Curb: Trusted and Scalable Software-Defined Network Control Plane for Edge Computing

Minghui Xu†, Chenxu Wang†, Yifei Zou†, Dongxiao Yu†, Xiuzhen Cheng†*, Weifeng Lyu‡
†School of Computer Science & Technology, Shandong University, China
‡School of Computer Science & Engineering, Beihang University, China
*Corresponding author
Email: mhxu@sdu.edu.cn, cx_wang@mail.sdu.edu.cn, yfzou@sdu.edu.cn, dxyu@sdu.edu.cn, xzcheng@sdu.edu.cn, lwf@nlsde.buaa.edu.cn

***Abstract*—The proliferation of edge computing brings new challenges due to the complexity of decentralized edge networks. Software-defined networking (SDN) takes advantage of programmability and flexibility in handling complicated networks. However, it remains a problem of designing a both trusted and scalable SDN control plane, which is the core component of the SDN architecture for edge computing. In this paper, we propose Curb, a novel group-based SDN control plane that seamlessly integrates blockchain and BFT consensus to ensure byzantine fault tolerance, verifiability, traceability, and scalability within one framework. Curb supports trusted flow rule updates and adaptive controller reassignment. Importantly, we leverage a group-based control plane to realize a scalable network where the message complexity of each round is upper bounded by $O(N)$, where N is the number of controllers, to reduce overheads caused by blockchain consensus. Finally, we conduct extensive simulations on the classical Internet2 network to validate our design.**

***Index Terms*—Software-defined network (SDN); edge computing; scalability; blockchain**

I. Introduction

The success of cloud computing and the proliferation of Internet of Things (IoT) gives birth to the new concept of edge computing, in which data processing occurs at the network edge rather than relies on cloud services. By lowering computing and storage resources from cloud to edge, end devices can achieve low communication overheads and high bandwidth in a decentralized network. To govern such center-free networks, software-defined networking (SDN) is an appealing technique enlightened by edge computing users. An integrated layered architecture of SDN and edge computing is presented in Fig. 1. SDN decouples the control plane and data plane to create a more expressive and programmable environment so that applications can flexibly configure network policies and realize network automation. SDN can benefit edge computing especially due to its advantages in data-intensive applications with massive edge devices.

Adopting a SDN control plane on edge brings about fresh attack surfaces. In common implementations, developers are used to adopting one single controller to manipulate multiple switches, which faces security problems and scalability concerns. A pivot controller, once crashes, can pose threats to a large area of interconnected edge servers. This problem becomes even more serious considering that third parties

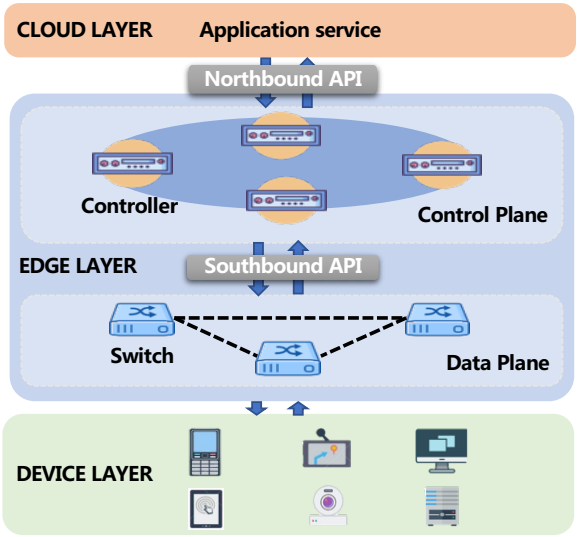


Fig. 1. Infrastructure of SDN-enabled edge computing

implementing application service are free to configure network policies on controllers. Besides, a centralized controller might fail to handle numerous requests generated by edge devices. Massive edge devices can incur large communication overheads and maliciously generate contention on the control plane. To address the above issues, researchers have proposed viable solutions, which essentially introduce redundant SDN controllers and gain fault tolerance by assigning each switch a controller group instead of a single controller. Hence when a controller becomes faulty, other group members can take charge to mitigate negative impacts. This fault tolerance can be achieved by the following approaches: BFT consensus algorithms [1]–[3], primary-backup control plane [4], [5], or blockchain techniques [6]–[9].

However, current approaches have limitations in achieving both security and scalability. On one hand, primary-backup or pure BFT consensus based methods confront with security concerns. For example, if a controller is corrupted and should be replaced with an honest one, how can we guarantee that the incoming controller is honest? In addition, when controller operates such as updating flow rules for switches, how can we

ensure that the operation logic is correct and the status after update is verifiable? Moreover, how to avoid that controllers create conflicting flow rules or configurations? On the other hand, introducing BFT consensus or blockchain systems (always embodied with BFT consensus) arises scalability concerns since a consensus process incurs much communication overhead due to the need of massive message exchanges, e.g., the message complexity of PBFT is $O(N^2)$ ($O(N^3)$ in the worst case). Therefore, letting each controller participate into verifying and achieving consensus on all proposals is not a proper and efficient way.

In this paper, we intend to answer the following question: **Can we design a both trusted and scalable SDN control plane for edge computing?** Concretely, we aim to advance existing BFT or blockchain-based control plane by first introducing a permissioned blockchain running BFT consensus as a subroutine to protect the control plane. Operations (e.g., flow table update) should be computed, verified, and agreed by the honest majority (by BFT consensus) before admitted by all nodes. Since the blockchain is a fully ordered chain of blocks, of which all nodes must hold an identical synchronized view, all participants can reach strong consistency on all valid operations that have been confirmed. Besides, we propose an adaptive reassignment approach to timely detect byzantine nodes and replace them with honest ones. This approach contributes to both safety and liveness of Curb. And even more critical is our proposed scalable group-based control plane. Specifically, we assign each switch a controller group of size $c = 3f + 1$ where c is a constant and at most f can be byzantine. The controller groups might have overlaps, but the number of groups is $O(N)$ where N is the number of controllers. We adopt an optimization programming solver to randomly and deterministically decide the assignment. To achieve scalability, the blockchain consensus is divided into two major stages: intra-group consensus and final consensus. Each group first executes an intra-group consensus to make internal decisions on transactions, which are further confirmed through the final consensus manipulated by the final committee. The message complexity of either intra-group consensus or the final consensus is $O(c^2)$. Hence the message complexity of the Curb protocol is $O(kc^2) = O(N)$, where k is the number of groups and is asymptotically $O(N)$. The primary contributions of this paper can be summarized as follows:

1) We propose Curb, a trusted and scalable SDN control plane on edge layer. Compared to existing approaches, Curb seamlessly incorporates blockchain and BFT consensus into group-based control plane, achieving byzantine fault tolerance, verifiability, consistency, and scalability within one framework. Extensive simulations are conducted to demonstrate the salient features of Curb.
2) To realize a scalable SDN control plane, controllers are organized into multiple groups, each taking charge of multiple switches and reaching intra-group consensus in parallel. The message complexity of each round is reduced to $O(N)$. This method significantly reduces the communication overhead incurred by blockchain consensus.
3) Last but not least, Curb provides a blockchain-secured adaptive reassignment approach for SDN control plane. So byzantine controllers can be timely detected and then rapidly replaced with honest ones. The correctness and security of reassignment are ensured by the blockchain.

The rest of the paper is organized as follows. Related works are presented in Section II. In Section III, we detail the Curb protocol. Simulation results are reported in Section IV, and this paper is concluded in Section V.

II. Related Work

In this section, we present a general overview on secure SDN for edge computing and a more specific overview on fault-tolerant SDN control plane.

A. Secure SDN for Edge Computing

Studies in this category concentrate on protecting SDN-enabled edge computing from various perspectives including service quality [10], load-balancing flow control [6], [11], device authentication [8], energy consumption [9], [12], trusted controller management [7], and information synchronization [13]. Concretely, Xu *et al.* [10] provided a secure service offloading scheme for SDN-enabled Internet of Vehicles (IoV). Zhang *et al.* [6] designed a message classification strategy for blockchain-enabled SDN, in which messages are classified before being transmitted so as to realize load balancing. Hu *et al.* [11] proposed a blockchain-assisted SDN to facilitate traffic control and flow verification. The authors also established a reward mechanism for IoT device verification. Tselios *et al.* wrote a position paper to discuss attack faces of current SDN frameworks. It points out the roles that blockchains can play in safeguarding SDN [8] and especially mentions that operations of IoT devices can be authenticated based on information recorded on a blockchain. Yazdinejad *et al.* [9] developed an energy-efficient blockchain-enabled SDN architecture to establish distributed trust among IoT devices. They saved energy by adopting an efficient routing protocol for clustered SDN-IoT networks. SmartBlock-SDN [12] presents an energy-efficient framework to integrate SDN and blockchain technologies. The framework saves energy by selecting part of the sensors to transmit messages so that the other sensors are free from heavy workloads. Gao *et al.* [7] analyzed the integration of blockchain, SDN, 5G and edge computing in a VANET system, in which a blockchain is run on the control plane for controller management. BEST [13] is a blockchain-enabled energy trading scheme for intelligent transportation system. Roadside units (RSUs) maintain a blockchain to record transactions generated in energy trading, and SDN facilitates efficient resource distribution and benefits the synchronization of blockchain databases among decentralized nodes.

Different from the above studies, Curb focuses on safeguarding the most important control plane by offering byzantine fault tolerance as well as blockchain-enabled security

features, and organizing a scalable SDN control plane to reduce the complexity brought by blockchain consensus.

B. Fault-tolerant SDN Control Plane

Next we review the existing studies that focus on the fault tolerance of SDN control plane. These studies shed light on implementing a trusted control plane, even though they are not completely designed for edge computing. The redundant controller assignment approach was first proposed by [14]. This approach provides a novel SDN architecture which assigns each switch with multiple controllers. It also formulates an NP-hard controller assignment in fault-tolerant SDN (CAFTS) problem. SimpleBFT and BeaconBFT [1] are two BFT SDN controllers that respectively integrate OpenFlowJ and Beacon with the BFT-SMaRt consensus algorithm. These controllers comprise a quorum of $3f + 1$ replicas to tolerate at most f byzantine controllers. The agreement-and-execution group-based approach [2] intends to improve the resource utilization of heterogeneous controllers by dividing them into fast and slow ones that can work at different levels of difficulty in the BFT consensus process. MORPH [4] is a primary-backup based framework designed for the BFT control plane. It makes use of distributed comparators to constantly identify erroneous configuration and inconsistent controller states and a centralized reassigner to reassign controller-switch connections. P4BFT [3] reduces traffic load of BFT consensus on the control plane through control packet comparison and deduction, which are executed by P4-enabled switches. Mohan *et al.* [5] mapped each switch to $f+1$ primary controllers and f back-up ones to defend against byzantine attacks. The authors also proposed two remapping algorithms namely MINCON and MINRUS, with the former minimizing the number of controllers used and the latter additionally considering the number of changes during remapping.

Compared to the existing fault-tolerant control plane schemes mentioned above, Curb additionally introduces blockchain for storing historical information, and provides a group-based method to alleviate communication overheads caused by the redundant controller assignment and the complex consensus process.

III. Curb Design

A. Design Goals

The proposed Curb is supposed to accomplish three major design objectives:

1) BFT control plane: All configurations should be agreed by the honest majority through a consensus process and thus the network cannot be maliciously configured or tampered with by byzantine controllers. Configuration changes need to be fully and immutably recorded by a blockchain to guarantee secure flow table maintenance.
2) Scalable network and linear message complexity: The throughput should linearly increases with the growing number of switches or hosts. The message complexity is $O(N)$, where N is the network size.

TABLE I
Summary of Notations

Symbol	Description
C	the set of all controllers
S	the set of all switches
A_c	agree messages stored by controller c
F_c	final-agreed messages stored by controller c
$\mathsf{R_s}$	REPLY message set of switch s
TX	a transaction
OP()	optimization programming solver
$\mathsf{ctrList_s}$	controller list of switch s
reqMsg	a request message
config	configuration
pk, sk	public and private keys
swList	list of all switches
ctrList	list of all controllers
txList	list of transactions
finalCom	final committee member list
LEADER	the set of all group leaders
reqBuffer	the buffer storing handled requests
blockBuffer	the buffer storing received and ordered blocks
PKT-IN	packet-in request
RE-ASS	reassignment request

3) Byzantine-fault controller detection and update: In the process of SDN running, the system should be capable of detecting byzantine controllers according to its behaviors, and the byzantine nodes should be efficiently and timely replaced with honest ones to keep the network live and safe.

B. System Elements

Host and switch: Hosts (or devices) send packets to switches (or routers)[1]. Switches then parse the header fields and match each packet to a flow rule. In Curb, switches can transfer packet-in/reassignment requests to controllers, maintain a controller list, and determine which controllers are byzantine based on their responses. These functionalities can be implemented by setting up a proxy on a switch [1], [4], or adopting SDN-enabled switches with simple intelligence [5].

Northbound and southbound API: Northbound APIs are responsible for the communication and orchestration between application services and the SDN control plane. Users are allowed to control the network by manipulating controllers through Northbound APIs. Southbound APIs (e.g., OpenFlow protocol) enable communications between controllers and switches. Switches can identify network states, request flow tables, and implement other requests via Southbound APIs.

Controller: SDN controllers, as the brain of a control plane, provide the network with flexible programmability. Controllers are to manage the data plane through southbound APIs. For example, a switch finding no flow rule matched to a certain packet can request updating the flow table from controllers with an `PACKET_IN` request. Commonly, controllers can be either state-independent (SI) or state-dependent (SD) based on the requirements of upper-layer application services. SI controllers do not require controller synchronization as SD ones do. SD controllers act based on historical configurations,

[1]For simplicity, we use "host" and "switch" throughout this paper.

thus compared to SI ones, they add complexity by introducing state synchronization techniques. In this paper, Curb supports both SI and SD application services since the blockchain can be regarded as a sate machine replication protocol, which requires all honest controllers to achieve consensus on ordered blocks, keeping the state machine fully synchronized.

Controller group and final committee: In our scheme, each switch is assigned a controller group which constitutes a number of controllers. The assignment is determined by controller-to-switch (C2S) and controller-to-controller (C2C) link delay, controller capacity, and fault tolerance level. When a single-point failure occurs such as packet dropping, denial of service, or misconfiguration of flow tables, the controllers within the same group can take responsibility. More importantly, each operation can be decided by the majority of the honest controllers through a consensus protocol. However, consensus and blockchain might incur extra communication overheads. Hence we adopt a group-based network topology and set up a final committee, which helps Curb to decrease message complexity from $O(N^2)$ to $O(N)$. The protocol specification is left to Section III-C.

Blockchain component: each controller is accompanied with a blockchain system, which mainly consists of a consensus core and a blockchain database. The consensus core determines the order of blocks. In this paper, the consensus algorithm we adopt is the practical byzantine fault-tolerant (PBFT) algorithm, but we would like to emphasize that other BFT consensus algorithms can also be applied to Curb. The blockchain database persistently stores the chain of blocks, and all the blockchain systems have an identical view of the blockcchain ledger. Each block contains in its block body serialized transactions in which critical information created in SDN scenarios is recorded. For example, a transaction can record new flow rules, controller assignment scheme, etc.

C. Workflow

Next, we illustrate the workflow of Curb step by step. Curb first runs the initialization procedure [Step 0], then carries out Steps 1-4 (denoted as one round) to handle requests and settle down blocks as shown in Fig. 2.

Initialization [Step 0]. Each controller first generates a pair of keys $(\mathsf{pk}, \mathsf{sk})$ and broadcasts pk as its identity (ID) to the network. Then we assign each switch a controller group using an optimization programming solver, namely $\mathsf{OP}()$. The basic $\mathsf{OP}()$ can take as inputs a switch list (denoted by swList), a controller list (denoted by ctrList), and a series of constraints concerning controller group size, controller capacity, and C2C and C2S link delays. The minimum controller group size is determined by the number of faulty nodes that a group is required to tolerate in a specific application. A controller's capacity represents the maximum number of switches that the controller can be concurrently connected to. The C2C and C2S link delays are determined by geographic distances and bandwidths. The basic $\mathsf{OP}()$ provides each switch a random and deterministic controller group denoted by ctrList. Note that the reassignment operation (explained in Step 4) can call a different $\mathsf{OP}()$ solver program, which adds new constraints to the basic $\mathsf{OP}()$ solver.

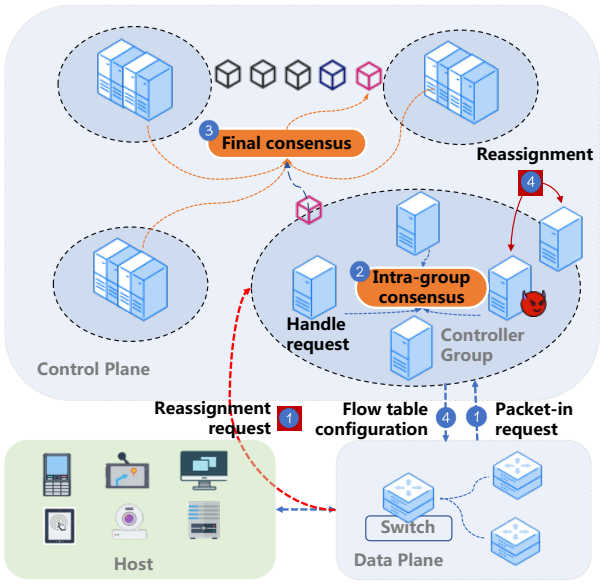


Fig. 2. The workflow of Curb protocol

Algorithm 1: Routine of s-agent s_i

upon event request *received from a host*
  broadcast $\langle \mathsf{reqMsg}, S_i \rangle$ to $c \in \mathsf{ctrList}_s$
upon event $\langle \mathsf{REPLY}, c, \mathsf{reqMsg}, \mathsf{config} \rangle$ *received from controller* c
  if config *matches with* f *messages in* R_s **then**
    if reqMsg = PKT-IN **then**
      update flow table with config
    else if reqMsg = RE-ASS **then**
      update $\mathsf{ctrList}_s$ with config
  else
    add $\langle \mathsf{REPLY} \rangle$ to R_s
  for *each* $c_j \in R_s$ **do**
    if config *does not match* f *messages* **then**
      broadcast $\langle \mathsf{RE\text{-}ASS}, s, c_j \rangle$ to $\mathsf{ctrList}_s$

In the following we illustrate the details of implementing the basic $\mathsf{OP}()$. To maximize the utilization of controller resources, we formulate the objective function in the controller assignment problem (CAP) as minimizing the controller usage. Let binary variable $A_{ij} \in \{0, 1\}$ denote whether the controller j is assigned to the switch i. Let $x_j \in \{0, 1\}$ denote whether the controller j is assigned to a switch or not. Denote the set of all controllers as C and the set of all switches as S. Then one can represent the objective function as follows.

$$[O1] \quad \min \sum_{j \in C} x_j, \tag{1}$$

Algorithm 2: Utilities of controller c_i

Function Initialization()
  generate $(\mathsf{sk}, \mathsf{pk})$ and broadcast $\mathsf{ID} = \mathsf{pk}$
  $\mathsf{ctrList}_i = \mathsf{OP}(\mathsf{swList}, \mathsf{ctrList}, \mathsf{constraints})$
  recognize the finalCom
  create the genesis block
Function HandleRequest(reqMsg, s, c, σ)
  if $\langle \cdot, \mathsf{reqMsg}, s, c, \cdot \rangle \in$ reqBuffer **then**
    discard the request
  else
    config = ComputeConfig(reqMsg)
    add $\langle \mathsf{TX}, \mathsf{reqMsg}, s, c, \mathsf{config} \rangle$ to reqBuffer
Function ComputeConfig(reqMsg)
  if reqMsg = PKT-IN **then**
    return new flow entries as config
  else if reqMsg = RE-ASS **then**
    remove byzantine nodes from ctrList
    config = OP(swList, ctrList, constrants)
    return config

where

$$\frac{1}{N}\sum_{i\in S} A_{ij} \le x_j \le 1, \qquad \forall j \in C.$$

To tolerate f byzantine faulty controllers in each controller group, each switch should be governed by a controller group of size at least $B_i = 3f + 1$. Hence we formulate such a constraint for each switch as below.

$$[C1.1] \quad \sum_{j\in C} A_{ij} \ge B_i, \qquad \forall i \in S. \tag{2}$$

In addition, each controller has limited processing capability for incoming requests generated from hosts, thus we should set a constraint for the controller capacity to realize load balance. Denote Q_i as the maximum message load the switch i sends in unit time and let C_j be the capacity of controller j. Then the total number of received messages in a given time is upper bounded by $\sum_{i\in S} A_{ij}Q_i$, which should not exceed C_j.

$$[C1.2] \quad \sum_{i\in S} A_{ij}Q_i \le C_j, \qquad \forall j \in C. \tag{3}$$

Furthermore, we establish constraints on the C2S and C2C link delays. This is useful in reducing the time cost of message transmissions and the consensus process. The constraint of C2S link delay is formulated as follows.

$$[C1.3] \quad A_{ij}d_{ij} \le D_{c,s}, \qquad \forall i \in S, \tag{4}$$

where d_{ij} denotes the actual unidirectional delay from controller j to switch i and $D_{c,s}$ denotes the threshold of delay for every C2S link. The d_{ij} is determined by the geographic distance between switch i and controller j. And the constraint of C2C link delay is depicted as follows.

$$[C1.4] \quad A_{ij}A_{ij'}d_{jj'} \le D_{c,c} \qquad j \ne j', \forall j, j' \in C, \forall i \in S \tag{5}$$

where $d_{jj'}$ denotes the real unidirectional delay from controller j to controller j' and $D_{c,c}$ denotes the threshold of delay for controller-controller links.

Algorithm 3: Event handlers of controller c_i

▷ intra consensus within each controller group
upon event $\langle \mathsf{reqMsg}, s \rangle$ *received from the switch* s
  if $c_i \in \mathsf{ctrList}_s$ **then**
    if c_i *is the leader of* $\mathsf{ctrList}_s$ **then**
      HandleRequest(reqMsg, s, c, σ)
      if *time out* ***or*** reqBuffer *is full* **then**
        pack $\langle \mathsf{TX} \rangle$s in reqBuffer into a txList
        launch Intra-PBFT(txList)
    else
      participate into Intra-PBFT(txList)
    if Intra-PBFT(txList) *is completed* **then**
      broadcast $\langle \mathsf{AGREE}, \mathsf{ctrListID}, \mathsf{txList} \rangle$ to finalCom
▷ final consensus across all controller groups
upon event $\langle \mathsf{AGREE}, \mathsf{ctrListID}, \mathsf{txList} \rangle$ *from a controller*
  if $c_i \in$ finalCom **then**
    if c_i *is the* finalCom *leader* **then**
      add txList to blockBuffer
      if *time out* ***or*** blockBuffer *is full* **then**
        pack all txList into a new block B_h
        launch Final-PBFT(B_h)
    else
      participate into Final-PBFT(B_h)
    if Final-PBFT(B_h) *is completed* **then**
      broadcast $\langle \mathsf{FINAL\text{-}AGREE}, c, B_h \rangle$
upon event $\langle \mathsf{FINAL\text{-}AGREE}, c, B_h \rangle$ *from controller* v
  if FINAL-AGREE *matches with* f *messages* $\in F_c$ **then**
    add B_i to blockchain
    for ***each*** $\langle \mathsf{TX} \rangle \in B_h$ **do**
      //$\langle \mathsf{TX} \rangle$ maps to $\langle \mathsf{TX}, \mathsf{reqMsg}, s, c, \mathsf{config} \rangle$
      send $\langle \mathsf{REPLY}, c, \mathsf{reqMsg}, \mathsf{config} \rangle$ to s
  else
    add $\langle \mathsf{FINAL\text{-}AGREE} \rangle$ into F_c

For a switch i, OP() returns a controller list $\mathsf{ctrList}_i$. The final committee finalCom is of size $3f+1$, and is chosen from the first $3f+1$ controller groups sorted by the group identity number. Since OP() might output two controller groups with overlapped controllers, we let each controller group elect only one finalCom member which should not be included by the members elected from previous groups. Since all switches and controllers have an identical view of the assignment, they can make the same choice on finalCom following the same finalCom selection rule. After settling down the controller list

and the final committee list, all initial controllers create the genesis block to record the initialization results. After initialization, the network can process and forward data packages generated by hosts.

Curb supports two basic functionalities: flow table configuration and controller reassignment. A flow table contains multiple flow entries and determines how to forward packets. A switch can receive configuration information from assigned controllers and then change flow entries accordingly. Besides, if a controller is found malicious, a switch can start a controller reassignment process to kick the malicious controller out. In the following, we depict the complete workflow of Curb in detail. As shown in Fig. 2, each round consists of four major steps.

Sending Packet-in/Reassignment Requests [Step 1]. To apply for new flow table entries or launch a controller reassignment, a switch s can broadcast PKT-IN (notation of a `PACKET_IN` request) or RE-ASS (notation of a `RE_ASSIGNMENT` request) to the controller group it belongs to through the southbound OpenFlow channels. For simplicity, we denote a request as reqMsg. Each request should be signed so that controllers can validate its authenticity and integrity. A switch sends a `PACKET_IN` request to a controller when the packet should be forwarded to the controller reserved port using explicitly specified flow entry or table-miss flow entry. In response, the controller sends the switch a `PACKET_OUT` message containing a list of actions that can be applied to the switch. For example, the controller can modify flow tables by embodying a `PACKET_OUT` message with `FLOW_MOD` commands. Switches can choose to configure packet-in events using buffered packets, which are processed after receiving `PACKET_OUT` messages or expire after a period of time. And a `PACKET_OUT` message can contain a buffer ID referencing a specific buffered packet.

Intra-group Consensus [Step 2]. In each controller group, only one controller is appointed as the leader, who needs to handle requests, generate new transactions, and launch intra-group consensus for the controller group it governs. Once receiving a reqMsg, the group leader invokes HandleRequest() to process it (line 6-11 in Algorithm 2). If $\langle \cdot, \mathsf{reqMsg}, s, c, \cdot \rangle$ has been recorded in a request buffer denoted by reqBuffer, the leader discards it; otherwise, the leader can compute a configuration as config by calling ComputeConfig(reqMsg). Concretely, computing config is to create new flow entries for a PKT-IN, or process reassignment for a RE-ASS by removing byzantine nodes from the full controller list ctrList and then computing OP(swList, ctrList, constrants). Afterwards, the leader generates a transaction TX and adds $\langle \mathsf{TX}, \mathsf{reqMsg}, s, c, \mathsf{config} \rangle$ to its local reqBuffer. Once reqBuffer is full, the leader packs all transactions into a new transaction list txList and launches intra-group consensus, namely Intra-PBFT(txList), which is a byzantine fault tolerant consensus algorithm. In this paper, we leverage PBFT as a standard consensus core due to its matureness and effectiveness validated by the Hyperledger Sawtooth, but Curb can be implemented with other BFT protocols including Tendermint and Hotstuff. The intra-group consensus starts from a proposal sent by the leader. Followers (controllers in a group except for the leader) in the same group can participate into the consensus process. For simplicity, we eliminate further details of PBFT in Algorithm 2. At the end of intra-group consensus, each controller broadcasts a $\langle \mathsf{AGREE}, \mathsf{ctrListID}, \mathsf{txList} \rangle$ to the final committee finalCom. Note that all controller groups can process requests in parallel, which can significantly reduce the consensus time. To ensure consistency of configurations across all groups, we set a final consensus process, which agrees on the order of blocks generated by each controller group. In the following we illustrate the process of carrying out final consensus.

Final Consensus [Step 3]. The final committee is in charge of the final consensus process with the same consensus algorithm (i.e., PBFT) to decide on the order of blocks received from controller groups. The final committee leader (with the highest ID number in the committee) is responsible for generating new blocks and proposing the blocks to the final committee members. A new block B_h serializes transactions in txList received from the controller groups. The final committee leader launches the final consensus procedure, namely Final-PBFT(B_h). Once final consensus is reached, the final committee members broadcast B_h to all controllers for information update. For each controller c, it should receive $f+1$ identical final-agreed messages to confirm that B_h is valid. This is to prevent from byzantine final committee members who propose malicious blocks. If B_h is valid, a controller can add B_h to its local blockchain database. Then for each $\langle \mathsf{TX} \rangle \in B_h$, each controller sends $\langle \mathsf{REPLY}, c, \mathsf{reqMsg}, \mathsf{config} \rangle$ to s.

Finish Configuration [Step 4]. The final step is to complete the configuration procedure following config received from a controller group. A switch s records the received reply messages in R_s. After receiving at least $f+1$ consistent config, s updates its configuration accordingly. If reqMsg = PKT-IN, s can update the local flow rules according to config. If reqMsg = RE-ASS, a switch should change links to controllers for a controller group update, which aims to remove faulty controllers from the network. Additionally, s also detects byzantine controllers based on the faulty configuration, which contradicts other $f+1$ identical config. Then s reports faulty controllers in a RE-ASS message to apply for a replacement of malicious controllers. A valid RE-ASS request can lead to a reconfiguration, which removes faulty controllers from and add new honest controllers to the current controller group.

The basic OP formulation can be described as follows.

$$
\begin{aligned}
&[O2] \quad \min \sum_{j \in C} x_j, \\
&[C2.1] \quad \sum_{j \in C} A_{ij} \geq B_i, \qquad \forall i \in S, \\
&[C2.2] \quad \sum_{i \in S} A_{ij} Q_i \leq C_j, \qquad \forall j \in C, \\
&[C2.3] \quad A_{ij} d_{ij} \leq D_{c,s}, \qquad \forall i \in S,
\end{aligned}
$$

$$
\begin{aligned}
&[C2.4] \quad A_{ij}A_{ij'}d_{jj'} \leq D_{c,c} \qquad j \neq j^{'}, \forall j, j^{'} \in C, \forall i \in S,\\
&[C2.5] \quad x_j = 0, \qquad \forall j \in C_{byz},\\
&[C2.6] \quad A_{ij} = 1, \qquad \forall (i,j) \in \mathsf{LEADER},
\end{aligned}
\tag{6}
$$

where the objective function $[O2]$ and the first four constraints $[C2.1-2.4]$ are consistent with the formulation $[O1, C1.1-1.4]$ of the basic OP. And $[C2.5] \quad x_j = 0, \forall j \in C_{byz}$ is the constraint requiring that all byzantine controllers are removed from the network, where C_{byz} denotes the set of all byzantine controllers. The constraint $[C2.6] \quad A_{ij} = 1, \forall (i,j) \in$ LEADER, where LEADER is the set of all group leaders, is to fix leader positions in all controller groups. $[C2.6]$ targets to reduce the link changes since a leader connects to all followers. There are two types of OP solvers proposed for the reassignment process. Eq. (6) represents the first type named trivial controller reassignment (TCR), while the second type called least-movement controller reassignment (LCR) adopts a different objective function as Eq. (7).

$$
[O3] \quad \min\{\sum_{j \in C} x_j + \sum_{i \in S \wedge j \in C} |A_{ij} - a_{ij}|\}. \tag{7}
$$

where A_{ij} and a_{ij} denote the new assignment and the previous assignment, respectively. Hence $\min\{\sum_{j \in C} x_j + \sum_{i \in S \wedge j \in C} |A_{ij} - a_{ij}|\}$ takes into consideration both minimized controller usage and minimized changed links. These two objective functions make trade off between the OP solving time and the controller movements. We show the performance of both OP solvers and their adaptions in Section IV.

D. Message Complexity of the Curb Protocol

To demonstrate the efficiency of Curb, we provide the following theorem and its informal proof regarding the message complexity. A more formal analysis can be found in the full version of this paper.

Theorem 1. *The message complexity of Curb is $O(N)$, where N is the number of SDN controllers.*

Proof. Assume the number of groups is k and the average group size is c. According to Eq. (6) [C2.1], c is lower-bounded by $B_i = 3f + 1$, where f is the maximum number of faulty nodes a controller group can tolerate. Besides, c is asymptotically equal to B_i since the OP solver satisfying Eq. (6) [C2.1] cannot output a value of $\sum_{j \in C} A_{ij}$ much higher than B_i but a value close to B_i. In Curb we set the value of f to be $O(1)$, so $O(c)$ and $O(B_i)$ both belong to $O(1)$. Since $O(k \cdot c) = O(N)$ [2], it is straightforward to obtain that k is $O(N)$. According to Algorithm 3, the message complexity of Curb is determined by four major steps: intra-group consensus, broadcasting AGREE messages to final committee, final consensus, and broadcasting FINAL-AGREE messages to controllers. The message complexity of the intra-group consensus is $O(kc^2)$ since there are k groups with each running PBFT ($O(c^2)$). Similarly, the message complexity of the final consensus is $O(c^2)$ since the final committee is of size $O(c)$. The message complexity of broadcasting the AGREE messages and broadcasting the FINAL-AGREE messages is $O(Nc)$ and $O(cN)$, respectively. Therefore, the message complexity of Curb is $O(kc^2 + c^2 + 2cN) = O(N)$. □

[2]Note that $N = k \cdot c$ might not hold since controller groups can overlap due to the characteristic of OP.

IV. Evaluations

Configuration: We conduct the simulation of Curb with around 6700 lines of Python codes atop Mininet [15] (version 2.3.0), and show the network topology with about 2200 lines of HTML codes. With Mininet, Curb first emulates Open vSwitch. Then Ryu [16] running the OpenFlow [17] protocol is used to implement SDN controllers. The communications among controllers and switches are based on gRPC [18], a widely adopted Remote Procedure Call (RPC) framework. We use the Gurobi optimizer [19] (version 9.5.0), a mathematical optimization solver to construct the OP solver, and NetworkX [20], a software for complex networks to compute the shortest paths and path lengths. The shortest paths determine the flow rules that controllers would send to switches. Moreover, key generation and digital signature schemes are supported by the pure-Python ECDH and ECDSA. To validate Curb in a realistic simulation setting, we adopt the publicly available Internet2 topology as shown in Fig. 3. The velocity of light in cables is 2×10^8 m/s and the bandwidth is set as 100 Mbps. Together with the path lengths, we can calculate the link delay of any path in Internet2.

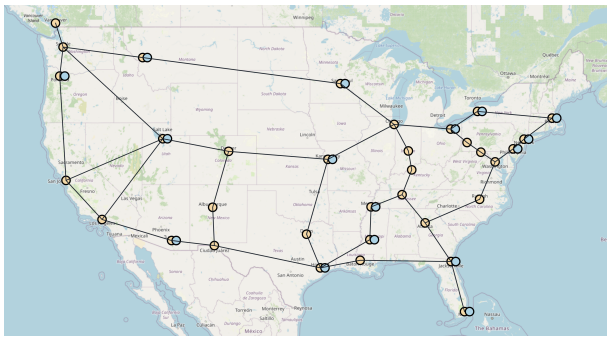
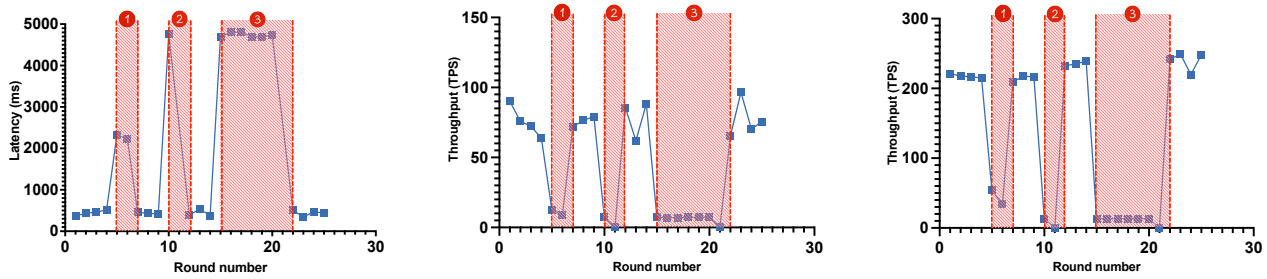


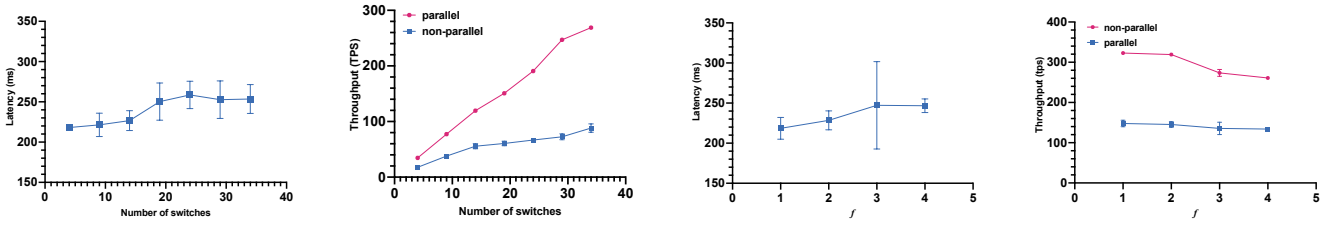
Fig. 3. An example of Internet2 topology with 16 controllers (blue points) and 34 switches (yellow points). Each edge represents the link between two nodes.

The simulation is run on a DELL PowerEdge R740 server with two Intel silver 4214R CPU, each having 12 CPU cores and 2.40 GHz frequency. The DRAM size is 128 GB (2×64 GB) with the type of DDR4. The L1 cache of each CPU core is 768 KB while the L2 cache is 12 MB. The 12 cores in each CPU processor share the same 16.5 MB L3 cache. Except where noted, we measure 200 times for each node to obtain the average of each data point after a warm-up of 30 seconds, and each test involves 34 switches and 16 controllers following the original Internet2 topology. Besides, the constraints of C2C link delay [C1.4] and [C2.4] are not adopted in all experiments since they are not compulsory and removing them avoids large computation overheads (details



(a) Latency *vs.* round (b) Throughput *vs.* round (non-parallel) (c) Throughput *vs.* round (parallel)

Fig. 4. Performance of handling the PACKET_IN requests



(a) Latency *vs.* the number of switches (b) Throughput *vs.* the number of switches (c) Latency *vs.* f (d) Throughput *vs.* f

Fig. 5. Performance of handling the PACKET_IN requests with the varying number of switches

can be found in Section IV-B1). Our evaluation quantitatively covers the following three aspects:

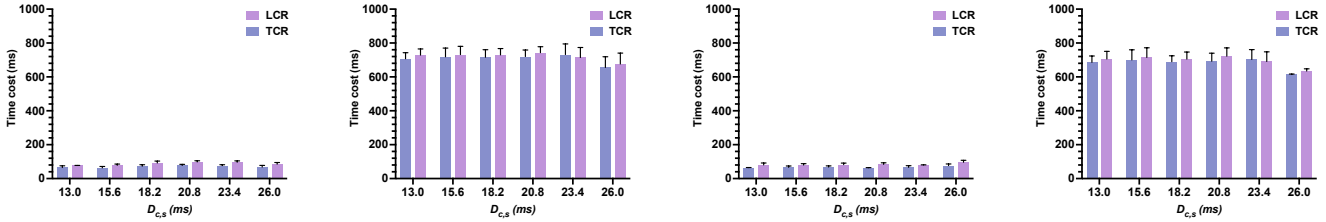
1) Curb's capability of defending against byzantine nodes;
2) the performance of handling PACKET_IN requests;
3) the performance of two types of OP solvers for controller reassignment and that of handling the RE_ASSIGNMENT requests.

A. Flow Table Update

1) Byzantine resilience test: To evaluate the capability of defending against byzantine, we conduct three experiments. Experiment ❶ presents the case when Curb handles one byzantine node which does not respond to requests within timeout (500 ms). As shown in Fig. 4, Curb finds out the byzantine node in the 5th round, and finishes reassignment and kicks out the byzantine node in the 6th round; after that, the latency and throughput recovers to the normal levels (latency is about 460.24 ms and throughput is about 71.90 TPS). Experiment ❷ simulates the case when Curb simultaneously handles three byzantine nodes in different controller groups. As shown in Fig. 4, three byzantine nodes cause large latency since they hinder the intra-group consensus of three different groups as well as the final consensus. Curb can remove these three byzantine nodes in one round by calculating OP once. The latency and throughput returns to normal within only two rounds. Finally, experiment ❸ illustrates a more intricate case when three byzantine nodes jointly cause large latency by keeping high response time in (200, 500) ms. In such a circumstance, we cannot recognize these "lazy" nodes since they do not exceed the designated timeout of 500 ms, but they can negatively impact the performance. Hence we choose to only allow lazy nodes to exist for 5 rounds (application-specific waiting time), after which they are being treated as byzantine nodes. Besides, in Fig. 4(c), Curb adopts a parallel processing mode. The intra-group consensus and final consensus proceed in parallel to achieve high performance. The results demonstrate that the parallel Curb has about $2 \sim 3$ times of the throughput of the non-parallel mode.

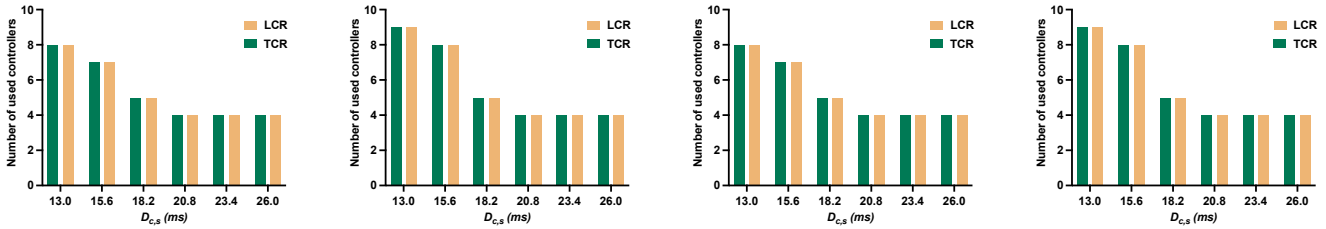
2) Performance of handling the PACKET_IN requests: In the following, we illustrate the performance of handling the PACKET_IN requests with the varying number of switches and f. The simulations measure the latency of handling the PACKET_IN requests and the throughput of responses to the PACKET_IN requests. The number of switches in Fig. 5 varies in [4, 34]. Note that the error bar would not be shown in figures if the error doesn't exceed a threshold. As shown in Fig. 5(a), the latency slightly increases, ranging from 218 ms to 258 ms with the increasing number of switches. Fig. 5(b) indicates that the throughput of both non-parallel and parallel Curb linearly increase with the number of switches. The throughput keeps on rising because 16 controllers are sufficient to handle all switches. With more switches, the users need to increase the number of controllers according to the results of OP.

Next, we discuss the results presented in Fig. 5(c) and 5(d), where f represents the maximum number of byzantine nodes that a controller group can tolerate. To protect against byzantine attacks, each controller group is of size $3f + 1$. For example, when $f = 4$, each switch is governed by 13 controllers. The larger the f, the more number of controllers are required. The latency of handling the PACKET_IN requests increases with f since assigning more controllers in each group results in larger communication overheads in both



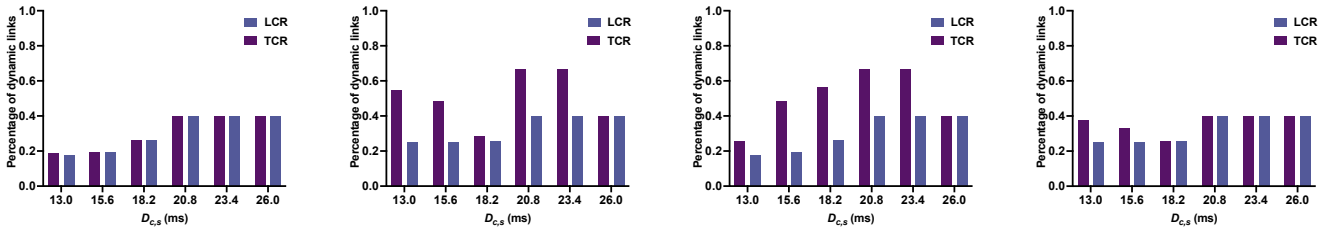
(a) With the leader constraint (b) With the $D_{c,c}$ constraint (c) Without the leader and $D_{c,c}$ constraints (d) With the leader and $D_{c,c}$ constraints

Fig. 6. Time cost *vs.* $D_{c,s}$



(a) With the leader constraint (b) With the $D_{c,c}$ constraint (c) Without the leader and $D_{c,c}$ constraints (d) With the leader and $D_{c,c}$ constraints

Fig. 7. The number of used controllers *vs.* $D_{c,s}$



(a) With the leader constraint (b) With $D_{c,c}$ constraint (c) Without the leader and $D_{c,c}$ constraints (d) With the leader and $D_{c,c}$ constraints

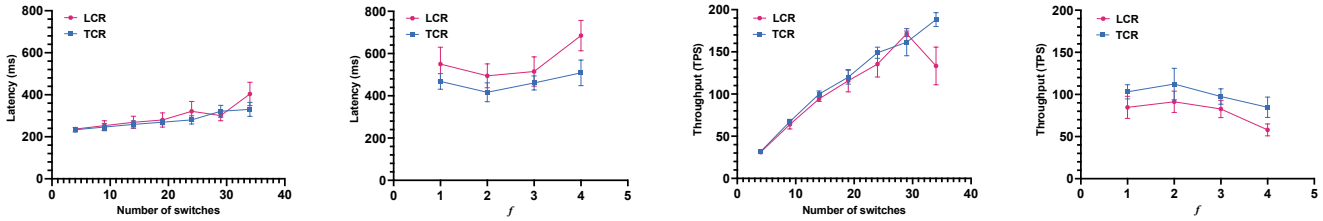
Fig. 8. The percentage of dynamic links (PDL) *vs.* $D_{c,s}$

intra-group and final consensus. Nevertheless, the throughput only slightly decreases since each controller group processes requests in parallel, which significantly avoids the communication bottleneck that might occur with centralized SDN controllers.

B. Reassignment

1) Performance of OP: The performance metrics include time cost, the number of used controllers, and percentage of dynamic links. The time cost reflects the efficiency of solving OP. The number of used controllers are determined by the OP whose objective function is to minimize the controller usage. We utilize the percentage of dynamic links (PDL) to depict the dynamicity of the SDN topology. PDL is the number of links changed over the total number of links that exist during a reassignment. For example, assume we have a network consisting of 30 links in total. If after a reassignment, Curb removes one controller which connects two switches (delete two links) and adds a new controller which connects three switches (add three new links). Then PDL is calculated by $(2+3)/(30+3) = 15\%$. In this study, we compare TCR and LCR with varying $D_{c,s}$ under different combinations of constraints. Recall that the leader constraint is $A_{ij} = 1, \forall (i,j) \in \mathsf{LEADER}$ and the C2C delay constraint is $A_{ij}A_{ij'}d_{jj'} \leq D_{c,c}$, where $j \neq j^{'}, \forall j, j^{'} \in C, \forall i \in S$.

Fig. 6 reports the time cost *vs.* $D_{c,s}$. We can draw the following useful conclusions. First of all, the leader constraint causes little extra time cost while the $D_{c,c}$ constraint brings large extra time cost of round 600 ms. Without the $D_{c,c}$ constraint, the time cost is lower than 100 ms. Secondly, the time cost of TCR is slightly lower than LCR since LCR considers a more complicated objective function. Thirdly, the $D_{c,s}$ does not clearly impact the OP solving time. Fourth, the $D_{c,c}$ constraint leads to significant time overheads since the constraint is quadratic. Solving an integer quadratically constrained programming (IQCP) problem takes more time than an integer linear programming (ILP) problem with similar constraints except for a quadratic constraint. In Fig. 7, Curb utilizes less controllers if $D_{c,s}$ is higher. A higher $D_{c,s}$ allows a switch to connect controllers in a wider range so using



(a) Latency *vs.* the number of switches (b) Latency *vs.* f (c) Throughput *vs.* the number of switches (d) Throughput *vs.* f

Fig. 9. Performance of handling the `RE_ASSIGNMENT` requests

less controllers can meet the constraints. Besides, the TCR and LCR methods output the same number of controllers being used since both objective functions aim to minimize the controller usage. Moreover, adding the $D_{c,c}$ constraint can result in more controllers enrolled because more controllers can compose a better connected network with lower link delay. The leader constraint does not change the controller usage but has impact on PDL.

Finally, we explore how much the reassignment can change the network links. Firstly, Curb changes less links with a lower $D_{c,s}$ as presented in Fig. 8. This is because a lower $D_{c,s}$ can result in a larger number of used controllers according to Fig. 7. Hence each controller can have fewer links with switches and substituting a byzantine controller causes less link changes. Secondly, LCR has better performance of PDL than TCR. The reason is that LCR's objective function considers the minimized controller usage plus the minimized number of changed links, but TCR only minimizes the controller usage. Thirdly, bringing the leader constraint can result in less PDL. The second and third observations indicate that to have a less dynamical network, we can set a lower $D_{c,s}$ if controllers are sufficient to be used, or bring the leader constraint.

2) Performance of handling the `RE_ASSIGNMENT` requests: Next we simulate the process of handling RE-ASS. In Fig. 9(a), the latency *vs.* the number of switches slowly increases. The latency with TCR and LCR solvers is very close since the time cost of TCR and LCR is nearly the same as shown in Fig. 6(a). In Fig. 9(b), Curb adopting LCR has larger latency than using TCR. With more switches and controllers, the extra time cost of LCR compared to TCR become more explicit. Concerning throughput, Fig. 9(c) shows that LCR and TCR have similar throughput, and the throughput of both cases increases with the number of switches. With an increasing f, the throughput decreases because a larger group size results in a slower consensus process.

V. Conclusion

We present Curb, a novel SDN control plane scheme that seamlessly integrates blockchain and BFT consensus into a group-based control plane, addressing security and scalability concerns of the state-of-the-arts. Curb supports trusted flow rule updates and adaptive controller reassignment. Importantly, Curb uses a group-based technique to realize a scalable network where the message complexity of each round is upper-bounded by $O(N)$. We also conduct extensive simulations on classical Internet2 network topology to validate our approach.

Acknowledgment

This study was partially supported by the National Key R&D Program of China under grant 2019YFB2102600, the National Natural Science Foundation of China under grants 61832012, 62102232 and 62122042, and the Blockchain Core Technology Strategic Research Program of Ministry of Education of China under grant 2020KJ010301.

References

[1] K. ElDefrawy and T. Kaczmarek, "Byzantine fault tolerant software-defined networking (sdn) controllers," in *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, vol. 2. IEEE, 2016, pp. 208–213.

[2] E. Sakic and W. Kellerer, "Bft protocols for heterogeneous resource allocations in distributed sdn control plane," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–7.

[3] E. Sakic, N. Deric, E. Goshi, and W. Kellerer, "P4bft: Hardware-accelerated byzantine-resilient network control plane," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2019, pp. 1–7.

[4] E. Sakic, N. Đerić, and W. Kellerer, "Morph: An adaptive framework for efficient and byzantine fault-tolerant sdn control plane," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 10, pp. 2158–2174, 2018.

[5] P. M. Mohan, T. Truong-Huu, and M. Gurusamy, "Primary-backup controller mapping for byzantine fault tolerance in software defined networks," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7.

[6] P. Zhang, F. Liu, N. Kumar, and G. S. Aujla, "Information classification strategy for blockchain-based secure sdn in iot scenario," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2020, pp. 1081–1086.

[7] J. Gao, K. O.-B. O. Agyekum, E. B. Sifah, K. N. Acheampong, Q. Xia, X. Du, M. Guizani, and H. Xia, "A blockchain-sdn-enabled internet of vehicles environment for fog computing and 5g networks," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4278–4291, 2019.

[8] C. Tselios, I. Politis, and S. Kotsopoulos, "Enhancing sdn security for iot-related deployments through blockchain," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 2017, pp. 303–308.

[9] A. Yazdinejad, R. M. Parizi, A. Dehghantanha, Q. Zhang, and K.-K. R. Choo, "An energy-efficient sdn controller architecture for iot networks with blockchain-based security," *IEEE Transactions on Services Computing*, vol. 13, no. 4, pp. 625–638, 2020.

[10] X. Xu, Q. Huang, H. Zhu, S. Sharma, X. Zhang, L. Qi, and M. Z. A. Bhuiyan, "Secure service offloading for internet of vehicles in sdn-enabled mobile edge computing," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 6, pp. 3720–3729, 2020.

[11] K. Kataoka, S. Gangwar, and P. Podili, "Trust list: Internet-wide and distributed iot traffic management using blockchain and sdn," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018, pp. 296–301.

[12] A. Rahman, M. J. Islam, A. Montieri, M. K. Nasir, M. M. Reza, S. S. Band, A. Pescape, M. Hasan, M. Sookhak, and A. Mosavi, "Smartblocksdn: An optimized blockchain-sdn framework for resource management in iot," *IEEE Access*, vol. 9, pp. 28 361–28 376, 2021.

[13] R. Chaudhary, A. Jindal, G. S. Aujla, S. Aggarwal, N. Kumar, and K.-K. R. Choo, "Best: Blockchain-based secure energy trading in sdn-enabled intelligent transportation system," *Computers & Security*, vol. 85, pp. 288–299, 2019.

[14] H. Li, P. Li, S. Guo, and A. Nayak, "Byzantine-resilient secure software-defined networks with multiple controllers in cloud," *IEEE Transactions on Cloud Computing*, vol. 2, no. 4, pp. 436–447, 2014.

[15] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as software defined networking testing platform," in *International Conference on Communication, Computing & Systems (ICCCS)*, 2014, pp. 139–42.

[16] R. S. F. Community. Ryu. [Online]. Available: https://ryu-sdn.org/ [Accessed: 2022-01-31]

[17] O. N. Foundation. Openflow. [Online]. Available: https://opennetworking.org/sdn-resources/customer-case-studies/openflow/ [Accessed: 2022-01-31]

[18] gRPC authors. grpc. [Online]. Available: https://grpc.io/ [Accessed: 2022-01-31]

[19] Gurobi. Gurobi optimizer. [Online]. Available: https://www.gurobi.com/products/gurobi-optimizer/ [Accessed: 2022-01-31]

[20] NetworkX. Networkx. [Online]. Available: https://networkx.org/ [Accessed: 2022-01-31]