



Aloe: Fault-Tolerant Network Management and Orchestration Framework for IoT Applications

Subhrendu Chattopadhyay , Soumyajit Chatterjee , Sukumar Nandi , Sandip Chakraborty 

Abstract—Internet of Things (IoT) platforms use a large number of low-cost resource constrained devices and generates millions of short-flows. In-network processing is gaining popularity day by day to handle IoT applications and services. However, traditional software-defined networking (SDN) based management systems are not suitable to handle the plug and play nature of such systems. In this paper, we propose Aloe, an auto-scalable SDN orchestration framework. Aloe exploits in-network processing framework by using multiple lightweight controller instances in place of service grade SDN controller applications. The proposed framework ensures the availability and significant reduction in flow-setup delay by deploying instances in the vicinity the resource constraint IoT devices dynamically. Aloe supports fault-tolerance with recovery from network partitioning by employing self-stabilizing placement of migration capable controller instances. Aloe also provides resource reservation for micro-controllers so that they can ensure the quality of services (QoS). The performance of the proposed system is measured by using an in-house testbed along with a large scale deployment in Amazon web services (AWS) cloud platform. The experimental results from these two testbeds show significant improvement in response time for standard IoT based services. This improvement of performance is due to the reduction in flow-setup time. We found that Aloe can improve flow-setup time by around 10% – 30% in comparison to one of the states of the art orchestration framework.

Index Terms—Orchestration Framework, Fault-tolerance, Programmable network, IoT, In-network processing, SDN

I. INTRODUCTION

The rapid proliferation and deployments of large-scale Internet-of-Things (IoT) applications have made the network architecture complicated from the perspective of end-user service management. Simultaneously, with the advancement of edge-computing, in-network processing and platform-as-a-service technologies, end-users consider the network as a platform for the deployment and execution of myriads of diverse applications dynamically and seamlessly. Consequently, network management has become increasingly difficult in today's world with this complex service-oriented platform overlay on top of the inherently distributed TCP/IP network architecture. The concept of software-defined networking (SDN) [1] has gained popularity over the last decade to make the network management simple, cost-effective and logically centralized, where a network manager can monitor, control and deploy new network services through a central controller. Nevertheless,

edge and in-network processing over an IoT platform is still challenging even with an SDN based architecture [2].

Motivation: The primary motivation for supporting a distributed network management and flow steering framework coupled with edge and in-network processing over an IoT platform are as follows: (1) The platform should be agile to support rapid deployment of applications without incurring additional overhead, ensuring the scalability of the system [3], [4]. (2) With micro-service architectures [5], in-network processing requires dividing a service into multiple micro-services and deploying them at different network nodes for reducing the application response time with parallel computations. However, such micro-services may need to communicate with each other, and therefore the flow-setup delay from the in-network nodes need to be very low to ensure near real-time processing. (3) The percentage of short-lived flows are high for IoT based networks [6]. This also escalates the requirement for reducing flow-setup delay in the network. (4) Failure rates of IoT nodes are in-general high [7]. Therefore, the system should support a fault-tolerant or fault-resilient architecture to ensure liveness.

Limitations of existing approaches: Although SDN supported edge computing and in-network processing have been widely studied in the literature for the last few years [2], [8], [9] as a promising technology to solve many of the network management problems, they have certain limitations. First, the SDN controller is a single-point bottleneck. Every flow initiation requires communication between switches and the controller; therefore, the performance depends on the switch-controller delay. With a single controller bottleneck, the delay between a switch and the controller increases, which affects the flow-setup performance. As we mentioned earlier, the majority of the flows in an IoT network are short-lived, the impact of switch-controller delay is more severe on the performance of short-lived flows. To solve this issue, researchers have explored distributed SDN architecture with multiple controllers deployed over the network [10]. However, with a distributed SDN architecture, the question arises about how many controllers to deploy and where to deploy those controllers. Static controller deployments may not alleviate this problem, as IoT networks are mostly dynamic with plug-and-play deployments of devices. Dynamic controller deployment requires hosting the controller software over commercially-off-the-shelf (COTS) devices and designing methodologies for controller coordination, which is a challenging task [11]. The problem is escalated with the objective of developing a fault-tolerant or fault-resilient architecture in a network where the majority of the flows are short-lived flows.

S. Chattopadhyay and S. Nandi are with the Department of CSE, Indian Institute of Technology Guwahati, India 781039. email:{subhrendu, sukumar}@iitg.ac.in. S. Chatterjee and S. Chakraborty are with Department of CSE, Indian Institute of Technology Kharagpur, India, 721302. email:{soumyachatt@iitkgp.ac.in, sandipc@cse.iitkgp.ac.in}

Our contribution: To alleviate the above mentioned challenges, we, in this paper, integrate a distributed SDN control plane with the in-network processing infrastructure, such that the control plane can dynamically be deployed over the COTS devices maintaining a fault-tolerant architecture. We design a distributed, robust, migration-capable and elastically scalable control plane framework with the help of docker containers [12] and state-of-the-art control plane technologies. The proposed control plane consists of a set of small controllers, called the *micro-controllers*, which can coordinate with each other and help in deploying new applications for in-network processing. The container platform helps in installing these micro-controllers on the COTS devices; a container with a micro-controller can be seamlessly migrated to another target device if the host device fails, yielding a fault-tolerant architecture. In addition to this, the deployment mechanism for the micro-controllers ensure elastic auto-scaling of the system; the total number of controllers can grow or shrink based on the number of active devices in the IoT network. We develop a set of special purpose programming interfaces to ensure fault-tolerant elastic auto-scaling of the system along with intra-controller coordination. Finally, we design a set of application programming interfaces (API) over this platform to ensure language-free independent deployment of applications for in-network processing. Combining all these concepts, we present Aloe, a distributed, robust, auto-scalable, platform-independent orchestration framework for edge and in-network processing over IoT infrastructures.

Aloe has multiple advantages for a IoT framework with in-network processing capabilities. (a) The distributed controller approach ensures that there is no performance bottleneck near the controller. (b) The flow-setup delay is significantly minimized because of the availability of a controller near every device. (c) The fault-tolerant controller orchestration ensures the liveness of the system even in the presence of multiple simultaneous devices or network faults. An initial version of Aloe has been presented in IEEE INFOCOM 2019 [13]. The current version of the paper gives multiple additional features of Aloe, along with formal proofs of its characteristics, such as the convergence, the correctness and the closure of the self-stabilization algorithm used in Aloe micro-controller deployment. We additionally discuss various trade-offs for Aloe deployment and service provisioning performance, based on thorough experimentation and performance analysis under various realistic scenarios. Accordingly, we introduce a resource management module to Aloe, which boost up the performance under dynamic workload scenarios.

We have implemented a prototype of Aloe using state-of-the-art SDN control plane technologies and deployed the system over an in-house testbed and a 68-node Amazon web services platform. The in-house testbed consists of 10 nodes (Raspberry Pi devices) with Raspbian kernel version 8.0. As mentioned, we have utilized docker containers to host the distributed control plane platform. We have tested Aloe with three popular applications for in-network IoT data processing – (a) A web server (simple python based), (b) a distributed database server (Cassandra), and (c) a distributed file storage platform (Gluster). We observe that

Aloe can reduce the flow-setup delay significantly (more than three times) compared to state-of-the-art distributed control plane technologies while boosting up application performance even in the presence of multiple simultaneous faults.

The rest of the paper is organized as follows. § II discusses about the existing works related to our problem. The basic design and various components of Aloe are described in § III. § IV and § V discuss the design and implementation of the proposed Aloe orchestration framework. In § VI, we present the evaluation results from both in-house testbed and large-scale AWS cloud deployments. § VII discusses about various extensions of Aloe exclusive to this paper, where we provide a time series based prediction method for resource reservation of Aloe micro-controllers to ensure the quality of services (QoS). Finally we conclude the paper in § VIII.

II. RELATED WORK

SDN distributed control plane [14]–[17] provides scalability which is necessary for practical large scale IoT systems. ONIX [18] and ONOS [19] are two popular distributed control plane architectures. ONIX uses a distributed hash table (DHT) data store for storing volatile link state information. On the other hand, ONOS uses NoSQL distributed database and distributed registry to ensure data consistency. Although both of them can scale easily and show a significant amount of fault-resiliency, they require high end distributed computing infrastructure for execution. Deployment of such infrastructure increases the cost of IoT deployment and leads to performance degradation of IoT services. To tackle the high resource requirements, Elasticcon [20] uses controller resource pool to enforce load balancing. They also proposed a hand-off protocol for switch controller co-ordination to ensure the serializability. However, Elasticcon is not suitable for failure prone IoT nodes. Similar problem is also faced by Kandoo [21] and [22]. A Markov chain based formal model has been proposed in [23] to analyse the dependability of optical networks over double ring topology. However, maintaining a double ring topology in a dynamic IoT system is challenging. The fault-tolerant SDN control plane is discussed in [24]–[26], where they have used multiple replica controllers, which can take the place of a failed controller. However, the use of replica controllers provides fault-resilience based on the number of replica controllers. On the other hand, managing state consistency between the primary controllers and the slave controllers is difficult in an IoT network with a large amount of flows. To overcome these issues, Aloe proposes an auto-scalable, distributed, and fault-tolerant SDN control plane using “self-stabilizing” controller deployment.

IoT applications generate short and bursty traffics. To avoid the impacts of short flows, DIFANE [28] uses special purpose authority switches. The authority switches can take localized decisions based on pre-installed wildcard flow entries depending on the traffic characteristics and network topology. However, local authority switches create a problem for the global state management of the network. DevoFlow [29] tries to solve this problem by proactively deciding wild-carded rules based on global state information. However, the dynamic

TABLE I: Comparison of Existing Works

| Names | Key contributions | Issues | | | |
|----------------------------|-----------------------------------|---------------|-----------------|---------------------|-------------------|
| | | Plug and Play | In-band control | Short flow friendly | Arbitrary failure |
| ONIX [18], ONOS [19] | Distributed network state | N | Y | N | N |
| Elasticon [20] | Controller load balancing | N | N | Y | N |
| Kandoo [21] | Hierarchical control plane | N | N | N | Y |
| BLAC [27] | Controller scheduling | N | N | Y | N |
| DIFANE [28], DevoFlow [29] | Pro-active flow installation | N | Y | N | N |
| SCL [30], ASCA [26] | Replication based In-band control | N | Y | Y | N |
| RAMA [25] | Consistency preservation | N | N | N | Y |

TABLE II: Selected Abbreviations Used

| Abbreviation | Explanation |
|--------------|-------------------------------|
| SDC | Service deployment controller |
| SNC | Super network controller |
| P2NM | PushToNode Module |
| TMM | Topology Management Module |
| SDM | State Discovery Module |
| RMM | Resource Management Module |
| MIS | Maximal independent set |
| DHT | Distributed hash table |
| COTS | Commercially off the shelf |

topology of the IoT platform prevents the proactive installation of flow entries. Therefore, although DIFANE and DevoFlow perform well in the case of a data center network, it delivers substandard performance in the case of IoT platforms.

IoT requires in-band control plane as most of the switches have limited network interfaces. Therefore, disruption in IoT links impacts severely on multiple IoT nodes due to disconnection from the control plane. To provide disruption tolerance, SCL [30] uses replication of controller applications on strategic places of a network. SCL uses a coordination layer inside the switch to provide consistent updates for a single image, lightweight controllers deployed in an in-band fashion. However, the use of a two-phase commit mechanism for consistency preservation increases higher latency and affects the flow setup delay for the short flows. Moreover, SCL assumes the existence of robust channels among switches and controllers, which is not possible in the case of low-cost and resource-constrained IoT platforms. On the other hand, DIFANE, DevoFlow and SCL exploit the data plane device capabilities to provide quicker response time. In order to do so, they require special purpose switches that can take decisions locally without requiring controller consultation. Such switch-level modifications may not be possible for every hardware device. To avoid such scenarios, the proposed Aloe exploits in-network processing platforms of IoT to deploy lightweight controller instances to realize “control plane as service”.

To avoid hardware modification, BLAC [27] uses a controller scheduling mechanism to dynamically scale the control plane to accommodate the need of the system. BLAC introduces a scheduling layer to achieve binding less architecture, where all the flows from a switch can be dynamically scheduled to one of the many controller instances. Although, BLAC reduces the switch hand-off issues, increases the flow setup time. Therefore, BLAC re-introduces performance bottleneck for IoT short flows.

From the discussion above and from the comparison of existing works given in Table I we can observe that the states of the art control plane architectures are not effective in managing IoT platforms. The reason behind the non-compatibility is the existing works pour more focus on consistency of the network state information than the availability and partition tolerance of the control plane. However, theoretically it is not possible to achieve all these goals simultaneously [31]. However, we feel that availability and partition tolerance requires more attention than providing strong consistency for IoT platforms. The reason behind this is a dynamic and failure prone net-

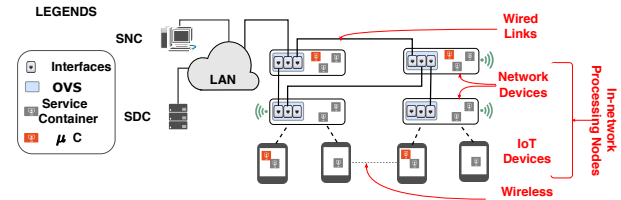


Fig. 1: Components of Aloe Infrastructure

work like IoT requires a highly available control plane than preserving consistency for volatile short flow information. The proposed Aloe ensures minimal flow initiation delay by using a self-stabilizing linear time convergent lightweight controller placement adjacent to the switches.

III. COMPONENTS OF ALOE

The Aloe orchestration framework exploits the capabilities of the in-network processing architecture over an IoT platform where devices work mostly in a plug-and-play mode. The main components of the architecture are shown in Fig. 1. It can be noted here that the proposed architecture does not bring new hardware or software platforms at its base; instead, we utilize the available COTS hardware and open-source software suites to design this entire architecture. Our objective is to design an orchestration platform that can be developed with market-available components while integrating innovations in the design such that the shortcomings of the existing systems can be mitigated. We discuss the individual components and their functionalities in this section.

A. Infrastructure Nodes

The networking equipment and devices are considered as the infrastructure nodes. Therefore, nodes are essentially embedded and resource-constraint devices like smart-gateways, smart routers, smart IoT monitoring devices, etc. These devices participate in communication and provide in-network processing platforms for lightweight services by utilizing residual resources. We consider that these nodes are either SDN-supported or can be configured with open-source software platforms like *open virtual switch* (OVS) to make them SDN capable.

We use containerized platforms like docker [12] to offload services in the IoT platform for in-network processing. The containerized service deployment helps in supporting service isolation and makes the architecture failsafe by supporting live migration of containers. Further, containers reduce a

programmer's overhead for service delegation and cost of deployment, as the same device can be used for in-network processing of IoT applications along with the execution of custom networking services.

B. Service Deployment Controller (SDC)

To identify the resource requirement and delegation of the services which require in-network processing, we use a centralized service deployment controller (SDC). The SDC periodically monitors the resource consumptions of the nodes. Once a new service is ready for deployment in the system, SDC identifies the schedules in which the nodes can execute the services without violating resource demands from individual services. Once the schedule is generated, the SDC is responsible for delegating the services based on the schedule. It can be noted that the load of an SDC is much less compared to the network management controller. Therefore we maintain a single instance of SDC in our system.

C. Super Network Controller (SNC)

Network management in an IoT platform is non-trivial due to the diversified inter-service communication requirements and the dynamic nature of the network. Aloe uses a two-layer approach. We deploy a high availability super network controller (SNC)¹ at the first layer, which is responsible for storing persistent network information, like routing protocols, QoS requirements, the periodicity of statistic collection from nodes, etc. A SNC also manages an access control list (ACL) to provide necessary security to the infrastructure nodes.

D. Micro-Controller (μC)

Although SDC(s) and SNC(s) are highly available, an IoT platform has a time-varying topology due to the use of the resource constraint devices and the devices being plug-and-play most of the time. Therefore, the use of a centralized controller cannot achieve fault-tolerance (failure of infrastructure nodes) and partition-tolerance (failure of network links resulting in network partitions). On the other hand, unlike SDC, SNC needs to be consulted by the nodes each time a new flow enters the system. This consultation overhead increases the communication overhead and flow initiation delay, which also affects the performance of the services deployed in the infrastructure. Therefore, Aloe uses a second layer of network controllers named as “micro-controllers” (μC).

μC s are lightweight SDN controllers. Each μC stores volatile link layer information of a small group of nodes placed topologically close to it. Thus a μC maintains information consistency by minimizing the delay between the governing μC and the nodes managed by it. The SNC can aggregate these statistics via REST API queries from the μC . Based on the changing QoS of the services, network service provisioning can be achieved in the μC via the same REST API. Based on the configuration of the SNC, a μC collects statistics from individual OVS modules of the nodes. Thus a μC can achieve a fine-tuned network control for the infrastructure nodes.

¹It is possible to use same physical device as SDC and SNC

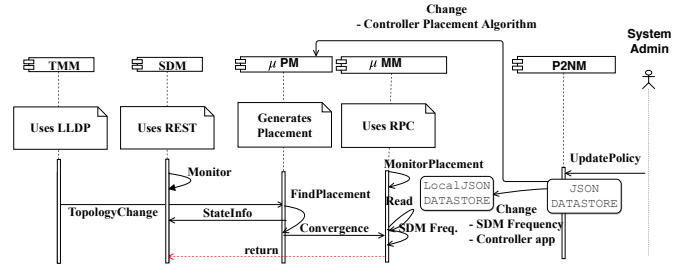


Fig. 2: Aloe function modules and their interactions

However, deployment of μC s in nodes might also create a network partitioning issue. To avoid such an undesirable scenario, Aloe uses a novel approach where the μC s are encapsulated inside a container and deployed as a service inside the infrastructure nodes itself. Thus Aloe supports μC as a Service ($\mu CaaS$), which ensures fault-tolerance of the system. μC containers can be migrated to a target node quite easily with the help of the live-migration technique of a container when the host node fails. Aloe ensures that a set of μC s is always live in the system, maintaining the requirements for minimized switch-controller delay. On the other hand, a μC container can be customized depending on the available capacity of the nodes and resource consumptions by the controller applications. It can be noted that this μC architecture is different from existing distributed SDN controller approaches, such as DevoFlow [29] and SCL [30], which require switch-level customization. μC s can run over the existing COTS devices without any requirement for switch-level modifications.

IV. DESIGN OF ALOE ORCHESTRATION FRAMEWORK

This section discusses the Aloe orchestration framework by highlighting various functional modules of Aloe and their working principles. Finally, we develop a set of APIs for language-independent and robust deployment of applications over the Aloe framework. The various functional modules of Aloe and list of acronyms used are shown in Fig. 2 and Table II respectively; the detailed description follows.

A. Aloe Functional Modules

The proposed framework consists of four node-level modules and one SNC-level module. The node-level modules run inside the infrastructure nodes and decide the topology and service parameters that need to be synchronized across various nodes. These modules collaborate with each other to take distributed decisions in a fault-tolerant way. It can be noted that in Aloe, infrastructure nodes are mutable and they can convert themselves as a μC if required. An interesting feature of Aloe is that this decision mechanism is executed in a pure distributed way, preserving the safety and liveness of the system in the presence of faults. The functionalities of various modules are as follows.

1) *Topology Management Module (TMM)*: We design Aloe as a plug-and-play service, where an Aloe-supported IoT device can be directly deployed in an existing system for flexible auto-scaling support. The TMM initializes the Aloe framework on a newly deployed node. The tasks of the TMM are as follows – (i) identify the nodes in the neighborhood, and (ii) determine whether an Aloe service is running in that node. An Aloe service is of two types – (a) μ C service, and (b) user application service. To find out the active nodes in the neighborhood, TMM uses *Link Layer Discovery Protocol* (LLDP) which is a standard practice for SDN controllers. We assume that each Aloe service deployed in the IoT cloud uses a unique predefined port address. TMM queries about the services in the local neighborhood via issuing a telnet open port requests. Apart from the initialization, this module is invoked on detecting a failure of a node/link or μ C.

2) *State Discovery Module (SDM)*: In case of a node or a link failure after the initialization through TMM, there is a possibility that the infrastructure nodes get disconnected from the μ C. To identify such a scenario, Aloe maintains various state variables for each node as follows. (i) *Controller State (CTLR)*: This state variable decides whether a node is in a general (does not host a μ C service), μ C (hosts a μ C) or undecided (an intermediate state between the general state and the μ C state) state. (ii) *Priority (PRIO)*: This state variable is required only if the node is undecided and denotes the priority of the node for becoming a μ C. The states associated to nodes are kept and managed by the nodes themselves. However, a node can access a copy of the states from its neighbor to decide its state. SDM is responsible for accumulating the state information collected from the neighbors. SDM uses REST for this purpose. Once a failure event occurs, TMM invokes the SDM. SDM keeps on executing periodically until the node finds at least one μ C in its neighborhood. The periodicity of the execution of this module is dependent on the link delay. For implementation purpose, we consider the periodicity as the largest delay observed to fetch data from a neighbor. The above functioning is different than control plane state discovery module which runs inside the μ C and keeps track of the network states. In contrast to that, the proposed SDM keeps track of the roles (i.e. acting as μ C or not) that a node is playing in its immediate neighborhood. The detailed state transitions are given in Section A

3) *μ C Placement Module (μ PM)*: Based on the neighbor states collected through SDM, every node independently determines whether it needs to launch a μ C service. This is done through the μ PM module. We consider the nodes as the vertices of a graph where the edges determined by the connectivity between two nodes, and place the μ C services to the nodes that form a *maximal independent set* (MIS) on that graph. An MIS based μ C placement ensures that there would be a μ C at least in one-hop distance from each node, which can take care of the configurations and flow-initiations for the application services running on that node. As we have claimed earlier and will show in § VI that the μ Cs utilized in Aloe are significantly light-weight but efficient for performing network and service management activities. Therefore the total overhead due to MIS based μ C placement is not significant.

For identification of a suitable set of μ C capable nodes, we develop a distributed randomized MIS algorithm given in Algorithm 1. The novelties of this algorithm are as follows. (1) **Randomized**: The algorithm selects different nodes at different rounds, ensuring that the load for μ C service hosting is distributed across the network and does not get concentrated on some selected nodes. (2) **Bounded set**: The number of deployed μ Cs are always bounded based on the total number of nodes in the network. (3) **Self-stabilized**: The algorithm is self-stabilized [32] and converges in linear time ensuring fault-tolerance of the system under single or multiple simultaneous faults until complete network partition occurs. The proofs of these properties are provided in the Section A.

Algorithm 1: μ PM Controller Placement Algorithm

Input: B : Any arbitrary large value greater than maximum degree of the network.

```

1 Function Trial():
   /* Breaks priority ties */
2   PRIO ← ⌊  $\frac{Rand()}{B}$  ⌋;
3   return;
4 Function Neighbor $\mu$ C():
5   if Another  $\mu$ C in one-hop neighborhood then
6     return true
7   else
8     return false
9 Function UMPriority():
10  /* If node has unique maximum priority */
11  if PRIO of this node > maximum PRIO in neighborhood then
12    return true
13  else if PRIO of this node = maximum PRIO in neighborhood then
14    return false
15  else
16    return None
17 Function Main():
18  while state change detected do
19    if CTLR=general & Neighbor $\mu$ C()=false then
20      /* No  $\mu$ C in neighborhood */
21      CTLR ← undecided;
22      Trial();
23      /* Initialize priority */
24    else if CTLR= $\mu$ C & Neighbor $\mu$ C()=true then
25      /* Two  $\mu$ Cs are adjacent */
26      CTLR ← general;
27    else if CTLR=undecided & Neighbor $\mu$ C()=true then
28      /*  $\mu$ C found in neighborhood */
29      CTLR ← general;
30    else
31      if UMPriority()=None then
32        /* Executor is not maximum */
33        continue;
34      else if UMPriority()=true then
35        /* Executor has unique maximum
36           priority, no need for further
37           trial. */
38        CTLR ←  $\mu$ C;
39      else
40        /* Executor has maximum but not
41           unique priority */
42        CTLR ← undecided;
43        /* Next round of trial starts */
44        Trial();
45  return

```

4) *μ C Manager Module (μ MM)*: Once a node decides its state through μ PM, the μ MM module initiates the μ C service on the selected nodes and establishes a controller-switch relationship between the μ C and the nodes with *general*

state in the one-hop neighborhood. As we mentioned earlier, a μC is initiated as a containerized service over the node designated for hosting a μC by the μPM algorithm. For a node with *general* state, this process may involve changing of controller services from one μC to another μC , which requires the reestablishment of the controller-switch relationship. For this purpose, the SDN flow tables need to be migrated from the old μC to the newly associated μC . The flow table migration mechanism is specific to the SDN controller software used, and will be discussed in §V.

5) *PushToNode Module (P2NM)*: Along with fault-tolerance, Aloe supports rapid deployment and runtime customization of the system. To implement this feature, we develop P2NM. Unlike the rest of the modules, P2NM is centralized and/or deployed in the SNC. It provides an interface for monitoring and changing the policy level information for the μC at runtime which is useful for system administrators. Aloe supported policy level information include (i) ACLs, (ii) controller application to be executed in the μC , (iii) routing protocols running in the μC , and (iv) SDM update frequency. Apart from the specified policies, Aloe also gives freedom to its user to customize the Aloe modules itself. This feature is achieved by developing a set of APIs as discussed next.

B. Application Programmer's Interfaces (API)

The primary objective of this orchestration framework is to deploy the *controller as a service* to the in-network processing infrastructure in the form of a μC . There are some significant differences between a user application service and a μC service, which makes the deployment of the later non-trivial. Unlike many user application services, performance of the management is dependent on the topological position of the μC services. A location transparent deployment of μC might allocate all the μC s in the same node if the node has sufficient resources. Such placement can degrade the network performance of the infrastructure. However, our placement algorithm is not an optimal solution. Therefore, during the design of Aloe, we consider the extendibility of this work. Many of the implemented functionalities of this framework can be reused as API for distributed controller application development. For ease of understanding, we only provide the python sample programs here. However, all the APIs can be invoked as *bash* shell commands over the SNC using P2NM.

1) *Topology Monitor*: Using this API, Aloe can detect a topology change event (*TopologyMonitor()*) and take actions accordingly. This API can also be used for general purpose routing application, as given in the following code.

```
1 ''' Find shortest path between dpidS and dpidD '''
2 G=TopologyMonitor()
3 path_dpid_list=FindShortestPath(G,"delay")
```

Listing 1: Topology Change Detector

2) *Distributed State Inspector*: We develop this API to observe the state of the nodes (*getNeighborStates()*), which helps in developing new placement algorithms for μPM . This API relies on a remote procedure call (*rpc*).

```
1 ''' Find max priority amongst neighbor '''
2 states=getNeighborStates()
```

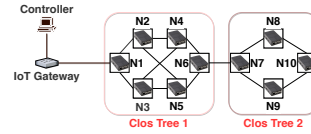


Fig. 3: Testbed:Topology



Fig. 4: Testbed:In Action

```
3 maxPrioUndecided=max([v["Prio"] for v in states.
                        values() if state["CTLR"]=="undecided"])
```

Listing 2: Distributed State Inspector

3) *Find Node Services*: The framework requires to identify the deployed services (*getNeighborServices()*) to enforce service level policies. We provide a python API to ease this task. The following example can be used for selective service blocking ACL.

```
1 '''Service blocking (ACL)'''
2 services=getNeighborServices()
3 bport=blocking_port
4 if (blocking_port in service["dpid"]):
5     Execute("ovs-ofctl add-flow match:src=dpid,
6            tcp_port=bport action:drop")
```

Listing 3: Find Node Services

Next, we discuss the details of the Aloe implementation as a general orchestration framework.

V. ALOE IMPLEMENTATION

We have implemented Aloe as a middleware over Linux kernel with the integration of open-source technologies, like docker containers, various SDN controllers, and REST based communication modules. The implementation details are described as follows.

A. Environmental Setup

We deploy an in-house testbed using the topology given in Fig. 3 for performance analysis of Aloe. The testbed follows clos tree based topology and spans across two different sites to resemble the topology given in [33]. The nodes in the testbed are Raspberry Pi version 3 Model B. The nodes are connected via multiple 100Mbps USB-to-Ethernet adapter-edges representing the physical Ethernet links among the nodes. Each links are configured to have 5Mbps of bandwidth and 100ms of propagation delay to match real life IoT deployment specification. Further, to analyze the scalability of Aloe, we have also deployed Aloe in a large-scale 68-node testbed using Amazon Web Services (AWS). For this purpose, we consider a sub-topology from *rocketfuel* [34] topology which consist 68 nodes. The nodes in the topology are deployed using 18 AWS *nano* instances (1 vCPU and 512 MB RAM) and 50 AWS *micro* instances (1 vCPU and 1 GB RAM). The AWS nodes are configured with Ubuntu 16.10 operating system with Debian kernel version 4.4.0. To emulate edges between the nodes, we use the *Layer 2 Tunneling Protocol* (l2tp) between the AWS instances. Every infrastructure node, both in the testbed and in the AWS, are configured with OVS.

B. Implementation Aspects

Here we discuss two important implementation aspects of Aloe – (i) flow-table consistency preservation during μC migration, and (ii) choice of controller service for μC implementation.

1) *Migration of μC and consistency preservation:* Due to changes in topology, node/link failure, network partition, or update in SNC policy, Aloe may require controller migration followed by a change in the node-controller association. We implement a `rpc` and `REST` based API (`changeCtrlr()`) which can dynamically change a switch’s allegiance from a source μC to target μC . This API can be invoked by either switch or participating μCs . `changeCtrlr()` forces the switch to invoke a “controller re-association request” to the target μC with its previous μC address. After receiving a “controller re-association request”, the target controller invokes migration of flow entries from the source μC . At the beginning of the migration procedure, Aloe preserves snapshots of the source μC flow table entries by sending `REST` queries to the source μC . To make the migration process lightweight, the container instance is not transferred from one node to another node; instead, the source μC container is terminated, and a new container is invoked at the target μC via `rpc`. In case of a network partitioning between the previous μC and the target μC , the target μC obtains the copy of flow table from the corresponding switch itself. In case of network partitioning between the source and the target μC , the target μC retrieves flow tables from the switches only. Since, during the migration procedure, new flow setup is deferred, the flow tables of the switches are unmodified. On the other hand, to tackle the inconsistency of neighbourhood information aroused due to the topology change, TMM updates the topology information eventually. In this way, the μMM preserves weak consistency of the system. Once the migration process is complete, the deferred flows are resumed. Due to the lower migration time, flow termination during the deferred period is negligible.

2) *Choice of controller service for μC :* The efficiency of Aloe is dependent on the efficiency of the choice of a controller service for the μC . Deployment of a heavy-weight controller can over-consume resources of the nodes; moreover, one μC is only responsible for managing a small set of nodes. Therefore, we target to opt for a light-weight μC for Aloe. In order to identify a suitable controller platform for μC , we compare a set of existing SDN controller services like “Open Day light (ODL)” [35], “ONOS” [19], “ryu” [36] and “Zero” [37] in our in-house testbed in terms of their resource utilization. Amongst these controllers, “ONOS” requires high

TABLE III: Wilcoxon Rank Sum Test (\uparrow indicates μC in top header consumes less resources, \leftarrow indicates μC in left header consumes less resources, **X** indicates the choice is undetermined)

| CPU | | | | |
|-----------------|------------|--------------|--------------|--------------|
| μC | No μC | Ryu | Zero | ODL |
| No μC | | \leftarrow | \leftarrow | \leftarrow |
| Ryu | < 0.0001 | | \uparrow | \leftarrow |
| Zero | < 0.0001 | < 0.0001 | | \leftarrow |
| ODL | < 0.0001 | < 0.0001 | < 0.0001 | |
| Memory | | | | |
| No μC | | \leftarrow | X | \leftarrow |
| Ryu | < 0.0001 | | X | \leftarrow |
| Zero | > 0.03 | > 0.01 | | \leftarrow |
| ODL | < 0.0001 | < 0.0001 | < 0.0001 | |
| CPU Temperature | | | | |
| No μC | | X | \leftarrow | \leftarrow |
| Ryu | > 0.39 | | \uparrow | \leftarrow |
| Zero | < 0.0001 | < 0.0001 | | \uparrow |
| ODL | < 0.0001 | < 0.0001 | < 0.0001 | |

memory consumption ($> 500MB$) which creates an instability in the docker environment. Further, we have observed that approximately 32% times, “ONOS” fails to execute over the testbed nodes due to unavailability of sufficient virtual memory. Therefore, we report the performance of the controllers other than “ONOS”. The performance is reported based on three major system parameters – CPU utilization, memory utilization and CPU temperature variation by generating 45 flows randomly in the system using Python `SimpleHTTPServer`. In Fig. 5a, we provide the comparison of the performance of the competing controllers in terms of CPU utilization. We observe that approximately 30% “ODL” μCs use more than 30% of the CPU utilization. In comparison to that, around 40% “Zero” μCs use 15% of the CPU utilization. In Fig. 5b, we observe that almost 80% “ODL” μCs use more than 600MB of memory space. All the other controllers show lower memory utilization. Fig. 5c shows the variation in CPU core temperature while executing different types of controller services. The consolidated pair-wise comparison of the controllers are provided in Table III (upper right triangle in blue color). The notation **X** signifies that the comparison cannot be obtained. On the other hand an upper arrow (\uparrow) suggests that the μC listed in the top header consumes less amount of resources. We use \leftarrow to denote the higher efficiency of the μC application mentioned in the left header. To ascertain the comparison, we perform a statistical hypothesis testing using non-parametric, one-tailed Wilcoxon rank sum test ($\alpha = 0.01$) [38]. Our alternative hypothesis assumes that the mean resource consumptions and the core temperature is higher than the one in the normal case. The left lower triangular part of Table III (given in green color) signifies the p -values obtained

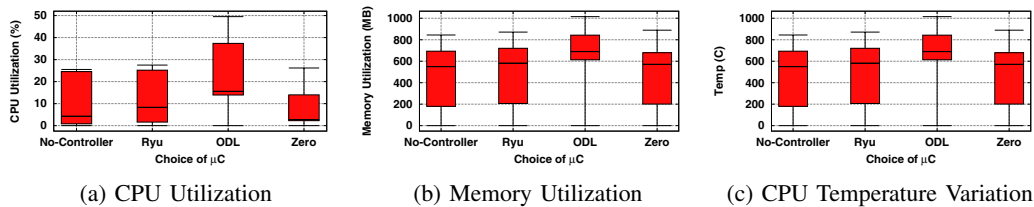


Fig. 5: Resource Utilization Comparison of Controller Applications

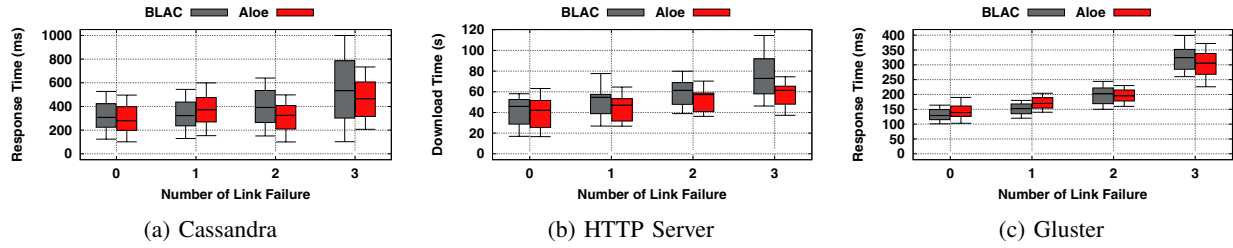


Fig. 6: Comparison of response time of services obtained from testbed: Average percentage improvement of Aloe – (a) HTTP server: 4% (0-fail), 11% (2-fails), 21% (3-fails), (b) Cassandra: 9% (0-fail), 26% (2-fails), 37% (3-fails), (c) Gluster: -8% (0-fail), 0.1% (2-fails), 6% (3-fails)

TABLE IV: Wilcoxon Rank Sum Test conclusions with p -values over response time of different applications:

X=Inconclusive, \checkmark =Aloe better, \bullet =In band better

| Services Failure | Cassandra | HTTP | Gluster |
|------------------|----------------------------|----------------------------|--------------------------|
| 0 | X (>0.11) | X (>0.20) | \bullet (<0.0001) |
| 1 | \checkmark (<0.0001) | X (>0.04) | \bullet (<0.0001) |
| 2 | \checkmark (<0.0001) | \checkmark (<0.0001) | X (>0.39) |
| 3 | \checkmark (<0.0001) | \checkmark (<0.0001) | \checkmark (<0.01) |

from the rank test. Our experimental results suggest that, “Zero” requires less CPU utilisation, less CPU temperature than almost all the competitors (except “ODL”). The memory utilisation of “Zero” is indistinguishable to that of “Ryu”. “Zero”’s micro-kernel architecture is responsible for the resource requirement benefits. Therefore, we use “Zero” as our choice of μ C in both testbed and AWS.

With this implementation, we evaluate the performance of Aloe, as discussed in the next section.

VI. EVALUATION

We have tested the performance of Aloe with three different categories of standard applications which are common and useful for an IoT based platform – (i) HTTP service (Python SimpleHTTPServer): used for bulk data transfer via web clients, (ii) distributed database service (Cassandra): for data-driven applications, and (iii) distributed file system service (Gluster): used for file sharing and fault-tolerant file replication over a distributed platform. We further compare the performance of Aloe with BLAC [27], a distributed SDN control platform. To emulate realistic fault models in the system, we have injected the faults using Netflix *Chaos Monkey* fault injection tool. We have taken the measurements under all possible link fault combinations² in the testbed and 100 different random fault combinations in AWS³.

A. Application Performance

Fig. 6b compares the download time of a 512MB file hosted using the HTTP service under the influence of both BLAC and Aloe over the in-house testbed. The results are obtained by varying all possible source-destination pairs in the topology. We observe that, even though Aloe results in higher download

time compared to BLAC when there is no failure in the system, the performance improves rapidly in the presence of link outage. While injecting failure, we observe that approximately 30% connections are timed-out while operating under the governance of the BLAC controller. However, Aloe reduces such flow termination⁴ ($< 5\%$ connection time-out for Aloe). To compare the differences of the nature of the results for each service, we performed a Wilcoxon rank sum test. The p -values and conclusion from the test is summarised in Table IV.

Fig. 6a shows the response time of Cassandra search queries. Here, we observe a significant difference in the characteristics of the plots due to the nature of the service. Unlike HTTP, Cassandra utilizes short flows to fetch query results. Therefore, we observe that Aloe provides a significant improvement in the query response time. However, in case of Gluster, Aloe performance is marginally poor compared to BLAC until there are 3 link failures (Fig. 6c). Gluster flows are short-distant flows, usually within one-hop. The flow-setup delay is almost negligible for a one-hop flow. Therefore the μ C deployment overhead of Aloe is more when the number of failures is less.

Similar behaviors are observed in the large-scale deployment of Aloe in the AWS cloud. In Figs. 7b and 7c, HTTP and Gluster response times show similar characteristics as observed in the testbed. In the case of Cassandra (Fig. 7a), all the cases perform significantly better than BLAC. From these observations, we conclude that Aloe performs significantly better for the services that generate long-distant mice flows (like database synchronization). For a long-distant flow, the flow setup delay is high, which gets further affected by link failures. As a consequence, Aloe performance is better for failure-prone systems, like IoT clouds, as the flow-setup delay gets increased with the recovery time due to a failure.

B. Dissecting Aloe

Aloe flow-setup time is dependent upon the convergence time of μ PM and the path restoration time. Fig. 8a shows the distribution of the average convergence time of Aloe in the presence of failure. We have an interesting observation here that as the number of simultaneous failures increases, the convergence time drops. This can be explained as follows. Let us consider two different faults. If the two faults are at two different sides of the network, then two waves of μ PM

²A node failure is equivalent to simultaneous failure of multiple links. Therefore, all possible link failure automatically covers node failure scenarios.

³All experiments are repeated 3 times

⁴Only in such cases, where the network is partitioned

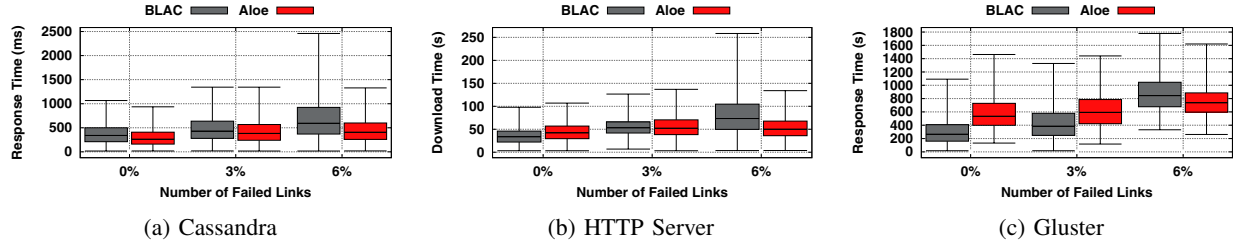


Fig. 7: Comparison of response time of services obtained from AWS cloud: Average percentage improvement of Aloe – (a) HTTP server: -2% (0-fail), 0.1% (2%-fails), 34% (6%-fails), (b) Cassandra: 20% (0-fail), 21% (2%-fails), 34% (6%-fails), (c) Gluster: -12% (0-fail), -6% (2%-fails), 14% (6%-fails)

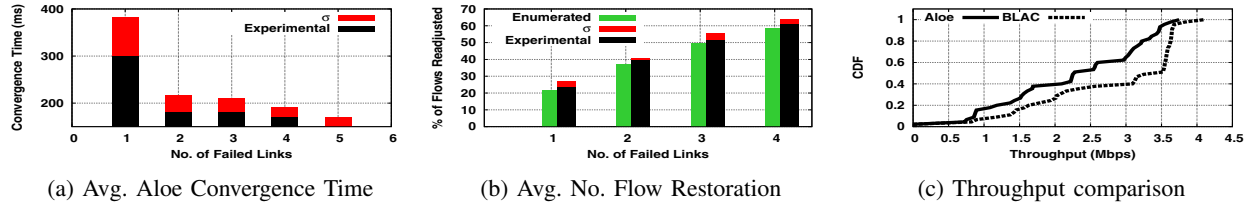


Fig. 8: Testbed: Effect of failure on Aloe performance (σ = standard deviation)

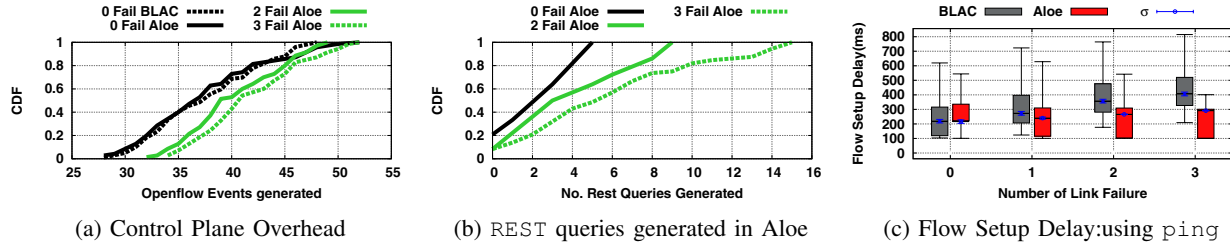


Fig. 9: Testbed: Comparison of Aloe overhead and Flow setup delay

starts executing simultaneously from two different ends of the network. These two waves get diffused in the network and meet in the middle of the network at convergence. That way, multiple faults create multiple such μ PM waves in the network in parallel, and as these individual waves need to deal with a smaller part of the network, they converge quickly.

The convergence phase is followed by the path restoration due to a change of the controller positions in case of a failure. To identify the performance of the path restoration, we measure the average number of flow adjustments done by the framework. The number of flow adjustment depends on the topology and the number of flows passing through the failed links. Therefore, to compare the result, we provide the enumerated number of flow adjustment required for all possible cases of link failures in Fig. 8b. We observe that the experimental observations closely match with the results obtained by enumeration. Further, these metrics have a direct impact on the flow-setup delay. To understand the effect of these factors, we compare the flow-setup delay for BLAC control plane and Aloe. To identify the flow-setup delay, we use ping to transfer a single ICMP packet. Fig. 9c shows that although Aloe marginally increases the flow setup-delay in the absence of a failure, it provides quick flow-setup when multiple faults occur in the network.

We observe that the overhead of distributed μ C in Aloe is responsible for the increase in the flow-setup delay during

the no-failure scenario. However, it is difficult to compare the exact overhead of the BLAC control plane and Aloe due to the differences in the nature of the overhead. We measure the overhead with respect to two different factors. Fig. 9a shows the comparison between BLAC and Aloe regarding the number of openflow events generated over a period of 100s. Aloe additionally generates REST queries to support inter-controller communication, therefore it has more number of openflow events compared to BLAC. Fig. 9b depicts the number of REST queries generated in Aloe. During the failure events, Aloe μ C may need to migrate from one node to another. Fig. 10c shows the data transfer overhead required for migration, which is in the order of a few KB. As the number of nodes in the IoT environment are increased, the number of flow table entries are also increased. Therefore, the transfer size per migration also increases when the number of nodes are increased. The size of the flow table entries also increases with more number of failures in the network, which introduces some of redundant flow entries (“zombie flows”). However, we observe that, the effect of redundant flows has marginal effect when the number of nodes in the system are significantly high. Due to these overheads, Aloe incurs higher communication overhead than the BLAC control plane. However, due to the significant reduction in flow-setup time, Aloe ensures better flow throughput than BLAC, as shown in Fig. 8c.

Although Aloe incurs communication overhead, Aloe en-

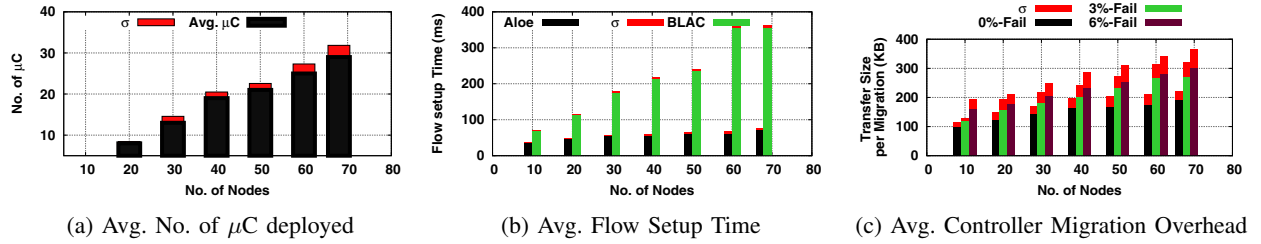


Fig. 10: Effect of Scaling Aloe μ C Deployments (σ = standard deviation)

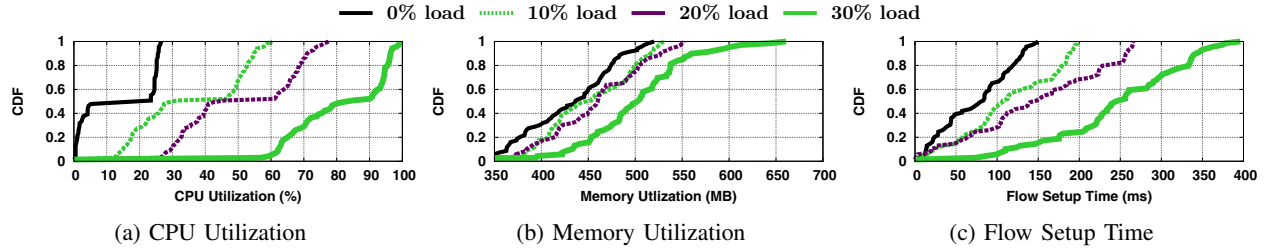


Fig. 11: Effect of Application Workload of the Host Devices on Aloe Performance

sures a significant drop in the average flow-setup delay. To limit the flow-setup delay, Aloe provides elastic auto-scaling by increasing the number of μ C instances to guarantee that each node can find a μ C in its neighborhood. Fig. 10a shows the average number of μ C instances when the network scales, as obtained from the AWS implementation. The effect of elastic auto-scaling is shown in Fig. 10b⁵ which indicates that the flow-setup delay only increases marginally in comparison to the BLAC controller, which incurs a significantly high flow-setup delay as the number of nodes in the network increases.

VII. ALOE PERFORMANCE OPTIMIZATION

Aloe μ Cs are deployed over existing network infrastructure which may have their own workloads due to the application services deployed in them; we call them as the application workloads of the devices. We perform a pilot study to check the impact of application workload of the devices on Aloe performance. We use the same AWS cloud-based deployment of Aloe as discussed earlier. Fig. 11 shows the impact of application workload on Aloe performance. To increase the application workload of the devices, we use “stress-ng” [39] tool. During the experiments, memory and CPU utilization of the devices have been increased from 0% to 30% of the actual capacity. When the application workload of the devices are increased, the system resources get over-utilized due to thrashing. Therefore, the system performance reduces aggressively as the μ C receives less CPU time. Additionally more swap events are generated which increases the flow setup time, as we observe from Fig. 11c. These observations confirms that, Aloe μ Cs require special attention in terms of resource reservation for severely loaded systems. We accordingly develop a resource management framework for Aloe, which is discussed next.

⁵Each experiments are repeated atleast 10 times

A. Effect of Resource Reservation

Reserving resources for μ C applications ensures QoS in terms of flow setup time. To optimize the performance of the system, the resource reservation must match the resource demand of the μ C. However, the resource demands of a μ C at a particular time depends on the amount of flows managed by that μ C. Therefore, we assume that the resource demand of a μ C follows a temporal pattern and depends on the network state of the IoT infrastructure. Although over-provisioning resources to the μ C improves the QoS, it might affect the primary workload of the devices; therefore, it can have negative impact on the overall application performance. Consequently, we implement a *Resource Management Module* (RMM) based on *Monitor-Forecast-Adapt* strategy which gets executed in each μ C to balance the μ C resource demands and the primary workloads of the devices. Fig. 12a shows the components of Aloe RMM which consists of three sub-modules. a) *Resource Monitor* periodically collects the usage statistics of the μ C and stores it in a JSON data-store. b) *Usage Estimator* periodically analyzes the time-series of the resource usage pattern of the μ C and predicts the probable resource demand for the next time period. c) *Resource Enforcer* is responsible for actually resource reservation for the μ C based on the predicted resource demand.

1) *Prediction of μ C Resource Demands*: For prediction of resource demands, it is important to identify the distributions of the resources which depend on the flow arrival pattern. However, in practice, it is difficult to estimate the flow arrival distribution for a large scale IoT platform with heterogeneous applications executing in it. Therefore, we choose a forecasting model based on the characteristics of the IoT applications. We focus on two basic characteristics of the IoT applications. (i) IoT applications generate bursty and short lived flows [6]. The bursty and short living nature of IoT flows reveal that, the flow arrival rates per μ C during a discrete time interval is cyclic. (ii) These characteristics also suggest that, the flow arrival

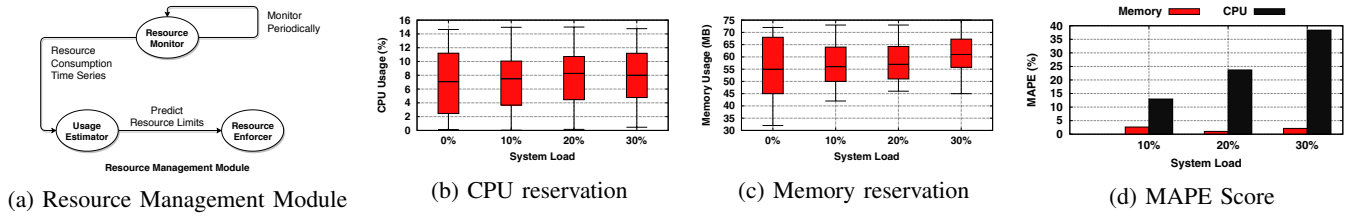


Fig. 12: Resource Reservation for Aloe μ Cs

rates follow a non-stationary property. Therefore, we use *Autoregressive Integrated Moving Average* (ARIMA) [40] model for forecasting of individual resource requirements. ARIMA relies on the mean reversion principle of non-stationary data to forecast the future strategy based on the time series by employing autoregression.

2) *Performance Improvement with Aloe RMM*: We have integrated the RMM module with Aloe and tested it over the AWS platform as discussed earlier. Like many statistical modeling methods, identification of parameters for ARIMA is a non-trivial challenge. Therefore, we use auto-ARIMA [40] based on our experimental observations to individually forecast the CPU and memory demands according to the resource utilization time series. Fig. 12b shows the amount of CPU reserved for the μ C in various load scenarios remains almost constant. Fig. 12c shows the memory reservation of the μ C application due to resource reservation module. From the results we can observe that, the memory reservation amount increases in case of 30% load. The reason behind this observation lies in the OVS to μ C mapping technique used in Aloe μ MM. An OVS chooses a μ C based on how quickly the μ C responds to its join request. As the system load increases, a lightly loaded μ C is more likely to provide a quick response time. Therefore, the lightly loaded μ Cs are likely to get connected with more switches with high number of flows passing through those switches, which in turn increases the memory overhead of those μ C. Next we check how accurate the RMM prediction model can perform based on the mean average percentage error (MAPE). Fig. 12d reveals that, the proposed RMM provides significantly low MAPE for prediction of the memory. On the other hand, higher MAPE was observed for CPU utilization prediction. The reason behind this observation is fluctuation of CPU is very frequent which the underlying ARIMA can not predict always. Therefore, the performance of the IoT applications are also influenced. Figs. 13a and 13b compare the resource utilization between the μ C and IoT application in terms of CPU and memory. From Fig. 13a, we observe that the accuracy of RMM does not significantly affect the performance of memory utilization by the IoT applications. However, the reduced accuracy of used ARIMA sometimes over provisions more CPU time to the μ Cs. As a result the IoT application receives less CPU time than its demand in such cases.

Interestingly, the IoT application (like GlusterFS) shares more host resources than the μ C; therefore, a slight resource biasing towards the μ Cs improve their performance significantly, while having marginal impact on the performance of the IoT application. We present this observation in Fig. 14a,

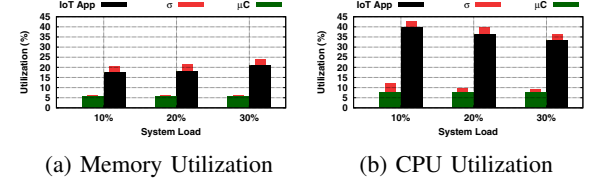


Fig. 13: Effects of Resource Reservation (σ = standard deviation)

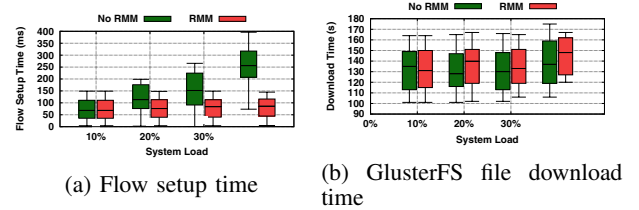


Fig. 14: Effects of Resource Reservation for Aloe μ Cs

where we compare the performance of the RMM in terms of flow setup time with that of no-RMM. The results justifies that the RMM ensures low variations in flow setup time as opposed to the no-RMM case. In fact use of RMM can significantly improve the performance of IoT short flows by reducing average flow set-up time by 13% – 120% in various load scenarios. However, due to the resource reservation of μ C, the application may suffer due to insufficient resources. To understand this effect, we compare the performance of the GlusterFS application before and after implementing the RMM in Fig. 14b. We find that, the increase in mean download time due to effects of RMM while downloading a 25MB file using GlusterFS varies between 2% – 7% for different load conditions which is considerably small.

VIII. CONCLUSION

In this paper, we present Aloe, an orchestration framework, for IoT in-network processing infrastructures. Aloe uses docker container to support lightweight migration capable in-band controllers. This design choice helps Aloe to provide elastic auto-scaling while keeping flow setup time under control. Aloe provides controller as a service to exploit in-network processing infrastructure and supports fault and partition tolerance. The performance of Aloe has been tested thoroughly and compared with a very recent orchestration framework (BLAC). The results indicate a significant improvement in response times for distributed IoT services. The low-level system properties of the Aloe framework can be validated through formal modeling of the Aloe functional algorithms, which can also provide the formal guarantee of the system performance. We keep this as a future direction of this work.

ACKNOWLEDGMENT

This work was supported by Science and Engineering Research Board (SERB) Early Career Research Award, File number: ECR/2017/000121 Dated 18/07/2017, funded by Department of Science and Technology, Government of India.

APPENDIX

The core characteristics of Aloe depend upon the proposed MIS algorithm (Algorithm 1) in μ PM. Here we analyze the theoretical properties of the MIS algorithm used in Aloe. For this purpose, we denote the topology of the in-network processing infrastructure as a undirected and unweighted graph $G = (V, E)$, where V represents the set of in-network processing nodes and E represents the communication links between them. We define a node v as the neighbor of another node u if there exists a communication link between u and v . The state of a node u is defined as the a variable $CTLR_u$ and is denoted as $CTLR_u$. As per Algorithm 1, $CTLR_u$ can take one of the following values – *general*, *undecided* and μC . A node goes through a state transition based on the state of its neighborhood and the value of the $PRIO$ variable of the closed neighborhood. For the ease of reference, we present the state transitions of a single node using Fig. 15. Based on the the individual state of the nodes, we define the term “*global state*” to represent the overall state of the platform. The global state of the platform is defined as $S = (CTLR_u : \forall u \in V)$ is an ordered tuple of length $|V|$. We define a “*legitimate state*” as any particular global state among the all possible global states, where no further statement can be applied at any node.

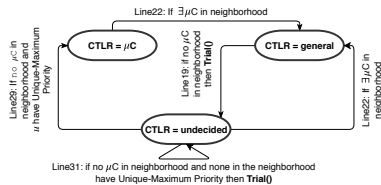


Fig. 15: State transition diagram of a node u

As mentioned earlier, we claim that the proposed algorithm is self-stabilizing. A proof of self-stabilization requires the proof of **Closure property** and **Convergence property**. The closure property of a system states that, the system will remain in a legitimate state if the system was in legitimate state and there is no perturbation in the system. On the other hand, if the convergence property is ensured by the system, the system will reach to a legitimate state in the absence of further perturbation.

A. Aloe μ PM Ensures Closure Property

Theorem 1. *If any node in the IoT platform is in Undecided state then the platform executing Algorithm 1 is not in a legitimate state.*

Proof: Let us assume that, $CTLR_u = \text{undecided}$ and the platform is in a legitimate state. In that case, only the following scenarios may occur.

Case-1 If $\exists v : CTLR_v = \mu C$ and v is in the neighborhood of u . In that case, $\text{Neighbor}\mu C()$ returns *true* and Line 24 will be applicable.

Case-2 $\nexists v : CTLR_v = \mu C$ for all v in the neighborhood of u . Let Q_u denotes the set of all v such that, each v is a neighbor of u and in Undecided state.

Case-2.i If u has unique maximum priority (i.e. $\text{UMPriority}() = \text{true}$) or $Q_u = \emptyset$, u executes Line 29.

Case-2.ii If u has maximum priority but not unique (i.e. $\text{UMPriority}() = \text{false}$) then, u executes Line 31.

Case-2.iii If u does not have maximum priority (i.e. $\text{UMPriority}() = \text{None}$) then, $\exists v$ which has/have maximum priority and it/they will execute either Line 29 or Line 31.

All these cases mentioned above contradicts the assumption that, the platform is in a legitimate state. Therefore, we can claim that, Algorithm 1 can not be in a legitimate state when at least one of the nodes is in the Undecided state. ■

Theorem 1 also proves that, if the platform is in a legitimate state, all the nodes are in either in the μC state or in the *general* state. As the system can not execute any statement in a legitimate state, then the following statements holds.

Corollary 1.1. (Closure property) *If no failure occurs, an IoT platform in “legitimate state” will remain in a legitimate state forever.*

The significance of legitimate state can be described by the theorem as follows.

Theorem 2. *If the platform is in a legitimate state, the set of μC s form a maximal independent set (MIS) of G .*

Proof: To prove this statement, we use proof by contradiction. We assume that the platform is in a legitimate state and the μC s do not form a MIS. There can be only two cases possible as per the definition of MIS.

Case-1 The set of μC does not hold the independence relationship. Therefore, $\exists u, v$ such that they are neighbor and both of them are in μC state simultaneously. In this case, Line 24 is applicable.

Case-2 The set of μC does not hold the maximality relationship. Therefore, $\exists u$ having *general* state can become μC without affecting other μC s. In this case, atleast one of the nodes can execute any one of statements given in Lines 19, 29 and 31.

Both of the cases mentioned above violates the the legitimate state assumption of the platform. ■

Theorems 1 and 2 proves that, Aloe is stable and consistent when there is no failure in the platform and after reaching legitimate state all μC s form a MIS of the original IoT platform. Thus, the flow setup delay is minimum as each switch has one μC in its vicinity.

B. Aloe Convergence after Each Failure

As a legitimate state can not accommodate any node in undecided state, the platform can only reach in a legitimate state once all the nodes in undecided state have changed their states by taking suitable transitions. So, in the following

theorem, we prove that traces of the state changes possible in individual nodes.

Theorem 3. *Only the following sequence or subsequence of state change is possible for each node during the execution of the algorithm, if no further failure occurs in between.*
 $(Undecided \rightarrow general \rightarrow Undecided \rightarrow general)$
 $(Undecided \rightarrow general \rightarrow Undecided \rightarrow \mu C),$
 $(\mu C \rightarrow general \rightarrow Undecided \rightarrow general),$
 $(\mu C \rightarrow general \rightarrow Undecided \rightarrow \mu C)$

Proof. As per Fig. 15, if a node u initially had μC or $general$ state, then it requires at most 2 or 1 state change to reach $Undecided$ state, respectively. On the other hand, if a node u executes Line 29, it will not execute any other rule unless a failure occurs; and no other nodes in the neighborhood of u can become μC after that. Therefore, if u executes Line 29, its neighbors can only execute Line 22. These properties ensure the statement of Theorem 3. \square

However, Theorem 3 does not ensure a bound on the number of statement execution required for a node to reach $general$ or μC from $undecided$ state. We prove the expected bound based on the distribution of the event of finding unique maximum value of $PRIOR$ variable in the close neighborhood.

Theorem 4. *If N is number of nodes in a closed neighborhood of any u , then $P(N, B)$ denote the probability of finding a unique maximum in the closed neighborhood of u . Then*

$$P(N, B) = \frac{(N \times \sum_{i=1}^B i^{(N-1)})}{(B+1)^N}$$

Proof. Let i be the highest priority in a configuration S after a round, where each round corresponds to the event of generating the priority by at most each nodes in the closed neighborhood of u . The unique maximum property suggests that, only one node has $PRIOR = i$ and the rest of nodes can have $0 < PRIOR \leq i - 1$. The value of i can vary from 1 to B . For a fixed i , the rest of $PRIOR$ values of the nodes can be arranged in $N \times i^{(N-1)}$ different ways. Hence the total probability calculated for a $UMPriority$ generation event can be expressed as follows.

$$P(N, B) = \frac{(N \times \sum_{i=1}^B i^{(N-1)})}{(B+1)^N}$$

\square

If all nodes in the closed neighborhood of u are in $Undecided$ state, all of the N nodes are executing Line 29 or Line 31 as there is no μC adjacent to them. To find the expected number of rounds for one of the intermediate node to become μC , we have to find the expected number of rounds in which there will be only one node with maximum priority in the neighborhood.

Theorem 5. *If X denote the random variable indicating the number of rounds required to find a unique maximum priority in the closed neighborhood of u then $E[X] \leq e$.*

Proof. The random variable and the distribution function can be represented as follows.

$$Pr[X = r] = P(N, B)(1 - P(N, B))^{(r-1)}$$

The expected number of rounds can be calculated as Eq. (1).

$$E[X] = \frac{1}{P(N, B)} = \frac{(B+1)^N}{N \sum_{i=1}^B i^{N-1}} \quad (1)$$

As the value of N is upper bounded by $(B+1)$, Eq. (1) can be expressed as follows.

$$E[X] \leq \frac{(B+1)^{B+1}}{(B+1) \int_0^B i^B di} = \frac{(B+1)^{(B+1)}}{B^{(B+1)}} = (1 + B^{-1})^{(B+1)}$$

The upper bound of $E[X]$ is attained at $B \rightarrow \infty$ and the value is calculated in Eq. (2).

$$\lim_{B \rightarrow \infty} (1 + B^{-1})^{(B+1)} = e \quad (2)$$

Therefore, $E[X] \leq e$. This result signifies that a node and its neighbors require expected “ e ” statement execution by each node for generation of unique maximum priority generation event. \square

Theorems 3 to 5 proves that, each node is expected to execute $2+e$ statements to reach a non $undecided$ state from any arbitrary state. We can also conclude that, the cumulative number statement execution of the entire platform to arrive at a legitimate state from any arbitrary state is expected to be $\mathcal{O}(|V|)$. Hence, the proposed algorithm is linear in terms of execution complexity.

REFERENCES

- [1] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmoly, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, pp. 14–76, 2015.
- [2] A. C. Baktir, A. Ozgovde, and C. Ersoy, “How can edge computing benefit from software-defined networking: a survey, use cases, and future directions,” *IEEE Communication Surveys & Tutorials*, pp. 2359–2391, 2017.
- [3] L. M. Vaquero and L. Rodero-Merino, “Finding your way in the fog: Towards a comprehensive definition of fog computing,” *ACM SIGCOMM Computer Communication Review*, pp. 27–32, 2014.
- [4] M. Chiang, S. Ha, I. Chih-Lin, F. Risso, and T. Zhang, “Clarifying fog computing and networking: 10 questions and answers,” *IEEE Communication Magazine*, pp. 18–20, 2017.
- [5] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, “Practical service placement approach for microservices architecture,” in *Proceedings of the 17th IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGRID)*, 2017, pp. 401–410.
- [6] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, “Large-Scale Measurement and Characterization of Cellular Machine-to-Machine Traffic,” *IEEE/ACM Transaction on Networking*, pp. 1960–1973, 2013.
- [7] O. Kaiwartya, A. H. Abdullah, Y. Cao, J. Lloret, S. Kumar, R. R. Shah, M. Prasad, and S. Prakash, “Virtualization in wireless sensor networks: fault tolerant embedding for internet of things,” *IEEE Internet of Things Journal*, pp. 571–580, 2018.
- [8] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo, “SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for wireless sensor networks,” in *Proceedings of the 34th IEEE International Conference on Computer Communications (INFOCOM)*, 2015, pp. 513–521.
- [9] F. Tang, Z. M. Fadlullah, B. Mao, and N. Kato, “An intelligent traffic load prediction based adaptive channel assignment algorithm in SDN-IoT: A deep learning approach,” *IEEE Internet of Things Journal*, 2018.
- [10] D. Wu, D. I. Arkhipov, E. Asmare, Z. Qin, and J. A. McCann, “UbiFlow: Mobility management in urban-scale software defined IoT,” in *proceedings of the 34th IEEE International Conference on Computer Communications (INFOCOM)*, 2015, pp. 208–216.
- [11] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli, “Large-scale dynamic controller placement,” *IEEE Transactions on Network and Service Management*, pp. 63–76, 2017.

- [12] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC)*, 2017, p. 11.
- [13] S. Chattopadhyay, S. Chatterjee, S. Nandi, and S. Chakraborty, "Aloe: An elastic auto-scaled and self-stabilized orchestration framework for iot applications," in *Proceedings of the 39th International Conference on Computer Communications (INFOCOM)*, April 2019.
- [14] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed multi-domain sdn controllers," in *Proceedings of 12th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–4.
- [15] M. A. Santos, B. A. Nunes, K. Obraczka, T. Turletti, B. T. De Oliveira, and C. B. Margi, "Decentralizing SDN's control plane," in *Proceedings of 39th IEEE Conference on Local Computer Networks (LCN)*. IEEE, 2014, pp. 402–405.
- [16] F. Bannour, S. Souihi, and A. Mellouk, "Distributed sdn control: Survey, taxonomy, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 1, pp. 333–354, 2018.
- [17] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the 2010 Internet Network Management Workshop/Workshop on Research on Enterprise Networking*. Berkeley, CA, USA: USENIX Association, 2010, pp. 3–3.
- [18] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2010, pp. 1–6.
- [19] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *et al.*, "Onos: towards an open, distributed sdn os," in *Proceedings of the 3rd Workshop on Hot topics in Software Defined Networking (HotSDN)*. ACM, 2014, pp. 1–6.
- [20] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elastic: an elastic distributed sdn controller," in *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, 2014, pp. 17–28.
- [21] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the 1st Workshop on Hot topics in Software Defined Networking (HotSDN)*, 2012, pp. 19–24.
- [22] B. Grkemli, S. Tatcolu, A. M. Tekalp, S. Civanlar, and E. Lokman, "Dynamic control plane for sdn at scale," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 12, pp. 2688–2701, Dec 2018.
- [23] U. Siddique, K. A. Hoque, and T. T. Johnson, "Formal specification and dependability analysis of optical communication networks," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017, 2017, pp. 1564–1569.
- [24] L. Sidki, Y. Ben-Shimol, and A. Sadovski, "Fault tolerant mechanisms for sdn controllers," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2016, pp. 173–178.
- [25] A. Mantas and F. M. V. Ramos, "Rama: Controller fault tolerance in software-defined networking made practical," *CoRR*, vol. abs/1902.01669, 2019. [Online]. Available: <http://arxiv.org/abs/1902.01669>
- [26] T. Hu, Z. Guo, J. Zhang, and J. Lan, "Adaptive slave controller assignment for fault-tolerant control plane in software-defined networking," in *2018 IEEE International Conference on Communications (ICC)*, 2018, pp. 1–6.
- [27] V. Huang, Q. Fu, G. Chen, E. Wen, and J. Hart, "BLAC: A Bindingless Architecture for Distributed SDN Controllers," in *Proceedings of 42nd IEEE Conference on Local Computer Networks (LCN)*, 2017, pp. 146–154.
- [28] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, pp. 351–362, 2010.
- [29] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM Computer Communication Review*. ACM, 2011, pp. 254–265.
- [30] A. Panda, W. Zheng, X. Hu, A. Krishnamurthy, and S. Shenker, "SCL: simplifying distributed SDN control planes," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2017, pp. 329–345.
- [31] A. Panda, C. Scott, A. Ghodsi, T. Koponen, and S. Shenker, "CAP for networks," in *Proceedings of the 2nd Workshop on Hot topics in Software Defined Networking (HotSDN)*. ACM, 2013, pp. 91–96.
- [32] S. Chattopadhyay, N. Sett, S. Nandi, and S. Chakraborty, "Flipper: Fault-tolerant distributed network management and control," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017, pp. 421–427.
- [33] "Cisco ACI Multi-POD and Multi-Site Demystified," May 2019, [Online; accessed 10. May. 2019]. [Online]. Available: <https://netdesignarena.com/index.php/2018/07/01/cisco-aci-multi-pod-and-multi-site-demystified/>
- [34] "UW CSE | Systems Research | Rocketfuel," May 2005, [Online; accessed 28. Jul. 2018]. [Online]. Available: <https://research.cs.washington.edu/networking/rocketfuel>
- [35] (2018) OpenDaylight. [Online]. Available: <http://www.opendaylight.org/>
- [36] "Ryu 4.30 documentation," May 2019, [Online; accessed 10. May. 2019]. [Online]. Available: <https://ryu.readthedocs.io/en/latest/>
- [37] T. Kohler, F. Drr, and K. Rothermel, "Zerosdn: A highly flexible and modular architecture for full-range distribution of event-based network control," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1207–1221, Dec 2018.
- [38] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, pp. 80–83, 1945.
- [39] "stress-ng - a tool to load and stress a computer system," Jun 2018, [Online; accessed 28. Jul. 2018]. [Online]. Available: <http://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html>
- [40] J. G. De Gooijer and R. J. Hyndman, "25 years of time series forecasting," *International Journal of Forecasting*, vol. 3, no. 22, pp. 443–473, 2006.



Subhrendu Chattopadhyay received the B.Tech. degree from West Bengal University of Technology, Kolkata, India, and the M.Tech. degree from Indian Institute of Technology Guwahati, Assam, India. He is currently pursuing the Ph.D. degree at Indian Institute of Technology Guwahati, Assam, India. He has received a Research Fellowship from TATA Consultancy Services, India. His research interests include the area of computer networking, distributed systems, and social network analysis



Soumyajit Chatterjee joined IIT Kharagpur in 2017, as a research scholar (Doctorate Program). He received his M.Tech in Computer Science from IIT (ISM), Dhanbad in the year 2016 and B.E. from University Institute of Technology, The University of Burdwan in 2012. He also has an industry experience of one year seven months. Currently, his domain of research is mobile systems and ubiquitous computing.



Sukumar Nandi received the Ph.D. degree from Indian Institute of Technology Kharagpur, India. Currently, he is a Professor with Indian Institute of Technology Guwahati, India. His research interests include traffic engineering, wireless networks, network security, and distributed computing. He is a Senior Member of IEEE and ACM, a Fellow of the Institution of Engineers (India), and a Fellow of the Institution of Electronics and Telecommunication Engineers (India).



Sandip Chakraborty received the Ph.D. degree from Indian Institute of Technology Guwahati, Assam, India, in 2014. Currently, he is an Assistant Professor with Indian Institute of Technology Kharagpur, Kharagpur, India. His research interests include wireless networks, mobile computing, and distributed computing. He is a Member of IEEE and the Association for Computing Machinery