DisProTrack: Distributed Provenance Tracking over Serverless Applications

Utkalika Satapathy*, Rishabh Thakur*, Subhrendu Chattopadhyay†, Sandip Chakraborty*
*IIT Kharagpur, Kharagpur, India 721302 †IDRBT, Hyderabad, India 500057

***Abstract*—Provenance tracking has been widely used in the recent literature to debug system vulnerabilities and find the root causes behind faults, errors, or crashes over a running system. However, the existing approaches primarily developed graph-based models for provenance tracking over monolithic applications running directly over the operating system kernel. In contrast, the modern DevOps-based service-oriented architecture relies on distributed platforms, like serverless computing that uses container-based sandboxing over the kernel. Provenance tracking over such a distributed micro-service architecture is challenging, as the application and system logs are generated asynchronously and follow heterogeneous nomenclature and logging formats. This paper develops a novel approach to combining system and micro-services logs together to generate a Universal Provenance Graph (UPG) that can be used for provenance tracking over serverless architecture. We develop a Loadable Kernel Module (LKM) for runtime unit identification over the logs by intercepting the system calls with the help from the control flow graphs over the static application binaries. Finally, we design a regular expression-based log optimization method for reverse query parsing over the generated UPG. A thorough evaluation of the proposed UPG model with different benchmarked serverless applications shows the system's effectiveness.**

***Index Terms*—Distributed provenance tracking,**

I. Introduction

Modern service-oriented architecture adopts DevOps [1], [2] practices and technologies to provide Software as a service (SaaS) [3] by leveraging distributed cloud infrastructure. Service deployment on top of the cloud widely adopts serverless computing (SLC) [4] to reduce operational expenditure whenever the service computations are stateless, elastic, and possibly distributed. Micro-services deployed on top of SLC architecture provide an abstraction of the underlying infrastructure where the developer can write, deploy and execute the code without configuring and managing the shared environment [4]–[7]. However, the available serverless-specific industry solutions [8]–[10] provide limited support for error reporting, execution tracing, and provenance tracking. Consequently, developers can only provide little attention to log vital forensic information. Some of the third-party observability tools [11]–[14] support distributed tracing as well as cost analysis features by instrumenting the source codes. However, these tools only support applications developed using a particular programming language. So, it is difficult to analyze the actual behavior of these micro-services.

Provenance Graphs: The non-invasive [1] frameworks rely on the logs generated by applications to identify the execution states. Since most of the production-grade micro-services are chosen from an available stable release, the executable files already contain meaningful log messages that can be used to identify event handling loops. In the domain of system security, provenance data is the metadata of a process that records the details of the origin and the history of modification or transformation that happened over time throughout its lifecycle [15]–[20]. The graph generated from this information is called the *provenance graph* of a process which is a causal graph that stores the dependencies between system subjects (e.g., processes) and system objects (e.g., files, network sockets). For example, an application event may generate a separate application log, error log, and operating system calls specific log, etc. In this context, a provenance graph is a directed acyclic graph (DAG) where individual log entries are the nodes, and the edges represent the causality relationship between the log entries. During attack investigation, an administrator queries this graph to find out the root cause and ramifications of an event. While a malicious entity performs some illegal events, the corresponding system logs are recorded as the provenance data. For example, when a compromised process tries to open a sensitive file, the OS level provenance can record the file-open activity, which can be referred for vulnerability analysis whenever an attack is detected in the system [21], [22]. Real-world enterprises widely use kernel or OS level information (logs) to perform provenance analysis to monitor their systems and identify the malicious events performed back in time.

A. Limitations of Existing Works and the Research Challenges

There have been several works that attempted to generate the provenance graphs by combining system and application logs together [16]–[20], [23]. However, these works primarily considered a monolithic application running directly over the Kernel, and thus combining the system log with the application log is not difficult. In contrast, the individual log files for each micro-service over an SLC application are physically distributed across the entire eco-system, which makes the generation of the provenance graph non-trivial. In such a scenario, a Universal Provenance Graph (UPG) that combines the interactions among all the micro-services can provide a meaningful platform for distributed provenance tracking. A few existing works [20], [24] have tried to address the issue of encoding all forensically-relevant causal dependencies regardless of their origin. Nevertheless, we have some non-trivial

[1]The non-invasive tools neither inject any piece of code inside the micro-service/container instances nor injects any special services into the SLC platform.

challenges in constructing a UPG for an SLC application.

• Challenge (1) – *Combining application logs from different micro-services:* Different micro-services generate separate logs that vary across the format for log messages, naming of the events and process descriptors, timestamp formatting, etc. Combining these logs towards generating the UPG is non-trivial as they may result in confounding nodes and edges in the graph.

• Challenge (2) – *Combining the system log with the application logs:* The next challenge comes in terms of combining the application logs with the system log (kernel `audit log`). The first issue is that the container-based sandboxing uses a common process identifier (`pid`) space; thus, mapping the kernel process logs with the micro-services event logs is not straight-forward, particularly when a micro-service run a multi-threaded process.

• Challenge (3) – *Identification of execution units:* As different micro-services may come from different production endpoints, there exists heterogeneity in terms of their implementation standards. Thus it is non-trivial to identify the functions or execution units that generate a specific entry in the log. In the same context, it is also difficult to identify the event handling loops, as the loops might produce asynchronous log entries. This problem magnifies in the case of SLC as the micro-services are deployed in different sandboxed environments.

• Challenge (4) – *Dependency explosion and handling confounding root causes:* To find out the root cause behind an event, the system administrator needs to execute a reverse query on the UPG. The results obtained from the reverse query can be multiple due to partial matching of the input query string. This results in more than one root cause behind a query event. Further, due to plausible circular dependencies among the micro-services, the resultant UPG might not be a DAG. Therefore, it is non-trivial to track all the causal paths behind an event.

Owing to the above challenges, in this paper, we develop a provenance tracing system, `DisProTrack` that generates a UPG which helps in root cause analysis of an event running on serverless by combining the system provenance with application's container logs. The core idea behind `DisProTrack` is to judicially use the applications' control flow graph (CFG) to avoid the runtime log tracking complexities.

B. Our Contributions

In contrast to the existing works, our contributions in this paper are as follows.

1. Design of the UPG from application and system logs: We implement a static analyzer module that generates the application-specific *Log Message String - Control Flow Graph* (LMS-CFG) from the application binaries. The LMS-CFG provides a profile of the application. We provide a novel approach for constructing a UPG from application logs and system logs using the LMS-CFG profiles for different application micro-services.

2. Runtime execution unit identification: We develop a Linux LKM (loadable kernel module) which can intercept the system calls generated during execution time to identify the semantic relationship between the system logs and the application logs. Furthermore, we propose a heuristic to segregate execution units which are challenging in a distributed system. Our proposed heuristic identifies the system calls (syscalls) to mark the application's event handling loops by tracing back the application binaries. These event handling loops can be refereed during the runtime partitioning of execution units across the micro-services.

3. Utilization of Regular Expression to improve search efficacy: Instead of storing the raw log messages in the UPG, we propose conversion and storage of an equivalent regular expression. This method improves the matching accuracy of log messages during the investigation phase and reduces the runtime search complexity by providing a faster response time. This method also reduces dependency explosion by decreasing the number of nodes in the generated UPG.

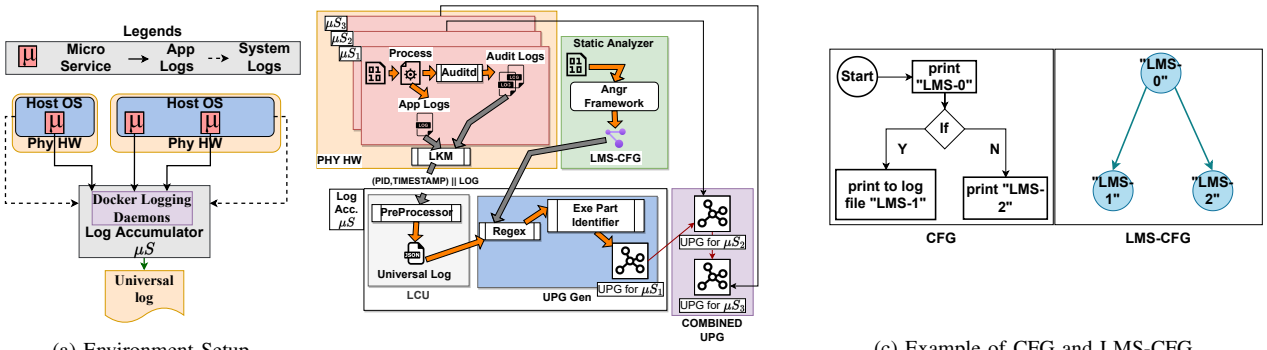
4. Implementation and evaluation: We have implemented the proposed framework, which can be deployed as a micro-service on top of the SLC without instrumenting the source code of the applications. We have made the implementation open-sourced[2]. Based on the experimental evaluation of `DisProTrack` with several benchmarked SLC applications, we found that the proposed method works well for identifying adversary activities. The framework has a minimal memory footprint (in the order of KB) and responds within $20s$-$30s$. The efficiency and efficacy of `DisProTrack` have also been tested with a proof-of-concept SLC application scenario.

II. Related Work

Existing third party log collection tools like, `FUSE` [25], `LSM` [26], `CamFlow` [27], `SPADE` [28], etc., allow hooking the kernel level objects and system calls; however, these methods require instrumentation of the OS. Additionally, the collection of system-level provenance in a containerized or serverless environment is difficult due to the micro-services' distributed and minimalistic deployed nature. Therefore, the existing threat detection and investigations with system-level provenance graph using kernel audit-log data, such as, `ETW` [29], `SLEUTH` [30], `Pagoda` [31], `POIROT` [32], `HOLMES` [33], `WATSON` [34], etc., do not suit an SLC environment. One common challenge for causality analysis with system provenance graph is the *"Dependency Explosion"* problem [35], [36] where too many *"root causes"* are behind one suspicious event. This dependency explosion increases the probability of *"security alert fatigue"* and *"missing threats"*.

`BEEP` [36] and `OMEGALOG` [16] have addressed dependency explosion problem by increasing input and output edge identification accuracy in the provenance graph. Other methods

[2]https://anonymous.4open.science/r/Project_ALV_2022-CEFD/ (Accessed: February 1, 2023)



(a) Environment Setup

(b) DisProTrack Execution Flow

(c) Example of CFG and LMS-CFG

Fig. 1: DisProTrack Overview

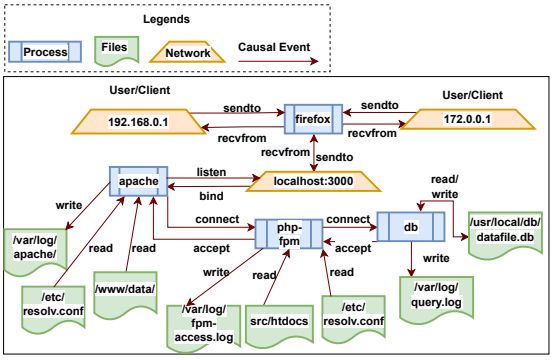


Fig. 2: An Example of System Provenance

like ETW [29], LogGC [37], MPI [35], ProTracer [38], etc., used execution partitioning to reduce the dependency explosion. However, most of these works require some instrumentation over the application source code. On the other hand, ALCHEMIST [20] and OMEGALOG used a combination of application-specific logs with system-level logs to track the information flow more accurately. In the same line, UIScope [39] proposed an instrumentation free, execution partition-based causality analysis for attack investigation system. However, the existing works are targeted toward monolithic systems and can not be deployed to secure SLC applications. Although, a few existing tools like, X-trace [40] and PivotTracing [41] are used to instrument SLC applications, they do not provide non-invasive property. Similarly, [11]–[14] are reliant on the used programming languages of the micro-service applications; thus lack flexibility [24]. The absence of language-independent, non-invasive causality analysis frameworks for SLC has motivated us to design DisProTrack.

III. DisProTrack Overview

The proposed DisProTrack is an open source provenance tracking system for containerized SLC as shown in Fig. 1a. DisProTrack accumulates the system and application logs of all the micro-service containers (μS) to generate the underlying directed edge-labeled provenance graph. The nodes of the provenance graph represent the system artifacts, for example, processes, socket connections, files, etc. The edges represent the causal dependencies between the nodes. Each edge is labeled with the generated system call (syscall) events (e.g. *read*, *write*, *connect*, *exec*, etc.) as shown in Fig. 2. The accumulated logs are merged together to create a *"universal log"* which is further used to identify the nodes of the provenance graph. Additionally, the framework also analyzes the CFG of the individual applications to understand the event handling loops before the deployment. So, the provenance graph generation requires two phases and, depending on that, DisProTrack is sub-divided into two major components; (a) *static analyzer* and (b) *runtime engine* as shown in Fig. 1b.

A. DisProTrack Static Analyzer

The static analyzer module analyzes the application executables and generates a semantic relationship between multiple *Log Message String* (LMS). Here, we define a LMS as a string present in the executable responsible for printing some log message. Typical LMSes contains format specifiers, error codes, debug level identifiers, etc. Since we only require the causal relationships between LMSes, the static analyzer identifies only the *Log Message Generating Functions* (LMGF).

Definition 1 (Log Message Generating Functions)**.** We define a LMGF as a library function that is directly used for printing a LMSes in either terminals, specific log store files, or log databases.

For example, consider the following C code snippet with a popular logging library Log4C.

```
log4c_category_log(NULL,
LOG4C_PRIORITY_ERROR,"Hello World!");
```

Here the LMS is Hello World! and the LMGF is log4c_category_log. More details about LMGF is described in Section IV-A.

Definition 2 (Log Message String CFG)**.** A LMS-CFG is formally defined as a directed graph $G' = (V', E')$ where $\forall_{e'_{i,j} \in E'} e'_{i,j} = (v'_i, v'_j) : v'_i, v'_j \in V'$ represents a directed edge between v'_i, v'_j where each $v' \in V'$ represents an LMGF. G' is constructed from CFG $G = (V, E)$ such that, $V' \subseteq V$ and,

$\underset{e' \in E'}{\exists} e' = (v'_i, v'_k) : v'_i, v'_j, v'_k \in V', \nexists_{v'_j} v'_j \in \mathfrak{P}(G, v'_i, v'_k)$. In this case $\mathfrak{P}(G, v'_i, v'_k)$ represents the directed path from v'_i to v'_k in G.

An example of a CFG and the corresponding LMS-CFG is provided in Fig. 1c where the constructed LMS-CFG contains only the LMS nodes from the CFG.

1) Contextualization of application log and system log (handling Challenges 1 & 2): The LMS-CFG is stored separately and frequently consulted by the runtime engine. During the execution, when the different levels of logs are generated, the constructed LMS-CFG is used to understand the relationship among those log messages. Based on the relationship, the logs coming from different sources are combined together (details in Section IV-B2).

B. `DisProTrack` Execution Path (Runtime Analyzer)

During the execution of the system, the applications generate multiple logs depending on the user's activities. So the task of the runtime engine is to collect and contextualize the log messages generated at the systems and application levels. We assume that the micro-services containers have `auditd` [42] installed for monitoring system-level logs. This assumption does not violate the *"without instrumentation of the application source code"* constraint, as it is straightforward to add an `auditd` layer to the existing containers without knowing anything about the application source codes.

1) Tracing the execution units from processes belonging to different micro-services (handling Challenge 3): To avoid confusion during the contextualization process of the accumulated logs, we append the Process ID (PID) of the process responsible for generating the log and the timestamp to add the semantic context to individual log entries. For this purpose, we develop a Loadable Kernel Module (LKM) which intercepts all the write syscalls caused by log printing functions to extract the PID information and timestamp of the system and append it to the log preamble. This LKM is deployed in the bare metal infrastructure where the containers are executing (details in Section IV-B2).

2) Dependency explosion and handling confounding root causes (handling Challenge 4): During the execution of the applications, the runtime engine periodically fetches the marked log entries from the micro-services. It generates the UPG after consulting the LMS-CFG. The LMS-CFG provides a relationship between the applications and file, which is exploited to construct UPG as depicted in Fig. 2. The details of the UPG construction procedure are described in the next section. The generated UPG is consulted by the system administrators for the system provenance tracking. The root cause of any suspicious log entry can be identified by backtracing the UPG (details in Section IV-C).

IV. Components of DisProTrack

In this section, we describe the design of individual components of `DisProTrack`. As mentioned previously, `DisProTrack` is subdivided into two parts; (a) *Static Analyzer*, and (b) *Runtime Engine*.

A. Static Analyzer

The static analyzer takes individual micro-service's application binaries as input and constructs the corresponding LMS-CFG for that micro-service application. A typical application will contain a set of statements to print the LMSes with syscalls in between. In our proposed framework, we load the executable application binary and use the python `Angr` module [43], [44] to identify the CFG from it. The generated CFG is a directed graph having the basic instruction blocks (syscalls, printing of LMSes, etc.) as nodes, and the edges of the graph represent jump/call/return statements from one block to another. Let the CFG be represented as $G = (V, E)$, where V and E are the nodes and the edges of the CFG, respectively. We then use the same `Angr` module to extract all the nodes $\mathbb{F}$ from the CFG $G(V, E)$ corresponding to various function calls. One significant issue with the generated CFG is that it does not always provide the complete graph due to the missing hardware-dependent features and system call information. Therefore, in this case, we only concentrate on the LMGF. Typically the stable versions of the applications use standard LMGF (e.g. `printf`, `log4c`, `syslog`, etc.). Let $\mathcal{L}$ be a list of standard LMGF names. We find $\mathbb{L} \subseteq \mathbb{F}$ where $\mathbb{L}$ contains the LMGFs from $\mathcal{L}$. We construct a LMS-CFG $G'(V', E')$ from $G(V, E)$ with the help of $\mathbb{L}$.

We propose Algorithm 1 to convert $G = (V, E)$ to $G' = (V', E')$ for a given $\mathbb{L}$ as the input. Each node in G represents a sequence of instructions. If a node v contains a LMGF name, then the algorithm extracts the arguments of the LMGF, which has been defined as a LMS apriori. For example, for a given LMGF `fprintf`, consider the following log entry function.

"`fprintf(stderr, "AH00526: Syntax error on line %d of %s:");`"

In this case, the algorithm extracts the LMS as `"AH00526: Syntax error on line %d of %s:"`, which is the constant string reference passed as an argument to the LMGF.

During the construction of LMS-CFG, we need to identify the caller functions of the LMGF, which is a time-consuming operation. Therefore, we limit the depth of backward tracing up to a certain threshold (BT), as shown in Algorithm 1:Line 13. The optimal value of BT depends on the complexity of the generated CFG, which we shall discuss during the experimental evaluation (Section V-B1). For accurate identification of the LMSes, we need to avoid the programming language-specific format specifiers. For that, we replace the format specifier of LMS with equivalent regular expressions (see Algorithm 1:Line 10). For example, consider the LMS as shown earlier: "`AH00526: Syntax error on line %d of %s:`". The regular expression equivalent to this LMS becomes "`AH00526: Syntax error on line -?[0-9]+ of .*:`", where `%d` and `%s` are replaced with `[0-9]+` and `.*`, respectively. Additionally, we mark the starting and ending LMS positions of the event handling loops with a flag. This generated and marked LMS-CFG is used by *Runtime Engine* explained next.

Algorithm 1: CFG to LMS-CFG Generation

Procedure `Main`
  Input: $G(V, E)$: CFG, $\mathbb{L}$: Set of LMGFs in $G(V, E)$
  Output: LMS-CFG $G' = (V', E')$
  Initialization:
    $V' \leftarrow \emptyset, E' \leftarrow \emptyset$
  for $v \in \mathbb{L}$ **do**
    /* The Angr tools return whether a node is a loop */
    if v *has a loop* **then**
      /* For a loop, we need to find out all the LMSes ($\mathbb{A}$) that are printed through the loop. */
      $\mathbb{A} \leftarrow$ `BackTraceOptimization`$(BT_{th}, G(V, E), v)$;
      /* Construct the subgraph $G_A(V_A, E_A)$ for A */
      $G_A(V_A, E_A) \leftarrow$ `CreateSubGraph`(A_i);
      $V' \leftarrow V' \cup V_A$;
      $E' \leftarrow E' \cup E_A$;
  for $v' \in V'$ **do**
    Identify format specifiers in v' and replace them with suitable Regular Expressions;
  return $G'(V', E')$;

Function `BackTraceOptimization`
  Input: BT: Backtrace Threshold, $G(V, E)$: CFG, v: A node from $\mathbb{L}$ having a loop
  Output: $\mathbb{A}$: A set of LMSes in the loop
  Initialization:
    $\mathbb{A} \leftarrow \emptyset$
  if v *has a set of LMSes* $\{l_0, \cdots l_k\}$ *printed through the loop with syscalls in between the LMSes* **then**
    /* We consider the loops having a syscall and associated LMSes */
    $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
    return $\mathbb{A}$;
  else
    /* The loop within v does not print any LMSes */
    do
      /* Backtrack to the LMGF that called v, until the backtrack threshold BT is reached. */
      if $\langle \bar{v} \rightarrow v \rangle$ *is a directed edge in* $G(V, E)$ **then**
        if $\bar{v}$ *has a loop with a syscall and a set of LMSes* $\{l_0, \cdots l_k\}$ **then**
          $\mathbb{A} \leftarrow \mathbb{A} \cup \{l_0, \cdots, l_k\}$;
          return $\mathbb{A}$;
      $v \leftarrow \bar{v}$;
      $BT \leftarrow BT - 1$;
    while $BT > 0$;
  return $\mathbb{A}$;

Function `CreateSubGraph`
  Input: $\mathbb{A}$
  Output: $G_A(V_A, E_A)$: A Subgraph generated from $\mathbb{A}$
  Initialization:
    $V_A \leftarrow \emptyset, E_A \leftarrow \emptyset$
  foreach $\{l_i, l_{i+1}\} \in \mathbb{A}$ **do**
    $V_A \leftarrow V_A \cup \{l_i, l_{i+1}\}$;
    $E_A \leftarrow E_A \cup \{\langle l_i \rightarrow l_{i+1} \rangle\}$;
  return $G_A(V_A, E_A)$;

B. Runtime Engine

We design `DisProTrack` Runtime Engine as a micro-service that can be deployed in the serverless platform. Since most of the serverless functions are deployed using multiple containers, log collection and analysis are challenging. `DisProTrack` is concerned about two types of logs; (i) Logs generated by the applications (i.e., inside the container) and (ii) System and/or serverless daemon logs (i.e., logs from the physical servers systems) – we call them Syslogs. The major challenge of obtaining a container's internal logs is that as the process spaces of the containers and the host system are isolated, identification of processes and syscalls become difficult. To avoid such issues, we use an audit daemon in each container by deploying a separate image layer[3] over the containers. The audit daemon is used to track the syscalls generated by the applications inside the containers.

However, audit logs provide the container internal PID, which might conflict with the audit logs obtained from different containers. The conflict must be resolved before the aggregation of the system log and application log. Therefore, we develop a LKM which intercepts LMS before it is printed in the log file and appends a unique tag (which is a combination of container ID, PID, and timestamp) to each LMS entry. This unique tag is used to establish a relationship by adding semantic context among the LMS.

The scope of operation for the runtime engine starts whenever the applications start generating logs. We have segregated Runtime Engine into three submodules; (a) Log accumulator, (b) Log processor, and (c) Provenance builder, which are described as follows.

1) Log Accumulator: Once the log files are generated, the *Log Accumulator* module periodically pulls the log files from all levels and performs operations on them to correlate them with the events. The non-persistent and ephemeral nature of micro-services implies the risk of data loss or the loss of logs generated during the execution phase of the container lifecycle when the container shuts down. Therefore, the proposed module periodically pulls the logs. To prevent data loss due to "`SIGKILL`", we deploy a signal handler inside the deployed image layer to instruct the container to save the logs in a persistent data volume.

2) Log Processor: Once the logs are accumulated, the *Log Processor* module aggregates the logs collected from various sources. However, simple concatenation of log files does not preserve the semantics relationship. Therefore, we convert the text-based log files into an equivalent `JSON` format. From the formatted application log files, the PID of the application is extracted from the tag generated by the LKM. Using the PID, the syscalls are identified from the audit logs. Now the LMS and the syscall-generated logs are merged to create an Application-Specific Common Log (ASCL) file such that the application logs appear just before the corresponding Syslogs.

C. Provenance Builder

At the runtime, the *Provenance Builder* takes the ASCL file and LMS-CFG of a process as input and constructs the corresponding UPG component for that process. The ASCL file contains interleaved application-level logs ($\langle pid, ts, lms \rangle$) and Syslog ($\langle pid, ts, syscall, path, exe \rangle$) entries. Using Algorithm 2, we identify the execution units in the ASCL file with

[3]https://docs.docker.com/storage/storagedriver/ (Accessed: February 1, 2023)

Algorithm 2: UPG Construction

```
1  Procedure Main
       Input: G' = (V', E'): LMS-CFG, F_pid: ASCL file for process pid
       Output: G^U_pid = (V^U_pid, E^U_pid): UPG component for process pid
2      Initialization:
3          V^U_pid ← ∅, E^U_pid ← ∅, U_pid ← ∅, E_pid ← ∅, G_pid ← ∅;
           /* U_pid: An execution unit denoting the
              set of LMSes matched with V'          */
           /* E_pid: Set of system logs corresponding
              to an execution unit                  */
           /* G_pid: Set of all E_pid from F_pid    */
4          end_unit ← false /* tracks execution units */
5      foreach e in F_pid do
           /* e can be an application-level entry
              ⟨pid, ts, lms⟩ or a syslog entry
              ⟨pid, ts, syscall, path, exe⟩         */
           /* pid: Process ID, ts: Log timestamp    */
           /* lms: LMS, syscall: Syscall in log     */
           /* path: System object (dir, socket,
              etc.) accessed by the executable      */
           /* exe: Name of the executable           */
6          if e is an application-level entry then
7              if e is the first entry in F then
                   /* Perform a regex match to find a
                      node n from G'(V', E') which
                      matches with e. n denotes the
                      current state of the search.  */
8                  n ← FindLMSinGraph(G', e);
9                  U_pid ← U_pid ∪ {n}
10             else
                   /* Perform a regex match to find a
                      node in the neighbor of n over
                      the graph G'(V', E'); the new
                      node becomes the current state
                      of the search                 */
11                 n ← MatchNeighbourNodes(G',n,e);
12                 U_pid ← U_pid ∪ n ;
13             if n is a leaf node in G'(V', E') then
14                 end_unit ← true /* Seen a block of
                      LMSes logged by the application
                      from a LMGF                   */
15         if end_unit is true then
16             E_pid ← E_pid ∪ e;
17             end_unit ← false /* tracked syslogs for an
                  execution unit                    */
18             G_pid ← G_pid ∪ E_pid;
19             E_pid ← ∅
20         else
               /* e is a syslog entry and end_unit is
                  false                             */
21             E_pid ← E_pid ∪ e;
       /* Tracked all syslogs for individual
          execution units, now construct the UPG */
22     partition ← 0 /* keep tracks of individual
          execution units                           */
23     foreach E_pid ∈ G_pid do
24         partition ← partition + 1 ;
25         foreach e ∈ E_pid do
26             if e has a valid exe and syscall then
27                 V^U_pid ← V^U_pid ∪ {e.exe} /* the name of
                      the executable application
                      binary becomes a node         */
28                 V^U_pid ← V^U_pid ∪ {partition||e.path} /* the
                      partition value appended with the
                      object path accessed in e
                      becomes the second node       */
29                 E^U_pid ← E^U_pid ∪ {⟨e.exe →
                      partition||e.path, e.syscall⟩} /* an
                      edge is added between the above
                      two nodes with e.syscall as the
                      edge label                    */
```

the help of LMS-CFG, where an execution unit represents a sequence of LMSes execution of a process. In this heuristic, we intelligently apply the LMS-CFG to mark the end of an execution unit by using the leaf nodes of the graph. In this heuristic, we assume that the micro-services are *weakly time-synchronized* with a small time drift λ. The bound on λ depends on how frequently the log messages from two different execution units are getting printed. In reality, λ can be in the order of a few hundred milliseconds as printing the logs too frequently is also an overhead for an application.

Extract Execution Units with the help from LMS-CFGs of application micro-services. Initially, the execution unit is empty. When the algorithm encounters an application-level log entry from the file, which also happens to be the first entry, it performs a regular expression matching on the LMS-CFG to find a node with a match of that entry. If a valid match, say, n, is found, then n becomes the current state. For the rest of the log entries, it performs the regular expression matching with the neighbors of n from the LMS-CFG to find the next state. This step is repeated for all the application-level entries till we find n as a leaf node in the LMS-CFG, which denotes an end unit of the execution unit. The intermediate Syslog entries are added to their respective PID's execution unit E_{pid}. When the end unit becomes true, it implies a block of LMSes has been printed along with a syscall execution. Hence, we save the state of this execution unit in a set G_{pid} and make the execution unit E_{pid} empty for the next set of LMSes to be added. We also set the end unit flag to false.

Interconnect execution units. Once all the syslogs for an execution unit is extracted, Algorithm 2 (Line: 23-29) constructs the UPG for that execution unit G_{pid}. We keep track of individual execution units in G_{pid} using a variable called $partition$. For every execution partition in G_{pid}, if an entry e contains exe and $syscall$, then the nodes of the UPG are – (i) the name of the executable application binary ($e.exe$), and (ii) the system objects such as files, socket, directory, etc. ($e.path$) appended with the $partition$ value. The edges are added between the above two nodes, where the corresponding Syscall denotes the edge labels from the Syslog entry ($e.syscall$). The resulting UPG is the union of all the UPG components constructed for each PID.

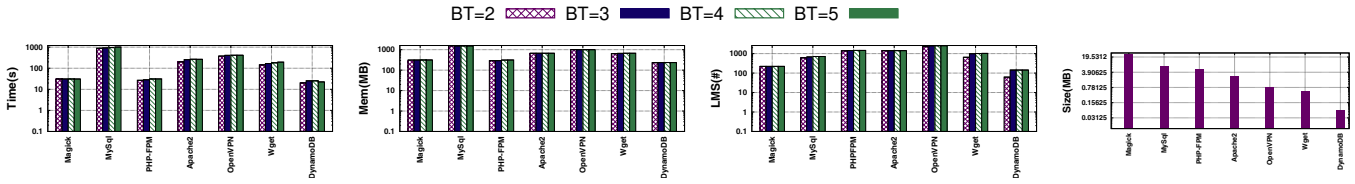
V. Performance Evaluation

The objective of our experimentation is two-fold as follows.

1) Since `DisProTrack` is targeted for serverless applications; resource overhead is a major concern. Therefore, experimentally we want to understand the resource overhead of the framework.
2) We also want to understand how effective `DisProTrack` is for identifying malicious activities.

For experimental analysis, we have implemented `DisProTrack`[4] and executed it in a testbed deployed in our lab. The implementation details are as follows.

[4]https://anonymous.4open.science/r/Project_ALV_2022-CEFD/



(a) Required Time (b) Memory Consumption (c) LMS Found (d) Application Size

Fig. 3: Static Analysis – The Y-axes are in logarithmic scale

A. Experimental Setup

The experiments are executed on a workstation having Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz dual-core processor and 32 GB of memory. To ensure bare metal infrastructural abstraction, we use multiple Virtual Machine (VM) instances running with Ubuntu 22.04 LTS with Kernel version $5.15.0 - 39$-generic. The compute functions are deployed using Docker[5] across the VMs. We use docker-swarm[6] for container management where one container instance is deployed as the manager. For communication, we use an overlay network driver to ensure direct connectivity among the containers. During our experimentation, we have used the standard docker images of the applications collected from `Dockerhub`[7] with an added image layer of `auditd` as described in Section IV-B1. On the other hand, the proposed static analyzer and runtime engine are deployed as a containerized micro-service.

B. Resource Utilization

We conducted experiments to analyze the resource utilization and overhead imposed by `DisProTrack`, which has a two-fold view – (i) the overhead of the static analyzer, which is a one-time event for every deployment scenario, and (ii) the overhead of the runtime engine, which is a periodic event. Therefore, we present different sets of experiments to understand these overheads as follows. Each experiment is repeated 10 times, and the mean is shown in the plots.

1) Overhead of Static analyzer: To understand the resource utilization overhead during static analysis, we have considered 7 different applications as given in Table I.

TABLE I: List of Benchmarked Applications

Name	Type of Application
`Apache Webserver`	Together popular as LAMP-Stack and widely used for web service deployment
`PHP-FPM`	
`MySql`	
`DynamoDB`	A noSQL based database used in Amazon Lambda stream processing usecase
`ImageMagick`	An image processing framework can be used for various Amazon Lambda file processing usecases
`OpenVPN`	A popular Secure IP tunnel daemon
`Wget`	Linux network downloader

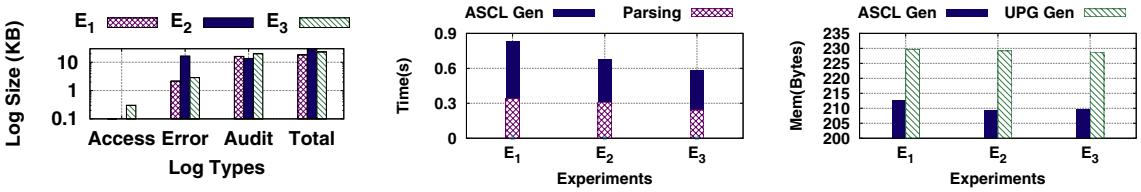
Based on the obtained results (Figs. 3a and 3b), we observe that the static analysis for `MySql` takes the longest time to complete and needs a greater amount of memory. The time and memory requirements increase when the BT increases. This is justified as the value of BT increases, the number of backtraces also increases (as mentioned in Section IV-A). However, an increase in the number of backtraces can identify more number of LMSes as shown in Fig. 3c. We also observe that even though the size of `Magick` is greater than the size of `MySql` (Fig. 3d), it takes more time and memory for `MySql` to trace the LMSes. This is due to the nature of the program control instructions used by developers while developing the application. Hence, it takes more time and memory to complete the backtrace in the presence of higher number of indirect branch instructions. For the same reason, `PHP-FPM` takes less amount of time and memory than `MySql`; however, it identifies more number of LMSes.

2) Overhead of Runtime Engine: Since the runtime engine works for a pipeline of micro-services, we execute multiple experiments on a web service application. Our deployed LAMP stack web service is composed of 3 micro-services (similar to Fig. 2); (a) [μ_1]:`Apache` based web server, (b) [μ_2]: `PHP-FHM` for server-side request handling, and (c) [μ_3]: `MySQL` for handling queries generated by the `PHP-FHM`. Each HTTP request incident on μ_1 forwards a `FastCGI` requests to μ_2. μ_2 executes the handler functions and accesses the database deployed in μ_3 for dynamic content. Once the page is constructed, μ_1 returns it as a response to the client. Specifically, we have hosted an application authorization portal where μ_1 interacts with the μ_2 via μ_3 to validate the user credentials. Upon receiving the valid credentials, μ_1 redirects from the login page to a user-specific home page. Otherwise, it remains on the same page with an error message pop-up. On top of the authorization service, we consider three experimental login scenarios as follows.

- E_1 New users register themselves, and the details are updated in the database. Once registration is successful, the user tries to log in and then log out of the application with valid credentials.
- E_2 A user logs in with a valid credential and goes to the home page. Once logged in, they try to reset the password and re-login with a new password.
- E_3 A user tries to log in to the deployed web service. On the homepage, they click on a link that triggers a background script and redirects the user to a different IP.

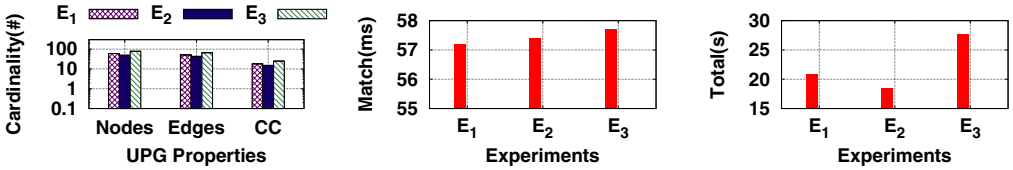
The size and types of log files generated during these experiments are presented in Fig. 4a. The results show that the error logs generated during E_2 are more significant than E_3. In contrary, the access log generated by E_3 is much greater than

[5]https://www.docker.com/ (Accessed: February 1, 2023)
[6]https://docs.docker.com/engine/swarm/ (Accessed: February 1, 2023)
[7]https://hub.docker.com/ (Accessed: February 1, 2023)



(a) Log File Size (b) Phase wise execution time (c) Phase wise Memory utilization

Fig. 4: Runtime Analysis – The Y-axis in (a) is in logarithmic scale



(a) UPG Parameters (b) RegEx Identification Time (c) UPG Construction time

Fig. 5: Runtime Analysis – The Y-axis in (a) is in logarithmic scale

the rest of the two scenarios. However, the audit log generated by E_3 is greater than the other two. The log file size depends on a user's actions while browsing the web service.

Next, we present the phase-wise time consumption analysis in Fig. 4b. Here we observe that the time required to parse the log files, merging them to create an ASCL and generation of UPG during provenance builder (shown in Fig. 5c) is directly related to the total amount of logs generated during the experiments which take approximately 600ms-900ms to complete the runtime processing. After contextualizing the log files, the system administrator provided suspicious log messages converted into an equivalent regular expression (RegEx) to avoid the values of the variables. This generated RegEx string is searched across the LMS-CFG which takes only a few milliseconds (as shown in Fig. 5b). We also provide the memory consumption during the individual phases as presented in Fig. 4c which suggests that the memory footprint is significantly lower for the modules of the runtime engine (in between 200B to 250B).

We observe that depending on the scenarios, the nature of the UPG is different, as shown in Fig. 5a. We find that E_3 has a significantly higher amount of nodes and edges than the rest of the two cases. The number of connected components in the UPG is considerably higher for E_3. Each connected component in the graph represents an execution unit of a process, which helps resolve the dependency explosion problem. Only the related events of a process form a connected component. More number of connected components implies that the graph is more densely partitioned.

VI. Analysis

This section discusses the effectiveness analysis of `DisProTrack`. In this context, we consider an adversarial scenario. The static analyzer can easily detect an adversary who can modify the application's source code. However, an adversary accessing the runtime platform can evade the static analysis and may issue malicious operations during code execution. Therefore, in this section, we primarily focus on detecting runtime adversaries. We have simulated multiple attack scenarios by considering different adversary models. We next discuss one of them.

A. Adversarial model & Attack Scenario

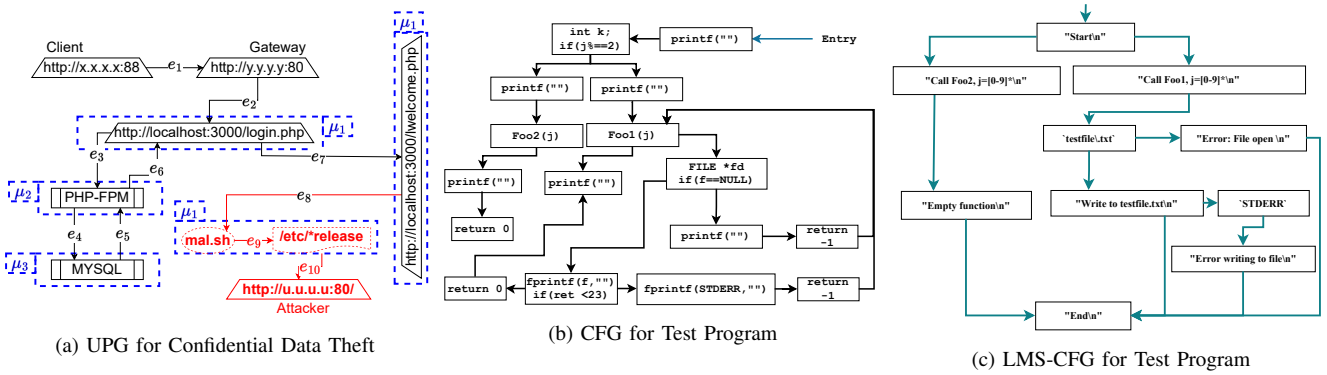
We assume that an adversary can somehow bypass the authorization mechanisms and gain access to one/more application container(s) except the runtime engine container without being detected. In the compromised container(s), the adversary can add, modify, and/or execute scripts and deploy webpages. However, We assume that the logs and audit rules are part of the trusted computing base (TCB). Moreover, the communication between the compromised container and the runtime engine can not be adulterated.

Using this adversarial model, the attacker may perform different types of malicious activities. However, here we present the case study in light of *"confidential data theft"* attack[8] where the attacker attempts to insert a malicious script to steal the confidential information from the compromised system. This script can be triggered during execution time. We simulated the attack on top of our web service application described in Section V-B2 by placing a malicious script named *"mal.sh"* in `PHP-FPM` container. This script is executed when an authorized entity login to the website after successful authentication and clicks on a masked link on the welcome webpage. Once the script gets triggered in the background, alongside normal execution, it tries to access and read a sensitive file on the server and forward them to an attacker's server IP address.

B. Provenance Builder for Attack Detection

Let us understand how this attack can be detected using `DisProTrack`. The UPG constructed by `DisProTrack` for above attack scenario is presented in Fig. 6a. To ensure

[8]https://bit.ly/trendmicro_shading_light (Accessed: February 1, 2023)



(a) UPG for Confidential Data Theft

(b) CFG for Test Program

(c) LMS-CFG for Test Program

Fig. 6: A PoC Case Study to Analyze the Accuracy of DisProTrack

readability, we have omitted a few UPG metadata and masked IP addresses. Using the relative event sequence numbers/edge identifier, the sequence of events can be traced in order. From the particular UPG instance, we can find that a client with IP address $x.x.x.x$ is connected to the service via port 88. The client takes the normal authentication route (from e_1 to e_7) to reach the `welcome.php` page. After that, the `mal.sh` script is triggered which results in collection of data from `/etc/*release` directory and forward it to $u.u.u.u$ (from e_8 to e_{10}). This step is a deviation from the standard behavior; therefore, the system administrator must intervene in this case and take some preventive action (e.g., suspend user access, block IP, etc.).

```c
#include<unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define Foo2(int j) ({printf("Empty function\n");})

int Foo1(int j){
    FILE *f = fopen("testfile.txt", "a");
    if (f == NULL) {
        printf("Error: File open\n");return -1;
    }
    if (fprintf(f, "Write to testfile.txt\n") < 23) {
        fprintf(stderr, "Error writing to file\n");
        return -1;
    }
    return(0);
}
/***********************/
int main(void){
    printf("Start\n");int j = rand();
    if(j%2==0){
      printf("Call Foo1(), j = %d\n", j);Foo1(j);
    }else{
      printf("Call Foo2(), j = %d\n", j);Foo2(j);
    }
    printf("End\n");
    return 0;
}
```

Listing 1: PoC Program for Accuracy Analysis

C. Accuracy analysis of DisProTrack

During our development of DisProTrack and experimentation with several other attack scenarios, we observed that the detection of attack depends on the accuracy of LMS-CFG construction from CFG in the static analyzer. Although we have presented the LMS identification results in Fig. 3c for different applications, the lack of gold standard values restricts us from claiming the accuracy of the proposed Algorithm 1. Therefore, to justify the accuracy of the static analyzer, we present a small sample proof-of-concept (PoC) program (Listing 1) here. The sample program presents two function calls depending on a random number generated. The functions can either generate a log message in the STDERR console or in a log file. As the program is simplistic in nature, it is easy to ascertain the accuracy of the generated LMS-CFG from the framework presented in Fig. 6c. For comparison purposes, we present the corresponding CFG in Fig. 6b, which is also obtained from the static analyzer module. From these two figures, we observe that both the figures have 8 listed LMSes and two file handles. The causal paths among the nodes are also verified to ensure that all the paths are covered in the corresponding LMS-CFG.

VII. Conclusion

This paper developed a non-invasive causality analysis framework, called DisProTrack, for provenance tracking over distributed serverless applications. The proposed framework is capable of adversarial attack analysis by identifying the root causes effectively. DisProTrack can be deployed on top of the SLC as a micro-service and has the virtue of being lightweight and provides results within 0.5 minutes. A PoC analysis of DisProTrack also shows its efficiency and efficacy in detecting attack instances for an SLC application.

A critical aspect of DisProTrack is that it uses a heuristic to identify the execution units by matching the CFGs generated from the micro-service binaries with the runtime application and system logs based on their temporal execution patterns. Thus, the framework might wrongly identify the execution units if the underlying servers running the micro-services are not weakly time synchronized (time drift within a small threshold). Nevertheless, this condition rarely occurs in a typical distributed SLC platform with multiple micro-services interacting with each other. Further, DisProTrack can be deployed as an additional micro-service along with the other application micro-services running over the servers, making it robust to be applied for a wide range of production-grade serverless application scenarios.

The authors have provided public access to their code and/or data at https://github.com/usatpath01/DisProTrack.

REFERENCES

[1] K. Kuusinen, V. Balakumar, S. C. Jepsen, S. H. Larsen, T. A. Lemqvist, A. Muric, A. Nielsen, and O. Vestergaard, "A large agile organization on its journey towards devops," in *Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*, 2018, pp. 60–63.

[2] K. Ojo-Gonzalez, R. Prosper-Heredia, L. Dominguez-Quintero, and M. Vargas-Lombardo, "A model devops framework for saas in the cloud," in *Advances and Applications in Computer Science, Electronics and Industrial Engineering*, 2021, pp. 37–51.

[3] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," in *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019, pp. 185–199.

[4] K. S.-P. Chang and S. J. Fink, "Visualizing serverless cloud application logs for program understanding," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2017)*, 2017, pp. 261–265.

[5] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Communications of the ACM*, pp. 76–84, 2021.

[6] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Software*, pp. 25–31, 2020.

[7] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–9.

[8] "Aws x-ray," https://aws.amazon.com/xray/.

[9] "Google cloud - viewing monitored metrics," https://cloud.google.com/functions/docs/monitoring/metrics/.

[10] "Microsoft azure monitor," https://cloud.google.com/functions/docs/monitoring/metrics/.

[11] "Iopipe - monitor serverless applications," https://www.iopipe.com/.

[12] "Dashbird - monitor serverless applications," https://dashbird.io/.

[13] "Thundra - monitor serverless applications," https://www.thundra.io/.

[14] "Epsagon - monitor serverless applications," https://epsagon.com/.

[15] S. Zawoad, R. Hasan, and K. Islam, "Secprov: Trustworthy and efficient provenance management in the cloud," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2018)*, 2018, pp. 1241–1249.

[16] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis," in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[17] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *IEEE Symposium on Security and Privacy (SP 2020)*. IEEE, 2020, pp. 1172–1189.

[18] M. M. Anjum, S. Iqbal, and B. Hamelin, "ANUBIS: a provenance graph-based framework for advanced persistent threat detection," in *ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1684–1693.

[19] H. Irshad, G. Ciocarlie, A. Gehani, V. Yegneswaran, K. H. Lee, J. Patel, S. Jha, Y. Kwon, D. Xu, and X. Zhang, "Trace: Enterprise-wide provenance tracking for real-time apt detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4363–4376, 2021.

[20] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran *et al.*, "ALchemist: Fusing application and audit logs for precise attack provenance without instrumentation." in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[21] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy {Whole-System} provenance for the linux kernel," in *USENIX Security Symposium (USENIX Security 2015)*, 2015, pp. 319–334.

[22] J. Tavori and H. Levy, "Tornadoes in the cloud: Worst-case attacks on distributed resources systems," in *IEEE Conference on Computer Communications (IEEE INFOCOM 2021)*, 2021, pp. 1–10.

[23] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter *et al.*, "You are what you do: Hunting stealthy malware via data provenance analysis." in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[24] P. Datta, I. Polinsky, M. A. Inam, A. Bates, and W. Enck, "ALASTOR: Reconstructing the provenance of serverless intrusions," in *USENIX Security Symposium (USENIX Security 2022)*, 2022.

[25] "Linux fuse," https://man7.org/linux/man-pages/man4/fuse.4.html.

[26] C. Schaufler, "Lsm: Stacking for major security modules," https://lwn.net/Articles/697259, 2016.

[27] T. F. J.-M. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 472–484, 2017.

[28] A. Gehani and D. Tariq, "Spade: Support for prmiscovenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[29] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Annual Computer Security Applications Conference (ACSAC 2015)*, 2015, pp. 401–410.

[30] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 487–504.

[31] Y. Xie, D. Feng, Y. Hu, Y. Li, S. Sample, and D. Long, "Pagoda: A hybrid approach to enable efficient real-time provenance based intrusion detection in big data environments," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 6, pp. 1283–1296, 2018.

[32] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019, p. 1795–1812.

[33] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time apt detection through correlation of suspicious information flows," in *IEEE Symposium on Security and Privacy (SP 2019)*. IEEE, 2019, pp. 1137–1152.

[34] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics," in *The Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[35] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security Symposium (USENIX Security 2017)*, 2017, pp. 1111–1128.

[36] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition." in *The Network and Distributed System Security Symposium (NDSS 2013)*, vol. 2, 2013, p. 4.

[37] ——, "Loggc: Garbage collecting audit log," in *ACM conference on Computer & communications security (SIGSAC 2013)*, 2013, pp. 1005–1016.

[38] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting." in *The Network and Distributed System Security Symposium (NDSS 2016)*, vol. 2, 2016, p. 4.

[39] R. Yang, S. Ma, H. Xu, X. Zhang, and Y. Chen, "UIScope: Accurate, instrumentation-free, and visible attack investigation for gui applications." in *The Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

[40] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-Trace: A pervasive network tracing framework," in *USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, 2007.

[41] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Symposium on Operating Systems Principles (SOSP 2015)*, 2015, pp. 378–393.

[42] (2007, Mar) auditd(8) - linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man8/auditd.8.html

[43] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (state of) the art of war: Offensive techniques in binary analysis," *IEEE Symposium on Security and Privacy (SP 2016)*, 2016.

[44] "Angr," https://angr.io/.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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: {soumyachat@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, Elasticon [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, Elasticon 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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 network 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.

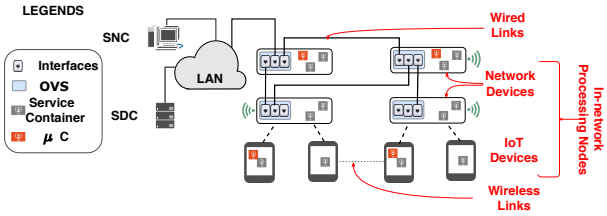


Fig. 1: Components of Aloe Infrastructure

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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).

μCs 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.

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

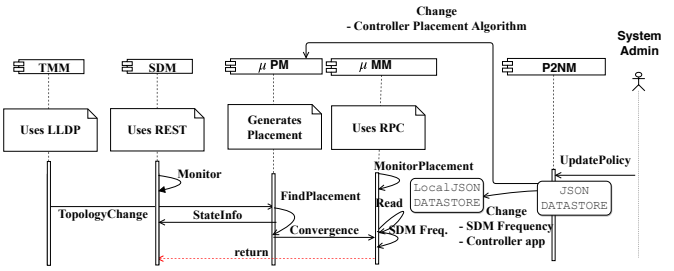


Fig. 2: Aloe function modules and their interactions

However, deployment of μCs in nodes might also create a 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 [29] and SCL [30], 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 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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) μC 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 μC.

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), ``$\mu$C`` (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. The above functioning is different than control plane state discovery module which runs inside the μC and keeps track of the network states. In contrast to that, the proposed SDM keeps track of the roles (i.e. acting as μC or not) that a node is playing in its immediate neighborhood. The detailed state transitions are given in Section A.

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 [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: μPM Controller Placement Algorithm

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

```
Function Trial():
    /* Breaks priority ties                                   */
    PRIO← ⌈Rand()/B⌉;
    return;
Function NeighborμC():
    if Another μC in one-hop neighborhood then
        return true
    else
        return false
Function UMPriority:
    /* If node has unique maximum priority                    */
    if PRIO of this node > maximum PRIO in neighborhood then
        return true
    else if PRIO of this node = maximum PRIO in neighborhood then
        return false
    else
        return None
Function Main():
    while state change detected do
        if CTLR=general & NeighborμC()=false then
            // No μC in neighborhood
            CTLR← undecided;
            Trial();
            // Initialize priority
        else if CTLR=μC & NeighborμC()=true then
            /* Two μCs are adjacent                           */
            CTLR← general;
        else if CTLR=undecided & NeighborμC()=true then
            /* μC found in neighborhood                       */
            CTLR← general;
        else
            if UMPriority()=None then
                /* Executor is not maximum                    */
                continue;
            else if UMPriority()=true then
                /* Executor has unique maximum
                   priority, no need for further
                   trial.                                     */
                CTLR← μC;
            else
                /* Executor has maximum but not
                   unique priority                            */
                CTLR← undecided;
                /* Next round of trial starts                 */
                Trial();
    return
```

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*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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 μCs 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.

```
''' Find shortest path between dpidS and dpidD'''
G=TopologyMonitor()
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`).

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

Listing 2: Distributed State Inspector

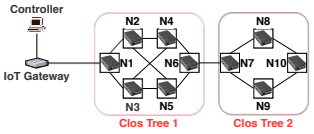


Fig. 3: Testbed:Topology

Fig. 4: Testbed:In Action

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)'''
services=getNeighborServices()
bport=blocking_port
if(blocking_port in service["dpid"]):
 Execute("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.

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*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

B. Implementation Aspects

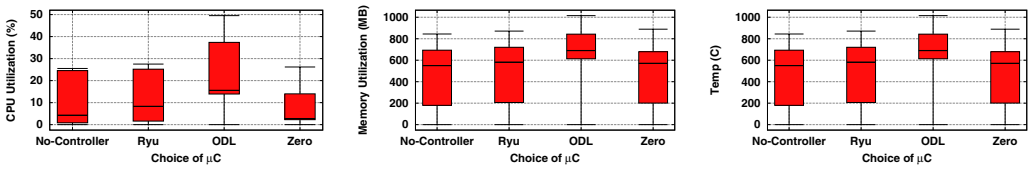
Here we discuss two important implementation aspects of Aloe – (i) flow-table consistency preservation during μC migration, and (ii) choice of controller service for μC implementation.

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 (`changeCtlr()`) 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. `changeCtlr()` 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 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 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

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)

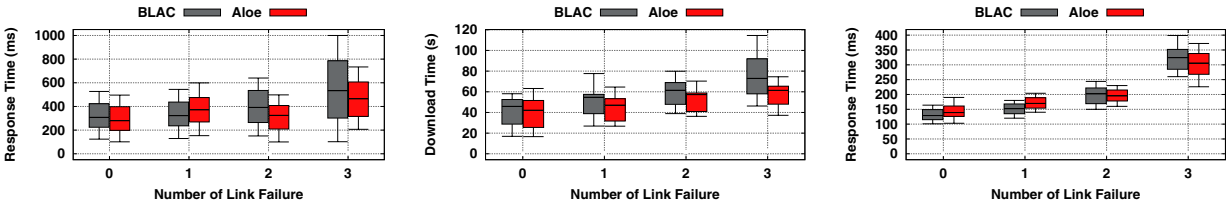
CPU				
μC	**No μC**	**Ryu**	**Zero**	**ODL**
No μC		$\leftarrow$	$\leftarrow$	$\leftarrow$
Ryu	< 0.0001		$\uparrow$	$\leftarrow$
Zero	< 0.0001	< 0.0001		$\leftarrow$
ODL	< 0.0001	< 0.0001	< 0.0001	
Memory				
No μC		$\leftarrow$	**X**	$\leftarrow$
Ryu	< 0.0001		**X**	$\leftarrow$
Zero	> 0.03	> 0.01		$\leftarrow$
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	



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

Fig. 5: Resource Utilization Comparison of Controller Applications

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management



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

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, ✓=*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)**

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 μC 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 combinations [2] in the testbed and 100 different random fault combinations in AWS[3].

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 termination[4] (< 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 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. 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

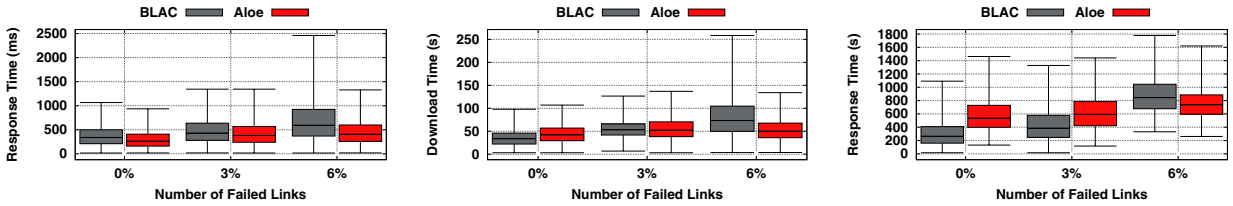
Aloe flow-setup time is dependent upon the convergence time of μPM and the path restoration time. Fig. 8a shows the distribution of the average convergence time of Aloe in the presence of failure. We have an interesting observation here that as the number of simultaneous failures increases, the convergence 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 μPM

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

[3]All experiments are repeated 3 times

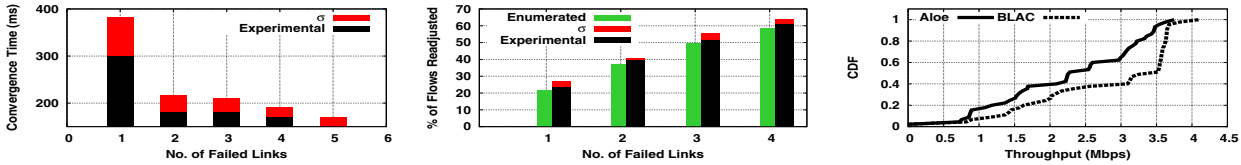
[4]Only in such cases, where the network is partitioned

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management



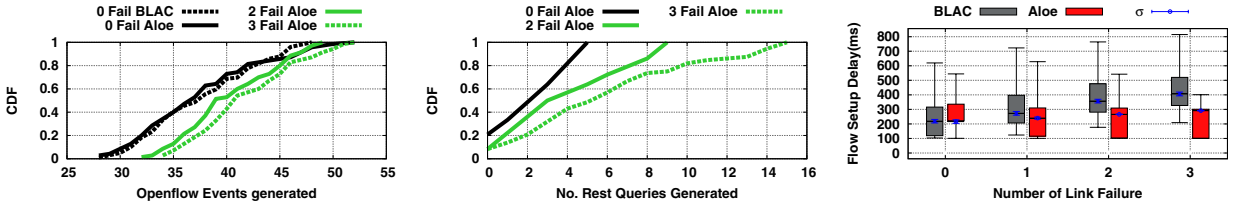
(a) Cassandra (b) HTTP Server (c) Gluster

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)



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

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



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

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

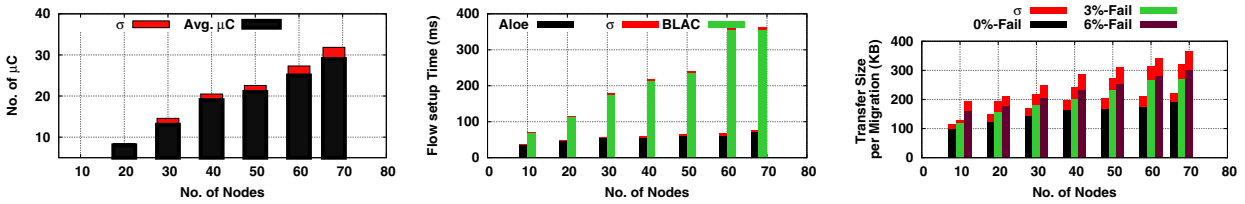
starts executing simultaneously from two different ends of the network. These two waves get diffused in the network and meet in the middle of the network at convergence. That way, multiple faults create multiple such μPM waves in the network in parallel, and as these individual waves need to deal with 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 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 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 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 μC 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.

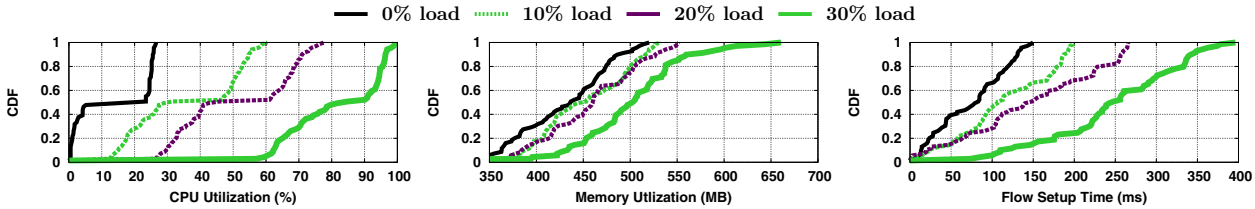
Although Aloe incurs communication overhead, Aloe en-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management



(a) Avg. No. of μC deployed (b) Avg. Flow Setup Time (c) Avg. Controller Migration Overhead

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



(a) CPU Utilization (b) Memory Utilization (c) Flow Setup Time

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 μC instances to guarantee that each node can find a μC in its neighborhood. Fig. 10a 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. 10b[5] 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 μCs 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 μC 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 μCs 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.

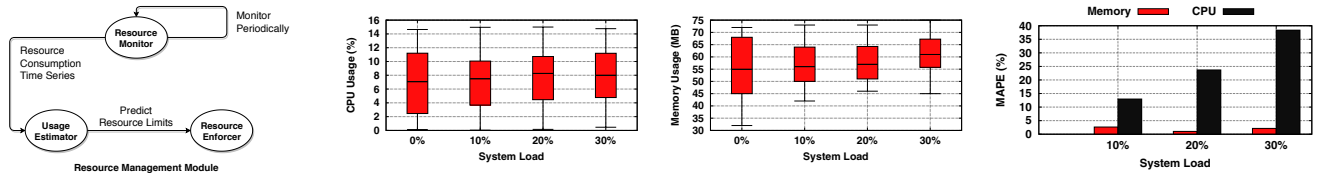
[5]Each experiments are repeated atleast 10 times

A. Effect of Resource Reservation

Reserving resources for μC 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 μC. However, the resource demands of a μC at a particular time depends on the amount of flows managed by that μC. Therefore, we assume that the resource demand of a μC follows a temporal pattern and depends on the network state of the IoT infrastructure. Although over-provisioning resources to the μC 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 μC to balance the μC 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 μC and stores it in a `JSON` data-store. b) *Usage Estimator* periodically analyzes the time-series of the resource usage pattern of the μC and predicts the probable resource demand for the next time period. c) *Resource Enforcer* is responsible for actually resource reservation for the μC based on the predicted resource demand.

1) Prediction of μC 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 μC during a discrete time interval is cyclic. (ii) These characteristics also suggest that, the flow arrival

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management



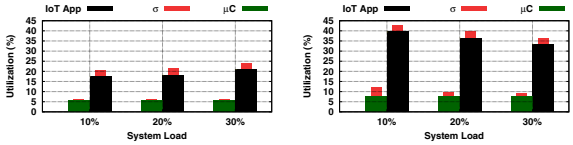
(a) Resource Management Module (b) CPU reservation (c) Memory reservation (d) MAPE Score

Fig. 12: Resource Reservation for Aloe μCs

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.

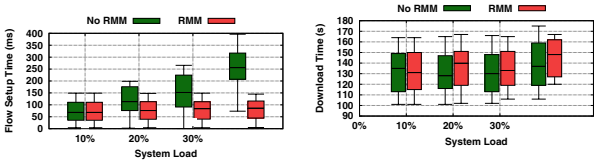
2) Performance Improvement with Aloe RMM: We have integrated the RMM module with Aloe and tested it over the AWS platform as discussed earlier. Like many statistical modeling methods, identification of parameters for ARIMA is a non-trivial challenge. Therefore, we use auto-ARIMA [40] based on our experimental 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 μC in various load scenarios remains almost constant. Fig. 12c shows the memory reservation of the μC 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 μC mapping technique used in Aloe μMM. An OVS chooses a μC based on how quickly the μC responds to its join request. As the system load increases, a lightly loaded μC is more likely to provide a quick response time. Therefore, the lightly loaded μCs 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 μC. Next we check how accurate the RMM prediction model can perform based on the mean average percentage error (MAPE). Fig. 12d revels 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 μC 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 μCs. 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 μC; therefore, a slight resource biasing towards the μCs improve their performance significantly, while having marginal impact on the performance of the IoT application. We present this observation in Fig. 14a,



(a) Memory Utilization (b) CPU Utilization

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



(a) Flow setup time (b) GlusterFS file download time

Fig. 14: Effects of Resource Reservation for Aloe μCs

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 μC, 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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 μPM. 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` and is denoted as $\texttt{CTLR}_u$. As per Algorithm 1, $\texttt{CTLR}_u$ 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 $\mathcal{S} = (\texttt{CTLR}_u : \forall u \in V)$ 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.

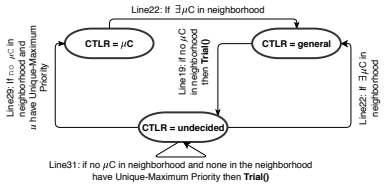


Fig. 15: State transition diagram of a node u

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 μPM 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, $CTLR_u =$`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, NeighborμC() 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. UMPriority()=`true`) or $Q_u = \emptyset$, u executes Line 29.

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

Case-2.iii If u does not have maximum priority (i.e. UMPriority()=`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 μCs 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 μCs 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 μCs. 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 μCs 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2020.3008426, IEEE Transactions on Network and Service Management

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`*),*
(μ`C`$\rightarrow$`general`$\rightarrow$`Undecided`$\rightarrow$`general`*),*
(μ`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 occures; 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. □

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 `PRIO` 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 an 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 $\mathcal{S}$ 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 `PRIO`$= i$ and the rest of nodes can have $0 <$ `PRIO` $\leq i-1$. The value of i can vary from 1 to B. For a fixed i, the rest of `PRIO` 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.

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

□

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).

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

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).

$$\lim_{B \to \infty} \left(1 + B^{-1}\right)^{(B+1)} = e \tag{2}$$

Therefore, $E[X] \leq e$. This result signifies that a node and it's neighbors require expected "e" statement execution by each node for generation of unique maximum priority generation event. □

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 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. Azodolmolky, 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 *Procedeeings 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, "Elasticon: 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. Tatlcolu, 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 UUSENIX 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

Amalgam: Distributed Network Control With Scalable Service Chaining

Subhrendu Chattopadhyay
IIT Guwahati
Guwahati, India 781039
subhrendu@iitg.ac.in

Sukumar Nandi
IIT Guwahati
Guwahati, India 781039
sukumar@iitg.ac.in

Sandip Chakraborty
IIT Kharagpur
Kharagpur, India 721302
sandipc@cse.iitkgp.ernet.in

Abhinandan S. Prasad
NIE Mysuru
Karnataka, India 570008
abhinandansp@nie.ac.in

***Abstract*—Management of virtual network function (VNF) service chaining for a large scale network spanning across multiple administrative domains is difficult due to the decentralized nature of the underlying system. Existing session-based and software-defined networking (SDN) oriented approaches to manage service function chains (SFCs) fall short to cater to the plug-and-play nature of the constituent devices of a large scale eco-system such as the Internet of Things (IoT). In this paper, we propose *Amalgam*, a composition of a distributed SDN control plane along with a distributed SFC manager, that is capable of managing SFCs dynamically by exploiting the in-network processing platform composed of plug-and-play devices. To ensure the distributed placement of VNFs in the in-network processing platform, we propose a greedy heuristic. Further, to test the performance, we develop a complete container driven emulation framework *MiniDockNet* on top of standard *Mininet* APIs. Our experiments on a large scale realistic topology reveal that *Amalgam* significantly reduces flow-setup time and exhibits better performance in terms of end-to-end delay for short flows.**

***Index Terms*—Service function chaining, Virtual network function, In-network processing, Programmable network, software defined network**

I. Introduction and Related Works

Due to the rapid deployments of connected environments, large-scale Internet of Things (IoT) networks [1] have become prevalent in recent years. Management of such large-scale heterogeneous ecosystems requires various network services such as network address translator (NAT), firewall, proxy, and local domain name server (DNS); these network services are called network function (NF)s. Generally, the network functions are deployed using virtual machines (VM)(s) to provide service isolation and reduce CapEx and OpEx; therefore, they are termed as virtualized network function (VNF) [1]. VNFs execution require computation platform to host the VM and execute the NF within the VM. Depending on network management policies, the application messages require steering through an ordered set of VNFs known as service function chaining (SFC) [2].

Among various existing architectures to execute VNFs over a network infrastructure [3]–[5] relies on software-defined network (SDN) [6] to steer flows from one VNF to another. On the other hand, [7], [8] takes a session-based approach where the end hosts control the SFC. Session-based approaches achieve lower host-based state management of VNFs, where SDN-based approaches achieve fine-grained quality of service (QoS). However, for a large-scale network spanning across multiple administrative domains, both of the SFC management (SCM) approaches fall short in several aspects as follows.

(a) Lack of scalability: Existing SCMs [6], [9] use a central controller that monitors the resource usage of the devices and use as the basis for the VNFs deployment. The use of a central controller for VNF deployment becomes challenging, especially when the network spans across multiple autonomous administrative domains that interconnected through different network service providers. On the other hand, the VNF placement is $\mathcal{NP}$-hard [10]. Existing distributed heuristics for VNF placement [11] require multiple rounds to deploy VNFs, which increases flow initiation delay leading to reduced IoT application performance since the majority of the IoT flows are short-lived [12].

(b) Dynamic service chaining: Usually, VNFss modifying the headers are common in a large-scale network. Consequently, the participating VNFs can change the SFCs during the lifetime of a flow based on the flow characteristics. For instance, a classifier VNF can add a load balancer based on the arrival rate of the packets in a flow. Existing scalable distributed VNF placement methods [11] and IP based traffic steering proposals [13] are not suitable for dynamic service chaining. On the other hand, [7] ascertains dynamic service chaining by adding an agent in each device, including hosts. Installation of agents on a large scale IoT becomes infeasible, where the devices with plug-and-play capability can dynamically enter and exit the ecosystem.

(c) Issues of flow monitoring over multi-administrative platforms: To steer the traffic through proper service chains while ensuring QoS, requires fine-grained flow monitoring. Existing flow identification methods using packet header fields are insufficient in the presence of a header modifying VNF in the SFC (such as NAT, load balancer, and proxy). Existing SDN-based flow monitoring schemes like FlowTags [9], Stratos [14] utilize *"vlan/mpls"* tagging which does not work through multiple administrative domains. On the other hand, the use of packet encapsulation in session-based approaches

This work was supported by TCS India Pvt. Ltd.

[1]https://www.sigfoxcanada.com/

Annex to ISBN 978-3-903176-28-7 © 2020 IFIP

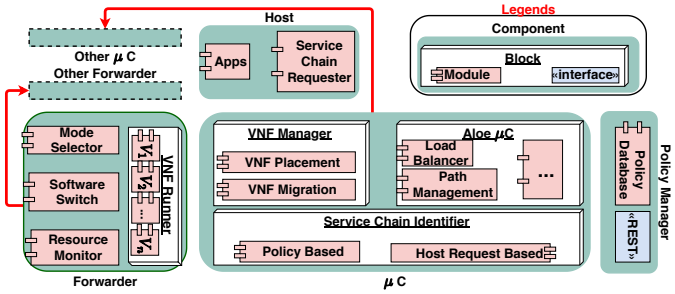


Fig. 1: Component diagram of Amalgam

[8] also fails to ensure QoS.

Therefore, in this paper, we propose *Amalgam* to provide a scalable SFC orchestration platform to provide fine-grained QoS over multiple administrative domains and dynamic SFC. The SCM *Amalgam* provides significant performance improvement in terms of flow initiation delay. The step by step contributions is as follows.

- *Amalgam* achieves distributed state management while ensuring fine-grained QoS by coupling distributed SDN and distributed SCM.
- We develop a distributed heuristic for the distributed deployment of VNFs, which provides performance improvement in terms of flow initiation delay.
- *Amalgam* design exploits the "micro-service"(μS) architecture of the in-network processing platform [15] to support dynamic service chaining and "`zero-touch deployment`" to support the plug and play nature of the devices.
- For performance evaluation, we develop an emulation framework *MiniDockNet* for VNF deployment using "`docker`" [2] over in-network processing, as the existing network name-space oriented mininet [3] emulator is not sufficient for in-network processing.
- Based on the experimental results over a realistic large-scale system (with 70 devices and 6 different service chain scenarios), we found that *Amalgam* can ensure fine-grained QoS without significant increase of resource utilization of the devices. After comparing the performance of *Amalgam* with two state-of-the-art distributed SFC platform *"Dysco"* [7] and *"WGT"* [11] we found that, *Amalgam* is capable of a significant reduction in the flow initiation delay and improves the performances of short-duration flows in terms of end-to-end delay.

This paper is organized as follows. Section II describes the architecture of proposed *Amalgam*. Section III describes the design choices taken during the development of *Amalgam*. We describe the experimental setup details, results and our observations in Section IV before concluding in Section V.

II. Architecture

The proposed *Amalgam*[4] is constructed on the top of *Aloe* [16] framework. *Aloe* provides an orchestration framework for a fault-tolerant self-stabilizing distributed control plane on top of the in-network processing platform using μCs (microcontrollers) instead of the standard SDN controller. Like *Aloe*, *Amalgam* supports 3 operating modes for each device; (a) **Host/(default) mode**: if the device executes only the client/server application, (b) **Forwarding mode**: If the device has multiple active network interfaces, (c) **μC mode**: based on the topology, if *Aloe* selects the device as a μC. At any instant, each device is in "`atleast`" any one of the above modes and each mode is a component of *Amalgam* framework as shown in Fig. 1. A mode selector module in *"forwarder"* component identifies the mode of operation and activates the respective components. Apart from the mode functional components, we add a seperate *"policy manager"* component in the *Amalgam* framework. Policy manager is a distributed database with a *"REST"* driven interface which contains (i) the flow identifier (i.e., "`OpenFlow`" match field) and (ii) ordered list of the types of VNF (service chain) through which the flow should be steered. This component is consulted by the mode functional components at time of determination of SFC.

Among the mode functional components, the host is the simplest. This component is responsible for traffic generation through the *"App"* module, which represents the client/server applications. Additionally, this module can request the nearest μC to change the SFC for the flows generated by the host. The *"mode selector"* module elevates the mode of the device with multiple active interfaces to *"forwarder"*. The forwarder component consists of *"software switch"*, *"VNF runner"*, and a *"resource monitor"*. The software switch module is responsible for forwarding data from one interface to another. On the other hand, the *"VNF runner"* block is reserved for execution of VNFs (e.g., V_1, V_2 etc. as shown in Fig. 1). This VNF runner block from each device constructs the in-network processing framework. During the execution of the VNFs *"resource monitor"* module periodically monitors the available resources in the device. The resource monitor module forwards the collected resource utilization statistics to the μC component.

The μC component is composed of three functional blocks namely Service chain identifier (SCI), ""*VNF manager*"", and "*Aloe* μC". The tasks of these blocks are as follows.

1) Service Chain Identifier: At the startup phase of the μC, SCI caches the policy in a local cache. The local cache is updated whenever the policy manager database is updated. SCI module is consulted when an "`OpenFlow`" "`packet in`" event is initiated at the μC. From the list of VNFs in the service chain, SCI chooses the first VNF, and it's execution status in the local domain. If the VNF is executing inside a forwarder connected to the μC, the SCI consults the path management module to establish the data flow path by installing flow table entries via standard "`OpenFlow`" protocol. Otherwise, it sends a search query to the other μCs to identify the target VNF address. If the address of the VNF is not found, then SCI consults the VNF manager module (Section II-2) to start the execution of the VNF. This procedure is iterated for all the VNFs in the service chain.

[2]https://www.docker.com/
[3]http://mininet.org/
[4]https://github.com/subhrendu1987/NFV_MiniDockNet

2) VNF Manager: The VNF manager module (VMM) works in a distributed fashion and communicates with the neighbor μCs. VMM tries to answers the following two questions; (a) should the VNF be placed in any of the forwarders associated with the μC? (b) which forwarder should take care of the VNF?. The detailed protocol to find an answer to these questions is described in next section. Additionally, VMM also takes care of the dynamic addition or removal of the VNFs to an ongoing flow.

3) Aloe μC: The *Aloe* μC module is the containerized SDN μ controller module as described in [16]. *"Aloe"* provides a distributed fault-tolerant controller module suitable for in-network processing frameworks that are responsible for path management. Use of *Aloe* ensures quick flow initiation along with fault and partition tolerance in *Amalgam*. However, during the design of *Amalgam*, we face several challenges exclusive to SFC deployment of IoT

III. Challenges and Design Choices

The goal of *Amalgam* is to provide a highly dynamic in-network processing platform. In this section, we describe the implementation challenges and the proposed solutions to overcome the scalability issues without affecting the dynamic behavior of the platform.

(a) Plug-and-Play Capability:A typical IoT platform is composed of plug-and-play devices where "`zero-touch deployment`" [5] is highly desired. It is necessary to configure a new device as soon as it enters the eco-system. To avoid individually configuring the devices, we design each component of *Amalgam* (except the host component) as Docker containers. Once a device enters the eco-system, it assumes the host mode of operation. Since the host mode does not require anything more than the IoT applications (clients and servers), they can work smoothly. Whenever the device wishes to change its mode, it can pull the container image of the *Amalgam* component from the nearest forwarder.

(b)Distributed VNF Placement: In a short-lived flow heavy system, minimization of the flow initiation delay is critical. The flow initiation delay consists of following components namely (a) Controller consultation delay (b) SFC deployment delay, and (c) path setup delay. The proposed VNF placement reduces the SFC deployment delay. A SFC for a particular flow is composed of multiple VNFs, which requires resource consumption. Each device of a IoT in-network processing platform has residual resources that can be used for deployment of these VNFs's. The proposed VNF placement identifies the set of devices where the VNFs of the SFCs can be placed for a given network and flow profile while satisfying the capacity constraints of the devices. Maintaining capacity constraints in a multi-domain system is non-trivial since the residual capacity of a device residing in a different administrative domain is difficult to collect. Therefore, we propose the greedy heuristic as given in the Algorithm 1.

[5]https://www.etsi.org/technologies/zero-touch-network-service-management

Algorithm 1: Distributed Placement of VNF

Function `GreedyPlace` (*Path:* P^a, *Service Chain:* C^j, μC: l):
  Find ordered set of unplaced VNFs from C^j;
  $I \leftarrow \{i : i \in P^a, \varphi_i = l\}$;
  Place as many VNFs as possible among I;
  return number of VNFs placed;

Function `Main` (*Flow:* f^j,μC: l):
  /* Find VNF placement profile for f^j in φ_i */
  Find set of paths (P) from s_j to d_j by querying "Path Management" module of l;
  $\underset{P^a \in P}{maximize}$ `GreedyPlace` (P^a, C^j, l);
  if $\exists c_{j,k} \cdots c_{j,j_{max}}$ *not placed* **then**
    /* All devices under l */
    Obtain the list of adjacent μC of l and store it in $N\mu$ **foreach** $l' \in N\mu$ **do**
      `Main` (f^j, l');
  return;

Each μC in the end-to-end path (P) executes the proposed heuristic for each flow (f^j represents jth flow) from source (s_j) to destination (d_j). We denote SFC of f^j with C^j. Certain μC with ID l maintains the topology information as the list of devices (D_l) and list of links (E_l) where each link $e_{i,i'} \in E_l$ represents the physical connection between two devices (i and i'). For the sake of simplicity, we denote the μC associated with $i$as φ_i. The proposed heuristic identifies a path P^a between s_j to d_j from the set of P such that, most of the VNFs of C^j are placed near s_j in a distributed fashion. This way, one μC does not need the resource utilization of devices from other administrative domains. Once the flow is established, the resource utilization of devices in the path (`info`) is piggybacked with the data packets. The VNF manager can re-solve the Algorithm 1 and find a new allocation of VNF with updated utilization.

(c)Migrations of the VNFs: A VNF may be relocated during (a) VNF readjustment due to prior sub-optimal placement and (b) addition or removal of the device. The μC nearest to the source node of the flow decides the VNF readjustment after it receives the piggybacked resource utilization of the devices. On the other hand, the addition and removal of devices trigger "`topology_change`" event, and the local μC initiates the decision about the VNF deployment. In both cases, the decider μC starts the migration process at the source device. Initially, the source device saves a snapshot of the executing container, and the snapshot is transferred and restored in the target device. Finally, the μC updates the existing flow table entries accordingly.

(d)Dynamic management of service chains: *Amalgam* provides support for dynamic service chaining. Dynamic service chaining enables VNFs to meet changing service requirements. For instance, consider a flow passes through a firewall VNF. Based on the signature of the flow, the firewall conditionally decides to steer the flow through an additional deep packet inspector (DPI) without interrupting the flow. To implement this, *Amalgam* allows the VNFs to interact with the local μC via "`REST`" interface. The local μC can deploy the DPI if it is not available and sends the "`OFPT_FLOW_MOD`" events to

the forwarder component to enable the flow steering without terminating the flow.

(e)Flow Tags for Monitoring: Once the VNFs are placed, the path management module of μCs set-up the flow table entries of the participating forwarders via `OpenFlow` protocol. One issue regarding path management through service chains is to identify an end-to-end flow that arises in the presence of the "5-tuple" changing VNFs (e.g., Load balancer, web proxy cache, and NATs). Since such VNFs may alter the packets in unpredictable ways, fine-grained management and monitoring of the flows passing through becomes difficult. To avoid this issue, *Amalgam* attaches a "`VLAN`" tag to the packets before it enters the VNF. The nearest μC of the VNF maintains a table for flows like f^a, which keeps track of the original match field of the flow and the modified field (alias).

(f)Providing QoS: *Amalgam* is developed on top of the SDN decentralized control plane, which enables us to ensure flow specific QoS guarantees. On the other hand, since the VNF deployment is done using containers, using "`cgroups`"can ensure the VNF specific QoS like reservation of CPU, Memory, etc. The policy server module contains the "`cgroups`" parameters for each VNFs of a service chain, which is used to ensure VNF specific QoS.

IV. Prototype and Experimental Results

The existing namespace oriented emulation frameworks (e.g. Mininet) is not suitable for VNF migration and in-networking platform emulation. Therefore, we develop docker based MiniDockNet which mimics real-life VNFs using the *"Docker-in-Docker"* configuration. This feature ensures rapid deployment from MiniDockNet emulation to real in-network processing environment. To implement the VNF migration, we use standard live container migration using CRIU[6]. For the emulation of the links between any two nodes, we use *"l2tp"*

A. Experimental Setup

For experimental purpose, we use "`rocketfuel topology`"[7]. Each link is configured to emulate $3ms$ of delay and $10Mbps$ of bandwidth using linux "`tc`" utility. We use "`iperf`" to generate long flows; for shorter flows we use "`ping`". The clients and server applications are hosted on the *diameter* of the topology. For background traffic we use python based "`HTTP`" client and server.

We use "`Apache cassendra`" to implement the policy server module. Rest of the *Amalgam* modules targeted for μC are implemented on top of "`Ryu`"[8], a python based SDN controller framework. For experiments, we use 3 different VNFs (NAT (N), Load Balancer (L) and Web Proxy(W)) to create 6 different combination of service chain as given in Fig. 5. In order to ensure the confidence on the results, each experiment is repeated atleast 30 times.

[6]https://criu.org/Live_migration

[7]http://tiny.cc/nv70mz

[8]https://ryu.readthedocs.io/en/latest/

B. Results

We compare the performance of *Amalgam* with the existing *"P4"*[9] based distributed session-oriented service function chaining framework called Dysco [7]. Since, Dysco ensures session related performance and does not provide any VNF placement strategy, the performance evaluation of the proposed distributed VNF placement algorithm is done with another existing work WGT [11] which proposes a distributed heuristic for VNF placement for the multi-domain network.

1) Session Related Performances: Fig. 2 shows the comparison between Dysco and *Amalgam* in terms of flow initialization delay. We found that *Amalgam* is capable of quicker flow initialization than Dysco. This reduction in flow initialization delay comes from the parallel deployment of VNFs as opposed to the hop by hop deployment of VNFs in Dysco. The advantage of flow initialization delay becomes much evident in the case of longer service chains like C^6 than the smaller service chain like C^1.

Since *Amalgam* uses containers to deploy the VNFs as opposed to the P4 applications used in Dysco, the deployment of VNFs using *Amalgam* incurs greater latency, as shown in Fig. 3. The increase in VNF deployment time for *Amalgam* depends on the VNF container size. Therefore, the deployment latency is higher for C^6 in compared to C^1. However, in a large scale network, VNF deployment events are far rare than a flow generation event. On the other hand, the use of containers provide greater flexibility as the creation of new middlebox application using container requires less programming overhead than the creation of a new P4 application. As a result, state management during the migration of VNFs from one node to another becomes easy when they are running inside a container as compared to the P4 applications of Dysco. However, these management benefits of containers come at the cost of resource utilization.

The placement of VNFs requires resource occupancy in the deployed devices, which is an important aspect of resource constraint IoT devices. In Fig. 6, we compare the performance of *Amalgam* with Dysco in terms of CPU utilization of devices due to the placement of VNFs. In order to normalize the additional resource consumption of *Amalgam* due to the use of containers, we also compare the resource utilization of *Amalgam* without using docker. Similarly, we provide a comparison of memory utilization for *Amalgam* and Dysco in Fig. 7. Based on these two experiments, we observe that Dysco incurs less utilization of resources than the proposed *Amalgam* with the container. However, based on the *"Wilcoxon Rank Sum test"* we find that, the difference of resource utilization of *Amalgam* without Docker and Dysco is statistically insignificant (i.e. $p-value > 0.05$) for C^4, C^5 and C^6. Fig. 8 shows the comparison of throughput between *Amalgam* and Dysco. The Wilcoxson rank sum test reveals that the throughput between *Amalgam* and Dysco are statistically indistinguishable (Here our alternate hypothesis H_a is *Amalgam* provides less throughput than Dysco).

[9]https://p4.org/

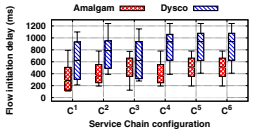


Fig. 2: Flow Initialization Delay

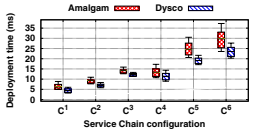


Fig. 3: Latency of Deployment

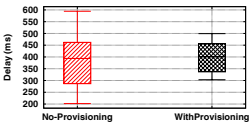


Fig. 4: Effect of QoS

Name	SFC	Name	SFC
C^1	(N)	C^2	(L)
C^3	(W)	C^4	(N→L)
C^5	(L→W)	C^6	(N→L→W)

Fig. 5: List of Service Chains

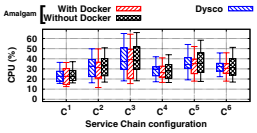


Fig. 6: CPU Utilization

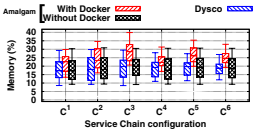


Fig. 7: Memory Utilization

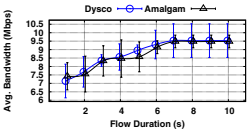


Fig. 8: Average Throughput

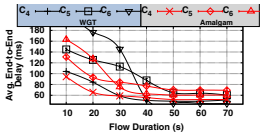


Fig. 9: Delay for C^4, C^5 and C^6

2) Performance of Distributed VNF Placement: To measure the performance of distributed VNF placement heuristic used in Amalgam, as mentioned earlier, we deploy WGT [11] on top of Dysco. However, it is difficult to deploy a centralized controller for a large scale multi-domain system. Therefore, we place the WGT in the micro-Controller (μC) nearest to the source device. We measure and compare the effect of delay for all the service chains when the flow duration increases. Based on the experimental results, we found that the effect of delay for single VNF does not change since *Amalgam* and WGT provides the same results for VNF placement. Hence, we omit the plots for C^1, C^2, C^3. For multiple VNF oriented service chains like C^4, C^5 and C^6, we provide the average end-to-end delay in Fig. 9. Based on the results, we can observe that *Amalgam* can perform significantly well for shorter flows as the iterative WGT requires a significant amount of feedback rounds to find the proper placements of VNFs.

C. QoS Provisioning

Amalgam is capable of showing QoS provisioning by reserving resources limiting CPU, memory, bandwidth, and link delay. We perform two experiments for each resource type, one with no provisioning and another with resource reservation limit set as the mean value found in the previous experiment. Based on the Wilcoxson rank-sum test on these results we found that, except the memory utilization (P-value $= 0.42$) rest of the resource reservation works significantly well (with P-value< 0.05). We also find that the resource reservation can reduce the jitter of the flow, as shown in Fig. 4.

V. Conclusion

In this paper, we present *Amalgam*, which integrates the distributed SDN orchestration framework with the distributed service chain management framework. The proposed *Amalgam* is suitable for large scale multi-domain IoT in-networking platforms. We also provide a distributed heuristics for the placement of constituent VNFs of service chains. The lack of an existing emulation platform for container oriented VNF service chain has motivated us to develop "MiniDockNet". Using this emulation platform, we found that *Amalgam* incurs a lesser flow initialization delay than that of a very recent distributed service chain management framework (Dysco). We also show that *Amalgam* is capable of ensuring less end-to-end delay for short flows.

References

[1] F. Duchene, D. Lebrun, and O. Bonaventure, "Srv6pipes: enabling in-network bytestream functions," in *17th IFIP Networking Conference and Workshops*, May 2018, pp. 1–9.

[2] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service function chaining use cases in mobile networks," *IETF*, 2015.

[3] D. Bhamare, R. Jain, M. Samaka, and A. Erbad, "A survey on service function chaining," *J. Netw. Comput. Appl.*, vol. 75, no. C, pp. 138–155, nov 2016.

[4] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *10th USENIX Symposium on NSDI*. Lombard, IL: USENIX, 2013, pp. 227–240.

[5] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," *SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 163–174, 2015.

[6] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *The 21st IEEE International Workshop on LAN/MAN*, April 2015, pp. 1–6.

[7] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford, "Dynamic service chaining with dysco," in *30th Proceedings of the ACM SIGCOMM*. ACM, 2017, pp. 57–70.

[8] IETF, "Rfc 8300 - network service header (NSH)," Sept 2019, [accessed 10. Sept. 2019]. [Online]. Available: https://tools.ietf.org/html/rfc8300

[9] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *11th USENIX Symposium on NSDI*. Seattle, WA: USENIX Association, 2014, pp. 543–546.

[10] M. Ghaznavi, N. Shahriar, S. Kamali, R. Ahmed, and R. Boutaba, "Distributed service function chaining," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2479–2489, Nov 2017.

[11] G. Sun, Y. Li, D. Liao, and V. Chang, "Service function chain orchestration across multiple domains: A full mesh aggregation approach," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1175–1191, Sep. 2018.

[12] 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 Transactions on Networking*, vol. 21, no. 6, pp. 1960–1973, Dec 2013.

[13] A. Wion, M. Bouet, L. Iannone, and V. Conan, "Let there be chaining: How to augment your igp to chain your services," in *18th IFIP Networking Conference*, May 2019, pp. 1–9.

[14] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, "Stratos: A network-aware orchestration layer for middleboxes in the cloud," *arXiv CoRR*, vol. abs/1305.0209, 2013.

[15] M. Charikar, Y. Naamad, J. Rexford, and X. K. Zou, "Multi-commodity flow with in-network processing," in *Algorithmic Aspects of Cloud Computing*. Cham: Springer International Publishing, 2019, pp. 73–101.

[16] S. Chattopadhyay, S. Chatterjee, S. Nandi, and S. Chakraborty, "Aloe: An elastic auto-scaled and self-stabilized orchestration framework for iot applications," in *37th IEEE Proceedings of INFOCOM*. Paris, France: IEEE, 2019, pp. 802–810.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

Alleviating Hidden and Exposed Nodes in High-Throughput Wireless Mesh Networks

Sandip Chakraborty, *Member, IEEE*, Sukumar Nandi, *Senior Member, IEEE*, Subhrendu Chattopadhyay

***Abstract*—This paper proposes an opportunistic approach to mitigate the hidden and exposed node problem in a high-throughput mesh network, by exploiting the frame aggregation and block acknowledgment (BACK) capabilities of IEEE 802.11n/ac wireless networking standard. Hidden nodes significantly drop down the throughput of a wireless mesh network by increasing data loss due to collision, whereas exposed nodes cause under-utilization of the achievable network capacity. The problem becomes worse in IEEE 802.11n/ac supported high-throughput mesh networks, due to the large physical layer frame size and prolonged channel reservation from frame aggregation. The proposed approach uses the standard 'Carrier Sense Multiple Access' (CSMA) technology along with an 'Opportunistic Collision Avoidance' (OCA) method that blocks the communication for hidden nodes and opportunistically allows exposed nodes to communicate with the peers. The performance of the proposed CSMA/OCA mechanism for high throughput mesh networks is studied using the results from an IEEE 802.11n+s wireless mesh networking testbed, and the scalability of the scheme has been analyzed using simulation results.**

***Keywords*-IEEE 802.11n; Frame Aggregation; Hidden and Exposed Nodes; Spatial Reuse**

I. Introduction

Wireless Mesh Networking [1] technology provides a key milestone to the next generation backbone access network, where a 'mesh' of wireless routers supports the backbone connectivity to the end-users through multi-hop communications. To assist commercial, enterprise and community mesh backbone technology, IEEE 802.11s [2] is gaining significant attention among the developers for its compatibility with commercially successful IEEE 802.11 wireless networking standard. The IEEE 802.11s amendment to the wireless networking standard contributes to the medium access control (MAC) specifications for the wireless mesh access, that can operate along with any physical layer technology supported by the IEEE 802.11 task groups. The recent developments over IEEE 802.11 technology enhances wireless communications for high data rate supports, up to 600 Mbps with IEEE 802.11n, and up to 6.77 Gbps with IEEE 802.11ac [3]. Therefore the mesh networking standard along with high data rate support has exorbitant potentials to provide high-throughput backbone network connectivity to the end users, that effectively directs towards the design of 'wireless world' [4].

The IEEE 802.11 coordination function for wireless channel access relies on 'Listen before Talk' which is known as 'Carrier Sense Multiple Access with Collision Avoidance' (CSMA/CA). The clear channel assessment (CCA) in CSMA/CA uses two listen or carrier sensing (CS) mechanism - physical CS (PCS) and virtual CS (VCS). Extensive studies on IEEE 802.11 CCA technologies have revealed that PCS fails to detect nodes which are beyond the CS range (known as hidden nodes), whereas VCS reduces spatial reuse by blocking the communications which are not susceptible to interference (known as exposed nodes) [5]. The performance impacts of hidden and exposed nodes in a network is characterized by the CS range and the interference range. In PCS, a node can initiate a new communication only if all other nodes in its CS range are inactive. However, the communication becomes successful only if the receiver is not within the interference range of any other active nodes in the network. As CCA is performed at the transmitter, whereas the receiver is susceptible to interference, there is a high probability of hidden nodes in a mesh network. VCS [6] is proposed to solve the hidden node problem through a handshaking between the transmitter and the receiver before the actual communication takes place, through a pair of messages termed as 'Request to Send' (RTS) and 'Clear to Send' (CTS). The transmitter sends a RTS message only if the CCA reports no active nodes in the CS range of the transmitter. The receiver replies back with a CTS message only if there are no active nodes within the CS range of the receiver. On overhearing the RTS and the CTS messages, other nodes within the CS range of transmitter and receiver block their communication for the time duration mentioned within the RTS and the CTS messages. As a consequence, VCS introduces the problem of exposed nodes which are outside the receiver's interference range, but within the CS range of the transmitter. Though the exposed nodes are harmless for an ongoing communication, they are blocked as a result of VCS and therefore cause under-utilization of the effective network capacity.

The performance issues experienced by hidden and exposed nodes in a multi-hop and mesh network have been well studied in the literature. Most of the existing approaches design mechanisms to eliminate either the hidden nodes or the exposed nodes, such as busy tone signaling [7], use of separate control channels [8], adaptation in CS threshold and temporary disabling the CS mechanism [9] etc. However, none of these approaches can eliminate hidden and exposed nodes simultaneously. In fact, existing research [9] has shown that it

Sandip Chakraborty is with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur 721302, INDIA (e-mail: sandipc@cse.iitkgp.ernet.in)

Sukumar Nandi and Subhrendu Chattopadhyay are with the Department Computer Science and Engineering, Indian Institute of Technology Guwahati, Guwahati 781039. INDIA (e-mail: {sukumar,subhrendu}@iitg.ernet.in)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

is very difficult to eliminate hidden and exposed nodes simultaneously, as these are two complementary problems. Further in general for a network with low to moderate traffic load, VCS may show negative impact (considerably less performance than PCS) because of RTS-CTS signaling overhead and large number of exposed nodes [10]. As a consequence, researchers have developed adaptive PCS mechanisms [11] where the CS range is dynamically tuned to reduce the number of hidden nodes in a network. In adaptive PCS, transmit power is tuned in such a way so that the CS range becomes almost equal to the interference range. In this way, the hidden nodes which are within the interference range, but outside the CS range of the transmitter, can be eliminated. Nevertheless, these mechanisms of dynamic CS range can not eliminate the hidden nodes which are within the CS range (or the interference range, assuming both are tuned to be same) of the receiver, but outside the CS range of the transmitter. In [12], the authors have proposed an access point (AP) cooperation method, where the APs share some common information among themselves to detect the hidden and exposed nodes. However, their scheme requires synchronization among the APs which is hard to achieve in a mesh network.

Our earlier paper [13] presents a theoretical model for the performance of IEEE 802.11n supported mesh networks in the presence of hidden and exposed terminals. The high throughput wireless networking technologies over IEEE 802.11n/ac support MAC layer frame aggregation and block acknowledgment (BACK) strategies to improve channel access performance by minimizing the MAC layer overhead [3]. The analysis reveals that the MAC layer frame aggregation and BACK are more susceptible to performance losses in the presence of hidden and exposed terminals. IEEE 802.11n supports a frame aggregation limit of 64 KB. Further in IEEE 802.11n/ac, high data rate support is obtained at the cost of higher and collocated channel losses during interference [14] due to channel bonding [1]. Though individual MAC protocol data units (MPDU) in an aggregated frame (called A-MPDU) can be recovered from random losses, a collocated loss due to collision from hidden terminals destroys a consecutive numbers of MPDUs. Further a BACK loss due to collision from hidden terminals may result in as large as 64 KB data loss, as the complete A-MPDU is required to be retransferred for a BACK loss. Similarly in VCS, a single exposed node unnecessary defers its transmission for a long time. For example with 1500 bytes MAC frame size and 32 Mbps data rate with IEEE 802.11g, the exposed node has to wait for approximately 0.04 ms, whereas with 64 KB MAC aggregated frame size and 600 Mbps data rate with IEEE 802.11n, an exposed node has to wait for approximately 0.1 ms, 2.5 times more. Possibility of an exposed nodes in a mesh network with VCS is very high. As shown in [13], reducing hidden nodes through a tunable CS threshold only is not sufficient to have a reasonable performance benefit in high throughput mesh networks, whereas pure VCS may show negative impact if number of exposed nodes crosses a limit. Considering the extortionate data loss due to hidden nodes in a high throughput mesh network with PCS, and possibility of capacity under-utilization due to exposed nodes in VCS, a scheme should be designed that reduces hidden nodes as much as possible, whereas allows communication opportunities to the exposed nodes.

This paper shows the effect of hidden and exposed terminals over the performance of high throughput mesh networks, using results from a testbed. An improved channel access mechanism is designed in this paper on the top of VCS, utilizing the frame aggregation and BACK strategies in high throughput wireless standards. The proposed mechanism, called CSMA with Opportunistic CA (CSMA/OCA) characterizes the exposed nodes based on their collision properties, and classifies them in transmitter side exposed (T-Exposed, the nodes within the CS range of the transmitter) and receiver side exposed (R-Exposed, the nodes within the CS range of the receiver) nodes. The CSMA mechanism is augmented to allow transmission from the T-exposed nodes which are outside the CS range of the receiver, and the R-exposed nodes which are outside the CS range of the transmitter. This augmentation in CSMA mechanism allows communication for the exposed nodes those do not result in a collision during MPDU communication. However, collision is still possible between MPDUs from one node, and control frames (BACK, RTS or CTS) from another. An opportunistic collision avoidance (OCA) mechanism is proposed in this paper, where the transmit power levels of control frames are determined in an adaptive and coordinated way, such that data-control collision and control packet loss can be avoided as much as possible. The performance of the proposed scheme is analyzed through the results obtained from a practical IEEE 802.11n mesh networking testbed. The scalability of the proposed scheme has been analyzed using simulation results.

II. Effect of Hidden and Exposed Nodes in a High Speed Mesh Network: PCS vs VCS

A number of works, such as [10] and the references therein, have shown that VCS may employ negative impact on channel access overhead due to signaling overhead and exposed nodes. Tinnirello *et al* [16] have shown both analytically and by simulation results that the situation may become worse in case of high speed networks, as the assumption of short transmission time for control frames does not hold for high speed networks. In this section, we report a series of results from a practical IEEE 802.11n+s testbed to analyze the impact of PCS and VCS over high speed multi-hop and mesh wireless networks.

A. Experimental Setups

In these experiments, we use IEEE 802.11n supported Ralink (presently MediaTek) RT-3352 router-on-chip [17] as the core of the wireless routers. The RT-3352 router-on-chip combines 802.11n draft compliant $2T2R$ MAC along with BBP/PA/RF MIMO, a high performance 400MHz

[1]In IEEE 802.11n, two 20 MHz channels are combined to obtain a 40 MHz channel, whereas in IEEE 802.11ac, four 20 MHz channels are combined to use a single 80 MHz channel. It is well known, that wider channels are more susceptible to random as well as collocated channel losses from interference [15].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

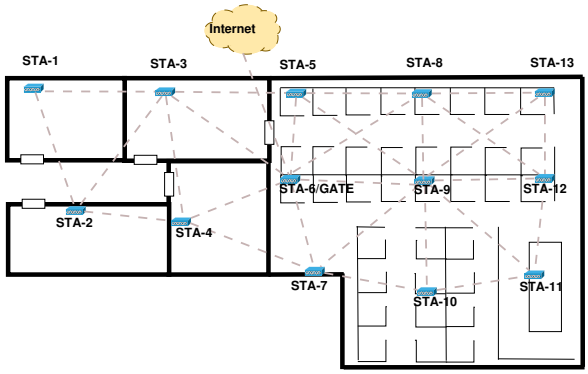


Fig. 1. High Throughput Mesh Testbed and Connectivity Layout

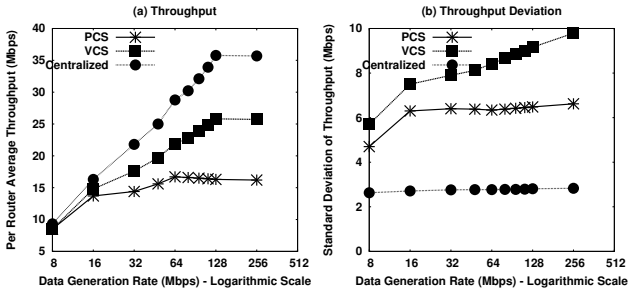


Fig. 2. Comparison among PCS, VCS and Centralized

MIPS24KEc CPU core, a Gigabit Ethernet MAC, 5-ports integrated 10/100 Ethernet Swtich/PHY, 64MB of SDRAM and 32MB of Flash. This chip can support up to 300 Mbps data rate with maximum transmit power of $16dBm$. The routers are configured with Linux kernel version 3.12 with open80211s support [18] which provides open source implementation of IEEE 802.11s mesh networking protocol stack.

B. Experimental Results

Fig. 1 shows the deployment and connectivity layout for a 13 node IEEE 802.11n+s mesh testbed, where 12 nodes act as mesh routers, and one node (STA-6) connects the mesh backbone to the Internet. All 13 nodes are able to act as an access point (AP) to the client nodes to provide backbone mesh connectivity. The dotted lines in the figure shows connectivity among the mesh routers based on the maximum transmit power and receiver sensitivity setup (maximum transmit power 16 dBm, receiver sensitivity -81 dBm). In the testbed, we have used two types of data flows generated through `iperf` tool - the upload traffic flow and the download traffic flow. The upload traffic flows are from mesh routers to the mesh gate (STA-6) and the download traffic flows are from mesh gates (STA-6) to mesh routers. Out of the total traffic flows, 60% data are from download traffic and 40% data are for upload traffic. The performance is evaluated by varying the data generation rate which in turn changes the traffic load of the network. The performance of PCS and VCS is evaluated with a comparison to an centralized scheduling scenario [19] which is generated based on a centralized formulation. The centralized formulation uses a spatial time division multiple access (STDMA) based scheduling, where exposed nodes are allowed to communicate while hidden nodes are blocked explicitly. This scheduling mechanism gives a theoretical upper bound along with protocol overhead, for IEEE 802.11 access scheduling. Therefore, we consider this scheme as the benchmark for performance comparison in this paper, and show how close our proposed distributed scheme can perform compared to the centralized solution. During the design of the centralized scheduling, we have assumed interference range equals to the CS range, as interference beyond the CS range (such as, additive interference) is difficult to design in a decentralized system [20]. In the testbed evaluation, the A-MPDU aggregation level is considered to be 64 KB. The effect of dynamic rate selection is also evaluated, and the performance data is reported later in this paper.

Fig. 2(a) depicts the channel access strategies for three mechanisms - PCS, VCS and the centralized access [19], with respect to average per router throughput. The x-axis shows the mean data generation rate, where every router generates data based on a log-normal distribution along the mean with standard deviation as 2 Mbps. The data generation rate does not include the forwarding traffic. Fig. 2(b) compares the three mechanisms with respect to the standard deviation of routers' throughput. While the average throughput shows the network performance, the standard deviation of throughput indicates effect of hidden and exposed terminals over the throughput performance. As all the nodes transmit traffic towards STA-6, the effect of hidden and exposed terminals is less for the nodes which are in one-hop away from STA-6. Fig. 2(a) shows that throughput loss becomes substantial at high traffic load, when the data generation rate is more than 10 Mbps. Though VCS improves performance at high traffic load by reducing hidden terminals, the throughput is still significantly less compared to the centralized scenario. At the same time, Fig. 2(b) indicates that the throughput deviation is very high for PCS and VCS at high traffic load. From extensive analysis of router traces, we have found that packet loss in PCS is very high for the nodes which experience hidden terminals. However, throughput variation for VCS is higher than PCS, that indicates severe network under-utilization at the presence of exposed nodes. In a high speed mesh network with high traffic load, a single exposed node at every transmission opportunity[2] may lead to network under-utilization equivalent to 64 KB data transmission. As a consequence, the throughput deviation inflates at high traffic load.

The above results indicate that IEEE 802.11 PCS and VCS fail to achieve even half of the centralized throughput in a high data rate wireless mesh network. Therefore, the standard CS mechanisms are required to be revisited for implementing high throughput mesh networks based on IEEE 802.11 standard. The target is to block hidden nodes to avoid data losses due to collision, and to allow transmissions to the exposed nodes for effective utilization of available network capacity. However, enabling communication to all the exposed nodes may create additional problems in a mesh network, as discussed in the

[2]A transmission opportunity indicates the time required to transmit an A-MPDU once a node gains access to the channel

next section.

III. Exposed Nodes: Spatial Transmission and Data-Control Collisions

Modern hardware supports with dynamic power adaptation and capture enabled devices [21] can reduce interference from hidden nodes by tuning CS range to make it equivalent to the interference range, although cannot eliminate them completely. Whereas receiver coordination similar to VCS is necessary to detect the nodes which are outside the CS range of the transmitter, however within the CS range (or interference range) of the receiver. In this design, we consider a network, where CS range is tuned to make it equal to the interference range through some existing mechanism similar to [22], and our target is to further reduce the hidden nodes which are beyond the scope of detection through tuning the CS range, while allowing communication to the exposed nodes as much as possible. The proposed CSMA/OCA protocol works on the top of VCS to improve the performance of high throughput mesh networks by reducing number of hidden nodes and opportunistically allowing communication to the exposed nodes.

As a protocol level simplification, CSMA/OCA considers only the nodes which are within the CS range, and are detectable through either PCS or VCS. The proposed protocol overlooks interference from outside the CS range (additive interference) which is not pre-detectable through PCS or VCS and difficult to capture in a real-time environment. For this purpose, this paper differentiates the term 'collision' and 'interference' as follows: collision is considered for the nodes which are within the CS range and are detectable, whereas interference is considered for the nodes which are outside the CS range, but results in data loss due to additive interference.

During PCS, the nodes which are within the CS (or interference) range of the receiver, but are outside the CS range of the transmitter can be a potential hidden node, as any transmission from it remains undetectable to the transmitter though it causes collision at the receiver. The VCS solves this problem by blocking all the nodes which are within the CS range of both the transmitter and the receiver. As a consequence exposed nodes may result network under-utilization as follows:

1) The nodes which are within the CS range of a transmitter, say T_1, but are outside the CS (or interference) range of a receiver, get blocked by overhearing the RTS message from T_1. These nodes can be a potential transmitter for some other nodes in the network, as they are not within the interference range of a receiver. These nodes are termed as *T-Exposed* nodes.
2) The nodes which are within the CS (or interference) range of the receiver, say R_1, however are outside the CS range of a transmitter, get blocked by overhearing the CTS message from R_1. Though these nodes can not be a transmitter, nevertheless they can be a potential receiver for some other nodes outside the CS range of R_1. These nodes are termed as *R-Exposed* nodes.

Fig. 3-Case 1 shows transmission and collision scenarios for T-Exposed nodes. The dark and light circles denote the transmission range and the CS range respectively. $C \rightarrow D$ communication is feasible as the data communications do not overlap at the receiver, and therefore collision does not exist. However, VCS blocks such communication resulting in node C to be an T-Exposed node. Although, allowing communication to such node may result in data-control collision, as shown in Fig. 3-Case 1(b). The BACK from node B may collide with the data from node C. Similar situation may arise for the CTS from node D. Therefore, special care has to be taken while allowing transmissions to the T-Exposed nodes. Fig. 3-Case 2 shows transmission and collision scenarios for R-Exposed nodes. The communications $A \rightarrow B$ and $C \rightarrow D$ does not result in a collision. However in VCS, node D gets blocked on overhearing the CTS frame from node B. Similar to the earlier scenario, allowing communication to R-Exposed nodes may result in a data-control collision, where one MPDU or two consecutive MPDUs may get lost in an A-MPDU. Based on these collision scenarios, an opportunistic access mechanism is proposed in this paper, as discussed in the next section.

IV. Opportunistic Access: Design and Analysis

The discussions till now have shown that the exposed nodes result in severe under-utilization of spatial reuse in high data rate mesh networks. Though VCS suffers from the signaling overheads, with a little modification in the CTS frame structure, the exposed nodes can be detected within the CS range of the transmitter and the receiver. However, allowing transmissions to all the exposed nodes may result in data-control collision, as shown in Fig. 3. Therefore, an opportunistic access mechanism is introduced in this section to deal with this problem. The opportunistic access mechanism utilizes VCS, where RTS and CTS messages are used for the detection of hidden nodes, as well as allowing transmission to the exposed nodes in an opportunistic way such that data-control collision can be minimized. The OCA mechanism uses an adaptive and coordinated transmit power calculation mechanism, where the control frames are transmitted based on a predefined power level such that data-control collisions can be avoided as much as possible. The detailed design of the opportunistic access mechanism is discussed in the following subsections.

A. Carrier Sensing: Detection of Hidden and Exposed Nodes

The legacy VCS mechanism uses a table, called the *network allocation vector* (NAV), to maintain the transmission blockage on overhearing the RTS and the CTS frames. The RTS and the CTS frames contain a duration field (DU) which indicates the time duration for the channel reservation. On overhearing the RTS and the CTS frames, every node sets its NAV for the time mentioned in the DU field. In the proposed augmentation of VCS, different NAVs are maintained for every overheard RTS and CTS frame, to detect the exposed nodes and to avoid the data-control collision. Let, RTS_{act} and CTS_{act} denote the sets of nodes from which a node has received the RTS or CTS frames, respectively, and $\mathcal{F}.DST$ denote the destination address associated with the frame $\mathcal{F}$. Further assume that $\mathcal{N}_i$ denotes the set of nodes which are in the CS range of node

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

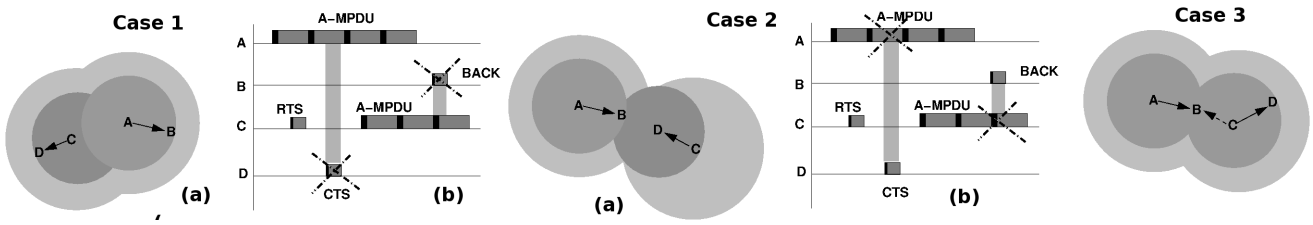


Fig. 3. **Case 1:** Spatial Transmissions and Collision Scenario for T-Exposed Nodes, **Case 2:** Spatial Transmissions and Collision Scenario for R-Exposed Nodes, **Case 3:** Hidden Node Scenario

Algorithm 1 Node S wants to transmit data to node R

1: **if** $(CTS_{act} = NULL) \wedge (\forall RTS_{act}.DST \notin \mathcal{N}_S)$ **then**
2: Calculate η_S.
3: Send <RTS,$\eta_S >$ using P_{max}; /*T-exposed nodes*/
4: **else**
5: Back-Off and retry;
6: **end if**

Algorithm 2 Node R receives RTS from node S

1: **if** $(RTS_{act} = NULL) \wedge (\forall CTS_{act}.DST \notin \mathcal{N}_R)$ **then**
2: Calculate $P_{Tx}^{(R)}, \eta_R$;
3: **if** $P_{Tx}^{(R)} \in \mathcal{P}$ **then**
4: Send $< CTS, \eta_R >$ using $P_{Tx}^{(R)}$; /*R-exposed nodes*/
5: **end if**
6: **end if**
7: $RTS_{act} \leftarrow this.RTS$ /*Append the received RTS in RTS_{act}*/

Algorithm 3 Node S receives CTS from node R

1: **if** $(CTS_{act} = NULL) \wedge (\forall RTS_{act}.DST \notin \mathcal{N}_S)$ **then**
2: Calculate $P_{Tx}^{(S)}$.
3: Send DATA using $P_{Tx}^{(S)}$;
4: **else**
5: Back-Off and retry;
6: **end if**
7: $CTS_{act} \leftarrow this.CTS$ /*Append the received CTS in CTS_{act}*/

i. This set can be populated using the standard IEEE 802.11 beaconing procedure.

We also include an additional 1 byte field, called background noise level (denoted as η_i for node i), at every control frame header. The background noise level indicates the total incident power on the receiver, before the communication starts. The background noise level is measured in terms of received signal strength, expressed in dBm, before transmitting a control packet. Assume that node B is an intended receiver. Therefore following the VCS mechanism, node B needs to transmit a CTS packet after it receives a RTS successfully. The received signal strength at node B is the indication of channel noise near the receiver. This information is propagated to the transmitter, in the form of background noise level, so that it can adjust its power level accordingly. This background noise level value is used for OCA along with the VCS mechanism, as discussed in the subsequent subsection.

The decision controls for hidden and exposed node detection are explained through Algorithm 1, Algorithm 2 and Algorithm 3. According to these algorithms, every data and control frames are transmitted using a predefined power level. The actual power level to transmit the data and control frames are calculated based on the OCA mechanism, as discussed later. In these algorithms, $P_{Tx}^{(i)}$ denotes the transmit power for the data or control packet to be transmitted, and P_{max} denotes the maximum available power level. $\mathcal{P}$ denotes the set of available power levels according to the physical layer modulation and coding technology.

The control statements in these algorithms define whether a node should initiate a communication based on the VCS mechanism. As mentioned earlier, whenever a node overhears a RTS or CTS control packet, it appends or updates the RTS_{act} and CTS_{act} sets accordingly. The lifetime of an entry in these sets is equal to the duration field mentioned in the corresponding RTS or CTS control packets, that denotes how much time a channel is going to be reserved by the corresponding node. Furthermore, it can be noted that a RTS transmission does not always indicate a data communication. It may happen that the sender fails to receive the CTS packet from the intended receiver due to a busy channel or interference. Therefore, after an entry is appended or updated in the RTS_{act} set, a node senses the channel after a timeout interval[3]. If the channel is found to be idle, the corresponding entry from RTS_{act} is deleted. These three algorithms can correctly identify the hidden nodes as well as both types of exposed nodes, as described in the following subsections.

1) Detection of Hidden Nodes: The hidden node scenario can occur for two cases - (i) a transmitter node is within the CS range of another receiver node, and (ii) a receiver node is within the CS range of another transmitter node. For the first scenario, considering Fig. 3-Case 3, let $A \rightarrow B$ communication starts first. Therefore, node C can overhear the CTS frame from node B, which is included in the CTS_{act} set. According to Algorithm 1, the condition is evaluated to be false, as $CTS_{act} \neq NULL$, and node C starts the back-off without initiating the communication.

For the second scenario, let us assume that $C \rightarrow D$ communication starts first. Node B is outside the CS range of node D, but within the CS range of node C. In this scenario node B acts as a receiver, and therefore, should not initiate a communication. Node B can overhear the RTS from node C, and includes it in the RTS_{act} set. On receiving the RTS frame from node A, node B executes Algorithm 2. However, the condition is evaluated to be false as $RTS_{act} \neq NULL$.

[3]In our implementation, this timeout is kept equal to the duration of $3 \times SIFS + T_{CTS}$, where SIFS is the short inter-frame space duration and T_{CTS} is the duration of a CTS frame in time unit (TU).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

Therefore it does not replies back with the CTS, and the communication is deferred avoiding possible collision due to hidden terminals.

2) Spatial Reuse by T-exposed Nodes: Considering Fig. 3-Case 1, let $A \rightarrow B$ communication starts first. Node C within the CS range of node A wants to initiate a transmission with node D, which is outside the CS range of node A. As node C is outside the CS range of node B, it does not receive any CTS. Further, $RTS_A.DST \neq C$. Therefore, the condition in Algorithm 1 is evaluated to be true, and node C forwards the RTS to node D. Node D is outside the CS range of both the nodes A and B. Therefore, the condition of Algorithm 2 is also evaluated to be true, and D replies back with the CTS, resulting in a communication initialization at node C following Algorithm 3.

3) Spatial Reuse by R-exposed Nodes: Considering Fig. 3-Case 2, let $A \rightarrow B$ communication starts first. As node D is outside the CS range of node A, it does not overhear the RTS. Therefore $RTS_{act} = NULL$. Assume, node D receives an RTS from node C. As node D is within the CS range of node B, it overhears the CTS from node B. However, $CTS_B.DST = A \notin \mathcal{N}_D$. Therefore following Algorithm 2, node D replies back with the CTS, resulting in the communication initialization at node C, as shown in Algorithm 3.

B. Opportunistic Collision Avoidance

As discussed earlier, allowing communications to the exposed nodes may result in data-control collision. The T-exposed nodes may result in control frame loss, whereas, the R-exposed nodes may affect data frames. To avoid such losses, we employ the dynamic power adaptation feature available with the commodity wireless routers. The modern commodity router hardwares support a set of transmit power levels for every physical modulation and coding schemes, that can be tuned by the MAC layer driver module. A higher transmit power improves the possibility of correct reception and decoding of a frame, however increases the interference range due to that communication. A frame can be received and decoded correctly if the received signal to interference and noise ratio (SINR) for that frame is beyond a certain threshold, called the *capture threshold*. For a communication $S \rightarrow R$ between two nodes S and R, this can be expressed as;

$$SINR_R = \frac{P_{tx}^{(S)} G_{SR}}{\eta_R} \geq \mathcal{S}_{thresh} \tag{1}$$

where $\mathcal{S}_{thresh}$ is the capture threshold required to correctly decode the frame, $P_{tx}^{(S)}$ is the transmit power level at node S, G_{SR} is the channel gain between S and R. As mentioned earlier, η_R is the background noise level at the receiver R, that denotes total power at the channel just before node R receives the data frame from S. This implies the total noise power contributed by all other communications except $S \rightarrow R$, near the node R. This noise power interferes with the $S \rightarrow R$ communication.

The OCA mechanism proposed in this paper dynamically adjusts the power level for the frames which may undergo a collision. Otherwise the frames are transmitted using the minimum power level such that the communication can sustain. Overall, the OCA mechanism works as follows:

- CTS and BACK control frames may get lost due to probable collision from a T-Exposed node, as shown in Fig. 3-Case 1. Therefore these control frames are transmitted using a higher power level, whenever the background noise level calculated near the sender is high enough. This power level is decided based on the sender side background noise level information (η) transmitted through the RTS and data frames. If the calculated power level is higher than the available power levels, then the communication is not initiated.
- Data frames may get lost due to probable collision from a R-exposed node, as shown in Fig. 3-Case 2. However, in this scenario, the sender side background noise level is very low, and therefore CTS and BACK control frames are transmitted using the minimum power level such that the communication can sustain. As a consequence, the possibility of data loss due to R-exposed nodes can be reduced.
- The RTS frames are transmitted using maximum available power level (P_{max}), since the maximum power level gives maximum possibility of receiving a frame correctly. The RTS frames are required to be received correctly for further coordination in the OCA mechanism.

The adjustment of the transmit power level for a CTS frame works as follows. Let, node S want to communicate to node R, and accordingly transmit the RTS frame with maximum power level P_{max}. The RTS frame also contains the parameter η_S, the background noise level at node S measured just before transmitting the RTS frame. On reception of the RTS frame, let the device driver of node R reports measured SINR as $SINR_{RTS}$. Then,

$$SINR_{RTS} = \frac{P_{max} G_{SR}}{\eta_R} \tag{2}$$

where η_R is the background noise level at node R measured before the reception of the RTS frame. It can be noted that the device driver of every node i periodically measures the background noise level and stores the value in the variable η_i. The background noise level is also used in the standard PCS and VCS mechanism to check whether the channel is free [23], [16]. For instance, the channel is assumed to be free if η_i is less than the CS threshold.

From Equation (2), node R can calculate the value of the channel gain G_{SR}. Now, let $P_{tx}^{(R)}$ denote the transmit power level of node R, for forwarding the CTS frame to node S. For correct reception and decoding of the CTS frame at node S, Equation (1) needs to be satisfied. Therefore,

$$\frac{P_{tx}^{(R)}\, G_{RS}}{\eta_S} \geq \mathcal{S}_{thresh} \tag{3}$$

Thus,

$$P_{tx}^{(R)} \geq \frac{\mathcal{S}_{thresh}\, \eta_S}{G_{RS}} \tag{4}$$

Since the communication interfaces for all the nodes are assumed to be homogeneous and the VCS mechanism has

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

already blocked the communications from the hidden nodes resulting in no hidden node interference, we can assume $G_{RS} \approx G_{SR}$. So,

$$P_{tx}^{(R)} \geq \frac{\mathcal{S}_{thresh}\ \eta_S\ P_{max}}{SINR_{RTS}\ \eta_R} \tag{5}$$

The CTS frames are transmitted only if $P_{tx}^{(R)} \leq P_{max}$. Otherwise, the communication is not initiated. As discussed earlier, a commodity wireless router supports a set of discrete power levels $\mathcal{P}$. Therefore, $P_{tx}^{(R)}$ is approximated to the next available power level from $\mathcal{P}$, higher than $P_{tx}^{(R)}$.

The transmit power levels for the BACK and data frames are calculated in a similar way. The power adaptation method opportunistically handles both the T-Exposed and R-Exposed nodes. For T-Exposed nodes, η_S becomes high. Therefore CTS and BACK frames are transmitted at a higher power level such that the control frames can be received and decoded correctly. For R-Exposed nodes, η_S is very low. Therefore, CTS and BACK frames are transmitted at lower power levels, reducing the possibility of collision with the data frames from the R-Exposed node.

1) Implementation Issue 1 (Measurement of Background Noise Level and SINR): The proposed OCA scheme relies on the measurement of background noise level and the calculated SINR value. In this work, we use an active measurement technique [24], [25] for measuring the received signal strength before the data communication, which is expressed in terms of background noise level. The MadWiFi like device drivers use a hardware abstraction layer (HAL) that measures the difference between the signal level and the noise level for each packet [26]. This measured value is used as the SINR value in our proposed scheme. Further, the noise level of the channel is measured in each interrupt, after reception of a sequence of packets. In our scheme, the latest channel noise reported by the HAL is used as the value of background noise level, expressed in terms of dBm.

However, it can be noted that both the SINR and the channel noise level fluctuates irregularly. To avoid sudden peaks of SINR and background noise level fluctuations, we use an exponentially weighted moving average (EWMA) mechanism to smooth down the reported SINR and background noise level values. Let $\mathcal{S}_{curr}$ is the reported SINR value by the HAL on receiving a packet, and $\mathcal{S}_{sm}$ is the smoothed average value. $\mathcal{D}_{sm}$ is the deviation in estimated SINR. We use the following iterative equations to estimate the smoothed value of SINR.

$$\mathcal{S}_{sm} = (1-\rho)\mathcal{S}_{curr} + \rho\mathcal{S}_{sm} \tag{6}$$

$$\mathcal{D}_{sm} = (1-\rho)|\mathcal{S}_{sm} - \mathcal{S}_{curr}| + \rho\mathcal{D}_{sm} \tag{7}$$

$$\mathcal{S}_{sm} = \mathcal{S}_{sm} - \mu\mathcal{D}_{sm} \tag{8}$$

Here ρ and μ are design parameters, and chosen to give the effect of previous measurement over the current measurement. In OCA, we use $\rho = 0.1$ and $\mu = 1$. These values are chosen based on experimental experiences that gives good smoothing by avoiding the sudden SINR peaks due irregular fluctuations. The calculated background noise level is also smoothed in a similar way with the smoothing parameters chosen as $\rho = 0.2$ and $\mu = 0.8$.

2) Implementation Issue 2 (Adaptive Modulation and Coding in IEEE 802.11): IEEE 802.11 supports adaptive modulation and coding (AMC) where every node has the flexibility to select the modulation and coding scheme (MCS) based on the channel quality and several other performance factors. The physical data rate and the transmit power depends on the selected MCS level. Therefore in a scheme where dynamic power selection is used to mitigate interference, AMC needs to collaborate with the transmit power selection mechanism to choose the appropriate MCS. It can be noted that for the control frames, the AMC always selects the MCS that provides the lowest data rate, because the lowest data rate sustain for maximum channel noise. Therefore, in the proposed scheme, the transmit power for the control frames is selected from the available power levels supported by the MCS corresponding to the lowest data rate.

The problem is non-trivial for the data frames. Equation (5) gives the minimum transmit power level required for successful decoding of the signal at the receiver. In the implementation of CSMA/OCA, once the minimum transmit power level is decided for a communication session, the AMC selects the MCS level that supports transmit power level greater than $P_{tx}^{(R)}$. The proposed CSMA/OCA and AMC works together as follows:

1) CSMA/OCA selects $P_{tx}^{(R)}$ according to equation (5).
2) AMC considers the MCS levels that support transmit power levels greater than or equals to $P_{tx}^{(R)}$. From this reduced set of supported MCS levels, the best MCS is chosen following the AMC algorithms. Let the selected MCS level be $\mathcal{M}$.
3) $\mathcal{M}$ supports a set of transmit power levels. Let it be $\mathcal{P}_{\mathcal{M}}$. CSMA/OCA selects the minimum power level from $\mathcal{P}_{\mathcal{M}}$ which is greater than or equals to $P_{tx}^{(R)}$.

Following this procedure, the proposed CSMA/OCA works together with the standard AMC procedure of IEEE 802.11.

V. Performance Evaluation of CSMA/OCA

The 13 node IEEE 802.11n+s high data rate mesh networking testbed, as shown in Fig. 1 is used to evaluated the performance of the proposed scheme and compare the results with the standard PCS, VCS as well as with the centralized solution as discussed earlier [19]. The performance is also compared with an adaptive PCS mechanism proposed by Park *et al* [27] and the AP coordination mechanism proposed by Nishide *et al* [12]. In [27], the authors have proposed a distributed carrier sense update algorithm (DCUA), that adaptively tunes the transmit power and carrier sense threshold of every node based on a pricing function calculated using the packet error rate. On the contrary, the AP coordination mechanism [12] detects hidden and exposed nodes based on the frame collision probability received through data and control packets.

The proposed CSMA/OCA is implemented as a loadable kernel module (LKM) in the IEEE 802.11s protocol stack for MAC layer channel access. In the implementation, CSMA/OCA is interfaced with the AMC (Minstrel protocol - default AMC for Linux kernel) and the HAL to handle the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

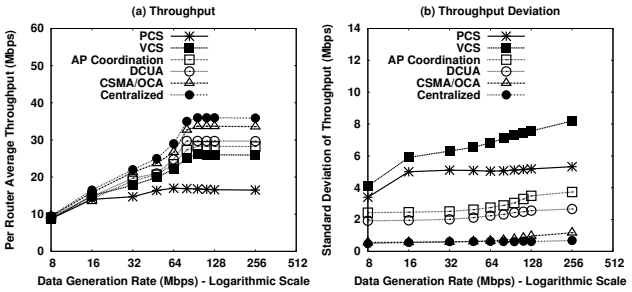


Fig. 4. For Different Channel Access Strategies (a) Throughput, (b) Throughput Deviation

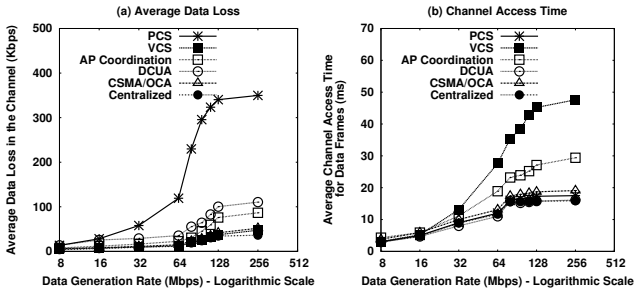


Fig. 5. For Different Channel Access Strategies (a) Channel Data Loss, (b) Average Channel Access Delay for Data Frames

issues related to dynamic MCS selection and calculation of SINR along with background noise level. The system setup is kept similar as discussed in Subsection II-A. The AMC uses the IEEE 802.11n supported MCS levels along with the supported transmit power levels and receiver sensitivity as directed by the RT-3352 hardware [17]. For instance, for MCS $0-8$ and with two different channel widths 20 MHz and 40 MHz, the available transmit powers are $\{3, 5, 8, 11, 14\}$ dBm and $\{5, 8, 11, 14, 16\}$ dBm, respectively.

Fig. 4(a) depicts different channel access protocols with respect to the average per user throughput. The average per user throughput is calculated as the average data transmitted successfully per second. Every mesh router uses A-MPDU aggregation level set to 64 Kbytes. The figure indicates that the network gets saturated (obtains maximum throughput) for PCS when the traffic load is more than 48 Mbps. After the saturation point, the throughput decreases slowly due to increased contention in the network. On the contrary, VCS and other channel access protocols get saturated near 80 Mbps traffic generation rate. The DCUA and AP coordination mechanisms improve average per router throughput compared to the PCS and VCS mechanisms, though the saturation throughput for these schemes are significantly less compared to the centralized strategy. The proposed CSMA/OCA mechanism attains maximum saturation throughput among all the schemes except the centralized strategy, and the performance is close to the centralized strategy. As indicated earlier, the centralized strategy is not implementation-feasible in a wireless mesh network. Being a decentralized scheme, the CSMA/OCA protocol can be easily implemented with marginal modifications in the existing mac80211 module, whereas the network can attain a performance benefit which is comparable with the centralized strategy.

Fig. 4(b) depicts throughput deviation among all the channel access strategies with respect to the traffic load in the network. As indicated earlier, throughput deviation is maximum for PCS and VCS, since hidden and exposed terminals increase unfairness in the network [9]. In presence of hidden and exposed terminals, performances of some nodes are affected by repeated channel access back-off which increases throughput deviation in the network. AP coordination and DCUA reduces this unfairness by eliminating a considerable number of hidden and exposed terminals. However, both of these mechanisms fail to reduce hidden and exposed terminals uniformly in the network. For instance, DCUA can eliminate only the hidden nodes which are within the maximum CS threshold. Similarly, AP coordination can detect only those hidden and exposed nodes which are within the CS range of the neighboring APs. Further AP coordination significantly elevates the control overhead in the network. The proposed CSMA/OCA can eliminate most of the hidden and exposed terminals with less control overhead.

Fig. 5(a) and Fig. 5(b) show the channel data loss and average channel access delay, respectively, for different channel access strategies. The channel data loss indicates the amount of data which are transmitted by a sender, however lost in the channel (acknowledgement is not received) and subsequently retransmitted by the sender. The channel data loss can result from two reasons - collision from the hidden terminals and data loss due to external and additive interference. As the CS range is tuned to make it equal to the interference range through the use of a high carrier sensing threshold value, data loss due to interference is almost negligible. Therefore, Fig. 5(a) indicates the effect of hidden terminals on the performance of the access strategies. On the other side, channel access delay is calculated as the time difference between the time when a MAC data frame is scheduled for transmission (it is at the head of the interface queue), and the time when it is actually transmitted. Therefore, Fig. 5(b) indicates the effect of exposed terminals over the performance. These two figures show that while channel data loss is maximum and grows exponentially for PCS, VCS results in maximum channel access time. AP coordination also has high channel access time, as the coordination requires significant amount of time. It can be noted that the time for control packet exchanges are included within the channel access time. The proposed CSMA/OCA mechanism reduces both the channel data loss as well as average channel access time, resulting in high throughput performance of the network.

Finally we evaluate the effect of A-MPDU aggregation level over the throughput attained through different channel access strategies, as shown in Fig. 6(a) and Fig. 6(b). The mean data generation rate for this experiment is considered to be 64 Mbps (that is just near the saturation level) for Fig. 6(a), and 32 Mbps (the network is unsaturated) for Fig. 6(b). In both the cases, the data generation varies along the mean data generation rate following a log-normal distribution with variance 16 Mbps. The performance of the centralized strategy

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

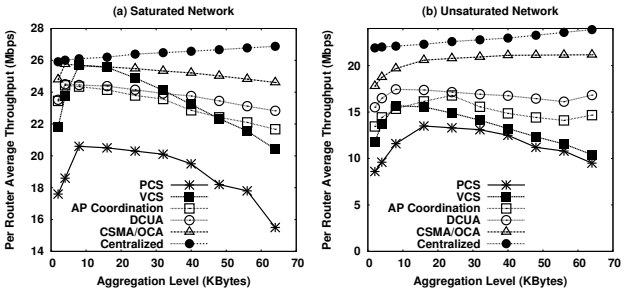


Fig. 6. Effect of A-MPDU Aggregation Level over Per Router Throughput (a) Saturated Network, (b) Unsaturated Network

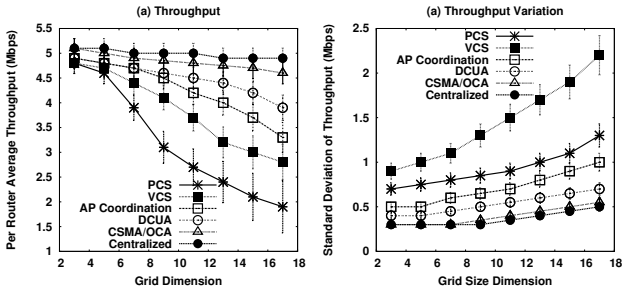


Fig. 7. Simulation Results for Different Channel Access Strategies (a) Throughput, (b) Throughput Deviation

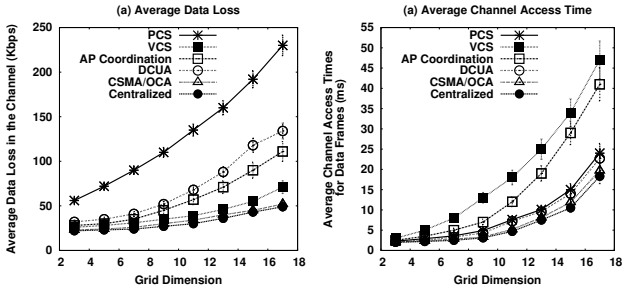


Fig. 8. Simulation Results for For Different Channel Access Strategies (a) Channel Data Loss, (b) Average Channel Access Delay for Data Frames

slowly increases with the increase in data aggregation level, as data aggregation level reduces the physical and MAC control overhead. The performance improvement with respect to the data aggregation level is more prominent in case of a unsaturated network, as shown in Fig. 6(b). With the increase in A-MPDU frame aggregation level, the performance of PCS and VCS first increases, and then drops linearly, and attain a very low throughput compared to the centralized strategy. The performance for PCS and VCS drops after a threshold aggregation level, as the data loss due to collision overshoots the effect of physical and MAC control overheads. With the increase in A-MPDU payload size, the loss due to collision also increases, as discussed earlier. Further VCS shows better performance at low aggregation level compared to AP coordination and DCUA, since AP coordination has significant control overhead, and DCUA can not eliminate hidden terminals completely. The proposed CSMA/OCA scheme performs similar to VCS when aggregation level is low, however, improves the network performance compared to VCS, as well as other schemes, as aggregation level increases. Further, the proposed scheme shows a consistent performance with the increase in the A-MPDU frame aggregation level. These experiments show that the proposed CSMA/OCA results in notable performance improvements in a high throughput wireless mesh network with minimum modifications to the standard mac80211 kernel module at the network protocol stack.

VI. Simulation Results

Although the testbed results give the performance boost for the proposed CSMA/OCA protocol, the experiments are limited with the number of nodes in the testbed. To analyze the scalability of the system, we have simulated the proposed CSMA/CA protocol in Qualnet-5.0.1 network simulator, and compared the results with other related schemes. In Qualnet-5.0.1, we have taken grid topologies of different sizes, with number of nodes in the grid varying from 9 (grid 3×3) to 289 (grid 17×17). Every intermediate node in the grid has 4 neighbors without any cross edges. The mesh gate is placed at the center of the grid. Similar to the testbed scenario, every mesh router has both upload and download traffic. The upload and download traffic are generated using Poisson distribution with the mean and variance as 3 Mbps and 2 Mbps respectively. This indicates a wide variation in traffic distribution across the network. We have executed 10 different scenarios with such random generated traffic model, and plotted the average and standard deviation (90% of the confidence interval) of the results in the graphs. The AMC and power setup for different data rates are kept similar to the testbed setup.

Fig. 7 shows the simulation results for throughput and throughput deviation comparison among different access strategies. In the grasp, grid dimension of n means a grid size of $n \times n$. Following the trend shown by the testbed results as discussed earlier, the simulation results also depict that the proposed CSMA/OCA scheme performs better than others. CSMA/OCA shows a consistent throughput with the increase in grid size.

Fig. 8 shows the average channel data loss and average channel access time for different access strategies. The figure indicates that the proposed CSMA/OCA is scalable to reduce average data loss similar to VCS and average channel access time similar to PCS. The simulation results reveal that the proposed CSMA/OCA mechanism is scalable and provides good performance benefit even in a large network.

VII. Conclusion

This paper presented the severity of the hidden and exposed terminal problem in case of a high throughput wireless mesh network using practical testbed results. The analysis revealed that the hidden terminals cause severe data loss, whereas the exposed terminals under-utilize the network capacity by reducing spatial reuse opportunities. Based on the IEEE 802.11n frame aggregation and BACK capabilities, this paper proposed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TWC.2015.2480398, IEEE Transactions on Wireless Communications

an opportunistic access protocol over the VCS paradigm to defend the hidden and exposed terminal problems. The effectiveness of the proposed scheme is analyzed through the results from a practical high speed indoor mesh testbed as well as from simulation.

REFERENCES

[1] I. F. Akyildiz and X. Wang, "A survey on wireless mesh networks," *IEEE Communications Magazine*, vol. 43, no. 9, pp. S23–S30, 2005.

[2] G. R. Hiertz, D. Denteneer, S. Max, R. Taori, J. Cardona, L. Berlemann, and B. Walke, "IEEE 802.11s: the WLAN mesh standard," *IEEE Wireless Communications*, vol. 17, no. 1, pp. 104–111, 2010.

[3] E. Perahia and M. X. Gong, "Gigabit wireless LANs: An overview of IEEE 802.11ac and 802.11ad," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 15, no. 3, pp. 23–33, Nov. 2011.

[4] B. Brown and N. Green, Eds., *Wireless World: Social and Interactional Aspects of the Mobile Age*. New York, NY, USA: Springer-Verlag New York, Inc., 2002.

[5] A. Tsertou and D. I. Laurenson, "Insights into the hidden node problem," in *proceedings of the 2006 IWCMC*, 2006, pp. 767–772.

[6] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "MACAW: a media access protocol for wireless LAN's," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 212–225, Oct. 1994.

[7] J. Monks, V. Bharghaban, and W. M. Hwu, "A power controlled multiple access protocol for wireless packet networks," in *proceedings of IEEE INFOCOM*, May 2007, pp. 219–228.

[8] A. Muqattash and M. Krunz, "Power controlled dual channel (PCDC) medium access protocol for wireless ad hoc networks," in *proceedings of IEEE INFOCOM*, vol. 1, March 2003, pp. 470–480.

[9] L. B. Jiang and S.-C. Liew, "Improving throughput and fairness by reducing exposed and hidden nodes in 802.11 networks," *IEEE Trans. Mobile Computing*, vol. 7, no. 1, pp. 34–49, Jan 2008.

[10] P. M. van de Ven, A. J. Janssen, and J. S. van Leeuwaarden, "Optimal tradeoff between exposed and hidden nodes in large wireless networks," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 179–190, Jun. 2010.

[11] P. van de Ven, A. Janssen, and J. van Leeuwaarden, "Balancing exposed and hidden nodes in linear wireless networks," *IEEE/ACM Transactions on Networking*, vol. 22, pp. 1429 – 1443, October 2014.

[12] K. Nishide, H. Kubo, R. Shinkuma, and T. Takahashi, "Detecting hidden and exposed terminal problems in densely deployed wireless networks," *IEEE Transactions on Wireless Communications*, vol. 11, no. 11, pp. 3841–3849, 2012.

[13] S. Chakraborty, S. Chattopadhyay, S. Chakraborty, and S. Nandi, "Defending concealedness in IEEE 802.11n," in *proceedings of the 6th COMSNETS*, Jan 2014, pp. 1–8.

[14] L. Deek, E. Garcia-Villegas, E. Belding, S. Lee, and K. Almeroth, "Intelligent channel bonding in 802.11n WLANs," *IEEE Transactions on Mobile Computing*, vol. 13, pp. 1536–1233, June 2014.

[15] Texas Instruments, "WLAN channel bonding: Causing greater problems than it solves," Tech. Rep., 2003.

[16] I. Tinnirello, S. Choi, and Y. Kim, "Revisit of RTS/CTS exchange in high-speed IEEE 802.11 networks," in *proceedings of the Sixth IEEE WoWMoM*, 2005, pp. 240–248.

[17] MediaTek RT3352 802.11n 2T2R platform (2.4GHz) - single-band 802.11n with 300mbit/s data rates for Wi-Fi access points and routers. Last visited 21 December, 2014. [Online]. Available: http://www.mediatek.com/en/products/connectivity/wifi/home-network/wifi-ap/rt3352/

[18] Open80211s: Open source implementation of IEEE 802.11s for Linux kernel. Last visited 21 December, 2014. [Online]. Available: www.open80211s.org

[19] G. Brar, D. M. Blough, and P. Santi, "Computationally efficient scheduling with the physical interference model for throughput improvement in wireless mesh networks," in *proceedings of the 12th MobiCOM*, 2006, pp. 2–13.

[20] A. Iyer, C. Rosenberg, and A. Karnik, "What is the right model for wireless channel interference?" *IEEE Transactions on Wireless Communications*, vol. 8, no. 5, pp. 2662–2671, May 2009.

[21] J. Jeong, S. Choi, J. Yoo, S. Lee, and C.-K. Kim, "Physical layer capture aware MAC for WLANs," *Wirel. Netw.*, vol. 19, no. 4, pp. 533–546, May 2013.

[22] T.-S. Kim, H. Lim, and J. C. Hou, "Improving spatial reuse through tuning transmit power, carrier sense threshold, and data rate in multihop wireless networks," in *proceedings of the 12th MobiCOM*, 2006, pp. 366–377.

[23] J. Deng, B. Liang, and P. Varshney, "Tuning the carrier sensing range of IEEE 802.11 MAC," in *proceedings of the IEEE GlobeCOM*, vol. 5, 2004, pp. 2987–2991.

[24] C. Reis, R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan, "Measurement-based models of delivery and interference in static wireless networks," in *Proceedings of the 2006 SIGCOMM*, 2006, pp. 51–62.

[25] W. L. Tan, P. Hu, and M. Portmann, "Experimental evaluation of measurement-based SINR interference models," in *Proceedings of the 2012 IEEE WoWMoM*, June 2012, pp. 1–9.

[26] RSSI in MadWiFi. Last visited 21 December, 2014. [Online]. Available: http://madwifi-project.org/wiki/UserDocs/RSSI

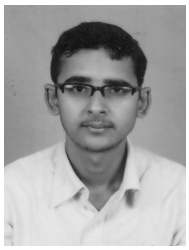
[27] K.-J. Park, L. Kim, and J. C. Hou, "Adaptive physical carrier sense in topology-controlled wireless networks," *IEEE Transactions on Mobile Computing*, vol. 9, no. 1, pp. 87–97, 2010.

Sandip Chakraborty Sandip Chakraborty received the PhD degree from Indian Institute of Technology Guwahati, India in 2014. At present, he is working as an Assistant Professor at Indian Institute of Technology Kharagpur, India. His research interests include wireless networks, mobile computing and distributed computing.

Dr. Chakraborty is a Member of IEEE, IEEE Communications Society and Association for Computing Machinery.

Sukumar Nandi Sukumar Nandi received the PhD degree in Computer Science and Engineering from Indian Institute of Technology Kharagpur, India. He is currently a Senior Professor of computer science and engineering with Indian Institute of Technology, Guwahati, Assam, India. He is coauthored of a book entitled *Theory and Application of Cellular Automata* (IEEE Computer Society). His research interests include traffic engineering, wireless networks, network security and distributed computing.

Dr. Nandi is a Senior Member of IEEE, a Senior Member of Association for Computing Machinery, a Fellow of The Institution of Engineers (India) and a Fellow of The Institution of Electronics and Telecommunication Engineers (India).



Subhrendu Chattopadhyay Subhrendu Chattopadhyay received his B.Tech degree from West Bengal University of Technology and M.Tech degree from Indian Institute of Technology, Guwahati, India. He is currently working towards his PhD degree with Indian Institute of Technology, Guwahati, Assam. He has received a research fellowship from TATA Consultancy Services, India. His current research interests are broadly in the area of computer networking, distributed systems and social network analysis.

The Journal of Supercomputing
https://doi.org/10.1007/s11227-021-04135-2

Containerized deployment of micro-services in fog devices: a reinforcement learning-based approach

Shubha Brata Nath[1] · Subhrendu Chattopadhyay[2] · Raja Karmakar[3] · Sourav Kanti Addya[4] · Sandip Chakraborty[1] · Soumya K Ghosh[1]

Accepted: 8 October 2021

Abstract

The real power of fog computing comes when deployed under a smart environment, where the raw data sensed by the Internet of Things (IoT) devices should not cross the data boundary to preserve the privacy of the environment, yet a fast computation and the processing of the data is required. Devices like home network gateway, WiFi access points or core network switches can work as a fog device in such scenarios as its computing resources can be leveraged by the applications for data processing. However, these devices have their primary workload (like packet forwarding in a router/switch) that is time-varying and often generates spikes in the resource demand when bandwidth-hungry end-user applications, are started. In this paper, we propose *pick–test–choose*, a dynamic micro-service deployment and execution model that considers such time-varying primary workloads and workload spikes in the fog nodes. The proposed mechanism uses a reinforcement learning mechanism, *Bayesian optimization*, to decide the target fog node for an application micro-service based on its prior observation of the system's states. We implement PTC in a testbed setup and evaluate its performance. We observe that PTC performs better than four other baseline models for micro-service offloading in a fog computing framework. In the experiment with an optical character recognition service, the proposed PTC gives average response time in the range of 9.71 sec–50 sec, which is better than Foglets (24.21 sec–80.35 sec), first-fit (16.74 sec–88 sec), best-fit (11.48 sec–57.39 sec) and mobility-based method (12 sec–53 sec). A further scalability study with an emulated setup over Amazon EC2 further confirms the superiority of PTC over other baselines.

Keywords Fog computing · Container deployment · Bayesian optimization · Internet of things · Reinforcement learning

✉ Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Extended author information available on the last page of the article

Published online: 27 October 2021

1 Introduction

The concept of fog computing [9, 18] is gaining popularity in recent times due to the availability of computationally powerful network devices, like routers, WiFi access points, core switches, etc. Various recent works have focused on developing innovative applications around it, ranging from smart cities applications [12], big data processing [25], pervasive health care [1, 19], etc. In a typical fog computing architecture, the intermediate network devices' excess computing resources are used for end-user application processing purposes. Thus, it provides low network latency along with added privacy to the data. However, the response time becomes higher when the end devices like the Internet of Things (IoT) devices do not have sufficient computing resources, and hence, the computation needs to be offloaded to a secondary device. Although a large number of research works [2, 7, 10, 17, 24, 27, 31, 33, 34] have been devoted to design application offloading mechanisms for fog computing, they primarily consider the workload placement as a one-time process and fail to capture the dynamicity of the system.

In the fog computing paradigm, the application workloads are divided into micro-services that can be executed in parallel [4]. These micro-services [15, 29, 30] are placed on the fog nodes (the routers, access points and core network switches) through some sandboxing mechanism, like the virtual machines (VMs) or the containers. Such a placement mechanism needs to consider two different aspects—(1) there should be excess computing resources available at the fog nodes, and (2) the execution of the micro-service should not interfere with the primary workload of the fog nodes. Although the existing approaches, as mentioned above, consider these two factors, they fail to properly characterize the nature of the fog nodes' primary workload. Such primary workloads are typically dynamic. Consider the following example. In a smart home environment, let the IoT devices and the corresponding applications use the home gateway as a fog node. Thus, the IoT data processing tasks are offloaded on the network gateway. Now, multiple smart home appliances are connected to that gateway, and the gateway needs to route the data packets to respective subnets, which is its primary workload. Now, this gateway's primary workload observes a spike whenever high-bandwidth applications, such as a smart television, get started. During that time, the gateway might not have sufficient computation resources to execute the fog micro-services. If we still force the micro-services to get completed on that fog node, the response time will possibly get increased, resulting in a negative performance impact.

The primary advantage of fog computing comes from the fact that it provides a quicker response than the cloud as the computation happens near the device. However, unless we properly place and schedule the micro-services over the fog nodes by considering their primary workloads, the advantages might get nullified and it might result in a very high response time. The primary requirements for developing such an execution architecture are as follows. (1) The placement algorithm should not be a one-time algorithm; it should continuously monitor the primary workload and excess resources in the fog nodes and dynamically update the target node where the micro-service can execute. (2) Depending on the dynamic

execution decision, micro-services should be able to seamlessly migrate from one fog node to another to complete its execution. (3) The migration should be fast, having negligible overhead. (4) The decision of placement and migration should be very fast so that it does not contribute significantly to the overall response time. The existing mechanisms fail to capture all of these four aspects together.

We propose *pick–test–choose* (PTC) framework which can be considered as a containerized micro-service placement mechanism for the fog nodes considering their primary workloads and their impact on the micro-service execution. PTC continuously monitors the fog nodes' excess resources and the required computation environment for the micro-service workloads to complete their executions. Accordingly, it models the problem as a constrained optimization such that the response time always remains within a predefined boundary. As solving such a constrained optimization is computationally expensive, we use a reinforcement learning technology to dynamically map the micro-service execution to the fog nodes' computation platforms. To satisfy the requirement of having a quick decision, we use *Bayesian optimization* [26] (BO) that can dynamically and quickly decide the target solution. It also considers the measurement noises that may come while calculating the resource availability (of the fog nodes) and the resource requirements (of the micro-services). We have implemented PTC over a prototype testbed for thorough performance evaluation and comparison with baselines; along with that, the performance has also been evaluated in Amazon Elastic Compute Cloud (Amazon EC2) for scalability analysis. The experimental results show that PTC can minimize the response time while effectively using fog nodes' excess resources. We further analyze PTC's performance in the context of a natural language processing (NLP) application as a use case, which has been evaluated over the testbed. We observe that the proposed framework can significantly reduce the response time of the NLP application's micro-service executions.

An initial version of this work has been published in [21]. We have made a significant extension of the initial framework in this current version. (1) We have extended the theoretical framework to formulate the dynamic optimization problem based on resource availability at the fog nodes and the fog micro-services' resource demands. We have shown how a BO-based solution model can reasonably fit in the proposed optimization framework. (2) We have extended the evaluation with a detailed analysis of PTC's performance under various scenarios. We have further analyzed a use case over the PTC framework. In summary, the salient contributions of this paper are as follows.

1. We have formulated an optimization problem considering the fog micro-services' dynamic demands along with the dynamic resource availability at the fog nodes. We have shown formally that the proposed optimization is $\mathcal{NP}$-hard.
2. In order to solve the proposed optimization, we utilized a Bayesian optimization-based framework while considering the dynamicity and measurement noises that may come during the runtime of such a system.
3. We have evaluated the proposed framework thoroughly with an optical character recognition application. The proposed PTC framework is also evaluated with a

natural language processing application as a use case. These evaluations have shown the superiority of PTC compared to other baselines.
4. We have also tested the proposed PTC framework for scalability in Amazon EC2. We have observed from the experiments in Amazon EC2 that the proposed PTC framework is scalable while ensuring minimization of response time.

The rest of the paper is organized as follows. An extensive study of the existing literature is provided in Sect. 2. The proposed system architecture and design goals are discussed in Sect. 3. The micro-service placement is modeled as an optimization, which is discussed in Sect. 4. Based on the hardness of the proposed optimization, we provide a reinforcement learning-based solution of the proposed optimization, as discussed in Sect. 5. The performance evaluation of the proposed framework is provided in Sect. 6. Finally, Sect. 7 concludes the paper.

2 Related work

Various works have focused on the micro-service deployment problem in fog computing. The authors in [28] present a metaheuristic approach to minimize the application response time in edge computing for micro-service-based applications. In [24], the authors have developed a programming infrastructure for geo-distributed applications, which launches the application modules and performs the migration of these modules between fog devices as containers. These approaches primarily choose the nearest node for application deployment and therefore do not consider that the nearest node might be loaded with clients' requests. In [7], Elgamal *et al.* have proposed a dynamic programming-based algorithm to partition operations in IoT applications. The operations are placed across edge and cloud resources to minimize the completion time of the end-to-end operations. VM-based micro-service placement and migration for mobile users have been discussed in [10] where the number of applications to be placed in fog devices has been maximized while minimizing the latency cost. In [11], the authors have studied the base station association, task distribution and VM placement for cost efficient fog-based medical cyber-physical systems. The authors have minimized the communication cost along with the VM deployment cost. Li et al.[16] have proposed a resource scheduling approach in edge computing-based smart manufacturing, ensuring latency constraints. However, all of these works have considered static workload at the fog devices; therefore, the placement algorithms are modeled mostly as a one-time optimization problem solved through some efficient heuristics.

A few works in the literature have focused on application placement at the fog devices based on dynamically monitoring of resources. Yigitoglu et al.[34] have proposed a containerized application placement strategy based on the current measurement of the fog workload. They have considered a hierarchical organization of fog devices based on their resource availability and placed the application at a suitable hierarchy. Taneja et al.[31] have presented a module mapping algorithm for efficient resource utilization by placing the application modules in fog–cloud infrastructure. However, these works still use one-time scheduling of the fog workloads depending

on the immediate availability of the computing resources at the fog nodes. Although such approaches can show good performance for short term workloads, the performance severely suffers when the execution time is a bit longer. The authors in [27] have proposed an orchestration system where the micro-services are placed in fog devices if the resources are available; otherwise, the micro-service would be placed in the cloud node. The authors have also analyzed if there is a need to offload the micro-service in the cloud or queue it to one of the busy fog devices. In [35], a container-based task scheduling and reallocation mechanism are designed to optimize the number of concurrent tasks in fog computing-based smart manufacturing. These approaches are mostly based on periodic measurements and re-execution of the full optimization, which is a significant overhead for the system. In [23], the authors have developed an orchestration tool based on Kubernetes. They have extended it with self-adaptation and network-aware placement capabilities. The authors have proposed a model-based reinforcement learning solution to solve the elasticity problem. The work of [13] has presented a distributed placement policy that optimizes energy consumption at fog nodes and communication costs. They have modeled the service placement problem as a combinatorial auction market. The authors in [22] have given a reinforcement learning-based approach that manages the containers' horizontal and vertical elasticity. In the next step, container placement is performed by solving an integer linear programming problem or using a network-aware heuristic. Nevertheless, these works do not consider the dynamic migration of fog workloads across different fog nodes depending on the nodes' primary workload. Consequently, the micro-services execution performance suffers when there is a spike in the fog nodes' primary workload. In [32], the authors have proposed an dynamic programming-based offline micro-service coordination algorithm. Also, they have proposed a reinforcement learning-based online micro-service coordination algorithm for dynamic micro-service deployment. Table 1 presents the comparison of these works.

Our proposed approach attempts to address the limitations of the works mentioned above as PTC can provide micro-service isolation by using lightweight virtualization, i.e., containers. Again, PTC considers the dynamic workload of the fog computing environment by doing BO-based online learning to decide the best micro-service placement and a migration strategy. Therefore, in contrast to the existing works, we consider a fully dynamic system where the time-varying behavior of both the fog workloads and the computing nodes has been taken care of.

3 System architecture and design goals

The overall objective of the proposed PTC orchestration framework is to support the following four requirements for micro-service placement over a fog architecture—(a) development of a lightweight technology suitable for micro-service deployment over fog devices, (b) support service isolation for micro-service execution over fog devices, (c) design of an online micro-service placement mechanism catering to the dynamic workload and resource availability characteristics, and (d) seamless migration of micro-services to maintain application QoS of both the primary workload

Table 1 Comparison with existing approaches

Work	Containers	Micro-service	Migration	Bayesian optimization	Scalability
Stevant et al., 2018 [28]	✓	✓	✗	✗	✓
Saurez et al., 2016 [24]	✓	✓	✓	✗	✗
Elgamal et al., 2018 [7]	✓	✓	✗	✗	✓
Gonccalves et al., 2018 [10]	✗	✗	✓	✗	✗
Gu et al., 2015 [11]	✗	✗	✓	✗	✓
Li et al., 2019 [16]	✗	✓	✗	✗	✗
Yigitoglu et al., 2017 [34]	✓	✓	✗	✗	✓
Taneja et al., 2017 [31]	✗	✓	✗	✗	✓
Souza et al., 2018 [27]	✗	✓	✓	✗	✗
Yin et al., 2018 [35]	✓	✗	✓	✗	✗
Rossi et al., 2020 [23]	✓	✗	✗	✗	✓
Kayal et al., 2019 [13]	✗	✓	✗	✗	✓
Rossi et al., 2019 [22]	✓	✗	✗	✗	✓
Wang et al., 2019 [32]	✗	✓	✓	✗	✓
PTC	✓	✓	✓	✓	✓

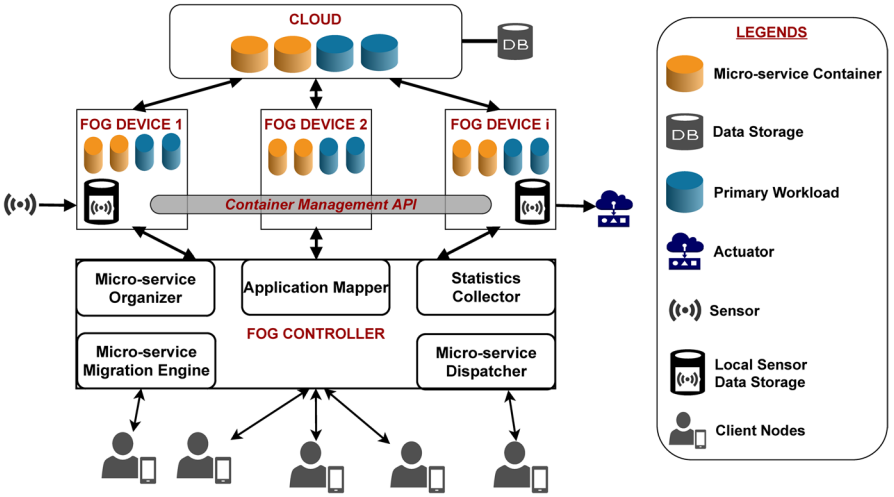


Fig. 1 PTC framework

and the micro-service workload. We discuss the architecture of the PTC framework (shown in Fig. 1) in this section.

Client Node The service request is generated by the client nodes. For example, a client might request the fog service to generate a consolidated report based on the sensing data captured by IoT devices. To do so, the client sends `REST` queries to the fog controller device.

Fog Controller Device The fog devices are connected to the fog controller device. This edge server receives the workload requests in terms of containerized micro-services and schedules them over the fog nodes. The different components of the *Fog Controller Device* are as follows. The `application mapper` maps the request–response to an application from where the requests originate. The `micro-service organizer` module divides the application into micro-services following the existing approaches such as ENTICE [14]. Also, the organizer module performs containerization of the micro-services. The `micro-service dispatcher` module finds the fog nodes to place the micro-services. The `statistics collector module` helps the dispatcher module to identify a suitable placement. The statistics collector module periodically monitors the available resources of the fog nodes. The micro-services are sent to the fog nodes by a `micro-service migration engine`, once the dispatcher module finds the placement. If the fog nodes become overloaded with their primary workloads, the micro-service dispatcher and the migration engines regenerate the placement.

Fog Device The containerized workloads are run in the fog devices. In Fig. 1, the blue-colored containers are the fog nodes' primary workloads. The micro-service containers utilize the remaining resources of the fog devices. We represent the micro-service containers with orange-colored containers (Fig. 1). The micro-service containers also need to communicate with the sensors and the actuators. The `local sensor data storage` module helps in this interaction. To seamlessly migrate the micro-service containers, we use a distributed `container management` API. The container management API is built using *"remote procedure call"* (`RPC`).

Cloud: The fog devices are connected to the cloud. Fog devices are always the first choice for placing a service. However, the proposed framework allows placement of a service in a cloud server if no fog server is available at a particular time instant.

In this framework, we combine two different approaches to design the overall execution model of PTC.

(a) We use containers as the sandboxing environment that supports the workload isolation along with the seamless migration of workloads across fog nodes.
(b) We design an online optimization strategy to decide the target fog node where a containerized micro-service can be placed. The optimization module facilitates both the initial micro-service deployment as well as the migration requirements. We use the standard application container-based approach [5] for executing the micro-services in the fog nodes and for migrating the micro-services based on the condition.

However, the challenge here is to design the dynamic micro-service placement in fog computing depending on the time-varying primary workload of the fog devices. The placement mechanism needs to be online, as well as it needs to cater to the dynamic workload characteristics of the system.

Next, we formally model the placement mechanism as an optimization problem. The details are given in Sect. 4.

4 Dynamic micro-service placement over fog nodes—an optimization formulation

This section presents the mathematical modeling of the fog micro-service placement problem and its theoretical analysis. The proposed execution model of the micro-services considers the following facts.

1. The fog devices have their primary workloads, which are time-varying, and thus can observe a spike in the resource demand. The priority is given to the primary workloads. Hence, the fog micro-service's execution might suffer if the primary workload withholds the resources used by the fog micro-services earlier.
2. To cope with such problems, PTC enables seamless migration of unfinished micro-services from one fog device to another. Such migrations are facilitated by container-based sandboxing. We consider that the migrations are live [20], indicating that execution states of the micro-services can be saved and later resumed on a different fog device.
3. The fog controller continuously monitors the available resource statistics on the fog devices and periodically decides the target fog device for a micro-service execution.

Considering the above mentioned facts, we present the system model for designing the optimization function for dynamic micro-service placement and execution over fog devices.

4.1 System model

We model the entire network, involving the end devices (sensors, actuators, edge devices), fog nodes (in-network devices like WiFi access point, gateway, router, core switches), fog controller, etc., as a weighted undirected communication graph. Table 2 shows the notations used. Let $G = (V,E,W)$ be an undirected and weighted communication graph. Here, the set of physical nodes is V, the set of physical communication links is E, and the set of edge weights is W. The sets of sensors, actuators and fog devices are denoted as $S = (S_i \in (1, \ldots, s))$, $\Lambda = (\Lambda_i \in (1, \ldots, \lambda))$ and $F = (F_i \in (1, \ldots, n))$, respectively. Here, the total number of sensors, actuators and fog devices are s, λ and n, respectively. Therefore, $V = \{S \cup \Lambda \cup F\}$. $e_{i,j} \in E$ represents the physical communication link between $v_i \in V$ and $v_j \in V$. In this case, the weight of each edge signifies the latency of the links.

We consider that the time is divided into l slots, where each slot is of length ι. Therefore, we define the system time vector $T = (\tau_t : t \in (1, \ldots, l))$. We also assume that each event either starts at the beginning of the slot or ends at the end of the slot. Therefore, all the time calculations done in this paper are in terms of number of slots. Let the weight of the shortest path between v_{i1} and v_{i2} be $D(v_{i1}, v_{i2})$, which signifies the communication latency between two nodes in the communication graph in terms of number of slots. Let k edge applications

Table 2 Notation table

Notation	Significance
G	An undirected and weighted communication graph
V	Set of physical nodes
E	Set of physical communication links
W	Set of edge weights
S	Set of sensors
Λ	Set of actuators
F	Set of fog nodes
s	Total number of sensors
λ	Total number of actuators
n	Total number of fog devices
$e_{i,j}$	The physical communication link between node v_i and node v_j
l	Number of time slots
ι	Length of a time slot
T	System time vector
$D(v_{i1}, v_{i2})$	Weight of the shortest path between node v_{i1} and node v_{i2}
k	Number of edge applications
$\mathbf{\Psi}$	Set of applications
$p_{x,y}$	y^{th} micro-service of a service P_x
h_x	Total number of atomic micro-services of a service P_x
$\mathbf{R}(F_i)$	The resource vector available in a fog node F_i
$\mathbf{\Gamma}(p_{x,y})$	The resource vector required by micro-service $p_{x,y}$
$\gamma_{x,y}^{q}$	The amount of resource type q required to execute $p_{x,y}$
f	The total different types of resources
r_i^q	The total available quantity of resource type q at F_i
A	Micro-service allocation matrix
$O(A, x, y)$	The occupancy of a fog node's resources by $p_{x,y}$ at a time instance
$Seq_{exe}(x, y)$	Fog device execution sequence by $p_{x,y}$
$[I]_{1\times n}$	The row vector of the fog device indices
T_{CPU}	Processing time
δ_i	Million instructions per second executed by fog node F_i
$\xi_{x,y}$	Million instructions required by $p_{x,y}$
$M_{x,y}$	Sub-vector of $Seq_{exe}(x, y)$
T_{MGR}	Migration time
$\Delta_{x,y}$	Constant delay factor for migration of $p_{x,y}$
ψ	The demand vector of sensors
α	The demand vector of actuators
$\psi_{x,y}$	The sensors required by $p_{x,y}$
α_x	The actuators required by $p_{x,y}$
$[I]_{1\times s}$	The row vector of the sensor indices
$[I]_{1\times \lambda}$	The row vector of the actuator indices
T_{DF}	The data fetch time
T_{ACT}	The actuation time

Table 2 (continued)

Notation	Significance
$\Omega(A, i, t)$	The executing micro-service vector at fog device F_i at time slot t
$\Theta(A, i, q, \Gamma, t)$	The amount of occupied resource of type q at time slot t
$T_{Resp}^{max}(A)$	The maximum response time taken by the applications
$\mathcal{R}(A)$	The resource availability at a particular time instant t
$\mathcal{D}_d$	The set of prior observations after d iterations
$\mu(\circ)$	The mean function
$K(\circ, \circ)$	The covariance kernel function
Φ	The standard normal cumulative distribution function
ϕ	The standard normal density function
EI	Acquisition function
ζ	Noise

are present in the system, such that $\mathbf{\Psi} = \left(P_x : x \in (1, \ldots, k)\right)$. $\mathbf{\Psi}$ is the set of applications. These applications need to execute their micro-services over the fog devices. An application can be decomposed into multiple micro-services, as mentioned earlier. We define $p_{x,y}$ as the y^{th} micro-service of P_x. Hence, $P_x = \left(p_{x,y} : y \in (1, \ldots, h_x)\right)$, where h_x is the total number of atomic micro-services of P_x. We consider that each micro-service can execute independently.

We denote $\mathbf{R}(F_i)$ as the resource vector available in a fog node F_i. This resource vector is time-varying and keeps on changing depending on the primary workload of the fog device. The resource vector required by micro-service $p_{x,y}$ is denoted by $\mathbf{\Gamma}(p_{x,y})$. We assume that the initial resource vector required by $p_{x,y}$ does not change over time, however depending on the partial completion of a micro-service's execution, the residual resource requirement might change. $\mathbf{\Gamma}(p_{x,y}) = (\gamma_{x,y}^q \in \mathbb{R} : q \in (1, \ldots, f))$, where each component $\gamma_{x,y}^q$ signifies the amount of resource type (central processing unit, memory, network bandwidth, etc.) q required to execute $p_{x,y}$ and f is the total different types of resources. Similarly, $\mathbf{R}(F_i) = (r_i^q \in \mathbb{R} : q \in (1, \ldots, f))$, where r_i^q is the total available quantity of resource type q at F_i. Each fog device F_i can host more than one micro-services depending on its available resources.

We define a micro-service allocation matrix $[A] = (A_{x,y,i,t} \in (0, 1) : (x \in (1, \ldots, k), y \in (1, \ldots, h_{max}), i \in (1, \ldots, n), t \in (1, \ldots, l))$ that gives the mapping between micro-services and fog nodes at each time stamp, where $h_{max} = \max_x(h_x)$, and each element $A_{x,y,i,t} = 1$ if $p_{x,y}$ is assigned to F_i at time slot t, otherwise $A_{x,y,i,t} = 0$. The task of the fog controller is to generate this occupancy metric dynamically based on its observation of the available resources at the fog devices. The allocation matrix changes over time, and such changes typically trigger a service migration from one fog device to another. Consequently, the occupancy ($O(A, x, y)$) of a fog node's resources by $p_{x,y}$ at a time instance is defined as follows.

$$\left[O(A,x,y)\right]_{n\times l} = \left(A_{x,y,i,t} \in (0,1) \,:\, i \in (1\ldots n), t \in (1\ldots l)\right) \tag{1}$$

Due to the dynamic workload characteristics of the fog devices, we consider that a micro-service may complete its execution over multiple fog nodes. Therefore, the time to calculate the total response time for an application has the following components—(a) processing time, (b) migration time and (c) data fetch and actuating time. We compute these three components as follows.

4.2 Computation of the processing time

As we mentioned earlier, a micro-service may get placed over a set of fog devices sequentially to complete its execution. Let the fog device execution sequence by $p_{x,y}$ be $Seq_{exe}(x,y) = [I]_{1\times n} \times [O(A,x,y)]_{n\times l}$, where $[I]_{1\times n} = [1,2,\ldots,n]$ represents the row vector of the fog device indices. We give the processing time (T_{CPU}) required by $p_{x,y}$ in order of number of time slots as follows. Here, F_i is capable of executing δ_i million instructions per second (MIPS), and $p_{x,y}$ requires $\xi_{x,y}$ million instructions.

$$T_{CPU}(x,y,A) = \sum_{i \in Seq_{exe}(x,y)} \frac{\xi_{x,y}}{\delta_i t} \tag{2}$$

4.3 Computation of the migration time

During the execution, a micro-service may require migration from one fog device to another, which incurs additional migration time. Let the sub-vector of $Seq_{exe}(x,y)$ be $M_{x,y} = (M^b_{x,y} = Seq_{exe}(x,y)_b \,:\, Seq_{exe}(x,y)_b \neq Seq_{exe}(x,y)_{b+1})$, where b is a particular time slot. Here, the fog node migration vector by $p_{x,y}$ is $M_{x,y}$. Therefore, the migration time (T_{MGR}) is computed as follows.

$$T_{MGR}(x,y,A) = \sum_{M^b_{x,y}} (\Delta_{x,y} \times D(M^b_{x,y}, M^{(b+1)}_{x,y})) \tag{3}$$

Here, $\Delta_{x,y}$ is a constant delay factor for migration of $p_{x,y}$, which depends upon the amount of data to be transferred from one fog device to another. Also, the weight of the shortest path between v_i and v_j is $D(v_i, v_j)$. $D(v_i, v_j)$ is signifying the communication latency between two nodes in terms of number of slots.

4.4 Computation of the data fetch and actuating time

Apart from migration, communication overhead plays a significant role in initial data fetching from sensors and triggering actuators. Let the demand vector of sensors and actuators are $\psi = (\psi_{x,y,i} \,:\, x \in (1,\ldots k), y \in (1,\ldots h_{max}), i \in (1,\ldots s))$ and $\alpha = (\alpha_{x,i} \,:\, x \in (1,\ldots k), i \in (1,\ldots \lambda))$, respectively. Here, $\psi_{x,y,i} = 1$ if $p_{x,y}$ requires S_i, otherwise it is 0. Similarly, $\alpha_{x,i} = 1$ if P_x requires Λ_i and it is 0 for all other cases. Let $\psi_{x,y} = [I]_{1\times s} \times (\psi_{x,y,i} \,:\, i \in (1,\ldots s))$ and $\alpha_x = [I]_{1\times\lambda} \times (\alpha_{x,i} \,:\, i \in (1,\ldots \lambda))$

are the sensors and actuators required by $p_{x,y}$, respectively. The row vector of the sensor indices is $[I]_{1\times s} = [1, 2, \ldots, s]$ and the row vector of the actuator indices is $[I]_{1\times\lambda} = [1, 2, \ldots, \lambda]$. The data fetch time ($T_{DF}$) of $p_{x,y}$ is given below.

$$T_{DF}(x, y, A) = \max_{i\in\psi_{x,y}}(D(M^1_{x,y}, i)) \tag{4}$$

The actuation time (T_{ACT}) is given below.

$$T_{ACT}(x, y, A) = \max_{i\in\alpha_x}(D(M^{|M_{x,y}|}_{x,y}, i)) \tag{5}$$

We define the executing micro-service vector ($\Omega(A, i, t)$) at fog device F_i at time slot t as follows.

$$\Omega(A, x, y, i, t) = \left(A_{x,y,i,t} \in (0, 1) : x \in (1 \ldots k), y \in (1 \ldots h_{max})\right) \tag{6}$$

The amount of occupied resource of type q at time slot t is given as follows.

$$\Theta(A, i, q, \mathbf{\Gamma}, t) = \left(\sum_{x,y}\left(\Omega(A, x, y, i, t) \times \gamma^q_{x,y}\right)\right) \tag{7}$$

According to the capacity property, cumulative occupied resources by micro-services executing on a single fog device must not surpass the total available resource at that fog device. A valid allocation matrix must satisfy the following capacity property given in Equation (8).

$$\underset{i,q,t}{\forall}\ \Theta(A, i, q, \mathbf{\Gamma}, t) \leq r^q_i \tag{8}$$

If the capacity property is satisfied by the allocation matrix, then the total response time required by $\mathbf{\Psi}_x$ can be calculated using Equations (2) to (5). Therefore, we calculate the total response time required by $\mathbf{\Psi}_x$ using Equations (2) to (5), as follows.

$$T_{Resp}(A, x) = \max_{y\in(1\ldots h_{max})}\begin{pmatrix} T_{DF}(x, y, A) + T_{CPU}(x, y, A) \\ +T_{MGR}(x, y, A) + T_{ACT}(x, y, A) \end{pmatrix} \tag{9}$$

4.5 Problem definition

We represent the formal definition of the micro-service placement problem as follows. Given the communication graph G and available resources ($\mathbf{R}(F_i)$), the micro-service placement problem finds an allocation schedule (A) for each micro-service ($p_{x,y}$) with required instructions ($\xi_{x,y}$) and resources ($\mathbf{\Gamma}(p_{x,y})$) such that the maximum response time taken by the applications ($T^{max}_{Resp}(A)$) is minimized. Therefore, mathematically, the problem can be represented as an optimization problem given by Equation (10). Therefore, we have

$$\begin{aligned} \underset{A}{\text{minimize}} \quad & T^{max}_{Resp}(A) \\ \text{subject to:} \quad & \\ & \mathcal{R}(A) \geq \mathbf{Z} \end{aligned} \tag{10}$$

where $\mathcal{R}(A)$ is the resource availability at a particular time instant t and $\mathbf{Z} = \left(z_{i,q,t} = 0 : i = (0, \ldots, n), q = (0, \ldots, f), t = (0, \ldots, l)\right)$.

In *"Minimax facility location problem"* (MFLP) [6], the cost to transfer the items to the demand site is optimized. In our micro-service allocation problem, we are interested to minimize the cost, i.e., the response time. We now show that the *"Minimax facility location problem"* (MFLP) [6] is polynomial time reducible to micro-service allocation problem (Equation (10)). Thus, the micro-service allocation is $\mathcal{NP}$-hard.

Theorem 4.1 *"Minimax facility location problem"* (*MFLP*) [6] *is polynomial time reducible to micro-service allocation problem given in Equation* (10).

Proof Let $\{L_i : i \in (1, \ldots, n)\}$ be the set of locations where a warehouse can be placed and $\{C_j : j \in (1, \ldots, m)\}$ be the set of demand sites that must be serviced. Suppose $c_{i,j}$ be the cost of delivery of items from L_i to C_j. In that case, the MFLP finds the subset of locations where the warehouses should be opened such that the maximum cost to transfer the items from facilities to the demand site is minimized. This is a known $\mathcal{NP}$-hard problem. To prove the $\mathcal{NP}$-hardness of our micro-service placement, we encode the instance mentioned above of MFLP.

The encoding of the instance mentioned above of MFLP can be done in the following way. In our case, the system consists of m applications, each having single micro-services (μ_j). Each of the micro-services (μ_j) needs to be placed in exactly one of the fog devices. Let $\{L_i : i \in (1, \ldots, n)\}$ be the set of fog devices and $\{C_j : j \in (1, \ldots, m)\}$ be the set of micro-services to be placed. $c_{i,j}$ is the cost (in terms of response time) of placing the j^{th} micro-service (C_j) in i^{th} fog device (L_i). The reduction can be made in polynomial time. This completes the polynomial time encoding of MFLP to our micro-service placement. The used fog devices represent the location where the warehouses can be set up from the solution allocation matrix. However, as the MFLP is a known $\mathcal{NP}$-hard problem; therefore, we can claim that our proposed micro-service placement problem is $\mathcal{NP}$-hard. □

5 Solution approach: Bayesian optimization for micro-service placement

As the optimization problem is *NP*-hard, we design an approximate solution of the proposed optimization, which is fast and provides quick decision about the placement depending on the system's dynamicity. We use Bayesian optimization (BO), a reinforcement learning framework that provides the solution of an optimization based on the historical observation and posterior distribution of the solution vector.

BO performs well in the scenarios when conducting a single experiment takes a higher time.

We define the micro-service allocation matrix as a solution configuration of BO. BO optimizes the objective function based on the prior observations and posterior distribution by conducting iterative experiments over the solution configurations. In our case, conducting an experiment is equivalent to test the performance of a given allocation matrix which is costly in terms of time taken to perform a test. Surprisingly, BO performs well in such cases where performing one single experiment takes a higher time.

To formulate the BO framework, we assume that the utility function $T^{max}_{Resp}(A)$ follows a normal distribution. BO executes initial experiments based on the prior belief function. After sufficient number of experiments, BO modifies the prior belief function based on the posterior distributions. BO uses an *"acquisition function"* (*EI*) to pick the configurations for the iterations. This leads to a near-optimal solution. Let $\mathcal{D}_d = \{(a_1, T^{max}_{Resp}(a_1)), \ldots, (a_d, T^{max}_{Resp}(a_d))\}$ be the set of prior observations after d iterations. We denote $p(T^{max}_{Resp}) = \mathcal{N}(\mu, K)$. Here, the mean function is $\mu(\circ)$, and the covariance kernel function is $K(\circ, \circ)$. We define the mean and covariance kernel function as follows.

$$\mu(a_u) = \mathbb{E}(T^{max}_{Resp}(a_u)) \tag{11}$$

$$K(a_u, a_v) = \mathbb{E}\Big(\Big(T^{max}_{Resp}(a_u) - \mu(a_u)\Big)\Big(T^{max}_{Resp}(a_v) - \mu(a_v)\Big)\Big) \tag{12}$$

Let us denote Φ and ϕ as the standard normal cumulative distribution function and the standard normal density function, respectively. We define $U_{min} = \underset{a \in \mathcal{D}_d}{min}(T^{max}_{Resp}(a))$, $\pi = \frac{U_{min} - \mu(a_d)}{\sigma(a_{d-1}, a_d)}$, and $\sigma(a_{d-1}, a_d) = \sqrt{K(a_{d-1}, a_d)}$. We choose the acquisition function (Equation (13)) which is recommended in [3].

$$EI(A|\mathcal{D}_d) = \begin{cases} 0 & \text{if:} \sigma(a_{d-1}, a_d) = 0 \\ \Big(\big(U_{min} - \mu(a_d)\big)\Phi(\pi)\Big) + \big(\phi(\pi)\sigma(a_{d-1}, a_d)\big) & \text{Otherwise} \end{cases} \tag{13}$$

However, $EI(A|\mathcal{D}_d)$ works well in case of unconstrained optimization. We take the procedure suggested by Gardner et al.[8] to satisfy our constrained optimization. By following their approach, we assume that $\mathcal{R} \Leftarrow \mathcal{A} \Rightarrow$ follows Bernoulli process, and *EI* can be modified to Equation (14).

$$EI^c(A|\mathcal{D}_d) = P(\mathcal{R}(A))EI(A|\mathcal{D}_d) \tag{14}$$

There can be observation noise in our setup due to a rise in loads in the fog devices, network delay, etc. Therefore, the proposed BO algorithm must handle noise. We assume that the noise (ζ) is a normally distributed random variable with zero mean, i.e., $\zeta = \mathcal{N}(0, \sigma_\zeta)$.

Algorithm 1: PTC

Input: Resource avalibility of the fog nodes, Resource requirement of the applications
Output: Allocation matrix for which the maximum response time taken by the applications is minimized

1 **Function** *Main()*:
  /* Function for Bayesian Optimization */
2 $\mathcal{D}_0 \leftarrow \emptyset$;
3 **for** $(d \leftarrow 1; HasConverged(\mathcal{D}_{(d-1)}); d++)$ **do**
4 $a_{nxt} \leftarrow \emptyset$;
5 **while** $Length(a_{nxt}) < n_{conf}$ **do**
6 Choose an allocation matrix a_i such that $a_i \notin \{\mathcal{D}_{(d-1)}\}$;
7 **if** *IsFeasible*(a_i) **then**
8 $a_{nxt} \leftarrow \{a_{nxt} \cup a_i\}$;
9 **else**
10 **go to** 5;
11 a_d = Configuration for which the aquisition function is minimum;
12 $\mathcal{D}_d = \left\{\mathcal{D}_{(d-1)} \cup \left(a_d, T^{max}_{Resp}(a_d) + \zeta\right)\right\}$;
13 **return** allocation matrix from $\mathcal{D}_d$ for which the maximum response time taken by the applications is minimized;

1 **Function** *IsFeasible*(a_κ):
  /* Checks feasibility of the allocation matrix */
2 **for** *all applications* $P_x \in \Psi$ **do**
3 **for** $p_{x,y} \in P_x$ **do**
4 $c = \{1, 2, \ldots f\}$;
5 **if** $\nexists_{i \in c} : r^q_i > \gamma^q_{x,y}$ **then**
6 **return** *False*;
7 **else**
8 Remove i from c;
9 **return** *True*;

PTC is described in Algorithm 1. In the Main() function, $\mathcal{D}_0$ is initialized to empty set in Line 2. Lines 3-12 iteratively finds the allocation or configuration for which the aquisition function is minimum. a_{nxt} is initialized to empty set in Line 4. Lines 5-10 generates $(n_{conf} - 1)$ numbers of allocation matrices (a_i). Each of the generated allocation matrices is checked for feasibility. If the allocation matrix is feasible, then that configuration is tested by applying it to the system. Once the testing process is over, the objective function is evaluated, and the next configuration (a_d) is determined based on the minimization of the acquisition function. Line 11 finds the configuration (a_d) for which the aquisition function is minimum. In Line 12, this new configuration (a_d) is added to the set of prior observations set, i.e., $\mathcal{D}_d$ set. This process is repeated until the system converges. PTC algorithm runs iteratively and it stops after reaching convergence. In the IsFeasible(a_κ) function, Line 2 runs for all applications. Line 3 runs for all micro-services of an application. In Line 4, c is the set of all resource types. Line 5-8 checks if any fog node is able to satisfy all the resource requirements of a micro-service. In each iteration, the PTC algorithm checks resource constraint for every micro-service. Therefore, the

computational complexity of PTC is $\mathcal{O}(k * h_x)$. Here, k is the number of edge services and h_x is the total number of micro-services of a service.

6 Performance evaluation

We have implemented the PTC framework in a testbed and a public cloud (Amazon EC2). In the testbed, we analyze the performance of PTC in a realistic environment under a constrained setup. To analyze the system scalability of PTC, we perform the experiments on a large scale over Amazon EC2.

6.1 Testbed setup

In the testbed, the fog nodes are implemented using `Raspberry Pi 3 model b` (https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ (Access: 2021/10/13 08:51:59)) devices, which works like a network gateway. The fog nodes are connected over a local network and provide packet forwarding services to the connected devices. The devices have been configured with `Ubuntu 16.04.2 long-term support (LTS)` (https://www.ubuntu.com/) operating system. `Docker` (https://www.docker.com/) containers are used to provide the sandboxing environment of the micro-services running over the fog nodes. Docker provides a platform for lightweight containers by application layer virtualization. The PTC framework is connected to a private cloud available in our institute.

Figure 2 shows the entire topology of the testbed architecture, which emulates the framework components, as shown in Fig. 1. We give the description of the components used in testbed experiments in Table 3. An ethernet switch has been used to connect various networking components to the local area network (LAN) in the testbed. We have used two switches to connect the fog devices and the client nodes to the ethernet switch. These switches interconnect various fog devices, the fog controller and the client node. A virtual machine running in the private institute cloud is used to host micro-services using Docker over a cloud environment. The fog controller device has `Ubuntu 16.04.2 LTS` (https://www.ubuntu.com/) as the operating system. Client applications are executed on another workstation having a similar configuration to that of the fog controller device. However, the client node only uses a custom TCP socket program to request the fog applications.

6.2 Experimental methodology

Table 4 provides the details of the values taken during the experiments in the testbed. The delays have been set up using the Unix `tc` utility (http://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html). We use `stress-ng` (http://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html) for emulation of memory load due to the primary workload in the fog devices. For memory status monitoring of fog devices and the fog controller, we use Linux `free utility` (http://manpages.ubuntu.com/manpages/bionic/man1/free.1.html). The CPU usage is measured

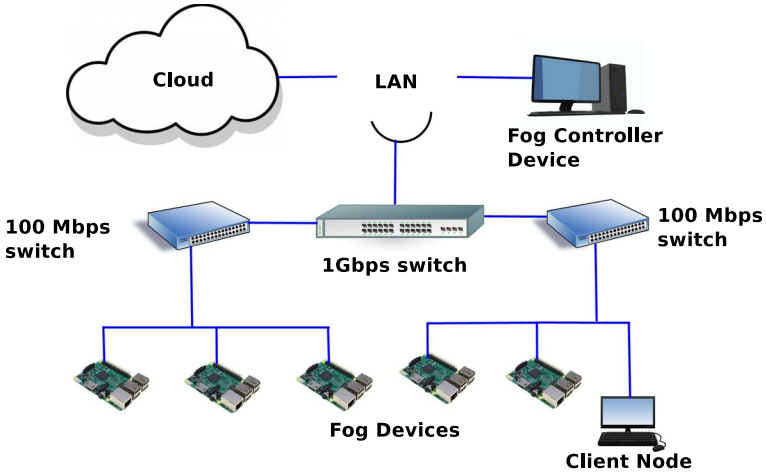


Fig. 2 Testbed topology for the experiments

Table 3 Devices used in testbed experiments

Component	Description
Fog devices	`Raspberry Pi 3 model b` devices (a quad-core 1.2 GHz Broadcom BCM2837 64-bit central processing unit (CPU) and 1 GB random access memory (RAM))
Ethernet wwitch	Netgear GS608 8-Port 1 Gigabit ethernet switch
Desktop wwitches	TP-Link TL-SF1008D 8-Port 10/100Mbps switches
Cloud VM	A cloud VM (4 GB RAM with 2 CPUs) running in the private institute cloud
Fog controller device	Quad-core 3.2 GHz Intel Core i5-4570 processor with 4 GB of RAM
Client node	Quad-core 3.2 GHz Intel Core i5-4570 processor with 4 GB of RAM

using the `iostat` utility (https://man7.org/linux/man-pages/man1/iostat.1.html). The network status is obtained by the `bmon` utility (http://manpages.ubuntu.com/manpages/bionic/man8/bmon.8.html). We use `scp` (http://manpages.ubuntu.com/manpages/trusty/man1/scp.1.html) to transfer files from the fog devices to the fog controller. `Scikit-Optimize (skopt)` (https://scikit-optimize.github.io/) python library is used for the implementation of the BO algorithm. We consider `optical character recognition (OCR)` as a service for testing; for this, we have run `tesseract-OCR web service` (https://hub.docker.com/r/guitarmind/tesseract-web-service/) as a docker container. Also, we have tested PTC with an NLP application, as discussed in Sect. 6.8.

6.3 Competing heuristics

We consider *Foglets* [24], *first-fit* [27], *best-fit* [27] and *mobility-based* [10] mechanisms as the baseline methods to analyze the performance of PTC. In Foglets, the

Table 4 Values of different parameters used in testbed

Parameter	Value
Number of fog devices	5
Number of application types	1
Number of micro-services per service or application	5
Network delay: RTT for fog device–fog device	10–18 ms (if not closest)/0.5 ms (if closest)
Network delay: RTT for fog device–cloud	200 ms
Network delay: RTT for controller device–cloud	200 ms
RAM load in fog devices	20 MB (low loaded)/100 MB (high loaded)
RAM needed for docker container, i.e., micro-service	40 MB
Number of parallel applications or number of scenarios	1 to 8
Number of runs per scenario	3-4

authors have proposed a programming infrastructure for geo-distributed applications, which uses a discovery and deployment protocol to find the fog computing devices with sufficient resources to host an application component. The first-fit mechanism selects the available fog devices for deployment and processing of an image. On the contrary, the best-fit method sorts the processing requests in ascending order according to their demands. In the mobility-based mechanism, the authors have developed a placement to minimize the applications' latency.

6.4 Application response time

We show the performance of PTC in terms of response time in Fig. 3(a). It is observed that PTC is faster than the baselines. The distribution of average response time for PTC converges within 50 seconds. At the same time, the distribution of average response time for baselines converges between 53 seconds and 88 seconds. It is found that PTC converges quickly to an optimal or near-optimal value (within 30 iterations). Table 5 presents a performance comparison of PTC with respect to the baseline approaches as shown in Fig. 3(a).

The PTC algorithm uses BO-based reinforcement learning. As BO can reach the desired solution in fewer iterations, the response time of the application reduces. BO finds the selected points in the search space with relatively few function evaluations. This optimization technique induces adaptive learning over the PTC framework and therefore makes PTC intelligent in dynamically selecting the target fog devices for micro-service placements. The BO-based adaptive reinforcement learning framework helps PTC optimize the utility function with prior observations and posterior distribution, such that PTC can reduce the overall response time. Consequently, the proposed framework learns the dynamic fog environment and their primary

Table 5 Performance of PTC in terms of average response time

Approach	Average response time
Foglets [24]	24.21–80.35 sec
First-fit [27]	16.74–88 sec
Best-fit [27]	11.48–57.39 sec
Mobility-based [10]	12–53 sec
PTC	9.71–50 sec

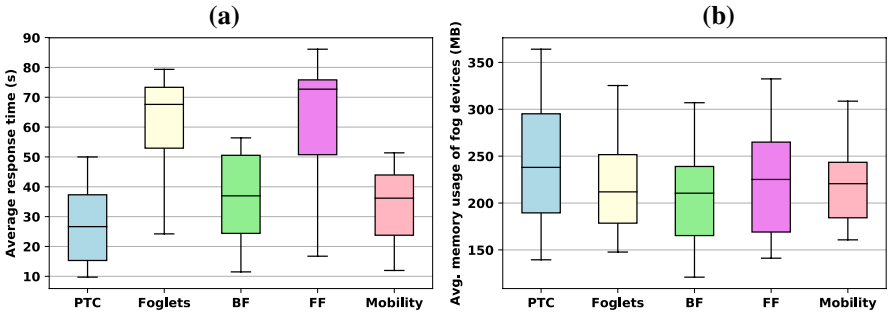


Fig. 3 **a** Distribution of average response time and **b** Distribution of average memory usage of the fog devices

workloads over time and can decide accordingly to support the applications' low response time.

On the contrary, the Foglets mechanism places the application components in fog devices that are closest. Therefore, it cannot find a suitable allocation matrix as the nearest fog nodes' computing resources might be blocked with their primary workloads. In the first-fit mechanism, the available fog devices which have sufficient resources are chosen for the placement. However, this method also does not consider the dynamic primary workload. Thus, in this case, the response time is more than PTC. The best-fit algorithm places the micro-services according to their resource demand. The best-fit response time is also more than PTC as the best-fit algorithm takes more time to deploy the micro-services in fog devices.

On the other hand, mobility-based placement only checks if the applications' latency is minimized while allocating them in the fog devices. This approach tries to maximize the number of tasks deployed in the fog landscape, ensuring the latency constraint. Thus, the average response time is higher in the mobility-based scheme than PTC, particularly when the primary workload observes a spike in the resource demand.

6.5 Resource usage of the fog devices

The memory consumption of the fog nodes is depicted in the Fig. 3b. It may be observed that the PTC consumes more amount of memory than the baseline

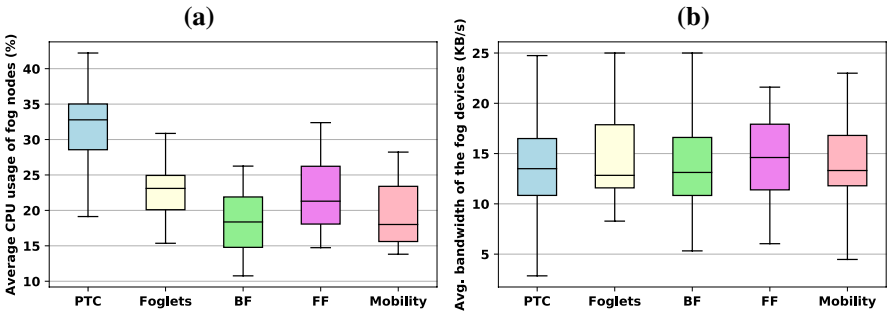


Fig. 4 **a** Distribution of average CPU usage of the fog devices and **b** Distribution of average network bandwidth usage of the fog devices

algorithms. The density of memory usage distribution is higher in PTC (in the range of 139 MB–366 MB) than the baselines. Further, we show the average CPU usage distribution of the fog nodes in Fig. 4a. The CPU usage is also higher in PTC (in the range of 18.43%–44%) than the CPU usage of the baseline algorithms. In a similar line, the bandwidth usage is also higher in PTC (in the range of 3 KB/s–24.74 KB/s) in comparison with other baselines, as shown in the Fig. 4b.

As these statistics are collected over the fog nodes, it indicates the average resource consumption by the micro-services placed on that fog nodes. The figures suggest that PTC can provide better resource utilization to the running micro-services compared to other baselines. We observe that the baseline mechanisms fail to place the micro-services in the most optimized fog devices having better availability of excess resources over time. They only consider the instantaneous measurements and, therefore, do not consider the fog devices' time-varying primary workload characteristics. Consequently, the running micro-services get better computing platform in PTC, resulting in a low response time, as we have seen earlier.

6.6 Resource usage of the fog controller

We show the resource consumption of the fog controller node in Fig. 5(a). As the proposed PTC algorithm runs for multiple iterations, the PTC algorithm's memory consumption is more than the baselines. The controller's average memory usage is higher (in the range of 756 MB–922 MB) in PTC than the baselines. The average CPU consumption of the fog controller device is shown in Fig. 5(b). It is also higher in PTC (in the range of 26%–74%). The average bandwidth usage of PTC is shown in Fig. 6(a). It is also higher (in the range of 11 KB/s–87.28 KB/s) in PTC compared to the baselines. In summary, we observe that the trade-off is in the fog controller's resource usage, where the PTC controller needs more resources than other baselines. However, considering the improvement observed in the application response time, this trade-off is acceptable as the average increase in the resource usage is not very significant, although it is higher than the baselines.

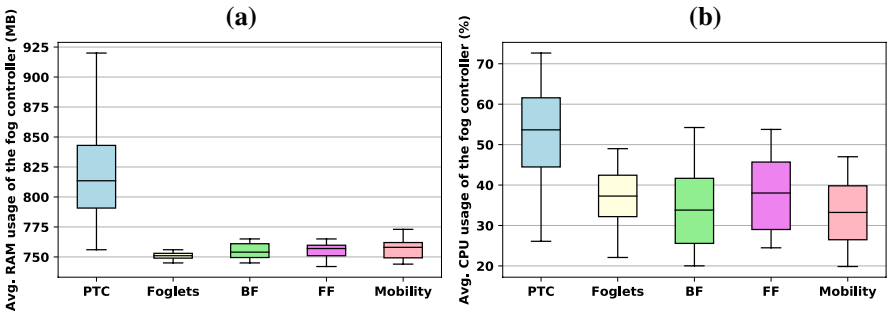


Fig. 5 **a** Distribution of average memory usage of the fog controller device and **b** Distribution of average CPU usage of the fog controller device

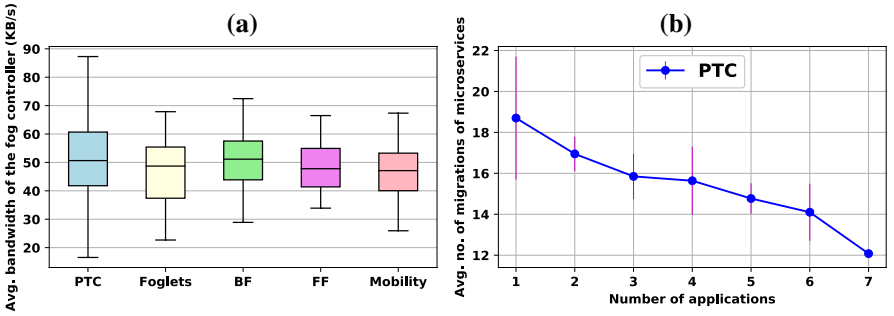


Fig. 6 **a** Distribution of average network bandwidth usage of the fog controller device and (b) Avg. number of migrations of the micro-services

6.7 Average number of micro-service migrations

The proposed PTC algorithm migrates the containerized micro-services to a new fog node if the current fog node becomes loaded. The migration ensures the minimization of service response time. The average number of migrations of the micro-services in Fig. 6b. It is observed that though the total number of migrations increases, the average number of migrations decreases with the increase in the number of applications. It indicates the system's stability at high load.

6.8 Case study with an NLP (Natural Language Processing) application

We have tested PTC with an NLP application—text summarization toolbox[1]. Such an application is important because it supports many practical situations. The evaluation is performed in our fog computing testbed with the configurations given in

[1] https://glowingpython.blogspot.com/2014/09/text-summarization-with-nltk.html (Access: 2021/10/13 08:51:59).

Table 6 Performance of PTC in terms of average response time for the NLP application

Approach	Average response time
Foglets [24]	59.12–131 sec
First-fit [27]	58.86–142.2 sec
Best-fit [27]	49.59–144 sec
Mobility-based [10]	49.37–143.29 sec
PTC	36.75–107 sec

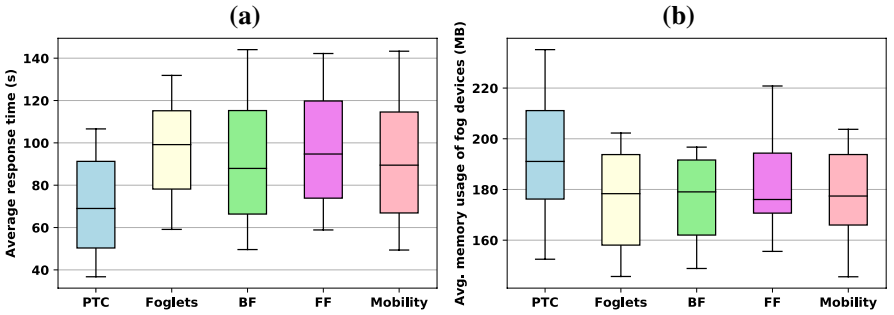


Fig. 7 **a** Distribution of average response time and **b** Distribution of average memory usage of the fog devices

Table 4. The testbed setup given in Sect. 6.1 has been used for the case study. It is observed that the average response time is less for PTC than the baselines. The distribution of the response time in PTC converges within 107 seconds (Fig. 7a), whereas distributions of the response time for the baseline algorithms converge between 131 seconds and 144 seconds. Table 6 shows a performance comparison of PTC with respect to the baseline approaches for the NLP application as shown in Fig. 7a.

The average memory usage is more in PTC, as we have seen earlier. It is in the range of 152 MB–235 MB (shown in Fig. 7b), whereas the average memory distribution of baselines converges within 220 MB for first-fit, within 202 MB for Foglets, within 203 MB for mobility-based and within 196 MB for best-fit, respectively. The CPU usage of the fog devices is higher in PTC in the case of the NLP application. It is in the range of 6%–19% (Fig. 8a) for PTC. The CPU usage distribution of baselines converges within 13% for first-fit, within 14% for best-fit and Foglets methods. The distribution of CPU usage converges within 15% for mobility-based algorithm. The distribution of bandwidth usage of the fog devices is also marginally higher in PTC than the baseline algorithms. For PTC, it is in the range of 5 KB/s–17 KB/s (Fig. 8b). On the contrary, the distribution of network bandwidth usage converges within 18 KB/s for the best-fit mechanism, within 17 KB/s for Foglets and within 16 KB/s for the first-fit and the mobility-based algorithm.

Similar to the previous observation, the average memory usage distribution of the controller is more in PTC compared to other baselines, which is in the range of 756 MB–930 MB (Fig. 9a). The fog controller's average memory usage distribution

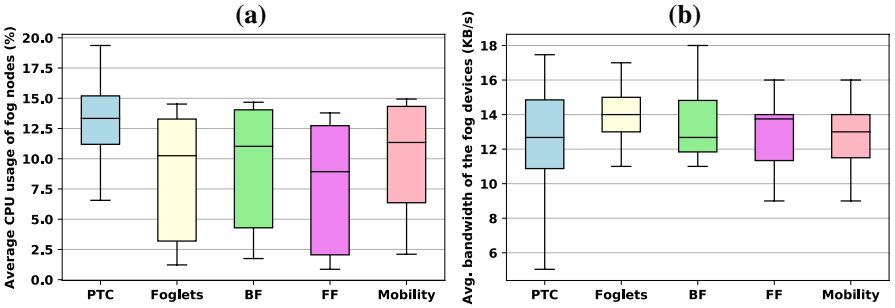


Fig. 8 **a** Distribution of average CPU usage of the fog devices and **b** Distribution of average network bandwidth usage of the fog devices

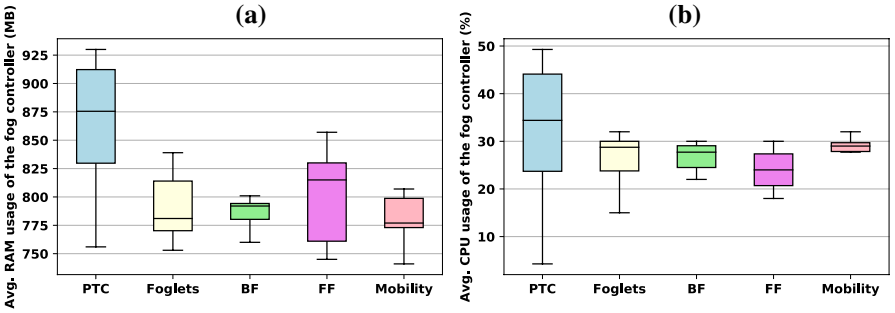


Fig. 9 **a** Distribution of average memory usage of the fog controller device and **b** Distribution of average CPU usage of the fog controller device

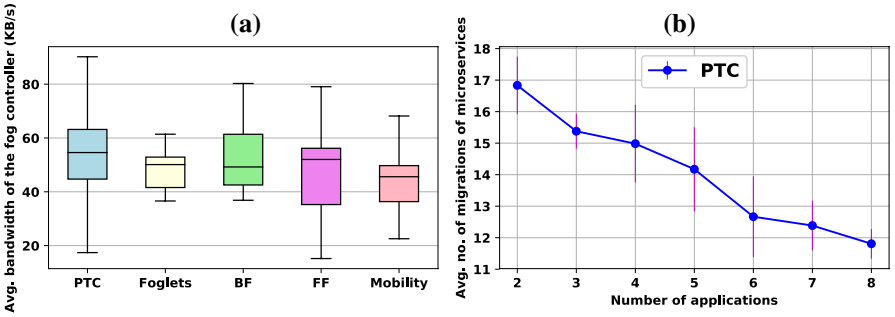


Fig. 10 **a** Distribution of average network bandwidth usage of the fog controller device and **b** Avg. number of migrations of the micro-services

converges within 839 MB for Foglets, within 800 MB for best-fit, within 857 MB for first-fit and within 807 MB for the mobility-based algorithm. The CPU usage of the controller is in the range of 4–49% for PTC. The CPU usage is shown in Fig. 9b. The distribution of baselines converges within 35% for mobility-based algorithm and 32% Foglets, respectively. It converges within 30% for best-fit and first-fit

Table 7 Performance of PTC in terms of average response time for scalability analysis

Approach	Average response time
Foglets [24]	175–650.82 sec
First-fit [27]	211.66–755.34 sec
Best-fit [27]	196–775 sec
Mobility-based [10]	214–634 sec
PTC	106.6–w318 sec

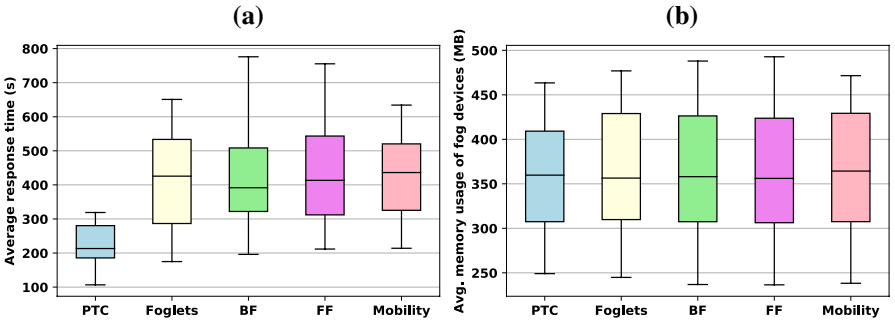


Fig. 11 **a** Distribution of average response time and **b** Distribution of average memory usage of the fog devices

algorithms. The distribution of the average network bandwidth usage of the PTC controller is marginally higher than other baselines. It is in the range of 17 KB/s–98 KB/s (Fig. 10a) for PTC. The distributions of baselines converge within 80 KB/s for best-fit, within 79 KB/s for first-fit, within 61 KB/s for Foglets and within 68 KB/s for the mobility-based mechanism.

The average number of migrations of the micro-services for the NLP application is shown in Fig. 10b. In this case, we also observe that though the total number of migrations increases, the average number of migrations drops with the increase in the number of applications, indicating the system's stability.

6.9 Scalability analysis in Amazon EC2

To test the scalability of the system, we have experimented with Amazon EC2. We have taken `tesseract-OCR web service` as a docker container for testing. We have taken 45 virtual machines (VM) in Amazon EC2. These VMs are considered as the fog devices for the experiments. The VMs have a 2.5 GHz Intel Xeon Family Processor with 1 GB RAM. The number of containers is increased from 80 to 360. The fog controller device has 1 GB of RAM. We show the average response time in 11a. We find that our proposed PTC framework can decrease the services' average response time than the baselines. The distribution of average response time for PTC converges within 318 seconds (Fig. 11a). Distributions of the baselines converge between 634 seconds and 775 seconds. On

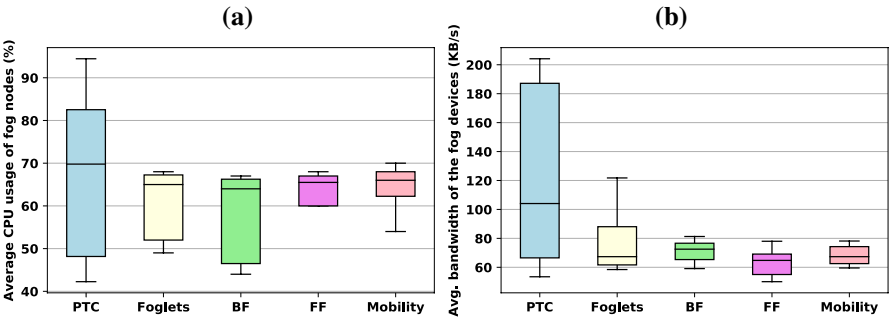


Fig. 12 **a** Distribution of average CPU usage of the fog devices and **b** Distribution of average network bandwidth usage of the fog devices

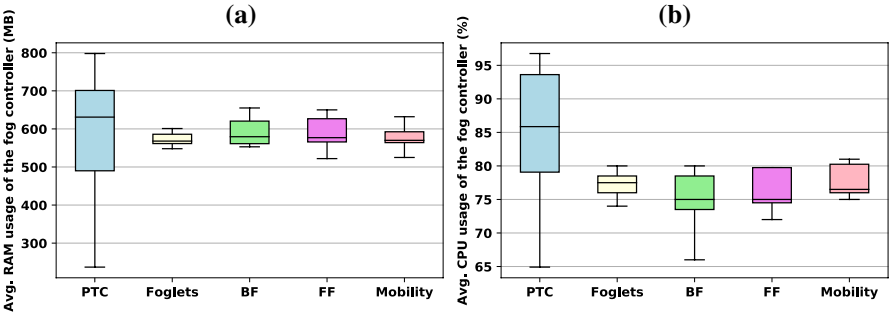


Fig. 13 **a** Distribution of average memory usage of the fog controller device and **b** Distribution of average CPU usage of the fog controller device

average, PTC performs better than the baselines in Amazon EC2 experiments. Thus, our PTC framework is scalable. Table 7 shows a performance comparison of PTC with respect to the baseline approaches for the scalability analysis as shown in Fig. 11a.

It may be observed that the average memory usage of the fog devices is in the range of 250 MB–463 MB in PTC in Amazon EC2 (shown in Fig. 11b). The average memory usage distribution for baseline algorithms converges within 492 MB for first-fit, within 476 MB for Foglets, within 471 MB for mobility-based and within 487 MB for the best-fit algorithm, respectively. The CPU usage of the fog devices is higher in PTC in Amazon AWS experiments. It is in the range of 42%–94% (Fig. 12a). The distribution for baselines converges within 68% for first-fit and Foglets, within 67% for best-fit and within 70% for the mobility-based algorithm, respectively. The distribution of bandwidth usage of the fog devices is marginally higher in PTC than other baselines when experimented with over Amazon EC2. It is in the range of 53 KB/s–204 KB/s (Fig. 12b) for PTC. On the contrary, such distributions converge within 81 KB/s for the best-fit mechanism, within 121 KB/s for Foglets, within 77 KB/s for first-fit and within 78 KB/s for the mobility-based algorithm. The overhead in terms of the fog controller's

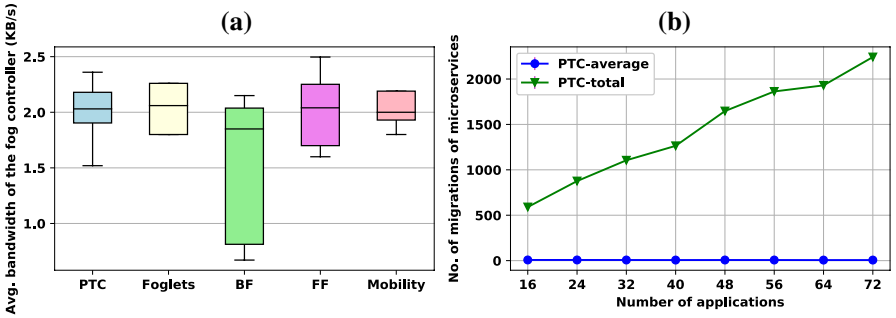


Fig. 14 **a** Distribution of average network bandwidth usage of the fog controller device and **b** Number of migrations of the micro-services

memory usage is shown in Fig. 13a. We have observed that the controller's average memory usage distribution is more in PTC in the Amazon EC2 experiments. It is in the range of 237 MB–798 MB (Fig. 13(a)). On the contrary, the fog controller's average memory usage distribution converges within 650 MB for first-fit, within 601 MB for Foglets, within 655 MB for best-fit and within 649 MB for the mobility-based algorithm.

Similar to our previous observations, the CPU usage of the controller is higher in PTC than other baselines and it is in the range of 65–97% (Fig. 13(b)). The distribution for baselines converges within 81% for mobility-based algorithm and 80% Foglets, respectively. It converges within 80% for best-fit and first-fit algorithm. The distribution of the average bandwidth distribution of the controller for PTC is in the range of 1.52 KB/s–2.36 KB/s (Fig. 14(a)). In contrary, the distribution for baselines converges within 2.15 KB/s for best-fit, within 2.49 KB/s for first-fit, within 2.26 KB/s for Foglets and within 2.19 KB/s for the mobility-based mechanism.

The number of migrations of the micro-services for the Amazon EC2 experiments is shown in Fig. 14(b). In these experiments, we observe that the total number of migrations increases with the number of applications. Also, the average number of micro-service migrations is in the range of 6.03 to 7.37.

7 Conclusion

With the widespread deployment of IoT-based services, the concept of fog computing can provide a useful solution for low-latency privacy-ensured data processing by utilizing the in-network processing capability of various devices. Although many existing works have focused on scheduling and deployment of application micro-services over fog nodes, the dynamicity of the fog devices' primary workloads is still a concern. In this paper, we have proposed PTC, a reinforcement learning-induced framework for continually monitoring the fog devices' primary workloads and accordingly dynamically deciding the execution platform for the application micro-services. PTC solves a hard time-varying optimization problem with Bayesian optimization, which dynamically finds a solution based on prior observations.

We have implemented PTC over a small-scale testbed setup and evaluated its performance over a large-scale emulation with Amazon EC2 clouds. The experiments confirm that PTC can reduce the application response time with better utilization of the fog device's excess resources, with the cost of a moderately complicated mechanism, which needs an additional edge server for execution.

We believe that the proposed framework can introduce dynamic execution of IoT data processing workloads by utilizing the fog devices' in-network computing facilities. Therefore, such a framework can be useful in various application scenarios, like smart home, smart manufacturing environments, etc. As future work, we plan to deploy the proposed framework with a complete IoT data collection and processing framework to build up an entire system by utilizing its performance. In future, we would like to perform evaluation of the proposed PTC framework with different IoT applications having different data processing needs.

References

1. Ahmad M, Amin MB, Hussain S, Kang BH, Cheong T, Lee S (2016) Health fog: a novel framework for health and wellness applications. J Supercomput 72(10):3677–3695
2. Ahmed A, Pierre G (2018) Docker container deployment in fog computing infrastructures. In: 2018 IEEE International Conference on Edge Computing (EDGE), pp. 1–8. IEEE
3. Alipourfard O, Liu HH, Chen J, Venkataraman S, Yu M, Zhang M (2017) Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp. 469–482
4. Alturki B, Reiff-Marganiec S, Perera C, De S (2019) Exploring the effectiveness of service decomposition in fog computing architecture for the internet of things. IEEE Transactions on Sustainable Computing
5. Bernstein D (2014) Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput 1(3):81–84
6. Drezner Z, Wesolowsky GO (1983) Minimax and maximin facility location problems on a sphere. Naval Res Logistics Quart 30(2):305–312
7. Elgamal T, Sandur A, Nguyen P, Nahrstedt K, Agha G (2018) Droplet: distributed operator placement for iot applications spanning edge and cloud resources. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 1–8. IEEE
8. Gardner JR, Kusner MJ, Xu ZE, Weinberger KQ, Cunningham JP (2014) Bayesian optimization with inequality constraints. ICML 2014:937–945
9. Goethals T, De Turck F, Volckaert B (2020) Near real-time optimization of fog service placement for responsive edge computing. J Cloud Comput 9(1):1–17
10. Gonçalves D, Velasquez K, Curado M, Bittencourt L, Madeira E (2018) Proactive virtual machine migration in fog environments. In: 2018 IEEE Symposium on Computers and Communications (ISCC), pp. 00742–00745. IEEE
11. Gu L, Zeng D, Guo S, Barnawi A, Xiang Y (2015) Cost efficient resource management in fog computing supported medical cyber-physical system. IEEE Trans Emerg Top Comput 5(1):108–119
12. Javadzadeh G, Rahmani AM (2020) Fog computing applications in smart cities: a systematic survey. Wireless Netw 26(2):1433–1457
13. Kayal P, Liebeherr J (2019) Distributed service placement in fog computing: an iterative combinatorial auction approach. In: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pp. 2145–2156. IEEE
14. Kecskemeti G, Marosi AC, Kertesz A (2016) The entice approach to decompose monolithic services into microservices. In: 2016 International Conference on High Performance Computing & Simulation (HPCS), pp. 591–596. IEEE
15. Li DC, Huang CT, Tseng CW, Chou LD (2021) Fuzzy-based microservice resource management platform for edge computing in the internet of things. Sensors 21(11):3800

16. Li X, Wan J, Dai HN, Imran M, Xia M, Celesti A (2019) A hybrid computing solution and resource scheduling strategy for edge computing in smart manufacturing. IEEE Trans Industr Inf 15(7):4225–4234
17. Liao S, Wu J, Mumtaz S, Li J, Morello R, Guizani M (2020) Cognitive balance for fog computing resource in internet of things: an edge learning approach. IEEE Trans Mobile Comput
18. Mukherjee M, Shu L, Wang D (2018) Survey of fog computing: fundamental, network applications, and research challenges. IEEE Commun Surv Tutorials 20(3):1826–1857
19. Mutlag AA, Abd Ghani MK, Arunkumar Na, Mohammed MA, Mohd O (2019) Enabling technologies for fog computing in healthcare iot systems. Future Gener Comput Syst **90**, 62–78
20. Nadgowda S, Suneja S, Bila N, Isci C (2017) Voyager: Complete container state migration. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 2137–2142. IEEE
21. Nath SB, Chattopadhyay S, Karmakar R, Addya SK, Chakraborty S, Ghosh SK (2019) Ptc: Pick-test-choose to place containerized micro-services in iot. In: 2019 IEEE Global Communications Conference (GLOBECOM), pp. 1–6. IEEE
22. Rossi F, Cardellini V, Presti FL (2019) Elastic deployment of software containers in geo-distributed computing environments. In: 2019 IEEE Symposium on Computers and Communications (ISCC), pp. 1–7. IEEE
23. Rossi F, Cardellini V, Presti FL, Nardelli M (2020) Geo-distributed efficient deployment of containers with kubernetes. Comput Commun 159:161–174
24. Saurez E, Hong K, Lillethun D, Ramachandran U, Ottenwälder B (2016)Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, pp. 258–269
25. Singh SP, Nayyar A, Kumar R, Sharma A (2019) Fog computing: from architecture to edge computing and big data processing. J Supercomput 75(4):2070–2105
26. Snoek J, Larochelle H, Adams RP (2012) Practical bayesian optimization of machine learning algorithms. Adv Neural Inf Proc Syst 25:1
27. Souza VB, Masip-Bruin X, Marín-Tordera E, Sànchez-López S, Garcia J, Ren GJ, Jukan A, Ferrer AJ (2018) Towards a proper service placement in combined fog-to-cloud (f2c) architectures. Futur Gener Comput Syst 87:1–15
28. Stévant B, Pazat JL, Blanc A (2018) Optimizing the performance of a microservice-based application deployed on user-provided devices. In: 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 133–140. IEEE
29. Taherizadeh S, Apostolou D, Verginadis Y, Grobelnik M, Mentzas G (2021) A semantic model for interchangeable microservices in cloud continuum computing. Information 12(1):40
30. Taherizadeh S, Stankovski V, Grobelnik M (2018) A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. Sensors 18(9):2938
31. Taneja M, Davy A (2017) Resource aware placement of iot application modules in fog-cloud computing paradigm. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 1222–1228. IEEE
32. Wang S, Guo Y, Zhang N, Yang P, Zhou A, Shen XS (2019) Delay-aware microservice coordination in mobile edge computing: a reinforcement learning approach. IEEE Trans Mobile Comput
33. Wang W, Zhao Y, Tornatore M, Gupta A, Zhang J, Mukherjee B (2017) Virtual machine placement and workload assignment for mobile edge computing. In: 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), pp. 1–6. IEEE
34. Yigitoglu E, Mohamed M, Liu L, Ludwig H (2017) Foggy: a framework for continuous automated iot application deployment in fog computing. In: 2017 IEEE International Conference on AI & Mobile Services (AIMS), pp. 38–45. IEEE
35. Yin L, Luo J, Luo H (2018) Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing. IEEE Trans Industr Inf 14(10):4712–4721

Authors and Affiliations

Shubha Brata Nath[1] · Subhrendu Chattopadhyay[2] · Raja Karmakar[3] · Sourav Kanti Addya[4] · Sandip Chakraborty[1] · Soumya K Ghosh[1]

Shubha Brata Nath
nath.shubha@gmail.com

Subhrendu Chattopadhyay
subhrendu@iitg.ac.in

Raja Karmakar
rkarmakar.tict@gmail.com

Sourav Kanti Addya
kanti.sourav@gmail.com

Soumya K Ghosh
skg@cse.iitkgp.ac.in

1 Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, 721302 Kharagpur, India

2 Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, 781039 Guwahati, India

3 Techno International New Town, New Town 700156, India

4 Department of Computer Science and Engineering, National Institute of Technology Karnataka, Surathkal 575025, India