Scientific Research Publishing

Communications and Network, 2020, 12, 122-154
https://www.scirp.org/journal/cn
ISSN Online: 1947-3826
ISSN Print: 1949-2421

Enhancing Cloud-Based IoT/M2M System Scalability by Dynamic Network Slicing

David de la Bastida[1], Fuchun Joseph Lin[2]

[1]EECS International Graduate Program, National Chiao Tung University, Taiwan
[2]Department of Computer Science, College of Computer Science, National Chiao Tung University, Taiwan
Email: josedelabastida.eed03g@nctu.edu.tw, fjlin@nctu.edu.tw

How to cite this paper: de la Bastida, D. and Lin, F.J. (2020) Enhancing Cloud-Based IoT/M2M System Scalability by Dynamic Network Slicing. *Communications and Network*, **12**, 122-154.
https://doi.org/10.4236/cn.2020.123007

Received: July 13, 2020
Accepted: August 25, 2020
Published: August 28, 2020

Abstract

With ever-increasing applications of IoT, and due to the heterogeneous and bursty nature of these applications, scalability has become an important research issue in building cloud-based IoT/M2M systems. This research proposes a dynamic SDN-based network slicing mechanism to tackle the scalability problems caused by such heterogeneity and fluctuation of IoT application requirements. The proposed method can automatically create a network slice on-the-fly for each new type of IoT application and adjust the QoS characteristics of the slice dynamically according to the changing requirements of an IoT application. Validated with extensive experiments, the proposed mechanism demonstrates better platform scalability when compared to a static slicing system.

Keywords

Internet of Things, Platform Virtualization, Quality of Service, Scalability, Software Defined Networking

1. Introduction

As millions of Internet of Things/Machine to Machine (IoT/M2M) devices are connected to the cloud, the IoT/M2M platform normally deployed in the cloud needs to be constructed with scalability design to serve a massive amount of IoT/M2M requests thus generated [1]. IoT/M2M system scalability can be achieved by horizontal scalability, which means scaling out or in server instances. Horizontal scalability of web applications has been extensively studied with many good results [2] [3] [4] [5]. Nevertheless, such horizontal scalability ignores the nature of IoT/M2M applications with heterogeneous and bursty Quality of Service (QoS) requirements.

Vertical scalability where the capability of server instances and their support networks can be upgraded or downgraded based on the changing requirements of QoS is a more viable approach. Normally, this implies providing different infrastructures of computing, storage, and network to IoT/M2M applications with different QoS requirements (e.g. reliability, delay and throughput). However, this will be too costly in terms of capital expenditures (CAPEX) and operating expenses (OPEX). The emergence of the fifth generation (5G) systems and its network slicing capability has paved a feasible path to tackle this problem [6].

With the upcoming 5G, network operators are expected to offer virtualized infrastructures with different QoS characteristics on top of the same physical network infrastructure based on Software-Defined Networking/Network Function Virtualization (SDN/NFV) technologies. Each virtual network infrastructure is thus called a "network slice" [7]. Network slices allow the realization of tailor-made networks according to the QoS requirements of specific applications. This is radically different from the one-size-fits-all approach as used in the fourth generation (4G) networks.

Our research proposes to improve the cloud-based IoT/M2M system scalability by assigning each different type of IoT/M2M applications to a different network slice configured with the appropriate QoS. Our published work [8] first addressed this problem by pre-provisioning a fixed number of network slices to serve the same number types of IoT/M2M applications. Nevertheless, the endless potential of IoT/M2M innovations makes such a static approach not practically applicable. First, considering the fact that many new types of IoT/M2M applications are yet to appear, it is not feasible to define in advance all network slices required by all potential varieties of IoT/M2M applications. Instead, an automatic procedure should be developed so that a new network slice can be created on-the-fly for each new kind of IoT/M2M applications. Second, to deal with the problem of potential fluctuation of incoming requests, a created slice will need to be dynamically adjustable whenever the QoS requirement of incoming requests is changed. Third, whenever a network slice created for a particular type of IoT application is no longer needed, it has to be removed.

As a result, in this work we propose the use of a dynamic SDN-based network slicing environment to effectively enhance IoT/M2M platform scalability in the cloud. Our main contributions are as follows:

1) Designing a Slice Manager to manage network slices including slice creation, deletion, and QoS adjustment.

2) Designing an Application Classifier to identify any new type of IoT/M2M applications and monitor, estimate and forecast its QoS requirements for further scalability management.

3) Building a system prototype to validate the usability of our dynamic approach to enhance IoT/M2M system scalability versus that of a static approach, comparing response time, number of requests, throughput, CPU and memory usage, as well as power consumption.

The rest of the paper is organized as follows: Section 2 discusses our previous effort in applying network slicing to IoT/M2M platform scalability, presents the system requirements and motivation of a dynamic network slicing environment, and discusses its related work. Section 3 introduces our proposed mechanism for achieving IoT/M2M scalability via dynamic network slicing. Section 4 explains the key components of our experimental setup and the results of scalability testing. Finally, in Section 5 we provide our conclusions and discuss future research topics.

2. System Requirements and Related Work

As stated in [9], multiple Standards Developing Organizations (SDOs) and Fora have proposed different definitions of network slicing depending on their own context [10] [11] [12] [13] [14]. Ultimately, all definitions fit into two categories: NFV-based or SDN-based network slicing.

In the NFV-based network slicing [15] [16], network slice instances (NSIs) are composed of one or more network slice subnet instances (NSSIs), which in turn are composed of one or more interconnected Physical Network Functions (PNFs) and/or Virtual Network Functions (VNFs) running on virtual machines or containers. In terms of implementation, NFV-based slicing follows the NFV-MANO (Management and Orchestration) specification [17] where NSSIs are instances of network services (NSs) created upon previously onboarded network service descriptors (NSDs) with some particular pre-established QoS configuration (*i.e.* deployment flavor). The NFV-MANO architecture includes an NFV orchestrator (NFVO) that manages the onboarding and instantiation of NSs, a VNF manager (VNFM) that takes care of the lifecycle of VNFs, and a virtual infrastructure manager (VIM) that interacts with the network function virtualization infrastructure (NFVI) for the actual deployment of VNFs.

In the SDN-based network slicing [18], network slice instances are isolated partitions of physical and/or virtual resources (e.g. compute, storage, and network) bridging clients with custom services. An SDN controller plays a centric role in the SDN-based architecture as it mediates client requirements (*i.e.* client context) with resource availability that enables custom services (*i.e.* server context). Moreover, some components of the NFV-MANO can be mapped to those in the SDN-based slicing architecture [9]. For example, the SDN controller plays the role of NFVO, the SDN controller's client context plays the role of NS, and the SDN controller's server context plays the role of NFVI. Furthermore, the interaction of SDN and NFV is feasible as proposed by the European Telecommunications Standards Institute (ETSI) [19].

In this research, we utilize an SDN instead of NFV-based network slicing approach to improve IoT/M2M system scalability because our goal is to automatically build network slices as partitions of the underlying network resources. This means that we do not create network slices based on pre-provisioned NSDs as in NFV-MANO, but do it dynamically and on-demand. In addition, by adjusting

the QoS of slices automatically we are not limited to a list of predefined deployment flavors of slices as in NFV-MANO.

To our best knowledge, very few works have utilized either SDN or NFV slicing to address the issue of system scalability for IoT/M2M platforms. E. Kapassa *et al.* [20] proposed an NFV-based framework that allows the creation of dynamic slices for IoT applications with diverse QoS requirements. However, their results on the improvement of scalability are not clear since their framework was not yet implemented. V. P. Kafle *et al.* [21] proposed a three-in-one (*i.e.* vertical, horizontal, and inter-slices) scalability approach based on SDN and NFV slicing. The authors provided results based on computer-based simulations and only for CPU resource provisioning while letting other resources like memory, storage, and throughput pending for future work.

Our previous efforts in [22] [23] proposed an OpenStack-based, horizontal, highly scalable system with good scalability results. However this method assumes that the network handles each IoT/M2M application in isolation from others. Such an assumption may not be realistic since multiple applications could arrive concurrently at the cloud. Hence, in a follow-up effort, we started to take the distinct QoS requirements (*i.e.* reliability, delay, and throughput) of different applications into consideration and assign each of them to a QoS-specific network slice [8] [23]. We have shown that by mapping various types of IoT/M2M applications to different network slices, it can significantly improve the platform scalability in terms of response time, power consumption, and computational cost.

Nevertheless, this mapping assumes that there is a fixed number of known IoT/M2M applications with the same number of SDN network slices. But realistically speaking, not only applications may not be known in advance, but also the network slices thus required may not yet exist. Furthermore, the QoS setting of each slice needs to be adjustable according to the bursty nature of IoT/M2M applications [24].

As a result, we set the requirements of our target system in this work as follows:

- Need identify the type of IoT/M2M applications based on a stream of incoming HTTP requests and consider the QoS requirements (*i.e.* reliability, delay, and throughput) of each application.
- Need automatically create network slices with the required QoS for each type of identified applications.
- Need monitor the changes on the QoS requirements of applications and adjust the QoS of their assigned slices accordingly.

Below we provide more details about these system requirements and their current state of the art in the literature.

2.1. Identifying IoT/M2M Applications and Estimating Its QoS

The system under scalability study in this research is the IoT/M2M platform

such as oneM2M [25] that receives the IoT/M2M applications in the form of RESTful requests. Below is a brief introduction to oneM2M as a typical example of such IoT/M2M platforms. These systems are normally defined as a standard for the IoT/M2M service layer. For oneM2M, its architecture is defined as shown in Figure 1. Its functional architecture comprises three entities: Application Entity (AE), Common Services Entity (CSE) and Network Services Entity (NSE), and four interface points: Mca, Mcn, Mcc, and Mcc', spanning across a field domain and an infrastructures domain. The CSE supports many Common Service Functions (CSFs) such as registration, group management, discovery, and security offered to AEs and other CSEs via Mca and Mcc/Mcc'.

Like many other IoT/M2M platforms, oneM2M adopts a resource-based information model. All entities in the oneM2M platform, such as AEs, CSEs, and data, are all represented as resources, and its resources form a hierarchical tree called resource tree. All oneM2M resources can be manipulated by RESTful APIs. The oneM2M currently supports HTTP, CoAP, MQTT and Websocket protocols. This research will focus on the IoT/M2M applications represented as a stream of HTTP RESTful requests in the form of CRUD (Create, Read, Update and Delete) over a TCP/IP infrastructure.

The issue of identifying IoT/M2M applications has been studied mostly by using machine learning methods [26] [27] [28]. Although such methods can deal with classification and clustering of applications, the referenced studies assume that IoT/M2M applications are predictable and can be categorized by predefined classes. However, IoT/M2M applications are, in reality, heterogeneous and it is impossible to construct a finite list of predefined categories for IoT/M2M application identification. Instead, a system should leverage some practical mechanisms embedded in the nature of IoT/M2M platforms (e.g. unique addressability of resources in the oneM2M) to achieve application identification.

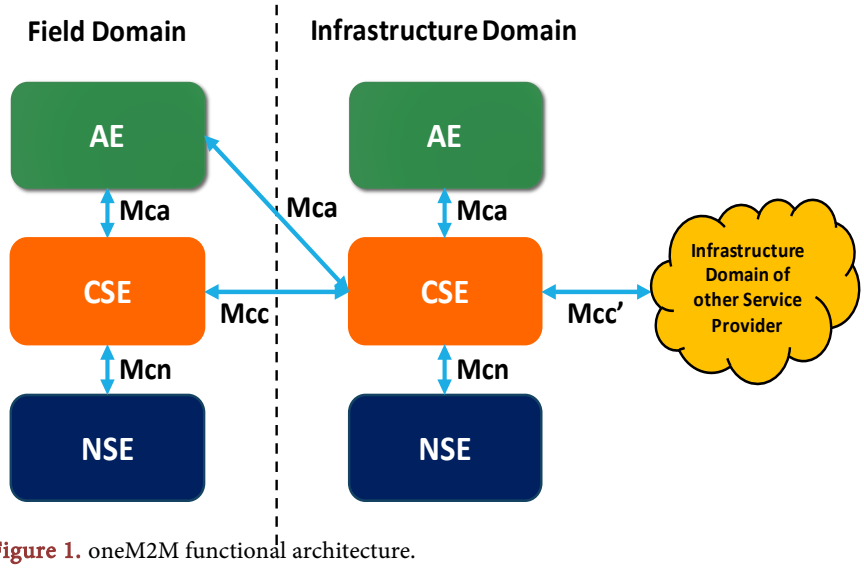

Figure 1. oneM2M functional architecture.

To estimate the QoS of IoT/M2M applications, previous research such as [29] proposed a collaborative approach, which consists of several IoT/M2M devices sharing their QoS usage experience with other devices in order to predict the QoS state of the network. This method assumes all the IoT/M2M devices have enough computational capacity to calculate and share their own QoS needs, which is not always true in the IoT/M2M. In addition, the research in [30] proposed a more general approach where the SDN controller prioritizes applications based on the context data acquired from devices, service providers, and users. This method requires collecting, categorizing and processing a substantial amount of data that could result in excessive consumption of computational resources.

In this research, we leverage the message transmission statistics of the applications to IoT/M2M platforms to conduct both application identification and QoS estimation. We use a multi-threaded traffic generator to simulate the HTTP requests sent by 50 to 100 User Equipment (UEs). Each thread simulates a UE for each type of IoT/M2M application. For the same type of application, its UEs will send the HTTP requests to access the same resource in the infrastructure domain. Thus, their HTTP requests all contain the same Uniform Resource Identifier (URI) destination [25]. Moreover, we assume each type of application is associated with a given reliability requirement. This enables a straightforward method for identifying the type of IoT/M2M application with its associated reliability from its unique addressability in the resource tree.

To estimate the remaining QoS-related metrics for each identified application, our approach consists of collecting enough statistics from the incoming HTTP requests so that those metrics can be fairly estimated. As illustrated by **Figure 2**, our approach first identifies which application (e.g. A, B and C in **Figure 2**) an incoming request belongs to, from a continuous stream of oneM2M requests. Then, with the collection of sufficient data, the delay and throughput of each application can be calculated. These along with the given reliability of the application gives us all QoS requirements of the application.

Once an application type and its QoS requirements are identified and estimated, a QoS-matched network slice will be created and configured to serve that application. Furthermore, the application will be continuously monitored in order to detect any changes in the QoS requirements of each application and update the QoS characteristics of its serving slice accordingly.

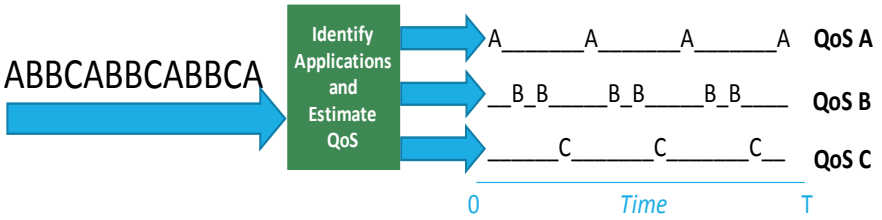

Figure 2. Estimation of QoS of different IoT/M2M applications.

2.2. Automatic Network Slice Creation

SDN technologies allow a single physical network be partitioned into multiple virtual networks with each configured with specific QoS capabilities. Each virtual network can be viewed as a SDN-based slice; it may consist of multiple end-to-end forwarding paths. However, in this research we assume each SDN-based slice consists of only one end-to-end forwarding path. Furthermore, each end-to-end forwarding path is allocated to a slice exclusively such that a slice can exercise its vertical scalability on the same path without interfering with the scalability operations of other slices. An IoT/M2M server instance will be deployed in a slice to serve the continuous stream of HTTP RESTful requests generated by the application.

Few approaches in the literature proposed an SDN system architecture for automated management of network slices. For example, in [31] authors proposed a framework that leverages SDN-based UE virtualization schemes and creates a representation of UEs in slices in the cloud. In [32] authors proposed a system architecture to enable the management of slices focused in the Industrial IoT 4.0. These works considered complex categories of IoT/M2M applications like mobile broadband and industry 4.0 but didn't suggest any general-purpose architecture capable of either automatically creating slices with appropriate QoS for new applications or updating the QoS of slices based on fluctuation of application QoS. We achieve these capabilities by constructing an experimental system based on Open vSwitch (OVS) [33] in Linux with OpenDaylight (ODL) SDN Controller [34]. Virtual networks are created on top of OVS switches through the southbound OpenFlow APIs and the northbound RESTCONF APIs of OpenDaylight [35].

The dynamic creation of network slices is enabled by an application making requests to the SDN controller using the RESTCONF northbound APIs. At receiving such a request, the SDN controller would convert it into the corresponding rules deployed in OVS switches through OpenFlow Southbound APIs. A network slice can then be formed by configuring the OVS switch behavior to create a sequence of paths among the connected OVS switches. Once the paths have been established, it is possible to set the QoS for a slice by configuring its reliability, delay, and throughput parameters on one of the Ethernet ports of the connected OVS switches.

2.3. Monitoring QoS Changes of Heterogeneous and Bursty IoT/M2M Applications

The heterogeneity of IoT/M2M applications is illustrated in **Table 1**, where the main QoS characteristics of eight typical types of IoT/M2M applications are illustrated. All eight applications are different in terms of their message's payload size, required throughput and minimum reliability needed to achieve normal operations, based on their respective references: smart meter [36], Bluetooth tags [37], eHealth [38], video [39], smart parking [40], intrusion detection [41], food monitoring [42], and air pollution [43].

Table 1. QoS characteristics of different IoT/M2M applications.

Application	Payload Size	Throughput	Reliability
Smart Meter [36]	1000 b	1000 b/s	99.99%
Bluetooth Tag [37]	220 b	190 b/s	99.99%
eHealth [38]	160 b	650 b/s	99.9999%
Video [39]	10000 b	10000 b/s	99.99%
Smart Parking [40]	380 b	2000 b/s	99.99%
Intrusion Detection [41]	3000 b	3000 b/s	99.9999%
Food Monitoring [42]	690 b	1500 b/s	99.99%
Air Pollution [43]	240 b	750 b/s	99.99%

Similarly, the bursty nature of IoT/M2M applications is illustrated in **Figure 3**, where the throughput requirements of two example IoT/M2M applications: eHealth and Smart Parking exhibit constant changes during a day [44]. The same concept holds true for the other six applications in **Table 1**, and to any other IoT/M2M application.

In this research, a stream of oneM2M requests belonging to different IoT/M2M applications constitutes the input for our proposed system while a network slice for each type of application with adequate QoS support constitutes the output. **Figure 4** illustrates the input that consists of requests sent by eight types of applications where each application UE is simulated by a thread sending requests at a particular frequency. In addition, as explained earlier each application embeds its desired reliability in the URI of each request. **Figure 4** also shows how our proposed system uses the throughput and the payload size information of **Table 1** to derive the frequency of requests and the delay, while enforces the requested reliability in the network slice created for each application.

For example, if one hundred threads of smart meter run at a given time, the input of the system becomes 100 requests per second, where each request has a payload of 1000 bytes and includes the desired reliability of 99.99% in the URI. Then, the system would create a slice and configure its QoS support with 99.99% for reliability, at least 100,000 bytes/second for throughput, and at most 655 milliseconds for delay (details in Section 3). In this research, we consider only reliability, delay and throughput as the main QoS requirements to keep focus on realizing a general framework to enhance IoT/M2M scalability. Other QoS requirements such as power utilization, level of security or availability are not considered at the moment but reserved as future work.

In [45] the authors fed their proposed algorithm with real-time and archived data of IoT applications for QoS monitoring and estimation purposes. In [29], the authors utilized machine learning algorithms fed with pre-provisioned datasets to conduct QoS monitoring of IoT applications. Despite the good results of these two methods, they could not handle newly appeared applications. In addition, the changing nature of IoT applications might turn their use of historical data unreliable.

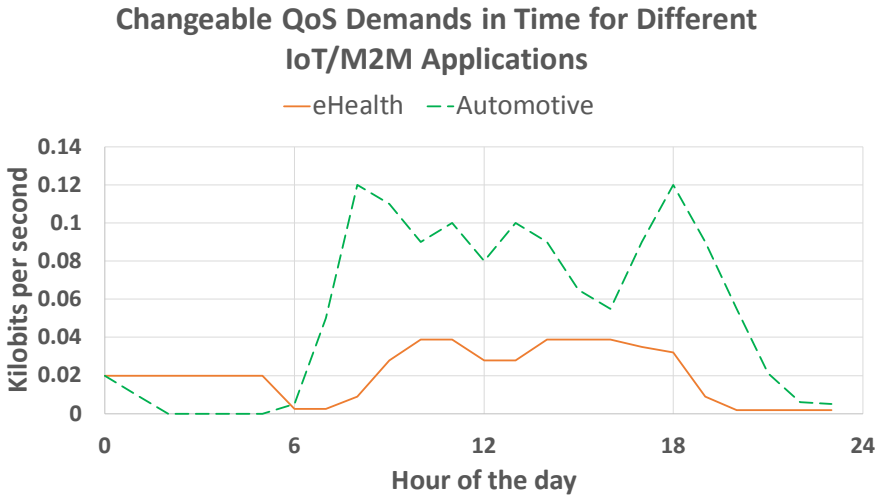

Figure 3. The fluctuation of throughput in a day for two IoT applications [44].

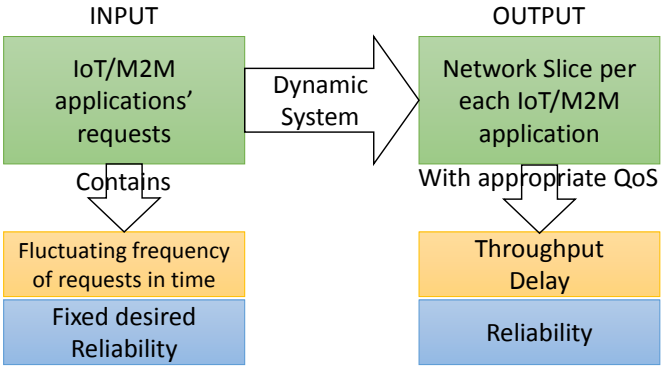

Figure 4. Input and output of our proposed system.

Also in [30], the author proposed an architecture for QoS monitoring and management of applications in a smart home by requesting data from IoT devices, services, and users. This approach enabled real-time QoS monitoring and estimation. But, it just set the QoS using a few predefined QoS policies.

Unlike the above efforts, we intend to design a system capable of not only identifying and estimating different QoS requirements of IoT/M2M applications but also performing scalability according to the dynamic QoS changes of the application behavior.

3. Dynamic SDN Network Slicing for Enhancing IoT/M2M Platform Scalability

In this research, we assume that when a new type of IoT application appears, it will persist and keep sending oneM2M requests to the system continuously, therefore the collection of enough data during a period of 5 minutes in order to derive the QoS requirements of the application is guaranteed. In addition, we assume that initially the system starts from a clean slate. That's, no application types have been mapped and the database that contains the mapping of application to network slice is empty. Furthermore, we assume the system under study

possesses sufficient underlying physical resources (*i.e.* computing, network, and storage) for creating network slices and for each application type, only one slice will be assigned.

During system operations, the incoming applications can be classified into two categories: mapped and unmapped. Mapped applications are those already appeared and mapped with network slices, while unmapped applications are those newly appeared and yet to be mapped with appropriate slices.

Figure 5 illustrates the high-level procedure of handling mapped types of applications. When the system receives incoming CRUD messages, it first performs identification of application type, and then forwards the message to the corresponding network slice. Simultaneously, the system continuously monitors the changes of QoS requirements of the application and scales up or down the serving slice accordingly. When scalability is needed, the system utilizes one thread to buffer all the incoming messages while employing another thread to execute the scalability process. The buffered messages would be sent to the corresponding slice after the scalability process is finished to continue with normal operations.

On the other hand, Figure 6 illustrates the high-level procedure of handling unmapped applications. Immediately after the system discovers a new type of application and its required reliability from its application identification, it launches one thread to put all the incoming messages in a buffer until the slice is created and another thread to accumulate message transmission statistics and estimate the other application QoS metrics (*i.e.* delay and throughput). After estimating the QoS, the system would dynamically create a network slice and assign it to serve the newly mapped application type. It would also update the application-slice mapping database, and clear the unmapped application messages buffer by forwarding its content to the newly created slice.

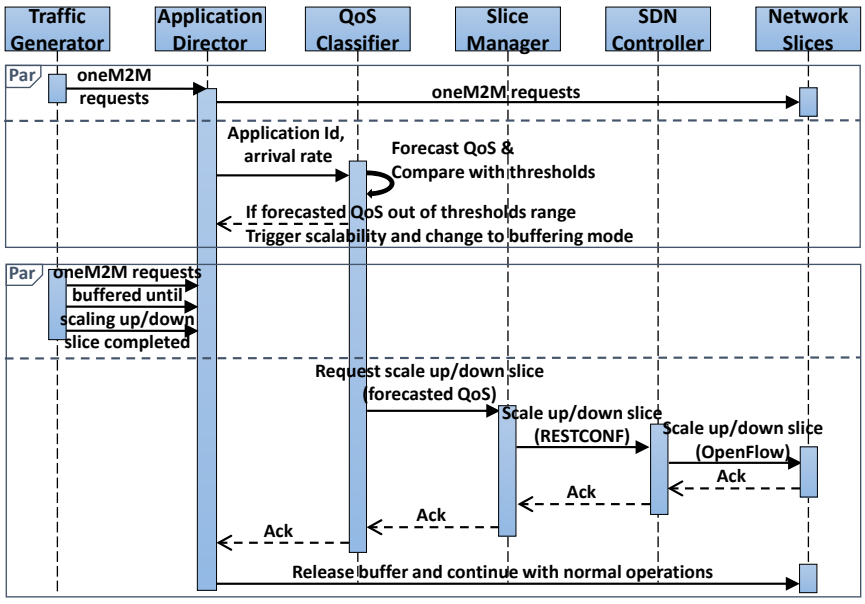

Figure 5. System operations for mapped applications.

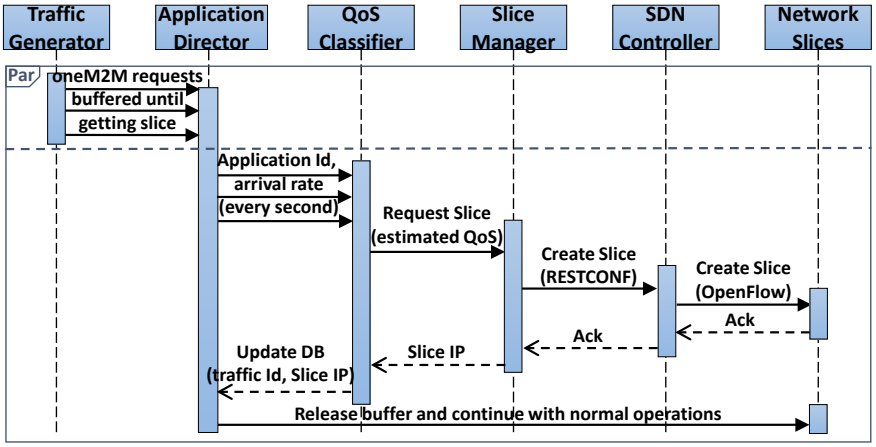

Figure 6. System operations for unmapped applications.

Instead of a static approach restricted to a fixed number of applications and slices, both procedures introduce the notion of dynamic network slicing for supporting system scalability. To realize the high-level procedures described in **Figure 5** and **Figure 6**, we propose a system architecture consisting of Traffic Generator, Application Classifier and Slice Manager as depicted in **Figure 7**. Each of them is explained below in detail.

3.1. Traffic Generator

Traffic Generator is designed to generate eight different types of IoT application CRUD messages as shown in **Table 1** for our experimental testing. In general, Traffic Generator uses the information of **Table 1** for messages generation as follows:

- Each application has a JSON object template created based on real data and whose size in bytes is equal to the value of the "Payload Size" column.
- The column "Throughput" defines the number of bytes transmitted per second by each application. To achieve such a throughput, Traffic Generator will generate a number of requests per second according to the following formula:

$$\text{Requests per second} = \text{Throughput}/\text{Payload\_Size}$$

- Traffic Generator embeds the required reliability into each application URI destination. This value is set based on the value of the "Reliability" column. For example, the reliability requirement of smart meter is 99.99% so its URI would be "/uri/of/smart_meter/99.99".

The JSON template for each application could include numerical, alphanumerical, and/or date-time values. As an example, **Table 2** shows the JSON template utilized for eHealth applications.

The numerical values in **Table 2** (*i.e.* patient_id, meter_id, and blood_sugar_measurement) are generated based on a uniform distribution random generator [46]. On the other hand, alphanumerical values (*i.e.* observation in **Table 2**) are words selected, based on a uniform distribution, from an English dictionary of more than 400,000 words [47]. Date, time values (*i.e.* date and time in **Table 2**)

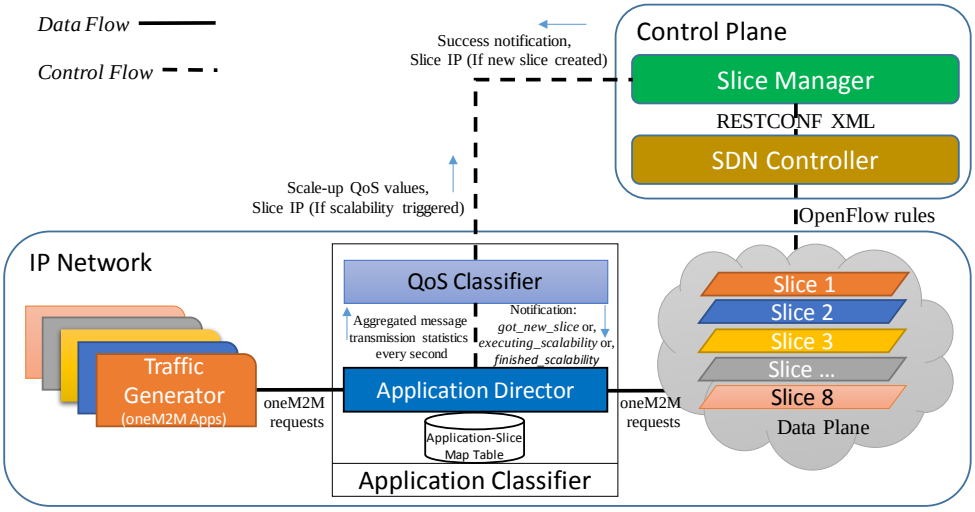

Figure 7. Proposed system architecture.

Table 2. eHealth JSON template.

```
eHealthJSON = {
'patient_id' : <randomNumber1>,
'meter_id' : <randomNumber2>,
'blood_sugar-measurement' : <randomNumber3>,
'observation' : <randomWord1>
'date' : time.strftime("%d/%m/%Y"),
'time' : time.strftime("%H:%M:%S")
}
```

are generated based on current system date_time. These procedures are similar to those utilized by popular traffic generator tools like Jmeter [48]. All the messages created by Traffic Generator go to Application Classifier.

In addition, Traffic Generator allows running multiple threads for each application type to simulate the fluctuation of IoT/M2M applications over time.

3.2. Application Classifier

Application Classifier is responsible for identifying IoT/M2M applications, estimating their QoS, and managing scalability. It is composed of Application Director and QoS Classifier that reside on the data plane and the control plane, respectively.

If the incoming messages belong to the mapped applications, Application Director forwards those to their corresponding network slices. Meanwhile, QoS Classifier would continuously watch for any change of QoS requirements of the mapped applications and triggers scalability whenever needed. If the incoming messages belong to the unmapped applications, Application Director will notify QoS Classifier to request Slice Manager to create a new slice after estimating its QoS requirements.

Application Director and QoS Classifier are explained in further detail next.

3.2.1. Application Director

Application Director performs three specific tasks: 1) identifying application types as mapped or unmapped, 2) sending the mapped applications messages to the corresponding serving network slice, and 3) sending applications message transmission statistics to QoS Classifier for QoS estimation and monitoring for both mapped and unmapped applications.

Table 3 shows the detailed steps of Application Director. First, Application Director spawns a thread to send message transmission statistics (*i.e.* application id and throughput) to QoS Classifier every second (Line 1). We use a 1-second time interval to have a uniform data acquisition timing. During this interval, the message transmission statistics of applications that have more than one request per second are aggregated. QoS Classifier would either utilize these statistics to estimate the QoS required for unmapped applications or monitor changes in QoS

Table 3. Application Director algorithm.

1:	Spawn a thread to aggregate and send application id and throughput to QoS Classifier for QoS estimation and QoS monitoring (every second).
2:	**While** Receiving a oneM2M request from Traffic Generator **Do**
3:	Extract URI from the request.
4:	Extract Reliability from URI.
5:	**If** URI does not exist in application-slice mapping table **Then**
6:	Insert URI with extracted Reliability as new application type with slice state ***NOT READY*** in the application-slice mapping table.
7:	**End If**
8:	If not yet executed, spawn a thread to execute the function *Handle_QoS_Classifier_Messages*().
9:	**If** slice state for URI is ***READY*** **Then**
10:	**If** buffer for URI application type is not empty **Then**
11:	Spawn a thread to clear buffer by forwarding all requests in it to the corresponding slice.
12:	**End If**
13:	Spawn a thread to forward oneM2M request to the URI corresponding slice.
14:	**Else**
15:	Store request in a buffer.
16:	**End If**
17:	**End While**
18:	**Function** ***Handle_QoS_Classifier_Messages*()**
19:	**While** received *notification* from QoS Classifier **Do**
20:	**If** *notification* == GOT_NEW_SLICE **Then**
21:	Change the URI corresponding slice state from ***NOT READY*** to ***READY*** in application-slice mapping table.
22:	**Else If** *notification* == EXECUTING_SCALABILITY **Then**
23:	Change the URI corresponding slice state from ***READY*** to ***NOT READY*** in application-slice mapping table.
24:	Suspend sending message transmission statistics of the application under scale-up/down.
25:	**Else If** *notification* == FINISHED_SCALABILITY **Then**
26:	Change the URI corresponding slice state from ***NOT READY*** to ***READY*** in application-slice mapping table.
27:	Resume sending message transmission statistics of the corresponding application.
28:	**End If**
29:	**End While**
30:	**End Function**

of mapped applications and if needed, request Slice Manager to create a new slice of the required QoS or execute scalability to adjust the QoS of an existing slice.

When a oneM2M request arrives at Application Director (Line 2), it first extracts the destination URI (Line 3) and application's reliability from URI (Line 4). Each destination URI in oneM2M uniquely identifies an IoT/M2M application [25]. If this URI doesn't exist in the application-slice mapping table kept by Application Classifier (Line 5), Application Director uses the URI along with its given reliability to create a new entry in the application-slice mapping table and mark its mapped network slice NOT READY (Line 6). This will trigger the system to store the requests in a buffer (Line 15) while the slice is being created.

Next, if not yet executed, Application Director would create a thread to handle the response from QoS Classifier asynchronously (Line 8) by calling Function Handle-QoS-Classifier-Messages() (Lines 18 to 30). This function is executed whenever a notification message from QoS Classifier is received (Line 19). There are three possible notifications from QoS Classifier to Application Director:

1) GOT_NEW_SLICE when a new slice has been successfully created and configured; in this case, Application Director would change the URI's slice state from NOT READY to READY in the application-slice mapping table (Lines 20 and 21).

2) EXECUTING_SCALABILITY when the need to scale a URI's slice is detected by QoS Classifier. Application Director would change the URI's slice state from READY to NOT READY (Lines 22 and 23) and suspend sending message transmission statistics of the application under scale-up/down (Line 24).

3) FINISHED_SCALABILITY when the process of scalability is completed by Slice Manager. QoS Classifier would relay this notice to Application Director to allow the latter to change the URI's slice state from NOT READY to READY (Lines 25 and 26) and resume sending message transmission statistics of the corresponding applications (Line 27).

At this point, all URIs are recorded in the application-slice mapping table, but for those URIs whose slice states are NOT READY (*i.e.* not fully configured yet or under scalability process), their requests would be temporarily stored in a buffer (Line 15). On the other hand, for the applications whose slice states are READY (Line 9), Application Director first verifies whether the previous requests for this application have been buffered. If true, it would create a new thread to clear the buffer by forwarding all its content to the corresponding slice (Lines 10 to 12). Finally, the current request is forwarded to the appropriate network slice (Line 13).

Note that we initiate several threads along with the tasks of Application Director with the aim of enabling concurrent processing needed in our system operations (e.g. sending message transmission statistics every second) while avoiding the system getting blocked on waiting for a response.

3.2.2. QoS Classifier

QoS Classifier has two groups of specific tasks: 1) estimating QoS of new applications then requesting Slice Manager to create a new network slice, and finally

instructing Application Director which slice to forward the new application messages, and 2) monitoring QoS changes of existing applications and invoking Slice Manager for scale-up or scale-down based on adjustment forecasting. In order to achieve these tasks, QoS Classifier utilizes the following three data structures during system operations:

moving_average = [application id, moving average throughput]

sliding_window = [application id, [[QoS]$_1$ [QoS]$_2$ [QoS]$_3$ … [QoS]$_{10}$]] and,

thresholds = [application id, [scale-up thresholds], [scale-down thresholds]]

- Moving_average: contains the average of the latest five received throughput values of each application. Moving average is a sampling technique for removing outliers from data samples. It helps to avoid triggering scalability unnecessarily (*i.e.* scaling up or scaling down repeatedly due to unstable message transmission flow).
- Sliding_window: contains the latest ten results of estimated QoS using the moving average of each application. Each is a tuple of the two estimated QoS metrics, *i.e.*, $[\text{QoS}]_1 \le i \le 10 = [\text{delay}_i, \text{throughput}_i]$ where Throughput (T) is the moving average of throughput and Delay (D) is derived from the ratio between a common TCP window size (*i.e.* 65,535 bytes) and the measured throughput [49] as follows:

$$D = \text{TCP\_Window\_Size/throughput.}$$

- Thresholds: specifies QoS values such as delay in milliseconds and throughput in bytes/second, of scale-up and scale-down thresholds for each application. In this research, they are set at +60% and −60% of the estimated delay and throughput, respectively.

The principles and advantages of the three data structures are illustrated in **Figure 8**. For simplicity, we show only an example of the throughput received during a period of 23 seconds. However, the same principle can be applied to the delay and other time sequences.

Assuming that the throughput values collected in **Figure 8** are for a new application type, a new slice will be created based on the forecasted throughput calculated at the 14th second, which is 79 bytes/second. This is derived from a linear regression model built upon the first sliding window of 10 QoS measurements in the sliding_window table. Each QoS measurement in the sliding window is calculated based on the moving average of the five received throughput values in the moving_average table. Based on such an initial value, the corresponding scale-up and scale-down thresholds, 126 bytes/second and 32 bytes/second, respectively, can also be calculated for the system, then stored in the thresholds table.

After creating a slice, the process continues with QoS monitoring. To do so, the moving average is recalculated with every new input. As mentioned before, the moving average helps to filter outliers. The value of throughput shown at the 16th second is an example of an outlier. It is out of range of the scalability thresholds,

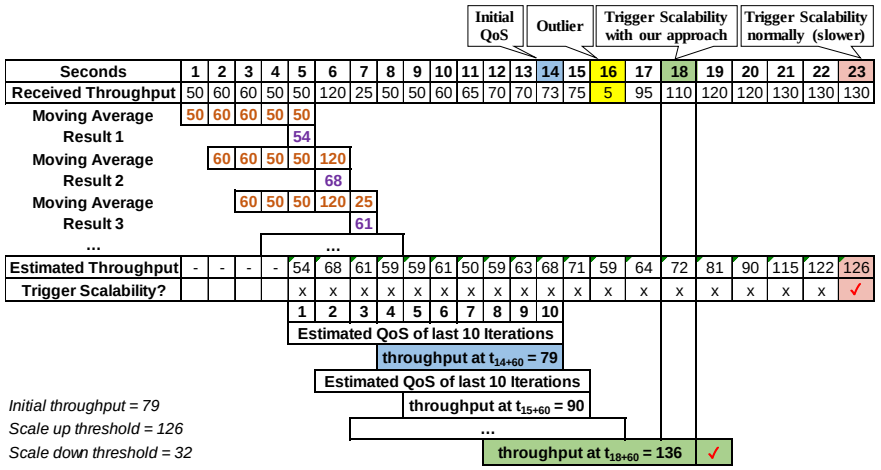

Figure 8. Moving average and sliding window for QoS forecasting and monitoring.

and without the help of the moving average, it could have triggered scalability unnecessarily. The system keeps updating the sliding window and recalculating the linear regression model to forecast the QoS at every second. Finally, the example also demonstrates how the forecasted throughput helps to trigger scalability before the system is overloaded (*i.e.* at the 18th second when the throughput is 72 bytes/second instead of at the 23rd second).

Table 4 shows the detailed tasks related to QoS Classifier. First, it defines global tables to keep the values of moving average, window of ten estimated QoS values, and values of scale-up and scale-down thresholds of each application (*i.e.* one entry per application) (Line 1) that will be used during system operations. Then, QoS Classifier continuously receives message transmission statistics from Application Director, including application id and throughput (Line 2). QoS Classifier uses this input to calculate the moving average of the received message transmission statistics (Line 3). When the next moving average is ready (Line 4), QoS Classifier estimates delay and throughput (Line 5) and stores those values in the next position of the corresponding sliding window of ten tuples (Line 6). When the sliding window is completed (Line 8), QoS Classifier uses a linear regression model to forecast the QoS values of the applications at time $t_n + m$ (Line 9). Where tn represents the current time and $+m$ represents a number of additional seconds. In this research, we assume m = 60. Therefore, it forecasts the status of the QoS metrics in 60 seconds in the future. Next, QoS Classifier queries the application-slice mapping table to check the state of the slice associated with the received application id and decide how to handle the forecasted QoS values such as forecasted delay and throughput.

If the slice state is NOT READY (Line 11), QoS Classifier needs to handle the provisioning of a new network slice in coordination with Slice Manager. Based on the forecasted QoS values, it establishes the thresholds for scalability on each metric (Line 12) and creates a new entry in the thresholds global table (Line 13). Then, QoS Classifier spawns a new thread to run the function Get_New_Slice()

Table 4. QoS Classifier algorithm.

```
1:  Create moving_average, sliding_window, and thresholds global tables in memory.
2:  While receiving application id and throughput from Application Director Do
3:      Compute moving average of the latest five received throughput and
        save them in the moving_average table
4:      If next moving average is ready Then
5:          Estimate Delay and Throughput out of moving average.
6:          Store the estimated QoS values along with application id in the
            sliding_window table
7:      End If
8:      If next window of the ten estimated QoS values is ready Then
9:          Set forecasted_qos = Linear regression of ten estimated QoS in t_n + m.
10:     End If
11:     If slice state of application id is NOT READY Then
12:         Calculate QoS values for scale-up and scale-down thresholds
            based on forecasted QoS
13:         Store thresholds in the thresholds table along with application id.
14:         Spawn a new thread to call the function Get_New_Slice() with
            the estimated scale-up QoS values and application's reliability
            from the application-slice mapping table as parameters.
15:     Else If slice state of application id is READY Then
16:         If forecasted_qos > 0 Then
17:             Retrieve corresponding scale_up and scale_down thresholds
                from the thresholds table
18:             If forecasted_qos ≥ scale_up threshold Or
                  forecasted_qos < scale-down threshold Then
19:                 Notify Application Director by sending
                    EXECUTING_SCALABILITY notification to it.
20:                 Recalculate scalability thresholds based on forecasted QoS
                    and update the thresholds table.
21:                 Spawn a new thread to call the function Trigger-Scalability() with
                    recalculated scale-up QoS values and IP address of slice as parameters.
22:             End If
23:         Else If forecasted_qos ≤ 0 Then
24:             Spawn a new thread to call the function Release_Slice()
                with IP address of slice as parameter.
25:         End If
26:     End If
27: End While
28: Function Get_New_Slice (arguments: QoS values)
29:     Invoke Slice Manager to get a new network slice including QoS values as parameters.
30:     After "success" notification from Slice Manager, notify
        Application Director by sending GOT_NEW_SLICE notification to it.
31: End Function
32: Function Trigger_Scalability (arguments: QoS values, IP address)
33:     Invoke Slice Manager to execute scalability with QoS values and IP address
        as parameters.
34:     After "success" notification from Slice Manager, notify
        Application Director by sending FINISHED_SCALABILITY notification to it.
35: End Function
36: Function Release_Slice (arguments: IP address)
37:     Invoke Slice Manager to delete a slice with its IP address as parameter.
38:     After "success" notification from Slice Manager, delete the
        corresponding entries from the global tables of application-slice
        mapping, moving_average, sliding_window, and thresholds.
39: End Function
```

(Lines 24 to 27) to request Slice Manager to create new slices by sending the values of delay and throughput corresponding to the scale-up threshold, in addition to the application's reliability stored in the application-slice mapping table (Line 14).

Note that QoS Classifier utilizes both scale-up and scale-down thresholds to decide when to trigger the execution of scalability, while Slice Manager only utilizes the scale-up threshold to configure the initial capacity of the slice. By doing so, it allows the incoming application messages to grow until the maximum capacity allowed by the scale-up threshold. After getting the information of the new slices from Slice Manager (*i.e.* slice IP address), QoS Classifier notifies Application Director about the readiness of the new slices by sending the GOT_NEW_SLICE notification to it (Line 26). This enables Application Director to set the state of the slices to READY in the application-slice mapping table in order to release the buffer and forward the subsequent incoming application messages of the same type to the newly created slice.

If the slice state is READY (Line 15), QoS Classifier will start or continue to watch the QoS changes and if required, either trigger scalability or release the slice. The forecasted QoS values can be used to distinguish between these two situations (*i.e.* triggering or releasing).

If the forecasted QoS values are positive (Line 16) (*i.e.* the slope of the linear regression is positive), QoS Classifier retrieves the scalability thresholds of the corresponding application stored in the thresholds table (Line 17) and evaluates the forecasted QoS values versus the scale-up and scale-down thresholds. If the forecasted QoS values are not within the thresholds (Line 18), the system would trigger the execution of scalability.

QoS Classifier first notifies Application Director about the imminent execution of scalability by sending an EXECUTING_SCALABILITY notification (Line 19). In this way, Application Director can change the state of the corresponding slices to NOT READY, suspend sending message transmission statistics of the corresponding application and buffer subsequent incoming oneM2M requests. Then, QoS Classifier calculates new QoS values for scale-up and scale-down thresholds based on the forecasted QoS values and updates the corresponding entry in the thresholds table (Line 20). Next, it spawns a new thread to call the function Trigger_Scalability() (Lines 32 to 35) that would invoke Slice Manager for executing scalability by providing the new scale-up QoS values and the IP address of the slice as parameters (Line 21). Finally, after receiving the acknowledgment from Slice Manager, QoS Classifier notifies Application Director by sending a FINISHED_SCALABILITY notification (Line 34). By doing so, Application Director can set the state of the corresponding slices to READY and resume normal operations.

On the other hand, if the forecasted QoS values are less than or equal to zero for at least $\dot{m}$ seconds (Line 23) (*i.e.* the slope of the linear regression is negative), it is the evidence that the application messages flow has stopped its operations, letting its corresponding network slice in an idle state, therefore that slice has to be released to save energy and computational resources. Here $\dot{m}$ is the

minimum amount of time needed by an inactive application to be safely determined as stopped. In this research, we assume $\dot{m} = 30$. Then, QoS Classifier spawns a new thread to call the function Release_Slice() (Lines 36 to 39) that would invoke Slice Manager for deleting the corresponding slice by providing its IP address as parameter (Line 24). Finally, after receiving the acknowledgment from Slice Manager, QoS Classifier removes all corresponding entries of such application and slice from all global variables in Application Classifier (Line 38).

Note that similar to Application Director, QoS Classifier also takes advantage of using multiple threads to execute concurrent tasks.

3.2.3. Slice Manager

Slice Manager resides on the control plane and it is an SDN application running on top of the OpenDaylight SDN controller. Its main purpose is to automatically provision network slices based on the QoS requirements estimated by QoS Classifier and each application's reliability. Each network slice fulfills the QoS requirements of a specific application type and offers the services of an IoT/M2M platform. In our system, each network slice comprises an end-to-end service, having a client sending a particular type of IoT/M2M application messages to a server (*i.e.* oneM2M platform).

In our previous work [8] [23], we created the network slices manually by defining a series of OpenFlow rules on each OVS and issuing multiple device-specific commands to set the actual QoS capabilities of each slice. Such a manual procedure is infeasible when deploying a large network. To tackle this problem, we apply SDN technologies to automate the aforementioned tasks. With SDN controllers, we not only can access the current network topology and functions but also extract the details of each network device in the current topology. Furthermore, SDN controllers can translate configuration settings written in JSON or XML into complex OpenFlow rules and directly install them in the corresponding forwarding devices. The configuration settings include device/vendor-specific rules to set QoS capabilities required by a network slice.

Table 5 introduces the tasks executed by Slice Manager. These tasks include a set of general instructions and three functions: one for creating a new slice, another to execute scalability, and the last one to delete an idle slice. Regarding the general instructions, when Slice Manager starts, it first connects to the SDN controller (Line 1), executes a full scan of all the forwarding devices (*i.e.* OVS switches) connected in the underlying topology, and creates an internal database of connected links among the forwarding devices (Line 2). Using the collected topology information, Slice Manager calculates all feasible network slices in the system and stores these results in memory (Line 3). This is done by looking at all the possible sequences of paths from the input switch (*i.e.* the switch where Traffic Generator is connected to) to the output switch (*i.e.* the switch where oneM2M servers are connected to) in the current network topology. Finally, Slice Manager loads into memory a provided list of IP addresses that can be assigned to new network slices (Line 4).

Table 5. Slice Manager algorithm.

```
1:   Connect to SDN Controller
2:   Get current network topology from SDN Controller
3:   Set All_Slices = Calculate all possible sequence of paths from input
       switch to output switch in the current network topology.
4:   Set All_IP_Addresses = List of IP addresses for new network slices
5:   While new request from QoS Classifier arrives Do
6:       If request == create slice Then
7:           Spawn a thread to run the function Create_Slice()
8:       Else If request == execute scalability Then
9:           Spawn a thread to run the function Execute_Scalability()
10:      Else If request == delete slice Then
11:          Spawn a thread to run the function Delete_Slice()
12:      End If
13:  End While
14:  Function Create_Slice (arguments: QoS values from QoS Classifier)
15:      Retrieve next unused slice in All_Slices & next unused IP
           address in All_IP_Addresses
16:      Foreach path in the retrieved slice Do
17:          Generate insert RESTCONF command
18:          Send RESTCONF to SDN Controller
19:          Set OpenFlow rules into the corresponding switch
20:      End For
21:      Set QoS using Linux tc qdisc
22:      Instantiate a oneM2M server virtual machine and assign IP
23:      Return the retrieved IP address
24:  End Function
25:  Function Execute_Scalability (arguments: new QoS values from QoS Classifier, slice IP)
26:      Retrieve slice using slice IP
27:      Set QoS using Linux tc qdisc
28:      Return success
29:  End Function
30:  Function Delete_Slice (arguments: slice IP)
31:      Retrieve slice using slice IP
32:      Foreach path in the retrieved slice Do
33:          Generate delete RESTCONF command
34:          Send RESTCONF to SDN Controller
35:          Delete OpenFlow rules from the corresponding switch
36:      End For
37:      Reset QoS using Linux tc qdisc
38:      Set retrieved slice as unused in All_Slices & set IP as unused in All-IP-Addresses
39:      Delete corresponding oneM2M server virtual machine
40:      Return success
41:  End Function
```

With all this information ready, Slice Manager continuously waits for requests coming from QoS Classifier (Line 5). If the request is for the creation of new slices (Line 6), Slice Manager spawns a new thread to run the function Create_Slice() (Line 7). On the other hand, if the request is for the execution of scalability (Line 8), it spawns a new thread to run the function Execute_Scalability() (Line 9). Finally, if the request is for the deletion of an idle slice (Line 10), it spawns a new thread to run the function Delete_Slice() (Line 11). We present the details of these three functions in the following paragraphs. Slice Manager exposes the function Create_Slice() to QoS Classifier for the creation of a new network slice that satisfies a given QoS as a parameter (Line 14). In this function, Slice Man-

ager first retrieves the next unused sequence of paths and the next unused IP address from memory (Line 15) to provision a new network slice. Then, for each path in the selected sequence (Line 16), Slice Manager translates the path into an ODL-based XML representation (Line 17) that is passed to the SDN controller via the RESTCONF protocol [50] (Line 18). The SDN controller then translates the XML declarations into OpenFlow rules to establish the logical links between the connected OVS switches and build the actual network slices (Line 19). Then, Slice Manager utilizes the tc qdisc commands [8] [23] on one of the ports of the connected OVS switches to set the required QoS (Line 21). Next, it utilizes Linux commands to launch a Kernel-based Virtual Machine (KVM) instance with an active IoT/M2M server and assigns the next unused IP address to it (Line 22). Finally, it returns the assigned IP address to QoS Classifier (Line 23).

Slice Manager also exposes the function Execute_Scalability() to QoS Classifier in order to scale-up or scale-down QoS capabilities of slices according to new QoS values (Line 25). Remember that QoS Classifier instructs Application Director to buffer the incoming application messages until the scalability procedure is completed (See **Table 4**, Line 19). In this research, we utilize a QoS update approach to execute scalability. This means setting new QoS configurations in the same existing slice. To do so, upon retrieving the corresponding sequence of paths (*i.e.* slice) based on the given IP address (Line 26), the new QoS configuration requested by QoS Classifier is translated into appropriate tc qdisc commands [51] that are executed on the Ethernet port of the corresponding OVS switch (Line 27). Finally, Slice Manager notifies QoS Classifier about its successful completion (Line 28), allowing the latter to work in coordination with Application Director to let the system continue its normal operations.

The third function offered by Slice Manager is Delete_Slice() which is used to delete idle slices from the system (Line 30). Slice Manager first retrieves the corresponding sequence of paths (*i.e.* slice) based on the given IP address (Line 31). Then, for each path in the sequence (Line 32), Slice Manager creates appropriate delete RESTCONF commands (Line 33), that are sent to the SDN controller (Line 34), converted into corresponding OpenFlow rules and installed in the respective OVS switches (Line 35). Next, Slice Manager resets the QoS settings of the Ethernet port of the corresponding OVS switch using appropriate tc qdisc commands (Line 37). Finally, Slice Manager sets both the slice and its IP address as unused in memory (Line 38), deletes the associated KVM virtual machine (Line 39), and notifies QoS Classifier about its successful completion (Line 40), allowing the latter to finish the slice releasing procedure.

4. Implementation and Evaluation of Our System

In this section, a scalability testbed implemented based on KVM, OVS, and OpenDaylight is presented. Two IoT/M2M system designs: 1) multiple dynamic network slicing, and 2) multiple static network slicing are tested and compared for their scalability features under various loads in terms of the average number of requests, response time, throughput, energy consumption, and memory/CPU cost.

4.1. Scalability Testbed

The scalability testbed is implemented in a desktop computer with a 4-core 3.2GHz CPU, 32 GB of memory and 1TB hard drive. It runs Ubuntu 16.06 and is installed with OpenDaylight Oxygen SR2, KVM 2.4, OVS 2.5.5, and OpenMTC [52] release 4 as the oneM2M platform. Figure 9 shows the logical network topology of the testbed: it is configured with five OVSs interconnected with each other in a mesh using patch links. The internal OVS-based virtual network can support a maximum throughput of up to 100 Mbps.

Traffic Generator, as described in Section 3.1, is used to simulate eight IoT/M2M applications with the characteristics as shown in Table 1:

- Smart meter [36] simulates data collection from appliances in a house.
- Bluetooth tag [37] simulates a Bluetooth Low Energy (BLE) tag attached to objects for location tracking.
- eHealth [38] simulates blood glucose measurements.
- Video [39] simulates video metadata messages from a video server.
- Smart parking [40] simulates a parking lot capable of detecting empty slots for car parking.
- Intrusion detection [41] simulates a video surveillance system equipped with motion detectors.
- Food monitoring [42] simulates the detection and maintenance of required temperature and humidity for food delivery from farm to supermarket.
- Air pollution [43] simulates the measurements of PM2.5 particles in the air, taken on an hourly basis.

With these eight applications, Traffic Generator is capable of generating application behavior characteristics simulating the heterogeneity and bursty nature of IoT/M2M applications. For example, we can categorize smart meter and video as applications that cannot tolerate long delay. However, they have different requirements in throughput: smart meter requires only low throughput,

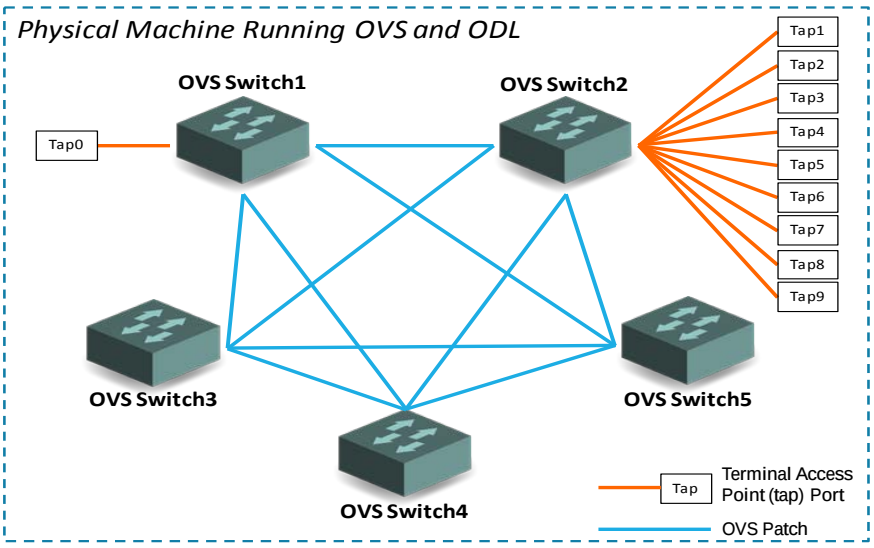

Figure 9. Network logical topology.

while video demands high throughput. On the other hand, both intrusion detection and food monitoring require low delay for data transmission. Nevertheless, intrusion detection demands high reliability, while food monitoring can tolerate low reliability.

4.2. Experiments Setup

The IoT applications considered in this research are based on HTTP RESTful communications, but we are aware that other application protocols like CoAP or MQTT are also popular among IoT applications. Our algorithm might need further adjustment for these kinds of protocols. In addition, some of these application protocols are based on UDP (e.g. CoAP) instead of TCP. Our algorithm assumes having a TCP/IP underlying infrastructure in which delay can be predicted based on TCP window size and throughput; on the other hand, if the applications are running under UDP/IP communications, other considerations might be required.

Every time we run an experiment, eight types of IoT/M2M applications are deployed concurrently and each of them follows the same messages flow load schema as depicted in Figure 10. It consists of continuous streaming of HTTP POST requests meant to insert data in the oneM2M resource tree.

As shown in Figure 10, the total duration of this test schema is 300 seconds (*i.e.* five minutes), distributed as follows:

- From Second 0 to Second 60, the messages flow load corresponds to 50 threads of an application, simulating 50 UEs sending requests simultaneously.
- From Seconds 60 to 120, the load increases in a ratio of 5 threads every 6 seconds, giving a total of 50 additional threads activated at the end of Second 120.
- Then, the load remains stable with 100 threads running from Seconds 120 to 180.
- Next, from Seconds 180 to 240 the load is decreased in a ratio of 5 threads every 6 seconds, with a total of 50 threads removed, and a total of 50 threads still running at the end.
- Finally, those active 50 threads continue sending data from Seconds 240 until 300.

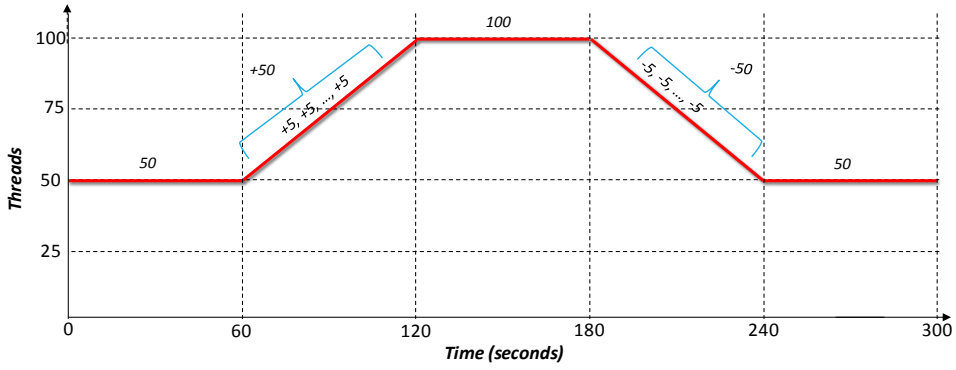

Figure 10. Messages flow load schema.

We plan to evaluate the scalability performance of our dynamic system versus that of a static system. Due to the absence of dynamism in a static system, the latter cannot change its capacity to suit the evolving QoS needs of individual applications. Therefore, we use two types of static systems for comparison: overdesign and underdesign. We call the system overdesign because it is pre-provisioned with more capacity than what is needed. On the other hand, we call the system underdesign as it is pre-provisioned with less capacity than what is actually needed.

Our objective is to prove that the dynamic slicing system has better scalability performance as follows:

1) In terms of efficiency (*i.e.* response time, number of requests, and throughput), it can come close to that of an overdesign system without the price of higher costs (*i.e.* CPU, memory, and power usage) paid by the overdesign.

2) In terms of cost (*i.e.* CPU, memory, and power consumption), it can come close to an underdesign system while still achieving better response time, number of requests, and throughput.

The three system designs for our verification are as follows:

- *The dynamic system*: This is our proposed system that consists of one dynamic slice for each type of IoT/M2M applications. It automatically creates each slice and sets its initial QoS with the given reliability and the scale-up threshold values (*i.e.* delay and throughput) after the first QoS forecasting iteration. The QoS of each slice and its scalability thresholds can be dynamically adjusted according to the changing needs of the incoming requests. Based on our study, such a dynamic system will vary its QoS from 60% higher than the initially detected traffic loading to 120% during the testing period.
- *The underdesign static system*: This system consists of one static slice for each of the eight IoT/M2M applications. We pre-provision each slice with a QoS equals to 60% higher than the initially detected traffic loading (*i.e.* similar to the initial QoS of the dynamic system) in addition to the given reliability. However, all eight slices created are static and cannot perform scale-up or scale-down.
- *The overdesign static system*: This system is similar to the underdesign system, except for having a pre-provisioned QoS on each slice equals to 120% higher than the initially detected traffic loading. Therefore, this design can accommodate all the QoS requirements of applications from beginning to end.

Each design is tested using the same eight types of IoT/M2M applications concurrently and each type was run based on the same general load schema as depicted in **Figure 10**.

4.3. Results of Evaluation

The results of the evaluation are presented in this subsection. **Figure 11** shows the results of the average number of requests handled by the systems during the 5-minute testing period. The proposed dynamic slicing system was able to handle a much higher number of requests than the underdesign slicing system. Taking eHealth applications as an example, the proposed system was able to handle

10,038 requests on average during the execution of the tests, while the underdesign system could only achieve 4947 requests. The same pattern is observed in the other seven applications. On the other hand, the overdesign system overpasses our dynamic system for the number of requests, but the difference is not significant. For example, for eHealth the difference is only at 952 requests.

Figure 12 shows the results of average response time for the three system designs. The proposed system demonstrated a much faster response time for all eight applications compared to the underdesign system and a compatible response time with that of the overdesign system. This is particularly significant for applications (e.g. eHealth and Bluetooth tags) that require quick response time. In particular for Bluetooth tags, the dynamic system obtained an average response time at 746 milliseconds, compatible to the 723 milliseconds of the overdesign system, whereas the underdesign system could only get an average response time at 2610 milliseconds.

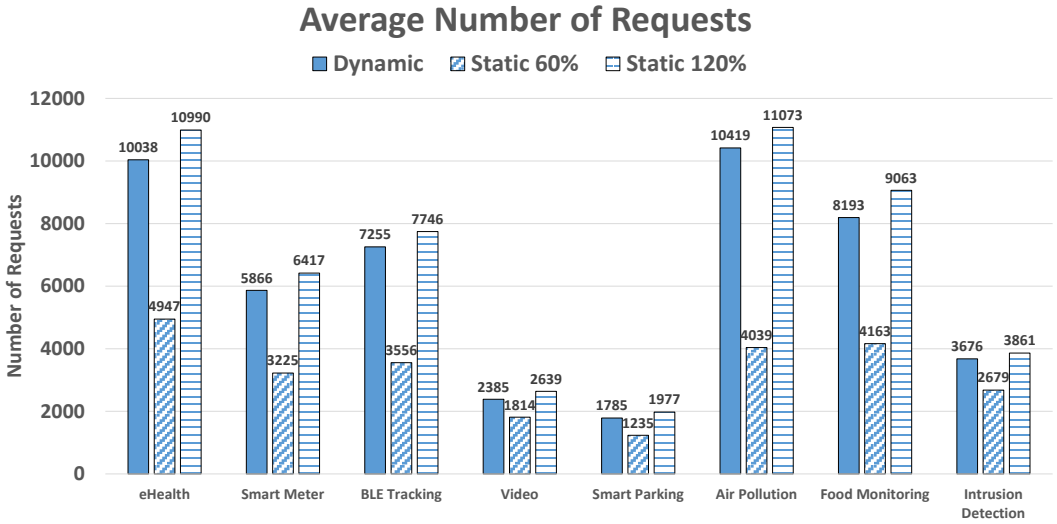

Figure 11. Average number of requests during test duration.

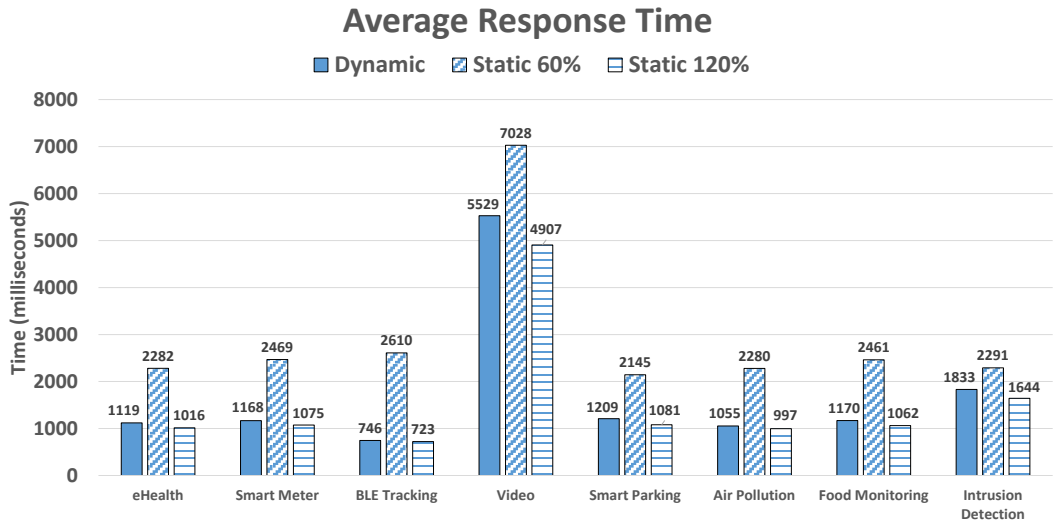

Figure 12. Average response time results.

As shown in Figure 13, our system also achieved a higher throughput for all applications when compared to the underdesign system, while compatible to that of the overdesign system. This is especially important for applications that require to transmit a large amount of data such as video and intrusion detection. In particular for video applications, our system achieved an average throughput at 79,507 bps, compatible to the 87,962 bps of the overdesign system, whereas the underdesign system reached 60,461 bps.

On the other hand, we also verified whether the dynamic slicing system would increase the computational and power cost because it requires more complex operations than the static slicing system and how far this cost is from that of the static system. The CPU and memory usage results are shown in Figure 14 and the amount of power consumed is proportional to the amount of CPU and memory utilized [8] [22] [23] as depicted in Figure 15.

These are measured as the overall system cost for all eight IoT/M2M applications running concurrently during the 5-minute tests on the same physical infrastructure. We verified that the resource utilization of our dynamic system would slightly increase when compared with that of the underdesign system. But it shows significant savings when compared to the overdesign system.

Certain types of applications might not work well with the demand of high CPU, memory usage, and power consumption (e.g. eHealth applications [53]). Hence, it is important to analyze whether any higher CPU and memory usage including power consumption would negatively impact any of eight applications under testing. Note the two thresholds shown in Figure 14: overloaded CPU at 90% and overloaded memory at 40%. They are used to verify whether our design is within the safe range of operations of a reference cloud system. Taken from [54] and [55], these two thresholds are derived as the maximum CPU usage of Google Cloud Spanner instances and the default threshold for Linux memory swapping, respectively. Though our dynamic system utilized up to 79.4% of CPU

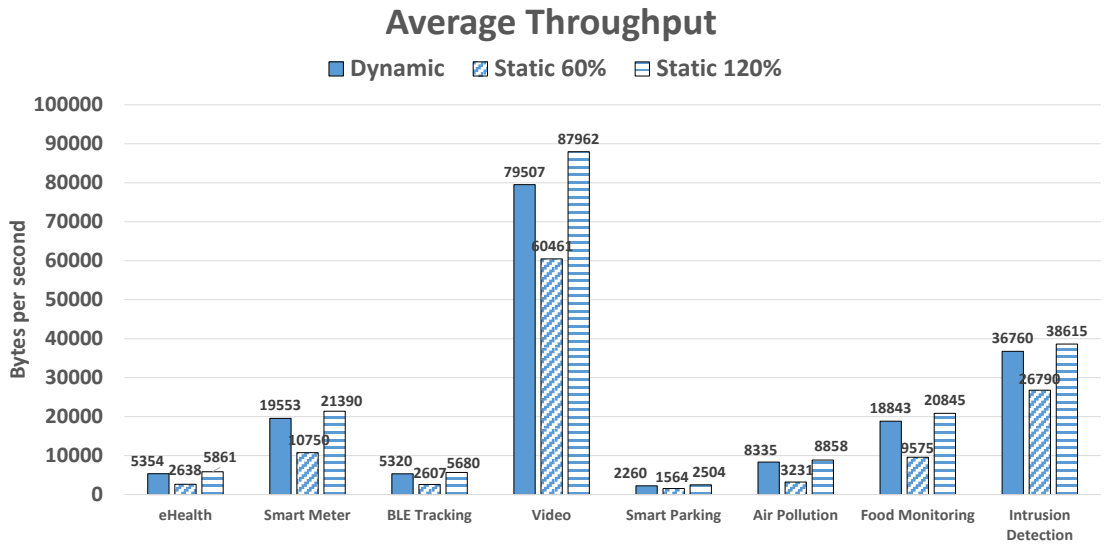

Figure 13. Average throughput results.

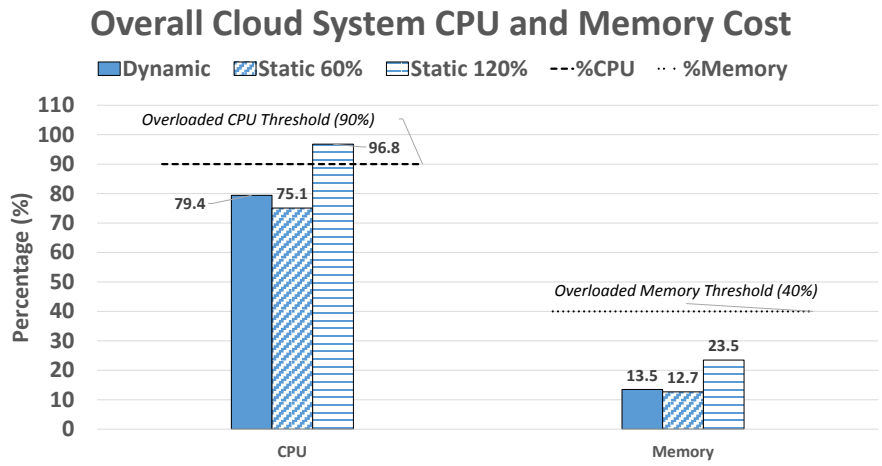

Figure 14. Overall CPU and memory cost.

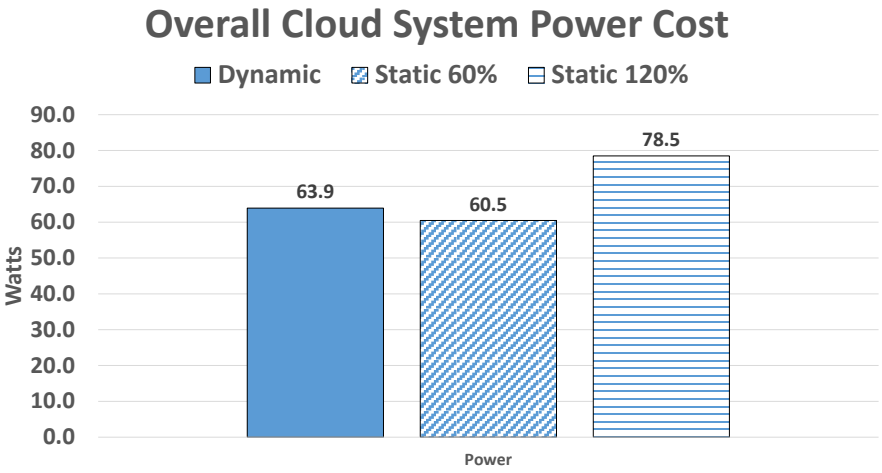

Figure 15. Overall power cost.

and 13.5% of memory, they are still below the overload thresholds of 90% CPU and 40% memory, respectively. Hence, we can conclude that the dynamic system indeed achieves higher scalability within the safe utilization of more computational resources. In other words, the increasing cost in CPU, memory usage, and power consumption of the dynamic slicing system can be seen as a valuable trade-off for improved scalability of the system. On the other hand, Figure 14 shows that the overdesign static system would exceed the CPU Threshold of 90%.

4.4. Discussion

Based on our experimental results, we have proved:

1) In terms of efficiency, *i.e.* average response time, number of requests, and throughput, the dynamic system performs very close to the overdesign system with a difference of only 8% in all three metrics. On the other hand, the efficiency of the underdesign system is much worse than that of the dynamic system as follows: response time lengthened to 103% slower, number of requests and throughput reduced to 82%.

2) In terms of cost, *i.e.* CPU, memory, and power utilization, the dynamic system stands very close to the underdesign system with a difference of 4%, 1%, and 3%, respectively. On the other hand, the overdesign system costs more than a dynamic system as follows: CPU at 18% more, memory at 10% more, and power utilization at 15% more.

As we have observed in the experimental results, every system design pays a trade-off between efficiency and cost. The dynamic system keeps a good balance between these two metrics. Through Application Director, QoS Classifier, and Slice Manager, it is able to incorporate every new application into operations in a good balance.

As mentioned in the beginning of this paper, scalability in the cloud has been well studied, particularly horizontal scalability enabled by virtualization technologies [2] [3] [4] [5]. While cloud scalability offers some promising features, scalability with IoT/M2M platforms in the cloud is more complex due to their heterogeneity and burstiness nature. This motivated us to look for better scalability solutions. After applying horizontal [22] [23] and static vertical [8] scalability approaches in our prior efforts, this research has furthered IoT/M2M scalability using a dynamic vertical approach. The obtained results demonstrate that existing cloud environments can take advantage of SDN-based network slicing and network virtualization technologies to enable a new level of vertical scalability with better results.

5. Conclusions and Future Work

In this work, we have proposed an innovative solution to the problem of scalability for IoT/M2M platforms in the cloud by leveraging SDN-based network slicing. Our solution is a dynamic network slicing system that improves system scalability by considering the heterogeneous and bursty nature of IoT/M2M applications. The proposed system can automatically create network slices on-the-fly for an IoT application with the matched QoS requirements. Furthermore, it would also constantly monitor the QoS demands of the ongoing messages flow and dynamically adjust the QoS characteristics of the serving slice accordingly.

The proposed system architecture in this research includes two main components. First, an Application Classifier for identifying IoT/M2M applications, estimating their QoS requirements, and monitoring changes in QoS to trigger scalability mechanism accordingly. Second, a Slice Manager for spawning and deleting network slices, and executing scalability. Slice Manager provides appropriate network slices by utilizing SDN technologies. It emits RESTCONF commands that are translated into OpenFlow rules by the SDN controller in order to manage the slices accordingly.

By Application Classifier and Slice Manager, our proposed system can dynamically scale-up or down network slices for any number of IoT/M2M applications with constantly changing QoS requirements. Each slice provides platform services based on the unique QoS requirements of the application behavior assigned to it and thus enables the IoT/M2M system to reach a new level of scalability.

Our unique contribution lies not only in the architectural and algorithmic design of a scalable IoT/M2M system but also in a better understanding of how network slicing can really improve scalability and what trade-off to be paid to improve system scalability.

Future exploration can include addressing scalability using both horizontal and vertical approaches (*i.e.* hybrid). Moreover, further study on how the underlying physical network infrastructure may affect the cost and efficiency of a slice could be done as our experiment now is all based on a virtual environment.

Another potential direction of future work consists of leveraging the result of this research and migrate it to the framework of NFV-based network slicing. NFV-based network slicing on top of MANO is expected to be the base of future 5G networks. Moreover, Virtual Network Functions (VNFs) in the NFV-based network slicing framework can be realized either in cloud or fog and in both virtual machines and containers, which have the potential to further improve the scalability of IoT/M2M systems.

Acknowledgements

The project reported in this paper is sponsored by the Ministry of Science and Technology (MOST) of Taiwan Government under Project Number MOST 106-2221-E-009-055-MY3 and the Center for Open Intelligent Connectivity from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Lueth, K.L. (2018) State of the IoT 2018: Number of IoT Devices Now at 7B-Market Accelerating. https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[2] Magalhaes, A., Rech, L., Moraes, R. and Vasques, F. (2018) REPO: A Microservices Elastic Management System for Cost Reduction in the Cloud. 2018 *IEEE Symposium on Computers and Communications* (*ISCC*), Natal, 25-28 June 2018, 328-333. https://doi.org/10.1109/ISCC.2018.8538453

[3] Saadaoui, A. and Scott, S.L. (2018) Lightweight Web Services Migration Framework in Hybrid Clouds. 2018 *IEEE* 4*th International Conference on Collaboration and Internet Computing* (*CIC*), Philadelphia, PA, 18-20 October 2018, 106-113. https://doi.org/10.1109/CIC.2018.00025

[4] Al-Said Ahmad, A. and Andras, P. (2018) Measuring the Scalability of Cloud-Based Software Services. 2018 *IEEE World Congress on Services* (*SERVICES*), San Francisco, CA, 2-7 July 2018, 5-6. https://doi.org/10.1109/SERVICES.2018.00016

[5] Nuño, P., Bulnes, F.G., Granda, J.C. and Suárez, F.J. and García, D.F. (2018) A Scalable WebRTC Platform based on Open Technologies. 2018 *International Con-*

ference on Computer, Information and Telecommunication Systems (*CITS*), Colmar, 11-13 July 2018, 1-5. https://doi.org/10.1109/CITS.2018.8440161

[6] Bizanis, N. and Kuipers, F.A. (2016) SDN and Virtualization Solutions for the Internet of Things: A Survey. *IEEE Access*, **4**, 5591-5606. https://doi.org/10.1109/ACCESS.2016.2607786

[7] 3GPP (2018) Technical Specification Group Services and System Aspects; System Architecture for the 5G System; Stage 2 (Release 15). 3GPP TS 23.501 V15.2.0. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144

[8] De La Bastida, D. and Lin, F.J. (2018) Extending IoT/M2M System Scalability by Network Slicing. *NOMS* 2018-2018 *IEEE/IFIP Network Operations and Management Symposium*, Taipei, 23-27 April 2018, 1-8. https://doi.org/10.1109/NOMS.2018.8406254

[9] ETSI (2017) Network Functions Virtualisation (NFV) Release 3; Evolution and Ecosystem; Report on Network Slicing Support with ETSI NFV Architecture Framework. ETSI GR NFV-EVE 012 V3.1.1.

[10] Afolabi, I., Taleb, T., Samdanis, K., Ksentini, A. and Flinck, H. (2018) Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions. *IEEE Communications Surveys & Tutorials*, **20**, 2429-2453. https://doi.org/10.1109/COMST.2018.2815638

[11] Taleb, T., Mada, B., Corici, M., Nakao, A. and Flinck, H. (2017) PERMIT: Network Slicing for Personalized 5G Mobile Telecommunications. *IEEE Communications Magazine*, **55**, 88-89. https://doi.org/10.1109/MCOM.2017.1600947

[12] Samdanis, K., Costa-Perez, X. and Sciancalepore, V. (2016) From Network Sharing to Multi-Tenancy: The 5G Network Slice Broker. *IEEE Communications Magazine*, **54**, 32-39. https://doi.org/10.1109/MCOM.2016.7514161

[13] Chen, J., Tsai, M., Zhao, L., Chang, W., Lin, Y., Zhou, Q., Lu, Y., Tsai, J. and Cai, Y. Realizing Dynamic Network Slice Resource Management Based on SDN Networks. 2019 *International Conference on Intelligent Computing and Its Emerging Applications* (*ICEA*), Taiwan, 30 August-1 September 2019, 120-125. https://doi.org/10.1109/ICEA.2019.8858288

[14] Meneses, F., Corujo, D., Neto, A. and Aguiar, R.L. (2018) SDN-Based End-to-End Flow Control in Mobile Slice Environments. 2018 *IEEE Conference on Network Function Virtualization and Software Defined Networks* (*NFV-SDN*), Verona, Italy, 27-29 November 2018, 1-5. https://doi.org/10.1109/NFV-SDN.2018.8725764

[15] NGMN Alliance (2015) 5G White Paper.

[16] 3GPP (2018) Technical Specification Group Services and System Aspects; Telecommunication Management; Study on Management and Orchestration of Network Slicing for Next Generation Network (Release 15). 3GPP TR 28.801 V15.1.0.

[17] ETSI (2014) Network Functions Virtualisation (NFV); Management and Orchestration. ETSI GS NFV-MAN 001 V1.1.1.

[18] ONF (2016) TR-526 Applying SDN Architecture to 5G Slicing.

[19] ETSI (2015) Network Functions Virtualisation (NFV); Ecosystem; Report on SDN Usage in NFV Architectural Framework. ETSI GS NFV-EVE 005.

[20] Kapassa, E., Touloupou, M., Stavrianos, P. and Kyriazis, D. (2018) Dynamic 5G Slices for IoT Applications with Diverse Requirements. 2018 5*th International Conference on Internet of Things: Systems, Management and Security*, Valencia, 15-18 October 2018, 195-199. https://doi.org/10.1109/IoTSMS.2018.8554386

[21] Kafle, V.P., Fukushima, Y., Martinez-Julia, P., Miyazawa, T. and Harai, H. (2018) Adaptive Virtual Network Slices for Diverse IoT Services. *IEEE Communications Standards Magazine*, **2**, 33-41. https://doi.org/10.1109/MCOMSTD.2018.1800018

[22] De La Bastida, D. and Lin, F.J. (2017) OpenStack-based Highly Scalable IoT/M2M Platforms. 2017 *IEEE International Conference on Internet of Things* (*iThings*) *and IEEE Green Computing and Communications* (*GreenCom*) *and IEEE Cyber, Physical and Social Computing* (*CPSCom*) *and IEEE Smart Data* (*SmartData*), Exeter, 21-23 June 2017, 711-718. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData.2017.110

[23] Lin, F.J. and De La Bastida, D. (2019) Achieving Scalability in the 5G-Enabled Internet of Things. In: Wu, Y.L., Huang, H.J., Wang, C.-X. and Pan, Y., Eds., *5G-Enabled Internet of Things*, CRC Press, Boca Raton, 418 p. https://doi.org/10.1201/9780429199820-5

[24] Peros, S., Janjua, H., Akkermans, S., Joosen, W. and Hughes, D. (2018) Dynamic QoS Support for IoT Backhaul Networks through SDN. 2018 3*rd International Conference on Fog and Mobile Edge Computing* (*FMEC*), Barcelona, 23-26 April 2018, 187-192. https://doi.org/10.1109/FMEC.2018.8364063

[25] oneM2M (2019) Functional Architecture. oneM2M Technical Specification TS-0001 version 2.22.0 Release 2.

[26] Shahid, M.R., Blanc, G., Zhang, Z. and Debar, H. (2018) IoT Devices Recognition through Network Traffic Analysis. 2018 *IEEE International Conference on Big Data* (*Big Data*), Seattle, WA, 10-13 December 2018, 5187-5192. https://doi.org/10.1109/BigData.2018.8622243

[27] Pinheiro, A.J., Bezerra, J. D.M., Burgardt, C.A.P. and Campelo, D.R. (2019) Identifying IoT Devices and Events Based on Packet Length from Encrypted Traffic. *Computer Communications*, **144**, 8-17. https://doi.org/10.1016/j.comcom.2019.05.012

[28] Aksoy, A. and Gunes, M.H. (2019) Automated IoT Device Identification using Network Traffic. *ICC* 2019-2019 *IEEE International Conference on Communications* (*ICC*), Shanghai, 20-24 May 2019, 1-7. https://doi.org/10.1109/ICC.2019.8761559

[29] White, G., Palade, A., Cabrera, C. and Clarke, S. (2018) IoTPredict: Collaborative QoS Prediction in IoT. 2018 *IEEE International Conference on Pervasive Computing and Communication*s (*PerCom*), Athens, 19-23 March 2018, 1-10. https://doi.org/10.1109/PERCOM.2018.8444598

[30] Kotani, D. (2019)An Architecture of a Network Controller for QoS Management in Home Networks with Lots of IoTDevices and Services. 2019 16*th IEEE Annual Consumer Communications & Networking Conference* (*CCNC*), Las Vegas, 11-14 January 2019, 1-4. https://doi.org/10.1109/CCNC.2019.8651866

[31] Meneses, F., Silva, R., Corujo, D., Neto, A. and Aguiar, R.L. (2019) Dynamic Network Slice Resources Reconfiguration in Heterogeneous Mobility Environments. *Internet Technology Letters*, **2**, e107. https://doi.org/10.1002/itl2.107

[32] Bagci, K.T. and Tekalp, A.M. (2019) SDN-Enabled Distributed Open Exchange: Dynamic QoS-Path Optimization in Multi-Operator Services. *Computer Networks*, **162**, Article ID: 106845. https://doi.org/10.1016/j.comnet.2019.07.001

[33] Sivaramakrishnan, S.R., Mikovic, J., Kannan, P.G., Choon. C. M. and Sklower, K. (2017) Enabling SDN Experimentation in Network Testbeds. *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization* (*SDN-NFVSec*'17), Scottsdale, March 2017, 7-12. https://doi.org/10.1145/3040992.3040996

[34] Sanchez Vilchez, J.M. and Espinel Sarmiento, D. (2018) Fault Tolerance Comparison of ONOS and OpenDaylight SDN Controllers. 2018 4*th IEEE Conference on Network Softwarization and Workshops* (*NetSoft*), Montreal, QC, 25-29 June 2018, 277-282. https://doi.org/10.1109/NETSOFT.2018.8460099

[35] Bakhshi, T. (2017) State of the Art and Recent Research Advances in Software Defined Networking. *Wireless Communications and Mobile Computing*, **2017**, Article ID: 7191647. https://doi.org/10.1155/2017/7191647

[36] Electricity Consumption Benchmarks. Australian Government. https://data.gov.au/dataset/ds-dga-0f3d60db-bd63-419e-9cd9-0a663f3abbc9/details

[37] Guy, N. (2020) The Best Bluetooth Tracker. Wirecutter. http://thewirecutter.com/reviews/best-bluetooth-tracker

[38] Kahn, M. Diabetes Data Set. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets/Diabetes

[39] VideoObject. http://schema.org/VideoObject

[40] Smart Parking Lots. ACT Government. https://www.data.act.gov.au/Transport/Smart-Parking-Lots/ff2q-wdgv

[41] Surveillance Datasets. https://data.world/datasets/surveillance

[42] International Food Composition Table/Database Directory. International Network of Food Data Systems (INFOODS). http://www.fao.org/infoods/infoods/tables-and-databases/en

[43] Taiwan Air Quality Monitoring Stations. Worldwide Air Qualilty. http://aqicn.org

[44] oneM2M (2015) The Interoperability Enabler for the Entire M2M and IoT Ecosystem. oneM2M White Paper. https://access.atis.org/apps/group_public/download.php/20435/OneM2MandIOT.pdf

[45] Aazam, M., St-Hilaire, M., Lung, C.H. and Lambadaris, I. (2016) MeFoRE: QoE Based Resource Estimation at Fog to Enhance QoS in IoT. 2016 23*rd International Conference on Telecommunications* (*ICT*), Thessaloniki, 16-18 May 2016, 1-5. https://doi.org/10.1109/ICT.2016.7500362

[46] Numpy. Random. Uniform. https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.random.uniform.html

[47] English-Words. A Text File Containing over 466k English Words. https://github.com/dwyl/english-words

[48] Random Variable. Jmeter User Manual. http://jmeter.apache.org/usermanual/component_reference.html#Random_Variable

[49] DMello, A., Foo, E. and Reid, J. (2018) Characterizing TCP/IP for High Frequency Communication Systems. 2018 *Military Communications and Information Systems Conference* (*MilCIS*), Canberra, 13-15 November 2018, 1-7.

[50] Jethanandani, M. (2017) YANG, NETCONF, RESTCONF: What Is This All about and How Is It Used for Multi-Layer Networks. 2017 *Optical Fiber Communications Conference and Exhibition* (*OFC*), Los Angeles, CA, 19-23 March 2017, Paper W1D.1. https://doi.org/10.1364/OFC.2017.W1D.1

[51] tc(8)—Linux man page, die.net. https://linux.die.net/man/8/tc

[52] FOKUS, F. The Open MTC Platform. http://www.open-mtc.org

[53] Islam, S.M.R., Kwak, D., Kabir, M.D.H., Hossain, M. and Kwak, K. (2015) The Internet of Things for Health Care: A Comprehensive Survey. *IEEE Access*, **3**, 678-708. https://doi.org/10.1109/ACCESS.2015.2437951

[54] CPU Utilization Metrics. https://cloud.google.com/spanner/docs/cpu-utilization

[55] Documentation for /proc/sys/vm/. Kernel Version 2.6.29. https://www.kernel.org/doc/Documentation/sysctl/vm.txt