Aloe: An Elastic Auto-Scaled and Self-stabilized Orchestration Framework for IoT Applications

Subhrendu Chattopadhyay*, Soumyajit Chatterjee†, Sukumar Nandi‡, Sandip Chakraborty§
*‡ Department of CSE, IIT Guwahati, India
†§ Department of CSE, IIT Kharagpur, India

***Abstract*—Management of networked Internet of Things (IoT) infrastructure with in-network processing capabilities is becoming increasingly difficult due to the volatility of the system with low-cost resource-constraint devices. Traditional software-defined networking (SDN) based management systems are not suitable to handle the plug and play nature of such systems. Therefore, in this paper, we propose Aloe, an elastically auto-scalable SDN orchestration framework. Instead of using service grade SDN controller applications, Aloe uses multiple lightweight controller instances to exploit the capabilities of in-network processing infrastructure. The proposed framework ensures the availability and significant reduction in flow-setup delay by deploying instances near the resource constraint IoT devices dynamically. Aloe supports fault-tolerance and can recover from network partitioning by employing self-stabilizing placement of migration capable controller instances. 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 testbed 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 state of the art orchestration framework.**

***Index Terms*—Network architecture, Fault-tolerance, Programmable network, software defined network**

I. Introduction

The rapid proliferation of Internet-of-Things (IoT) has made the network architecture complicated and difficult to manage for service provisioning and ensuring security to the end-users. Simultaneously, with the advancement of edge-computing, in-network processing (also known as fog computing) and platform-as-a-service technologies, end-users consider the network as a service platform for the deployment and execution of myriads of diverse applications dynamically and seamlessly over the network. Consequently, network management is becoming 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) 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 [**?**].

The primary requirements for supporting edge and in-network processing over a networked IoT platform are as follows: (1) The platform should be agile to support rapid deployment of applications without incurring additional overhead for in-network processing [**?**]. This also ensures scalability of the system [**?**]. (2) Many times, in-network processing requires dividing a service into multiple microservices and deploying the microservices at different network nodes for reducing the application response time with parallel computations [**?**]. However, such microservices 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 [**?**]. This also escalates the requirement for reducing flow-setup delay in the network. (4) Failure rates of IoT nodes are in-general high [**?**]. Therefore, the system should support a fault-tolerant or fault-resilient architecture to ensure liveness.

Although SDN supported edge computing and in-network processing have been widely studied in the literature for the last few years [**?**], [**?**] as a promising technology to solve many of the network management problems associated with large-scale IoT networks, they have certain limitations. First of all, 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 the switch and the controller increases, which affects the flow-setup performance. As we mentioned earlier that the majority of the flows in an IoT network are short-lived flows, 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 [**?**]. 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 a plug-and-play deployment of devices. Dynamic controller deployment requires hosting the controller software over commercially-off-the-shelf (COTS) devices and designing methodologies for

Email:subhrendu@iitg.ac.in*,soumyachat@iitkgp.ac.in†,sukumar@iitg.ac.in‡, sandipc@cse.iitkgp.ac.in§

This work is partially supported by TCS India, Microsoft India, LRN Foundation and CNeRG IIT Kharagpur.

controller coordination which is a challenging task [**?**]. 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.

In contrast to the existing architectures, the novelty of this paper is as follows. We integrate an 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. This has multiple advantages for an 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. To achieve these goals, we first design a distributed, robust, migration-capable and elastically scalable control plane framework with the help of docker containers [**?**] 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 sink 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 coordinations. 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[1], a distributed, robust, elastically auto-scalable, platform-independent orchestration framework for edge and in-network processing over IoT infrastructures.

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.

[1]Aloevera: A plant known for its healing property.

II. Related Work

Traditional single controller architecture is not suitable for IoT infrastructure, where the network is dynamic and failure prone. One way to address such a problem is to deploy a distributed control plane. However, existing distributed control planes are not efficient for handling large-scale IoT systems that require in-band control. ONIX [**?**] and ONOS [**?**] 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 which rely on short flows due to the increase in the number of controller consultation. On the other hand, IoT devices require in-band control as most of them have limited network interfaces. Therefore, a disruption in IoT link can have a severe impact on multiple IoT nodes due to disconnection from the control plane. Similar problems can be observed in case of Kandoo [**?**] and Elasticon [**?**].

To avoid these challenges DIFANE [**?**], DevoFlow [**?**] and BLAC [**?**] design control plane by utilizing data plane services. DIFANE and DevoFlow use special purpose switches which can take decisions in its local neighborhood in the absence of the controller. However, this requires switch-level modifications which may not be possible for every hardware components. On the other hand, BLAC uses a controller scheduling mechanism to dynamically scale the control plane to accommodate the need the system. However, BLAC increases the flow setup time and not suitable for real-time IoT applications.

In SCL [**?**], the authors have developed a coordination layer which can provide consistent updates for a single image, lightweight controllers deployed in an in-band fashion. However, SCL uses two-phase commit mechanism for consistency preservation, which incurs high latency. Moreover, SCL is not suitable for IoT services as it assumes the existence of robust channels among switch and controllers, which is not suitable for low-cost and resource-constrained networked IoT systems.

III. Components of Aloe

The Aloe orchestration framework exploits the capabilities of the in-network processing architecture over an IoT based 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

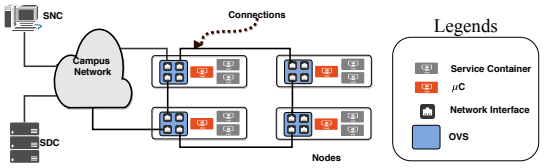


Fig. 1: Components of Infrastructure

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 platform like *open virtual switch* (OVS) to make them SDN capable.

We use containerized platforms like docker [**?**] 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 execution of custom networking services.

B. Service Deployment Controller

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 services can be executed by the nodes 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

Network management in an IoT system 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) at the first layer, which is responsible for storing persistent network information, like routing protocols, quality of service (QoS) requirements, the periodicity of statistic collection from nodes, etc. An SNC also manages an access control list (ACL) to provide necessary security to the infrastructure nodes.

D. Micro-Controllers

Although super controllers are highly available, an IoT infrastructure has a time-varying topology due to the use of the resource constraint devices and the devices being plug-and-play most of the times. 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 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).

μCs are lightweight SDN controllers. A 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.

However, deployment of μCs in nodes might also create network partitioning issue. To avoid such an undesirable scenario, Aloe uses a novel approach where the μCs are encapsulated inside a container and deployed as a service inside the infrastructure nodes itself. Thus Aloe supports *μC as a Service* (μCaaS) 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 μCs 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 [**?**] and SCL [**?**], which require switch-level customization. μCs 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 are shown in Fig. 2; the detailed description follows.

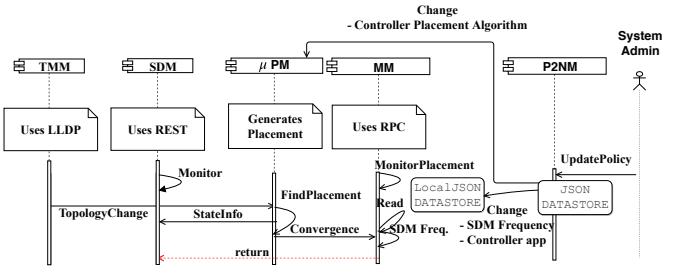


Fig. 2: Aloe function modules and their interactions

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 TIM 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) μC service, and (b) user application service. To find out the active nodes in the neighborhood, TMM uses *Link Layer Discovery Protocol* (LLDP). We assume that each Aloe service deployed in the IoT cloud uses a unique predefined port address. TIM queries about the services in the local neighborhood via issuing a `telnet open port` requests. Apart from the initialization, this module is invoked whenever a node/link failure or μC failure event is detected.

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 μC. 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 μC service), `μC` (hosts a μC) or `undecided` (an intermediate state between the general state and the μC 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 μC. 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 μC 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.

3) μC Placement Module (μPM): Based on the neighbor states collected through SDM, every node independently determines whether it needs to launch a μC service. This is done through the μPM module. We consider the nodes as the vertices of a graph where the edges determined by the connectivity between two nodes, and place the μC services to the nodes that form a *maximal independent set* (MIS) on that graph. An MIS based μC placement ensures that there would be a μC 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 μCs utilized in Aloe are significantly light-weight but efficient for performing network and service management activities. Therefore the total overhead due to MIS based μC placement is not significant. For identification of a suitable set of μC 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 μC service hosting is distributed across the network and does not get concentrated on some selected nodes. (2) **Bounded set**: The number of deployed μCs are always bounded based on the total number of nodes in the network. (3) **Self-stabilized**: The algorithm is self-stabilized and converges in linear time (the proof is omitted due to space-constraint), ensuring fault-tolerance of the system under single or multiple simultaneous faults until complete network partition occurs. Intuitively, we can see that; a node cannot retain its status as `undecided` forever. In a neighborhood, if it finds another μC then it changes to `general` (Line 24), otherwise, it competes with other nodes having `undecided` status in a series of tiebreaker rounds by choosing a random `PRIO` value (Line 25). Until one of the nodes receives a unique maximum priority in the neighborhood and gains `μC` status (Line 29), the random trials continue (Line 31). It can be shown that the expected number of such trials for a node is less than thrice the number of competing nodes (proof omitted due to space constraint). Therefore, the Algorithm 1 always converges in linear time.

4) μC Manager Module (μMM): Once a node decides its state through μPM, the μMM module initiates the μC service on the selected nodes and establishes a controller-switch relationship between the μC 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

Algorithm 1: μPM Controller Placement Algorithm

```
1  Function Trial():
2      PRIO← Rand();
       ;                          // Breaks priority ties
3      return;
4  Function NeighborμC():
5      if Another μC in one-hop neighborhood then
           ;                      // If μC in neighborhood
6          return true
7      else
8          return false
9  Function UMPriority:
       /* If node has unique maximum priority     */
10     if PRIO of this node > maximum PRIO in neighborhood then
11         return true
12     else if PRIO of this node = maximum PRIO in neighborhood then
13         return false
14     else
15         return None
16 Function Main():
17     while state change detected do
18         if CTLR=general & NeighborμC()=false then
               // No μC in neighborhood
19             CTLR← undecided;
20             Trial();
               // Initialize priority
21         else if CTLR=μC & NeighborμC()=true then
               /* Two μCs are adjacent                */
22             CTLR← general;
23         else if CTLR=undecided & NeighborμC()=true then
               /* μC found in neighborhood            */
24             CTLR← general;
25         else
26             if UMPriority()=None then
                   /* Executor is not maximum         */
27                 continue;
28             else if UMPriority()=true then
                   /* Unique maximum priority.        */
29                 CTLR← μC;
30             else
                   /* Maximum but not unique priority
                      */
31                 CTLR← undecided;
                   /* Next round of trial starts      */
32                 Trial();
33     return
```

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 therefore, we discuss it 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 Application Programmer's Interfaces (API) 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 user application services, μC services should not act location transparently. A location transparent deployment of μC might allocate all the μCs in the same node, which can degrade the network performance of the infrastructure. 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 a `script` over the SNC using P2NM.

1) Topology Monitor: Using this python 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.

```
''' Find shortest path between dpidS and dpidD'''
import networkx as nx
G=TopologyMonitor()
path_dpid_list=nx.algorithms.shortest_path(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`).

```
''' Find max priority amongst neighbor'''
import json
states=getNeighborStates()
maxPrioUndecided=max([v["Prio"] for v in states.
    values() if state["CTLR"]=="undecided"])
```

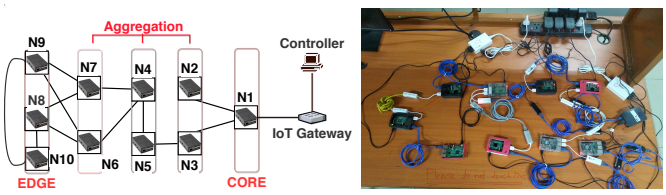
Listing 2: Distributed State Inspector

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.

```
'''Service blocking (ACL)'''
import json,os
services=getNeighborServices()
bport=blocking_port
if(blocking_port in service["dpid"]):
 os.system("ovs-ofctl add-flow match:src=dpid,
    tcp_port=bport action:drop")
```

Listing 3: Find Node Services

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



(a) Topology (b) In Action

Fig. 3: Testbed

TABLE I: Wilcoxon Rank Sum Test (↑indicates μC in top header consumes less resources, ←indicates μC in left header consumes less resources, **X** indicates the choice is undetermined)

CPU				
μC	**No μC**	**Ryu**	**Zero**	**ODL**
No μC		←	←	←
Ryu	< 0.0001		←	←
Zero	< 0.0001	< 0.0001		←
ODL	< 0.0001	< 0.0001	< 0.0001	
Memory				
No μC		←	**X**	←
Ryu	< 0.0001		**X**	←
Zero	> 0.03	> 0.01		←
ODL	< 0.0001	< 0.0001	< 0.0001	
CPU Temperature				
No μC		**X**	←	←
Ryu	> 0.39		←	←
Zero	< 0.0001	< 0.0001		↑
ODL	< 0.0001	< 0.0001	< 0.0001	

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. We first discuss the implementation environment that we utilized, followed by a brief description of two different implementation aspects.

A. Environmental Setup

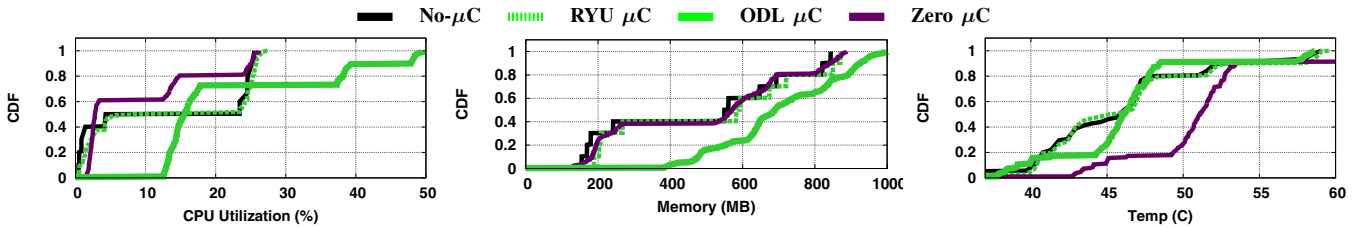
To analyze the performance of Aloe, we have deployed an in-house testbed using the topology given in Fig. 3a (Fig. 3b shows the live-snapshot of the testbed). The nodes in the testbed are Raspberry Pi version 3 Model B, which are configured with Raspbian 8.0 operating system with kernel version 4.4.50-v7+. The nodes are connected via multiple 100Mbps USB-to-Ethernet adapter-edges representing the physical Ethernet links among the nodes. We use Linux `tc` to configure each link to use 5Mbps of bandwidth and added $100ms$ 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` [**?**] 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 *open virtual switch* (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. Here, we discuss these two aspects in details.

1) Migration of μC and consistency preservation: A change in a policy level parameter requires a migration of the flow tables from the old μC instances to the new instances of the μC. Similarly, after a μPM execution, there might be a need for change in node-controller association. To implement such functionality, we have implemented a `rpc` and `REST` based API (`changeCtlr()`) which can dynamically change a switch's allegiance towards a μC. `changeCtlr()` forces the node 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 previously assigned μC. During the migration procedure, it is important to keep track of the previous state informations. To ensure the consistency, Aloe preserves snapshots of the μC flow table entries by sending `REST` queries to the μCs before the migration process starts. To make the migration process lightweight, the container instance is not transferred from one node to another node; instead, the source node container is killed, and a new container is invoked at the destination node 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 requester node itself. In this way, the μMM preserves weak consistency in the system.

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)`" [**?**], "`ONOS`" [**?**], "`ryu`" and "`Zero`" in our in-house testbed in terms of theirs resource utilization. Amongst these controllers, "`ONOS`" requires high 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. In Fig. 4a, we provide the comparison of the performance of the competing



(a) CPU Utilization (b) Memory Utilization (c) CPU Temperature Variation

Fig. 4: Resource Utilization Comparison of Controller Applications

TABLE II: Wilcoxon Rank Sum Test conclusions with p-values over response time of different applications:
X=Inconclusive, ✓=*Aloe* better, •=In band better

Services / Failure	Cassandra	HTTP	Gluster
0	**X (>0.11)**	**X (>0.20)**	**•(<0.0001)**
1	**✓(<0.0001)**	**X (>0.04)**	**•(<0.0001)**
2	**✓(<0.0001)**	**✓(<0.0001)**	**X (>0.39)**
3	**✓(<0.0001)**	**✓(<0.0001)**	**✓(<0.01)**

controllers in terms of CPU utilization. The plot suggests 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. 4b, we observe that almost 80% "ODL" μCs use more than $600MB$ of memory space. All the other controllers show lower memory utilization. Fig. 4c 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 I (upper right triangle in blue color). The notation **X** signifies that the comparison cannot be obtained. On the other hand ↑suggests that the μC listed in the top header consumes less amount of resources. We use ←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 ($\alpha = 0.01$) [**?**]. Our alternative hypothesis assumes that the mean resource consumptions and the core temperature will be higher than the one in the normal case. The left lower triangular part of Table I (given in green color) signifies the p-values obtained from the rank test. Based on our observation from Table I, we choose "Zero" as our choice of μC in 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 a networked IoT based system – (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 [**?**], 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 fault combinations in the testbed and 100 different random fault combinations in AWS.

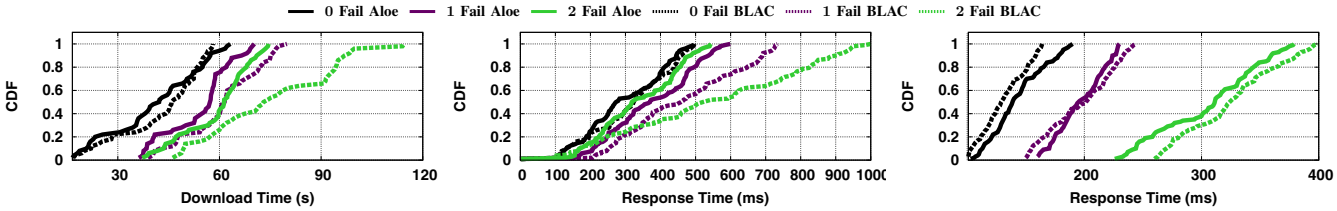
A. Application Performance

Fig. 5a 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 termination[2] ($< 5\%$ 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 II.

Fig. 5b 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. 5c). `Gluster` flows are short-distant flows, usually within one-hop. The flow-setup delay is almost negligible for a one-hop flow. Therefore the μC deployment overhead of Aloe is more when the number of failures is less.

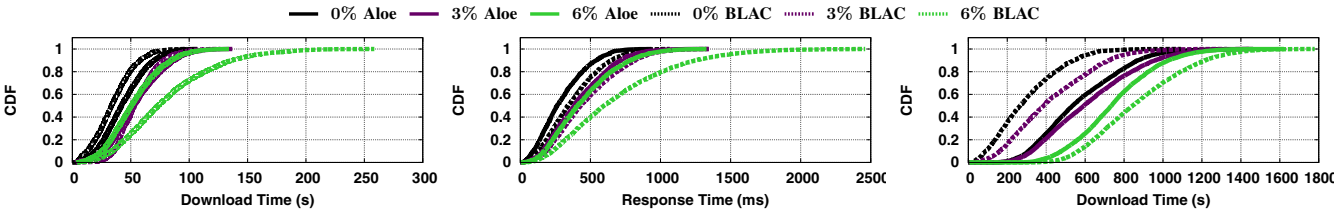
Similar behaviors are observed in the large-scale deployment of Aloe in the AWS cloud. In Figs. 6a and 6c, `HTTP` and `Gluster` response times show similar characteristics as observed in the testbed. In the case of `Cassandra` (Fig. 6b), all the cases perform significantly better than BLAC. From these 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

[2]Only in such cases, where the network is partitioned



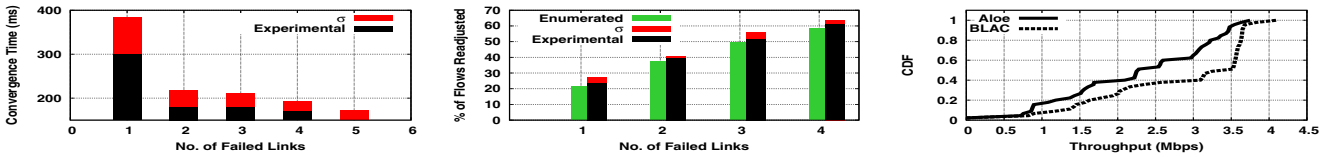
(a) HTTP Server (b) Cassandra (c) Gluster

Fig. 5: 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)



(a) HTTP Server (b) Cassandra (c) Gluster

Fig. 6: 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)



(a) Avg. Aloe Convergence Time (b) Avg. No. Flow Restoration (c) Throughput comparison

Fig. 7: Testbed: Effect of failure on Aloe performance (σ= standard deviation)

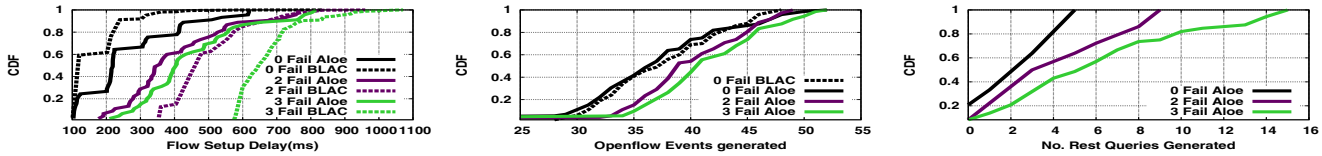
link failures. As a consequence, Aloe performance is better for failure-prone systems, like IoT clouds, as the flow-setup delay gets increased with the recovery time due to a failure.

B. Dissecting Aloe

Aloe flow-setup time is dependent upon the convergence time of μPM and the path restoration time. Fig. 7a 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 μPM 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 μPM waves in the network in parallel, and as these individual waves need to deal with a smaller part of the network, they converge quickly.

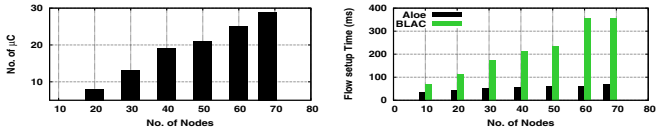
The convergence phase is followed by the path restoration due to a change of the controller positions in case of a failure. To identify the performance of the path restoration, we measure the average number of flow adjustments done by the framework. The number of flow adjustment depends on the topology and the locations of the failed links. Therefore, to compare the result, we provide the enumerated number of flow adjustment required for all possible cases of failures in Fig. 7b. 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. 8a 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 observed that the overhead of distributed μC 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 difference of the nature the overhead. We measure the overhead regarding two different factors. Fig. 8b shows the comparison between BLAC and Aloe regarding the number of `openflow events` generated over a period of 100s. However, Aloe additionally generates `REST` queries to support



(a) Flow Setup Delay:Measured using `ping` (b) Control Plane Overhead (c) `REST` queries generated in Aloe

Fig. 8: Testbed: Flow setup delay and overhead comparison



(a) No. of μC (b) Avg. Flow Setup Time

Fig. 9: AWS: Effect of Scaling

inter-controller communication. Fig. 8c depicts the number of `REST` queries generated in Aloe. Due to these overheads, the 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. 7c.

Although Aloe incurs communication overhead, Aloe ensures 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 μC instances to guarantee that each node can find a μC in its neighborhood. Fig. 9a shows the average number of μC instances when the network scales, as obtained from the AWS implementation. The effect of elastic auto-scaling is shown in Fig. 9b 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. 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 using two real testbeds and compared with a very recent orchestration framework (BLAC). The results indicate a significant improvement in response times for distributed IoT services.