo[1,2] Gongming Zhao[1,2] Hongli Xu[1,2] Chun-Jen Chung[3] Liguang Xie[4]
[1]School of Computer Science and Technology, University of Science and Technology of China
[2]Suzhou Institute for Advanced Research, University of Science and Technology of China
[3] Vancouver Research Center, Huawei Technologies Inc, Canada
[4] Virginia Tech, USA

***Abstract*—Driving southbound messages from a cloud control plane down to the distributed data plane on every compute node is one of the critical challenges in public clouds. Existing message delivery solutions solely based on remote procedure call (RPC) or message queue (MQ) tend to overlook strict resource constraints, *e.g.*, network bandwidth and CPU capacity. This often results in extensive overhead in the control plane or message redundancy in the data plane, especially when a cloud receives highly concurrent user requests or experiences a rapid expansion. To this end, we design a dual-channel southbound message delivery framework, namely SMART, which combines an RPC channel with an MQ channel, to maximize the resource utilization in the cloud network. In the control plane, we implement a message parsing mechanism and propose a delivery channel selection algorithm based on the deep reinforcement learning (DRL) approach to support efficient dual-channel delivery under resource constraints. In the data plane, we design a message agent on each compute node to ensure the order preservation and state consistency of southbound messages. Both experimental and large-scale simulation results show that SMART demonstrates a reduction in control plane overhead by 64% compared to RPC and redundant messages by 45% compared to MQ, respectively.**

***Index Terms*—Southbound Message Delivery, Message Queue, Remote Procedure Call, Virtual Private Cloud.**

I. Introduction

In a multi-tenant cloud, tenants deploy virtual machines (VMs) on compute nodes in the distributed cloud data plane and manage the VMs through unified restful APIs by the cloud control plane [1], [2]. The control plane processes tenants' requests, and sends network configuration messages, also called *southbound messages*, to compute nodes [3]. Over the past decade, we are observing rapid growth of the number of customers and the continuous expansion of individual network size. As a result, the number of southbound messages is mounting a rapid pace [4]–[6]. Thus, how to achieve efficient and resource-saving southbound message delivery has become a critical issue for hyper-scale cloud deployments [7]–[9].

With the development of cloud computing, many distributed end-to-end communication protocols have emerged, such as message passing interfaces (MPI) [10], JMS (Java Message Service) [11], and remote procedure call (RPC) [12]–[14]. As one of the most common and efficient protocols in distributed systems, RPC establishes TCP connections between controllers and compute nodes. In this way, each compute node directly communicates with controllers and receives all the required configuration messages without redundancy. The downside is that, each TCP connection with a compute node requires a certain amount of resources (*e.g.*, CPU and bandwidth) [15]. With the increasing network scale, the direct communication method will cause a high load on the control plane, leading to message congestion or loss, especially when the burst southbound traffic happens [8], [9]. We perform the testbed experiments on gRPC [16], a widely used open-source RPC framework, in Section VII-D. The experimental results show that too many connections will lead to a significant delay in the southbound message delivery.

As an alternative, the message-oriented middleware (MOM) model [17], [18] has become one of the most widely adopted solutions for building cloud infrastructures and tenant applications in a large-scale cloud (*e.g.*, Azure Queue and Amazon Message Queuing Service) [19]. It reduces southbound control overhead and improves system ability to deal with a large amount of southbound messages by decoupling the data plane from the control plane with the help of a messaging middleware [7], [19], [20]. Message Queue (MQ), as the most common messaging middleware implementation, sets up multiple queues in the MQ server for storing and forwarding southbound messages between the control plane and the data plane. Under this model, the controller pushes southbound messages to the MQ server, while compute nodes pull southbound messages from one or more queues on demand, thus avoiding high overhead on the control plane [21], [22].

However, an MQ server commonly supports a relatively small number of queues due to its capacity constraint (*e.g.*, disk I/O) [23], [24]. For example, the experimental results in [21] show that setting up a few hundred queues will cause the Apache Kafka server [23] (a well-known open-source message queue) to crash frequently. Considering a large number of southbound messages, we should send several types of messages to one queue [5]. In this way, once a compute node pulls messages from a queue, it has to receive all messages in this queue so as to catch valid messages. As a result, some useless messages from this queue may be received. It means, the limit on the number of queues causes inevitable message redundancy on compute nodes. Accordingly, the redundant messages will occupy valuable network bandwidth and memory of compute nodes, leading to the increased delay in southbound message delivery.

From the observation above, though RPC provides redundant-free southbound message delivery, it may cause a massive overhead on the control plane and unacceptable delay when the cloud network scales up to a certain amount. On the contrary, MQ provides lower control overhead, but it will lead to a considerable overhead in the data plane due to redundant messages. In order to integrate their pros, but mitigate the cons of these two solutions, we design an efficient southbound message delivery system with respect to both the overhead on control and data planes. However, given the multitude of messages and strict resource constraints, devising an efficient message delivery channel selection is not trivial. To solve the problem, a complete message delivery system and an efficient channel selection algorithm are both needed.

II. Background and Motivation

A. Network Configuration Messages in Clouds

In cloud networks, network configuration messages (southbound messages) play a crucial role in enabling efficient and dynamic virtual cloud network. These messages are responsible for conveying critical information and instructions that shape the behavior and configuration of network devices and services within the cloud infrastructure. Network configuration messages can be categorized into unicast and multicast messages, each serving distinct purposes.

Unicast messages are used for point-to-point communication, where a message is sent from a source to a specific destination. These messages enable targeted and individualized configuration updates, such as assigning specific IP addresses or routing information to a particular network device or virtual machine. For example, a unicast message may be used to instruct a network device to update its firewall rules or to inform a virtual machine about its assigned network resources.

In contrast, multicast messages are employed for one-to-many communication, allowing the simultaneous delivery of a message to multiple recipients. These messages are particularly useful for network-wide configuration updates or announcements that need to reach a group of devices or services simultaneously. For instance, a multicast message can be used to distribute network policy changes or to propagate routing information across multiple network switches or routers. By utilizing both unicast and multicast messages, cloud networks can achieve a fine-grained and scalable network configuration management. Unicast messages ensure precise and targeted updates, while multicast messages provide efficient and synchronized dissemination of configuration changes across the network infrastructure.

B. Resource Constraints in Southbound Message Delivery

In general, both the bandwidth and CPU resources of the control plane are limited. Specifically, the bandwidth of sending southbound messages to the data plane is typically only tens of Gbps [8], which is insufficient compared to a large amount of southbound messages in clouds (up to hundreds of Gbps). Besides, a classic server typically offers generic CPUs that fail to keep pace with the high-speed data stream and numerous parallel threads between the server itself and compute nodes in the data plane (range from tens of thousands to hundreds of thousands). For example, a typical server with Ubuntu 18.04 supports up to several thousand threads, each of which will occupy system resources (mainly the CPU resources) [25]. The overall network performance will drop significantly when the control plane is overloaded.

Meanwhile, the bandwidth capacity in the data plane may also be a potential bottleneck. In practice, since each compute node usually uses a single dedicated port for receiving southbound messages while the remaining ports are configured for normal packet processing between VMs [6], the bandwidth for southbound messages is typically only a few hundred Mbps. As a result, we should consider the bandwidth resource constraint of each compute node for delivering southbound messages. Besides, from the perspective of saving bandwidth resources in the data plane, we need to reduce the overall bandwidth consumption of the data plane for southbound messages delivery.

C. Observations and Intuitions

We observe that the existing two typical models for southbound message delivery have pros and cons. Cloud networks using RPC allow each compute node to receive the required messages without redundancy, but lack sufficient resource capacity to cope with the large cloud scale. On the other hand, the use of MQ reduces the load on the control plane while the load on the data plane (redundant messages) increases significantly .

Besides, we find that there are two types of southbound messages, *unicast message* and *multicast message*. More specifically, when users need to configure a specific instance, such as setting IP, adding or deleting ports, only the compute node where the instance is deployed requires these messages. That is, the controller needs to send each unicast message to one compute node. Furthermore, when a user sets the subnet or security group policy, all compute nodes that contain related VMs need to receive these messages. The controller will send the same multicast messages to multiple nodes. Obviously, the performance of RPC and MQ is different when processing different types of messages. That is, when delivering unicast messages, RPC provides direct communication, while MQ brings redundancy since all compute nodes subscribed to the same topic will receive the same message. When delivering a multicast message, MQ only needs to send it to the MQ server once, while RPC treats it as multiple unicast messages and sends them one by one.

A question immediately following the above discussion is that *can we do better by combining these two methods while taking into account the resources constraints on both the control and data planes?* Intuitively, we can choose MQ to send multicast messages to reduce the control plane overhead and use RPC to deliver unicast messages to reduce the data plane overhead. However, evaluation results in Section VII show that this intuitive dual-channel selection method does not work well due to resource constraints in the cloud.

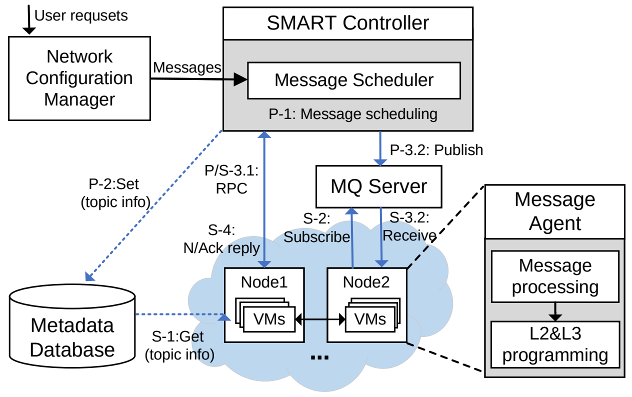


Fig. 1: SAMRT architecture and workflow. Our work consists of message parser, message scheduler and message agent. The southbound message delivery mainly includes two stages: publish and subscribe (P/S). The controller is responsible for processing configuration messages (P-1), updating the database (P-2), and delivering messages in two ways: operating RPC (P-3.1) or publishing to the MQ server (P-3.2). Message agent on each node is responsible for accessing the database (S-1), subscribing to topics (S-2), receiving messages (S-3.1/3.2), and returning L2&L3 programming results via RPC (S-4).

Furthermore, enabling two delivery channels simultaneously introduces additional challenges to the system design, such as out-of-order messages and inconsistent states, which need to be resolved in more details.

III. SMART Overview

We design and implement a dual-channel southbound message delivery system, called SMART, to achieve efficient message delivery under resource constraints (*e.g.*, the bandwidth resource constraint). To achieve this, we propose three fundamental components, including a message parser and a message scheduler in the controller and a message agent in each compute node, as shown in Fig. 1.

- **Message Parser** in the controller (Section IV) provides a unified southbound message abstraction and batching mechanism. As the network configuration manager parses the user's requests into several messages, the message parser uses a hash function to classify messages to different categories based on their destinations. Besides, it uses a counter to store the traffic amount of each category of messages and periodically estimates the sending rate of each category of messages for real-time monitoring of message traffic.
- **Message Scheduler** in the controller (Section V) determines the message channel (queue) selection for each message with resource constraints. When messages come into the controller, the message scheduler utilizes the deep reinforcement learning mechanism to make delivery decisions for each message. According to the decision result, the scheduler will send messages through RPC or MQ server with the specific topic.
- **Message Agent** on each compute node (Section VI) is mainly responsible for pulling and processing southbound messages from the two channels. The processing logic is specially designed to ensure the orderliness and consistency of the messages. The message agent uses the timestamps and concurrent locks in the message processing flow. Noting that how to execute the southbound message is not the focus of this paper.

TABLE I: Specifications for configuration message format.

Type	Name	Description
uint32	message_id	identify of a message
uint32	request_id	corresponding request id
uint32	vpc_id	VPC a message belongs to
string	timestamp	message generation time
Configuration	configuration_type	type of the configurations
Operation	operation_type	type of the operations

The overall workflow of SAMRT is shown in Fig. 1. The southbound message delivery mainly includes two stages: publish and subscribe (P/S). In the publish stage, the scheduler determines the message delivery channel regarding the current network overhead and writes the corresponding topic mapping into the database (P-2). Next, messages are delivered through the RPC channel (P-3.1) or MQ channel (P-3.2) based on the selection results. In the subscribe stage, the message agent gets the subscription information from the database (S-1) and subscribes to the corresponding topics (S-2). Then it receives the messages from two channels (S-3.1 and S-3.2). Finally, the L2&L3 programming results are returned via RPC (S-4).

IV. Message Parser Design

A. *Unified Message Abstraction*

Due to the variety of southbound messages, designing a processing logic for each message type is laborious and time-consuming. In actual situations, when delivering southbound messages, our emphasis is not on the content, but the size, sending rate, and destination of messages, since they are the key indicators for resource optimization of the control and data planes. To this end, we design a unified abstract model to efficiently deliver and process southbound messages.

Abstraction. We observe that southbound (network configuration) messages are diverse, involving different configuration types (*e.g.*, VPC, Subnet) and operation types (*e.g.*, Delete, Create). For the ease of message processing, we design a unified abstraction for messages. Table I gives a specification for the configuration message format. The *message id* is used to identify different messages, and the *request id* corresponds to the user's request. The *timestamp* is set as the message generation time to ensure the message order. Besides, *configuration type* and *operation type* are used to specify the content of the configuration. For example, when a tenant requests the creation of a subnet on two VMs and generates a subnet configuration message, the message parser sets the current system clock as the timestamp, and the configuration type and operation type are set as `Subnet` and `Create`, respectively.

Benefits. Message abstraction offers benefits to both message delivery and system design. For message delivery, the

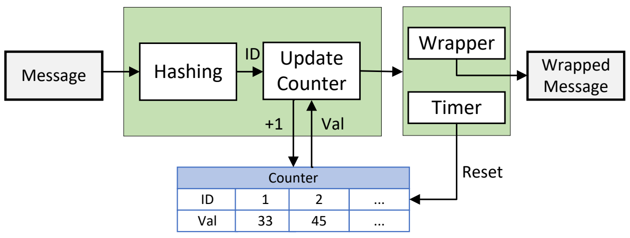


Fig. 2: Illustration of processing steps in message parser.

Algorithm 1 Processing steps for the rate estimation

```
1:  Initialize the counter $C$ for different message types
2:  Initialize the timer and interval $T$
3:  while TRUE do
4:      for each interval of rate estimation do
5:          for each incoming message $m$ do
6:              Encapsulate the message as in Table I
7:              $i \leftarrow hash(m.hosts)$
8:              $Counter_i \leftarrow Counter_i + m.size$
9:          end for
10:         Calculate $f(m_i) = \frac{Counter_i}{T}$ for each message type $m_i$
11:         Reset the timer.
12:     end for
13: end while
```

controller and compute nodes use a unified interface for communication and only need to pay attention to the configurations and operations during transmission, not other details. For system design, it abstracts message contents at the granularity of configurations, which enables further configuration-level processing of messages. We illustrate them in Section IV-B.

B. Destination-based Message Batching

Message Aggregation. Creating a separate thread for each message and sending through TCP connection consumes a certain amount of computing resources. Processing multiple messages at the same time will occupy lots of computing resources on the controller, which contradicts with the limited computing resource and may result increased delivery delay. In fact, if two or more messages have the same destination(s), we can deliver them in one shot. For example, there is one message for creating subnets on VMs distributed across two compute nodes, and another message for creating routers on the same compute nodes. Then, we aggregate the two messages into one message with configuration type {`Subnet`, `Router`}. Moreover, the parser uses a hash function to classify the messages. For each message m, the parser hashes the host IP list of m into ID i and updates the target value in the counter by its size, as shown in Fig. 2. Messages with the same ID are aggregated before delivery. As a result, message aggregation will significantly reduce the controller's CPU consumption.

Rate Estimation. The sending rate of messages is a crucial indicator for SAMRT to realize resource-optimal southbound message delivery in Section V. To this end, SAMRT offers a periodic estimation of the sending rate of messages. More specifically, to estimate the sending rate $f(m_i)$ for message type m_i, SAMRT sets a timer for periodically checking the current sending rate. For each time interval T, $f(m_i)$ is estimated as $f(m_i) = \frac{Counter_i}{T}$, where T is the interval in seconds and $Counter_i$ is the value in the counter with ID i.

SAMRT fine-tunes T with respect to traffic dynamics to update its estimation and avoid over-provisioning resources. The overall processing steps are described in Algorithm 1. The counter and timer are first initialized (Lines 1-2). In each interval, the parser encapsulates each incoming message as in Table I (Line 6), hashes it based on its host IPs (Line 7), updates the corresponding value in the counter by the message size (Line 8), and estimates the sending rate $f(m_i)$ for each message type (Line 9). After the end of each interval, the parser resets the timer (Line 10).

V. Message Scheduler Design

A. System Model

In general, SAMRT consists of the control plane and the data plane. Specifically, the SAMRT controller in the control plane manages the network, including southbound message delivery. We use $\mathcal{C}_b$ and $\mathcal{C}_p$ to represent the bandwidth and CPU capacity for handling concurrent messages of the control plane. The data plane consists of compute nodes and provides compute resources to tenants. We use $N = \{n_1, n_2, ..., n_{|N|}\}$ to represent the set of compute nodes. For each compute node n, we denote its bandwidth as $s(n)$. Since there are different types of southbound messages, we use $M = \{m_1, m_2, ..., m_{|M|}\}$ to denote the set of message types.

Two messaging protocols are used to implement southbound messaging in SAMRT: RPC and MQ. The MQ server contains a set of queues for storing and forwarding southbound messages from the control plane to the data plane. Each queue is identified by a topic. The topic set is defined as $\mathcal{T} = \{t_1, t_2, ..., t_K\}$, where $K = |\mathcal{T}|$ is the number of queues on the MQ server.

B. Problem Formulation

We present the dual-channel southbound message delivery (DSMD) problem with the following three constraints. (1) *Channel Constraint.* A message will only be delivered through one channel and one queue if through the MQ channel. (2) *Bandwidth Constraint.* The total traffic on the controller and each compute node should not exceed their capacity to prevent congestion. (3) *CPU Constraint.* Since each sent message takes up a certain amount of CPU resources (*e.g.*, threads, cycles), we hope the total number of concurrent messages should not exceed the CPU capacity. Accordingly, we formulate DSMD as follows:

$$\min \sum_{n \in N} b(n)$$

$$S.t \begin{cases} x_m + \sum_{t\in T} y_m^t = 1, & \forall m \in M \\ \sum_{m\in M} f(m)(\sum_{n\in N}\Gamma_m^n x_m + \sum_{t\in \mathcal{T}} y_m^t) \le \mathcal{C}_b & \\ \sum_{m\in M}(\sum_{n\in N}\Gamma_m^n x_m + \sum_{t\in \mathcal{T}} y_m^t) \le \mathcal{C}_p & \\ z_n^t \ge \Gamma_m^n y_m^t, & \forall n,t,m \\ \sum_{m\in M} f(m)(\Gamma_m^n x_m + \sum_{t\in \mathcal{T}} y_m^t z_n^t) = b(n), & \forall n \in N \\ b(n) \le s(n), & \forall n \in N \\ x_m, y_m^t, z_n^t \in \{0,1\}, & \forall m,t,n \end{cases} \tag{1}$$

Where binary variable $x_m \in \{0,1\}$ denotes whether or not the controller will deliver message type m via RPC, binary variable y_m^t indicates whether the message type m is published to topic t or not, and z_n^t indicates whether the compute node n subscribes topic t or not. The constant Γ_m^n presents whether the compute node n requires the message type m or not.

The first set of inequalities indicates that each message type needs to be delivered to nodes in one way (either RPC or MQ). The second set of inequalities shows that the total amount of messages sent by the controller should not exceed its bandwidth capacity. The third set of inequalities indicates that the number of concurrent messages sent by the controller should not exceed its CPU capacity. The fourth set of inequalities represents that compute node n should subscribe to topic t if one or more messages it requires are published to topic t. The fifth set of equalities represents the message traffic amount on each compute node n, denoted as $b(n)$. The sixth set of inequalities expresses the bandwidth constraint on each compute node n. Our objective is to minimize the total message traffic amount on compute nodes, that is, $\min \sum_{n\in N} b(n)$.

Theorem 1: The DSMD problem is NP-hard.

Proof: The proposed DSMD problem remains NP-hard even if the resource constraints on the control plane are ignored. Under this case, the problem turns to be a Weighted Set Covering Problem (WSCP) [26] for each compute node. More specifically, we can calculate the weight of each message set as the message redundancy it introduces if we put all messages from the set into the same queue. Then the problem can be described as selecting message sets so as to cover all messages. Since WSCP is a special case of our problem, we can conclude that the DSMD problem is NP-hard too. ■

C. Feasibility of Solving DSMD using DRL

It is difficult to directly solve the DSMD problem in Eq. (1) due to its NP-hardness. At the same time, the following three characteristics of cloud networks also increase the difficulty of solving DSMD. First, the multi-resource heterogeneity of the cloud system poses a tough problem to provide users with efficient messaging service because heterogeneous compute nodes increase the hardness of message delivery. It is challenging to design an efficient method to consider the massive nodes in large-scale clouds. Second, a large number of resource constraints may cause algorithms complicated and even unsolvable [27]. Third, the time-varying network conditions also aggravate the difficulty of this problem, which requires fast responsive algorithms at runtime. The traditional approaches using predefined rules or heuristics lack the ability to quickly respond to traffic changes.

Fortunately, we found that a learning-driven method (*e.g.*, deep reinforcement learning) helps deliver southbound messages while satisfying resource constraints with the abundant network. In fact, deep reinforcement learning (DRL) method has been widely adopted and proven to be feasible in the large-scale networks [28]–[30]. By constructing optimization objectives reasonably, DRL can learn an optimal strategy for message delivery that satisfies resource constraints in continuous iterations. Moreover, the message aggregation and rate estimation mechanisms given in Section IV-B provide the premise for designing a environment-adaptive algorithm, which ensures a fast response to changes in network traffic. Hence, we propose a design using a DRL-based method to learn an effective solution for the DSMD problem.

D. DRL-based Algorithm Design for DSMD

To implement DRL techniques, we first specify the state space, the action space, and the reward function below.

1) **State space**: Assume that the input state at epoch i is represented by $\mathcal{S}_i = (X(i), Y_t(i), C_b(i), C_p(i), S(i))$, where $X(i) = \{x_1(i), x_2(i), ..., x_m(i)\}$ and $Y(i) = \{y_1^t(i), y_2^t(i), ..., y_m^t(i)\}$ denote the current channel selection to deliver each message. $C_b(i)$ and $C_p(i)$ denote the available bandwidth and CPU resource on controller. $S(i) = \{s_1(i), s_2(i), ...s_n(i)\}$ represents the available bandwidth on compute nodes.
2) **Action set**: On receiving a message m, the scheduler needs to take an action a_i from the action set for channel selection. The action set encoding the channel selection decision can be represented as $\mathcal{A} = \{\hat{x}, \hat{y_1}, \dots, \hat{y_t}\}$, where $\hat{x}$ means "delivering through RPC channel", and $\hat{y_t}$ means "delivering through MQ with topic t".
3) **Reward function**: At each training epoch i, the system will get a reward $r(s_i, a_i)$ under a certain state s_i after executing action a_i. In practice, the reward function should be positively correlated with the system objective. As in Eq. (1), the objective function of this work is to minimize the total amount of messages received by compute nodes. Hence, the reward function should be negatively correlated with the $\sum_{n\in N} b(n)$. Moreover, a reward penalty should be incorporated into the reward function if resource constraints are violated. We define the reward function as:

$$\begin{aligned} r(s_i, a_i) &= \frac{\kappa}{1 + \sum_{n\in N} b_n(i)} \\ &- \beta[\frac{\sum_{m\in M} f(m)(\sum_{n\in N}\Gamma_m^n x_m(i) + y_m^t(i)) - C_b(i)}{C_b(i)}]^+ \\ &- \beta[\frac{\sum_{m\in M}(\sum_{n\in N}\Gamma_m^n x_m(i) + y_m^t(i)) - C_p(i)}{C_p(i)}]^+ \\ &- \beta[\sum_{n\in N}\frac{b_n(i) - s_n(i)}{s_n(i)}]^+ \end{aligned} \tag{2}$$

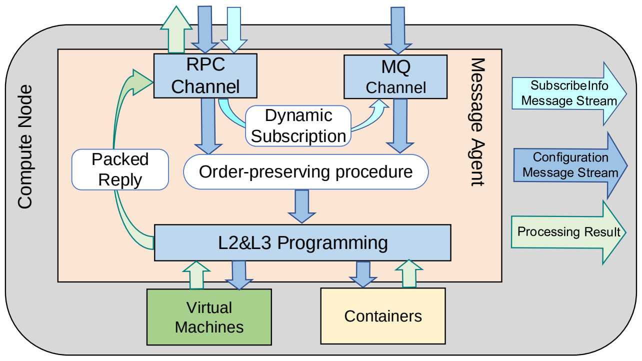


Fig. 3: Architecture and workflow of a message agent. Three types of data streams are processed during runtime.

where $[x]^+ \triangleq max(0,x), \kappa = 100$ and $\beta = 50$ are two positive constants as in [28]. The positive constant hyperparameters are typically determined through empirical studies or prior knowledge and are chosen to provide a good balance between exploration and exploitation in the reinforcement learning algorithm. The first term on the right-hand of Eq. (2) indicates that the reward function should be positively correlated with the system objective. The remaining three terms incorporate the reward penalty for constraint violation, including bandwidth and CPU capacity constraints on the control plane, the bandwidth constraints on each compute node.

Given the design of state space, action set, and reward function, we adopt the efficient framework for DRL as in [31], [32] to train the model and solve DSMD. The DRL-based method aims to learn a general action set based on the current system state and the given reward. At each epoch i, it observes a state s_i, takes an action a_i and receives a reward r_i by executing the action. The objective is to find a policy mapping a state to an action that maximizes the cumulative reward as $R_0 = \sum_{i=0}^{I} \gamma^i r(s_i, a_i)$, where $r(\cdot)$ is the reward function, I is the number of epochs until model convergence and $\gamma \in [0,1]$ is a discount factor of the future reward.

The SMART system implements a two-phase approach consisting of offline training and online execution to ensure the stability of the cloud control plane system. The offline-training procedure is specifically designed to mitigate any negative impact caused by the training process on system stability. During this phase, the system leverages historical data and machine learning techniques to train a model that learns the optimal policy for making control decisions.

VI. Message Agent Design

The message agent is deployed on each compute node and offers a stable and consistent southbound messaging service. The overall architecture of the message agent is shown in Fig. 3, where two monitoring programs (RPC and MQ) are in charge of continuously listening to the ports and receiving messages. There are three types of data streams that are processed. 1) The agent receives SubscribeInfo messages through the RPC channel and dynamically configures MQ's subscriptions to cope with the time-varying cloud network. 2) It receives southbound configuration messages through two channels in parallel and then processes them in an order-preserving procedure to avoid out-of-order situations. 3) The L2&L3 programming results are aggregated into packed replies and sent back through the RPC channel to ensure eventual consistency.

A. Dynamic Subscription

To realize the decoupling of control and data planes, we rely on a metadata database to store the topic mapping information as shown in Fig. 1, which enables the agent to quickly establish a connection with the MQ server when it initializes or recovers from a crash. Considering the dynamic of user requests and the environment-adaptive selection method in Section V-B, SAMRT requires a real-time update of the topic subscription. To this end, we design a `SubscribeInfo` metadata to specify the corresponding topic subscription. Table II lists the detailed properties of subscription messages, which specifies 1) the MQ server's IP address, 2) topic and subscription names that the node needs to subscribe to, and 3) the corresponding operation.

Once the SAMRT controller decides to update the topic subscription of a certain compute node, it will push a `SubscribeInfo` message to the corresponding node through RPC and modify the database at the same time. However, in order to achieve the lossless dynamic subscription, the agent does not immediately update the subscription when receiving a message. Instead, the agent checks the timestamps of messages delivered through the MQ channel and waits until receiving a configuration message generated after the `SubscribeInfo` message. In this way, the compute node can quickly switch to the specified topic without message loss.

B. Order Preservation of Messages

Due to the variable network and system load, the message delivering delay is unpredictable. As a result, the order of message delivery between RPC and different queues of MQ cannot be strictly guaranteed. Once the out-of-order occurs, it may cause serious consequences. Therefore, for order-preserving of southbound messages in compute nodes, SAMRT controller sets a timestamp in each delivered message. The compute node itself maintains a local cache that stores the latest timestamp for each configuration resource. When receiving a message, the compute node inspects its timestamp and determines if it's the latest message . If not, the node discards it and reports an out-of-order error to the controller. Otherwise, it performs L2&L3 programming based on the message content.

However, the order-preserving procedure may lead to the read/write conflicts in a scenario when the compute node receives two messages simultaneously through two channels, and they configure the same resource. Therefore, we add concurrent locks for each resource in the cache to avoid the conflict. Fig. 4 shows an example where two configuration messages for Subnet #2 through two channels at the same time. The first message locks the resource key for Subnet #2

TABLE II: Specification for the SubscribeInfo message format.

Type	Name	Description
string	url	url of the MQ server
string	topic	topic name
string	subscription	subscription name
string	timestamp	generation time
bool	operation	subscribe/unsubscribe

and updates the timestamp in the cache. The second message is blocked until the resource key is released. However, as the timestamp of the second message is earlier than the timestamp of the same resource in the cache, the agent needs to drop this message and send an error to the controller in a packed reply.

C. *Packed Reply for Consistency*

In the SMART system, ensuring eventual consistency of programming results is crucial for the stability and reliability of the cloud control plane. However, the message-oriented middleware (MQ) lacks a callback channel for programming results, which can lead to short-term state inconsistencies. To address this limitation, the SMART system employs a combination of asynchronous communication through MQ and synchronous Remote Procedure Call (RPC) for packing and returning programming results to the controller.

After processing messages in the data plane, the compute nodes generate programming results, which need to be eventually returned to the controller. Instead of immediately sending individual programming results through RPC, the SMART system packages these results in the past short period of time and returns them together through synchronous RPC. This approach reduces the frequency of RPC calls, leading to lower CPU and bandwidth consumption for programming results.

However, there might be cases where a specific compute node does not receive the programming results through RPC within a period of time. In such situations, the compute node takes the initiative to feed the programming results back to the controller. For this purpose, an RPC server is established on the controller side. Compute nodes can send the packed reply, containing the accumulated programming results, to the controller through the RPC channel using the provided API. In response to error reports, the controller will initiate reprogramming requests to the affected compute nodes. These requests instruct the nodes to update their state and apply any necessary modifications to ensure the system's consistency.

VII. Implementation and Evaluation

A. *System Implementation*

We implement SAMRT framework based on Linux 5.4 kernel. The Message Scheduler are written in Java code for simplicity and high portability. The Message Agent is written in C++ for ease of deployment on compute nodes. Besides, we implement MQ and RPC channels using widely used open-source frameworks Apache Pulsar (version 2.6.1) [24] and gRPC (version 3.8.0) [16], respectively.

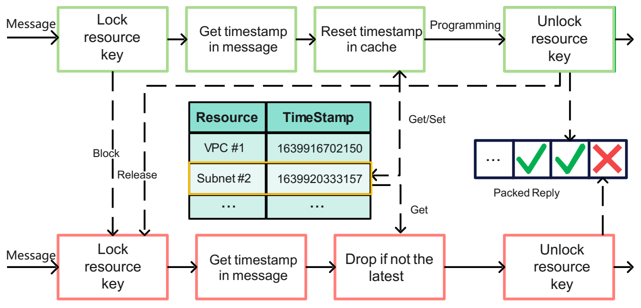


Fig. 4: An example of order-preserving procedure and packed reply where two messages update Subnet #2 at the same time.

In the DRL training, we implement a prototype on top of Pytorch [33] and use a 2-layer fully-connected feed-forward neural network to serve as the actor network, which utilizes the Rectified Linear Unit (ReLU) for activation. In the output layer, we employ tanh as activation function. For the critic network, a 2-layer fully-connected feed-forward neural network is adopted. The offline training takes 7 hours on a server with Intel Xeon 6152 CPU, RTX 3090 and 128G memory to converge in a scenario with five thousand compute nodes. The online execution model takes about 0.5 milliseconds on our server to make decisions.

B. *Performance Metrics and Benchmarks*

This paper studies how to deliver southbound messages in clouds with the lowest redundant messages in the data plane, considering both constraints in control and data planes. We adopt six main metrics for performance evaluation. The first set of metrics is about control plane overhead, including (1) *the traffic on controller*, (2) *the number of southbound messages* and (3) *the CPU utilization*. Specifically, *the traffic on controller* represents the bandwidth resource consumption of the controller for sending southbound messages. We record the total traffic amount of southbound messages as the traffic on the controller. *The number of southbound messages* represents the total number of southbound messages sent by the controller. Note that a multicast message is sent as multiple unicast messages using RPC while it only needs to be sent once using MQ. Thus, this metric can indicate the CPU resource consumption of the controller for sending southbound messages. *The CPU utilization* is recorded as the CPU resource consumption, which is measured as the multi-core CPU utilization by the System Activity Reporter (SAR) command. The second set of metrics is about data plane overhead, including (4) *the total traffic on CNs (compute nodes)* and (5) *the maximum traffic on CNs*. We measure the traffic amount of southbound messages received by each compute node and calculate the total (or maximum) traffic amount of all compute nodes. Furthermore, we test (6) *the message delivering delay* to show the efficiency of the SAMRT system. We also measure the time interval for each southbound message from the controller to the compute node and record the 95th percentile as the message delivering delay.

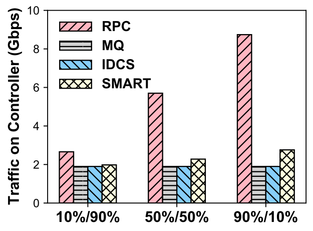


Fig. 5: Traffic on Controller vs. Percentage of Multicast/Unicast

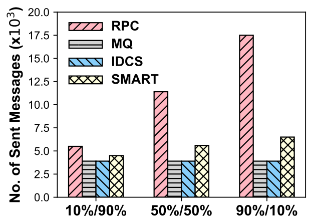


Fig. 6: No. of Sent Messages vs. Percentage of Multicast/Unicast

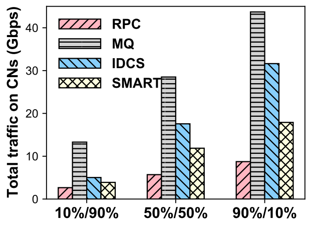


Fig. 7: Total Traffic on CNs vs. Percentage of Multicast/Unicast

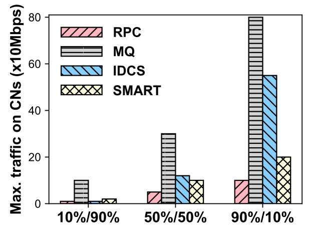


Fig. 8: Max. Traffic on CNs vs. Percentage of Multicast/Unicast

We compare SAMRT with three state-of-the-art benchmarks.

1) The first one is RPC [15], which is a widely used method in distributed microservice system framework. RPC establishes TCP connections between the controller and all compute nodes in clouds. Messages are sent from the controller to compute nodes one by one.
2) The second one is VITA [34], which performs southbound message delivery using a message queue server. To deal with a limited number of message queues on the server, VITA divides messages into certain groups according to their VPC ID. Messages of the same group are sent to the same specified topic. Each node subscribes to the topics due to the their own needs.
3) The third one is an intuitive dual-channel selection scheme, denoted by IDCS. We use a simple channel selection method to compare the effect of our DRL-based algorithm. By default, IDCS sends all unicast messages through RPC and sends all multicast messages through MQ. We compare IDCS and SAMRT in large-scale simulations to evaluate the superiority of our proposed dual-channel selection algorithm.

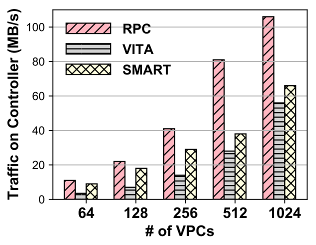
C. Large-scale Simulations

Simulation Settings. We refer to a private cloud deployed in CERN (European Organization for Nuclear Research) [4] to design our simulation. The CERN cloud contains 5,500 compute nodes, 12000 virtual machines, and 1900 VPCs. We assume that the VMs are distributed on the compute nodes randomly. The expected message traffic intensity for each VPC is set as 1Mbps. Moreover, we use power law for the message-size distribution, where 20% of all messages account for 80% of traffic volume as observed in [35], and the average message size is set to be 512kB. Unless otherwise specified, the proportional relationship between multicast/unicast messages is 50%/50%. The number of topics of the message queue server is set to 1000 by default.

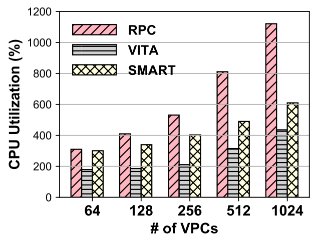
Simulation Results. In the large-scale simulation, we observe the traffic on the controller, the number of sent messages, and the total/maximum traffic on CNs by changing the proportional relationship between multicast and unicast messages while the total amount of messages remains the same. The results are shown in Figs. 5-8. These figures show that the RPC method achieves the lowest data plane overhead with the highest control overhead (both in traffic amount and the number of southbound messages). Furthermore, as the proportion of multicast messages increases, the traffic on the controller significantly increases, which may cause congestion of the controller with high probability. This is because RPC sends multicast messages multiple times according to the destination nodes. In contrast, VITA has the lowest overhead on the controller, and the results remain the same when the proportional relationship between multicast and unicast messages changes. However, it results in the greatest message redundancy on the data plane since each original unicast message is sent to all nodes subscribed to the same topic. The enormous message redundancy will lead to an increase in overall latency and probability of compute node overload. This set of experiments shows that regardless of the proportional relationship between multicast and unicast messages, dual-channel-based methods (SAMRT and IDCS) achieve lower control overhead than RPC and lower message redundancy than VITA. For example, when the percentage of multicast messages is 50%, the traffic on the controller is 5.82Gbps and 2.11Gbps for RPC and SAMRT, respectively. That is, SAMRT reduces traffic on the controller by 63.7% compared to RPC. Meanwhile, the total traffic on compute nodes for VITA and SAMRT is 28.3Gbps and 11.4Gbps, respectively, which means SAMRT reduces the total traffic on compute nodes by 59.7% compared to VITA. This is because dual-channel-based methods can leverage the pros of two message channels compared with RPC and VITA. In addition, SAMRT always performs better than IDCS since SAMRT comprehensively considers the resource constraints of the control and data planes compared with IDCS. We should note that too much overhead on the control plane (*e.g.*, RPC) or too much overhead on the data plane (*e.g.*, VITA) can break the resource constraints, resulting in long message delivery delay. Furthermore, even though the control overhead of SAMRT is higher than that of VITA, the message delivery delay is not affected because the controller is not overloaded. These scenarios will be verified in Section VII-D.

D. Testbed Evaluation

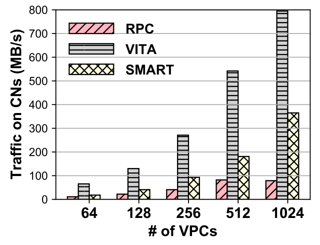
Implementation on the Platform. In general, we use ten servers running Ubuntu 18.04 with Linux kernel 5.4 to build the testbed. All the servers are equipped with a 22-core Intel Xeon 6152 processor, 128GB memory, and an Intel X710 10GbE NIC. Two servers are used as the controller and the message queue server, respectively. The remaining eight



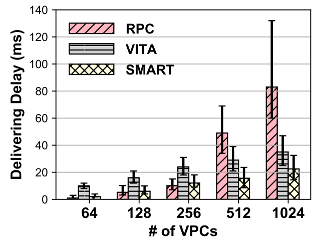
(a) Traffic on Controller vs. # of VPCs



(b) CPU Utilization vs. # of VPCs

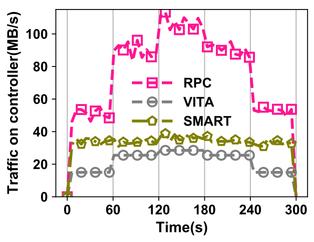


(c) Total Traffic on CNs vs. # of VPCs

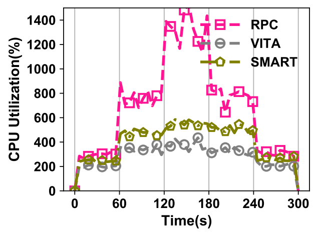


(d) Delivering Delay vs. # of VPCs

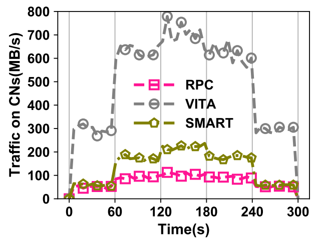
Fig. 9: Performance of Scalability vs. Number of VPCs



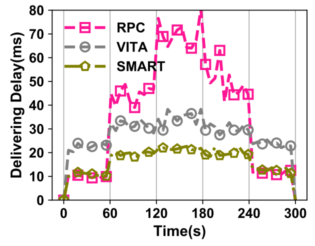
(a) Traffic on Controller vs. Time



(b) CPU Utilization vs. Time



(c) Traffic on CNs vs. Time



(d) Delivering Delay vs. Time

Fig. 10: System Performance Timeline under Different Work Load

servers are used as the data plane. We take a small cloud deployed in GoDaddy [36] as a reference, which contains 350 compute nodes. To expand the testing topology and collect testing data conveniently, we rely on virtualization technology for system implementation. Specifically, we deploy 350 VMs, each equipped with one vCPU and 1GB memory, as compute nodes on the remaining servers.

Testbed Settings. Unless otherwise specified: the number of VPCs and topics is by default set to 512 and 200, respectively; the expected traffic intensity for each message is set to 1Mbps, and the bandwidth constraint of each compute node is 1Gbps by default. The message-size distribution is the same as that of simulations, where 20% of all messages account for 80% of traffic amount. According to [37], we generate two types of messages: (1) unicast messages, whose sources and destinations are randomly picked, *e.g.*, IP address segment configuration messages; (2) multicast messages, which simulate the traffic with multiple destinations, *e.g.*, subnet and security group configuration messages. Each type of message accounts for half of the total traffic amount.

Scaling Provisioning. We measure the traffic on the controller, the CPU utilization, the total traffic on CNs, and the message delivering delay by scaling the number of VPCs from 64 to 1024. The error bars represent the maximum and minimum delivering delay.The number of messages increases linearly with the increase in the number of VPCs. Fig. 9 reports the detailed results. As shown in Figs. 9(a)-9(b), RPC has the highest control overhead (both on bandwidth and CPU), which leads to a relatively high delivering delay at 512 and 1024 VPCs as shown in 9(d). This is because RPC communicates directly with compute nodes and sends each message one by one, creating a bottleneck in the controller. As for VITA, it achieves slightly less control overhead than SAMRT as the number of VPC increases as shown in Figs. 9(a)-9(b). However, the message redundancy of VITA is quite huge, as shown in Fig. 9(c). As a result, Fig. 9(d) shows that SAMRT scales better than the other two methods with the number of VPCs increasing. For example, SAMRT achieves a message delivering speedup of 3.1x than RPC and 1.9x than VITA at 512 VPCs.

To summarize, Figs. 9 show that SAMRT scales well when the number of VPCs increases in the cloud. That means, by leveraging the dual-channel southbound message delivery, SAMRT offers significant delivering delay reduction compared to both RPC and VITA, especially in larger-scale scenarios.

Dealing with the Burst Traffic. Fig. 10 shows the results where messages are delivered during a 300-second period. It is divided into five 60-second phases with different sending rates: 50%, 80%, 95%, 80%, 50% of 30MB/s, to simulate medium, medium-heavy and heavy workloads of the system. Although Figs. 10(a) and 10(b) show that the traffic and CPU utilization on the controller of VITA are quite good, Fig. 10(c) indicates that the total traffic on compute nodes of VITA is much more than the other two methods. High load on compute nodes results in higher delivering delay shown in Fig. 10(d). On the contrary, RPC has the lowest traffic on compute nodes but brings the highest CPU utilization and bandwidth consumption as shown in Figs. 10(a) - 10(c). Fig. 10(d) shows that the message delivering delay of SAMRT is smaller than the other approaches throughout the five phases. More specifically, SAMRT offers a 1.9x and 3.2x speedup compared with VITA and RPC with the medium-heavy load. Moreover, SAMRT provides a relatively stable delay guarantee since the delay is not significantly affected when the workload varies.

VIII. Conclusion

In this paper, we design the dual-channel southbound message delivery system, which consists of three main modules: Message Parser/Scheduler on the controller and On-host Message Agent. Both the simulation and experimental results clearly indicate the high efficiency of our proposed system.

Acknowledgement

The corresponding authors are Gongming Zhao and Hongli Xu. This work was supported in part by the National Science Foundation of China(NSFC) under Grant 62372426 and Grant 62102392, in part by the National Science Foundation of Jiangsu Province under GrantBK20210121, in part by the Hefei Municipal Natural Science Foundation under Grant 2022013, in part by the Youth Innovation Promotion Association of Chinese Academy of Sciences under Grant 2023481.

References

[1] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer *et al.*, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 373–387.

[2] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson *et al.*, "Network virtualization in multi-tenant datacenters," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 203–216.

[3] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *2010 Proceedings IEEE INFOCOM*, 2010, pp. 1–9.

[4] T. Bell, B. Bompastor, S. Bukowiec, J. C. Leon, M. Denis, J. van Eldik, M. F. Lobo, L. F. Alvarez, D. F. Rodriguez, A. Marino *et al.*, "Scaling the cern openstack cloud," in *Journal of Physics: Conference Series*, vol. 664, no. 2. IOP Publishing, 2015, p. 022003.

[5] L. Luo, G. Zhao, H. Xu, L. Xie, and Y. Xiong, "Vita: Virtual network topology-aware southbound message delivery in clouds," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022.

[6] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano *et al.*, "Orion: Google's {Software-Defined} networking control plane," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 83–98.

[7] H. Qu, O. Mashayekhi, C. Shah, and P. Levis, "Decoupling the control plane from program control flow for flexibility and performance in cloud computing," in *Proceedings of the thirteenth euays conference*, 2018, pp. 1–13.

[8] K. Zheng, L. Wang, B. Yang, Y. Sun, and S. Uhlig, "Lazyctrl: A scalable hybrid network control plane design for cloud data centers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 115–127, 2016.

[9] S. Maheshwari, P. Netalkar, and D. Raychaudhuri, "Disco: Distributed control plane architecture for resource sharing in heterogeneous mobile edge cloud scenarios," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 519–529.

[10] Y. Gong, B. He, and J. Zhong, "Network performance aware mpi collective communication operations in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3079–3089, 2013.

[11] S. Seleznev and V. Yakovlev, "Industrial application architecture iot and protocols amqp, mqtt, jms, rest, coap, xmpp, dds," *International Journal of Open Information Technologies*, vol. 7, no. 5, pp. 17–28, 2019.

[12] R. Thurlow, "Rpc: Remote procedure call protocol specification version 2," RFC 5531, May, Tech. Rep., 2009.

[13] P. Stuedi, A. Trivedi, B. Metzler, and J. Pfefferle, "Darpc: Data center rpc," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–13.

[14] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[15] A. Pourhabibi Zarandi, M. J. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the rpc tax in datacenters," in *Proceedings of the 54th International Symposium on Microarchitecture (MICRO'21)*, no. CONF, 2021.

[16] "grpc," https://www.grpc.io/.

[17] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, 2017, pp. 1–7.

[18] S. Lima, A. Rocha, and L. Roque, "An overview of openstack architecture: a message queuing services node," *Cluster Computing*, vol. 22, no. 3, pp. 7087–7098, 2019.

[19] W. Lu, J. Jackson, and R. Barga, "Azureblast: a case study of developing science applications on the cloud," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 413–420.

[20] G. Fu, Y. Zhang, and G. Yu, "A fair comparison of message queuing systems," *IEEE Access*, vol. 9, pp. 421–432, 2020.

[21] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, 2017, pp. 227–238.

[22] L. Malina, G. Srivastava, P. Dzurenda, J. Hajny, and R. Fujdiak, "A secure publish/subscribe protocol for internet of things," in *Proceedings of the 14th international conference on availability, reliability and security*, 2019, pp. 1–10.

[23] "Apache kafka," https://kafka.apache.org/.

[24] "Apache pulsar," https://pulsar.apache.org/.

[25] D. B. De Oliveira, R. S. De Oliveira, and T. Cucinotta, "Untangling the intricacies of thread synchronization in the preempt_rt linux kernel," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 1–9.

[26] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[27] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[28] Z. Meng, H. Xu, M. Chen, Y. Xu, Y. Zhao, and C. Qiao, "Learning-driven decentralized machine learning in resource-constrained wireless edge computing," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[29] X. Li, F. Tang, J. Liu, L. T. Yang, L. Fu, and L. Chen, "{AUTO}: Adaptive congestion control based on {Multi-Objective} reinforcement learning for the {Satellite-Ground} integrated network," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 611–624.

[30] M. Cheng, J. Li, and S. Nazarian, "Drl-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers," in *2018 23rd Asia and South pacific design automation conference (ASP-DAC)*. IEEE, 2018, pp. 129–134.

[31] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[32] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[33] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[34] G. Zhao, L. Luo, H. Xu, C.-J. Chung, and L. Xie, "Southbound message delivery with virtual network topology awareness in clouds," *IEEE/ACM Transactions on Networking*, 2022.

[35] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 202–208.

[36] "Godaddy," https://www.godaddy.com/.

[37] S. Paul, R. Jain, M. Samaka, and J. Pan, "Application delivery in multi-cloud environments using software defined networking," *Computer Networks*, vol. 68, pp. 166–186, 2014.