An Experimental Study of Kubernetes Cluster Peer-to-Peer Application-Level Federation via Istio Service Mesh

Chanpol Kongsute
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University, Thailand
cchhch123@gmail.com

Chaodit Aswakul
Wireless Network and Future Internet Research Unit
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University, Thailand
chaodit.a@chula.ac.th

***Abstract*— A new paradigm of cloud computing, cloud federation, provides new business opportunities and many benefits to cloud providers, including resource utilization and improvements in reliability. However, the existing implementations of the peer-to-peer cloud federation model require intricate configurations and time-consuming management. In this paper, we propose a pragmatic implementation framework in which Kubernetes clusters are federated in application-level as a peer-to-peer federation through Istio service mesh. This framework can be instantiated by configuring an Istio gateway to act as a cross-cluster load balancer handling traffic between clusters. This approach allows use cases such as cloud bursting and canary deployment. As a verification of this proposed federation framework, we use two Kubernetes clusters installed with echo servers. And by sending requests to the echo server locating in the cluster, we observe the behaviors and limitations of our proposed model. After proving the feasibility of this approach, future works worthy of further investigations are finally given in this paper.**

***Index Terms*—Cloud Computing, Cloud Federation, Istio, Kubernetes, Peer-to-Peer Federation**

I. INTRODUCTION

Cloud federation, where two or more cloud computing service providers (cloud providers) are interconnected and agree to share resources, is the new paradigm of cloud computing. This cooperation between cloud providers provides new business opportunities and many benefits to cloud providers, including resource utilization and improvements in reliability [1].

This paper is a part of IoTcloudServe@TEIN [2], a multisite cloud testbed dedicated to researchers and developers for conducting experiments. This multisite cloud, spreading across 4 countries: Thailand, Laos, Malaysia, and South Korea, interconnects with an underlying OF@TEIN+ network infrastructure [3]. Our main goals aim to obtain a framework design and an implementation model for a multisite IoT-cloud service platform as well as to demonstrate the utility and practicability of our IoT-cloud service platform. In order to achieve our objectives, we have investigated on an approach to attain our multisite environment which will be presented in this paper.

Usually, a cloud federation has two major trust models [4]. The first one is cross-cloud federation. It is a dynamic model where two or more cloud providers come into contact at runtime. The second model is relatively more static and long-term model called horizontal federation. In this model, cloud providers agree to resources exchange and QoS policies beforehand. This model does not have time-consuming phases of dynamically on-demand matching or finding the provider that satisfies the requirements as the previous model, but it lacks scalability. Since IoTcloudServe@TEIN is a collaborative community with a steady set of requirements from closely co-working members, we propose the horizontal federation as our trust model. Aside from trust evaluation, the next crucial aspect of the federation that we have to consider is federation deployment models [5]. For simplification, we simplify all the deployment models to three major models: centralized third-party deployments, hierarchical deployments, and peer-to-peer deployment. In the first deployment model, all the federation management resources are maintained in a centralized, trusted third-party or broker. This model requires a high availability of a third party that all members trust. This may be hard to implement in the diversified group of cloud providers because the unanimous decision to choose the third-party must be made. The second model, hierarchical deployments, has a parent-child relationship where policies propagate from the parent to child. This model can also be hard to implement in collaborative communities because this model implies that there are members with more privileges than others. The last model is peer-to-peer deployment where all the federation management resources are distributed across all members with no hierarchical relationship between members. Although troublesome to maintain, we consider this the most suitable model for international collaborative communities because of its flexibility and fair-shared style of implementation. However, the existing implementations of the peer-to-peer cloud federation model, such as CometCloud federation [6], require intricate configurations and time-consuming management. In this work, we propose a pragmatic implementation framework in which Kubernetes clusters [7] are federated in application-level through Istio service mesh [8], allowing use cases such as cloud bursting, where one of the cloud providers acquire underutilized resources from another cloud provider to accommodate their spikes in demand. In this paper, a small-scale federation with two clusters of cloud has been developed in order to test the proposed idea. For convenience, during this phase of completing the integration of IoTcloudServe@TEIN and OF@TEIN+ clusters, we have tested first our idea between the IoTcloudServe@TEIN cluster and the public Google cloud. After ensuring the proof-of-concept feasibility, the next stage of research work has also been suggested as an ongoing future work towards the federation with the OF@TEIN+ clusters.

II. Kubernetes and Istio

A. *Kubernetes* [7]

Kubernetes is a container orchestration tool for handling the deployment and management of containerized applications. Application instances can be deployed in a Kubernetes cluster as a desired state called 'deployment' which provides a self-healing mechanism. Once the deployment is created, the application can be exposed to the external traffic by creating an abstraction layer called 'service' in which a policy to access the application is defined.

B. Istio [8]

Istio service mesh is a policy-based infrastructure layer that enforces the desired behavior of the network of services. This infrastructure layer provides functionalities such as load balancing, failure recovery, and monitoring. Moreover, a service mesh can be implemented for operations like canary deployment, where a new version of application is slowly released to a subset of users before rolling it out to every users, and blue/green deployment, an application release model in which user traffic is transferred from an old version to a new release that are both running in production.

Istio is made of two components: a control plane which provides network policy and configurations for services, and a data plane that handles both inbound and outbound network traffic. The traffic configuration is applied to Istio by using Istio's traffic management API resources. These resources consist of the following components:

- Virtual service, a set of routing rules, is a configuration of how requests are routed to a service within an Istio service mesh. Unlike plain Kubernetes, Istio address service by the hostname, which will be resolved via DNS, instead of IP address. Therefore, each of the virtual services is applied to the particular service according to the hostname that virtual service is assigned to.

- Destination rules, working along with virtual services, define the traffic policies of calls that are coming to the destination service, allowing the user to customize settings such as load balancing model and outlier detection (i.e., the conditions that are used to mark endpoints of the service as unhealthy and eject them from the connection pool).

- Service entries are used to add services to the Istio's service registry. This can be used to map incoming requests of one hostname to another hostname or IP address.

- Gateway handles inbound and outbound traffic to the service mesh, by allowing specific types of traffic to pass through. For instance, users can configure their gateways to allow only HTTPS request to enter the mesh.

Istio also supports concepts like locality load balancing which prioritizes traffic to the service closest to the locality of the service sending the request when that service becomes unhealthy (when service continually return gateway errors or connection timeouts).

III. Proposed Cloud Federation

In this section, the proposed cloud federation suitable for IoTcloudServe@TEIN and OF@TEIN+ project's clouds or resource clusters is detailed. Our design is based on the peer-to-peer federation model, where all management resources are distributed across all clusters, so there is no hierarchical relationship between clusters. In other words, each cluster will install its own Istio control plane. The only shared component is a common root certificate authority, which is needed for authentication in cross-cluster communication. Fig. 1 depicts the diagram exhibiting the proposed model.

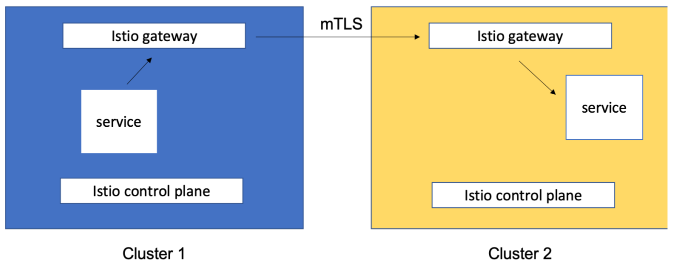

Fig. 1. Cross-cluster communication

In this model, cross-cluster communication is sent to the local Istio gateway and routed to the Istio gateway of another cluster by mTLS protocol. And this routing process can be configured so that an Istio gateway acts like a load balancer that controls traffic between clusters.

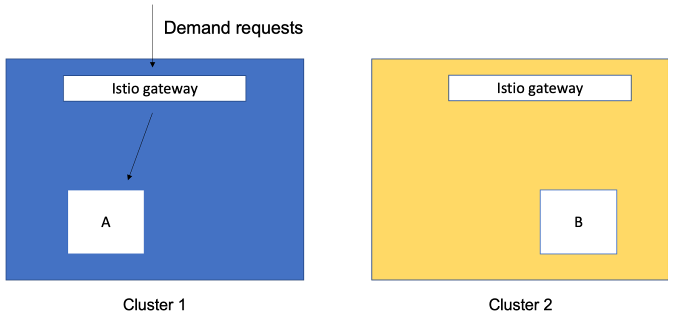

Fig. 2. Ingress request to service

For instance, in a use case where there is a request calling a service in cluster 1 (hereafter "service A") as shown in Fig. 2. If there is a burst in demand of service A and resource in the local cluster (cluster 1) is insufficient, users can replicate the service A and deploy it in a remote cluster (cluster 2) as a service B, then redirect the incoming requests to service B. Although this redirect process can be done manually by configuring the virtual service of service A, we propose a method where this redirect process occurs automatically when the number of requests reaches a certain threshold. This method utilizes the locality load balancing functionality of Istio. In particular, when the service A is marked as unhealthy, i.e., incoming requests reaches a certain threshold, the service will be ejected from connection pool and the Istio gateway of cluster 1 will automatically redirect requests to service B. This can be implemented by creating service entry of service A in cluster 1 pointing to the gateway of cluster 2. YAML file of this service entry is shown in Fig. 3. This service entry adds another endpoint of service A where incoming requests can reach. The 'hosts' field is for matching the hosts addressed in VirtualServices and DestinationRules. The incoming traffic will be forwarded to the endpoint on a specified port assign in the 'ports' field. 'MESH_INTERNAL' signifies Istio to use mTLS to communicate with the endpoint. The 'address' field is the address of the endpoint. The '15443' port is Istio default port for cross-cluster communication.

After setting up the service entry, the destination rule of service A has to be created in cluster 1, to activate locality load balancing and limit the number of requests that service A can accept. Fig. 4 shows an example of YAML file of the destination rule. Here, one can specify different combination of threshold values for triggering parameters. The 'host' field declares the name of a service associated with the destination rule. Traffic criteria for triggering the redirecting process and marking the service as unhealthy are assigned in the 'connectionPool' and 'consecutiveErrors' field. Some of the criteria are:

1) 'maxConnections': Maximum number of HTTP/TCP connections to the host.

2) 'http2MaxRequests': Maximum number of HTTP requests to the host.

3) 'maxRequestsPerConnection' : Maximum number of requests per connection to the host.

4) 'http1MaxPendingRequests': Maximum number of pending HTTP requests to the host.

If the incoming requests exceed one of these threshold, the requests will be rejected with gateway error responses ('502 Bad Gateway') for HTTP protocol. 'consecutiveErrors' set the number of gateway errors before the host is marked as unhealthy which will be inaccessible for the duration set in 'baseEjectionTime'. 'interval' set the time interval between each ejection analysis. In practice, the proper thresholds can be configured so that the redirecting process is triggered only during the peak load. It should also be noted that each cluster has the full authority to decide on its own choice of threshold values. In other words, cluster 2 can create its own destination rule to limit the amount of overflown demand requests from cluster 1. This implementation framework is necessary for the peer-to-peer mode of federation.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: a
spec:
  hosts:
  - a.istio.svc.cluster.local
  ports:
  - name: http
    number: 80
    protocol: http
  resolution: STATIC
  location: MESH_INTERNAL
  endpoints:
  - address: #IP address of Istio gateway in cluster 2
    locality: us-west1/us-west1-b
    ports:
      http: 15443
```

Fig. 3. YAML file of service entry.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: a
spec:
  host: a.istio.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http2MaxRequests: 1
        maxRequestsPerConnection: 1
        http1MaxPendingRequests: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 1m
      maxEjectionPercent: 100
    tls:
      mode: "ISTIO_MUTUAL"
```

Fig. 4. YAML file of destination rule.

The result of these configurations is the Istio gateway that behaves like a cross-cluster load balancer, by which the overflown traffic is redirected to another cloud. The overflown conditions can be controlled by parameters in the destination rule in the YAML file as in Fig. 4. The overall view of the redirecting process is depicted in Fig. 5 and the redirecting process is illustrated in Fig. 6.

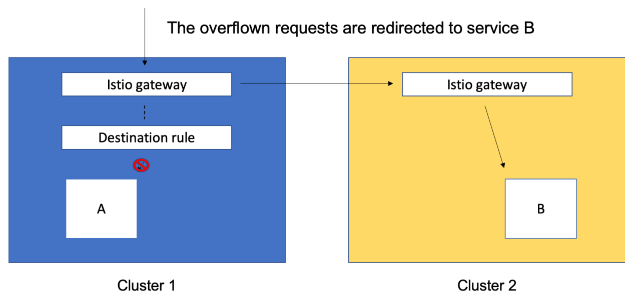

Fig. 5. Cloud bursting

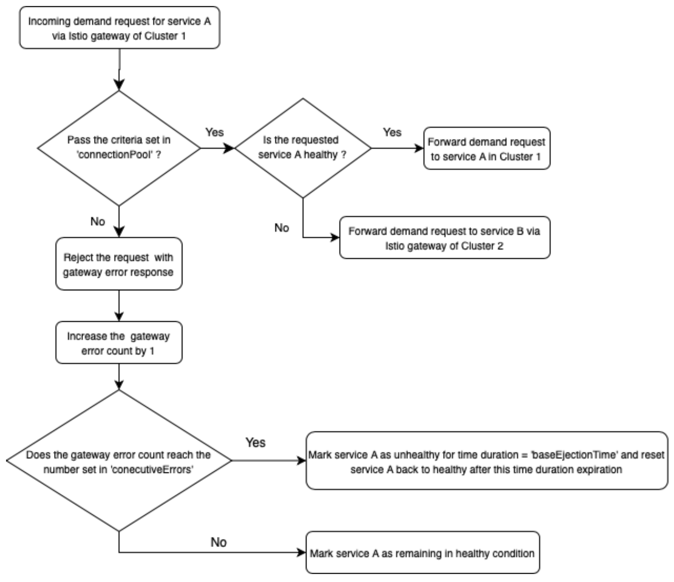
Incoming demand request for service A via Istio gateway of Cluster 1
Pass the criteria set in 'connectionPool' ?
Yes
Is the requested service A healthy ?
Yes
Forward demand request to service A in Cluster 1
No
Forward demand request to service B via Istio gateway of Cluster 2
No
Reject the request with gateway error response
Increase the gateway error count by 1
Does the gateway error count reach the number set in 'conecutiveErrors'
Yes
Mark service A as unhealthy for time duration = 'baseEjectionTime' and reset service A back to healthy after this time duration expiration
No
Mark service A as remaining in healthy condition

Fig. 6. Flow chart of redirecting process

Since the Istio service meshes must be installed separately, all the Istio components such as destination rule can be configured separately for each cluster. Therefore, there are no centralized control plane or hierarchical relationship where one cluster has the ability to control others. For this reason, the proposed model is suitable for the desired peer-to-peer federation.

IV. EXPERIMENTAL RESULT

As a verification of this proposed federation framework, we use two Kubernetes clusters. One of them has been created in Google Cloud Platform (GCP) and another in IoTcloudServe@TEIN. And we have installed Istio service mesh on both clusters. After installation, we have configured the service meshes as described in Section III, where cluster 1 and cluster 2 are IoTcloudServe@TEIN and GKE respectively and service A and B are echo servers (with the docker image: k8s.gcr.io/echoserver:1.10) that return responses when receiving requests. These responses contain information such as the locations of the server (i.e., the name of the pod), headers and bodies of the requests. So, we can extract from the obtained echo responses from which cluster the corresponding requests have been served The result of these configurations is illustrated in Fig. 7.

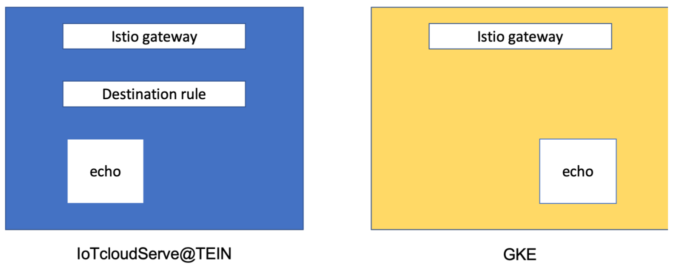

Fig. 7. Example setup of the proposed model

To setup clusters as we described, we start by installing Istio on both clusters. Then we have created the echo services and virtual services on both cluster. After that, in cluster 1, we have created a service entry that map the hostname of the echo service to the Istio gateway of cluster 2. Lastly, we have created destination rule for the echo service. These steps are illustrated in Fig. 8.

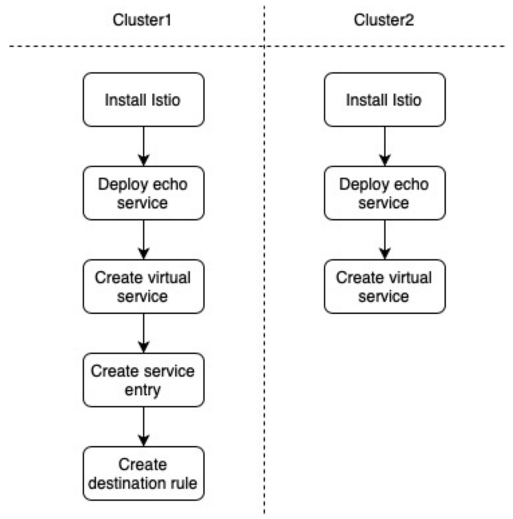

Fig. 8. A flow chart summary of installation process to establish the proposed peer-to-peer federation before starting experiment. The guideline to set up cluster 1 is on the left side while the guideline for cluster 2 is on the right.

We have investigated on the operation of the proposed practice by sending requests to the echo server in IoTcloudServe@TEIN. The expected behaviors are:

1) The echo server in IoTcloudServe@TEIN will accept every request if the number of simultaneous connections does not exceed the maximum threshold that has been set in the destination rule. In this configuration, this threshold has set to 1 connection as an example.
2) When the number of simultaneous requests exceeds the maximum threshold, the subsequent requests will be redirected to the echo server in GKE after the echo server in IoTcloudServe@TEIN is marked as unhealthy and becomes inaccessible for the duration that has been set in the destination rule (1 minute in this example configuration).
3) After having been marked as unhealthy for 1 minute and the demand does not exceed the maximum threshold, the echo server in IoTcloudServe@TEIN

will return to the normal state and accept requests as usual.

These behaviors can be observed by reading the responses of the requests. Because echo servers will return the names of the pods in which they are located, we can confirm the destination where these requests have been sent. For instance, if the responses return the name of the pod in GKE, we can know that these requests have been redirected to the echo server in GKE. We have developed a Python script for response logging as shown in Fig. 9.

```
import requests
from concurrent.futures import ThreadPoolExecutor
cluster1 = []
cluster2 = []
n = input('# of parallel connections: ')
url = ["http://IP address of Istio gateway in IoTcloudServe@TEIN "] * 100
def a(url):
    response = requests.get(url)
    if response.status_code == 200:
        get_body = response.text
        a1 = get_body.find('Hostname:')
        if get_body[a1+10:a1+31] == 'echo-74957c7d55-zljkm':
            if len(cluster1) == 0:
                cluster1.append(1)
            else:
                cluster1.append(cluster1[-1]+1)
        elif get_body[a1+10:a1+31] == 'echo-5c7dd5494d-gzlcn':
            if len(cluster2) == 0:
                cluster2.append(1)
            else:
                cluster2.append(cluster2[-1]+1)
    elif response.status_code == 503:
        a(url)
with ThreadPoolExecutor(max_workers=int(n)) as pool:
    for _ in pool.map(a, url):
     pass
```

Fig. 9. Developed Python script for response generation and logging

Now we test these configurations by sending HTTP requests to the Istio gateway of IoTcloudServe. Firstly, the request has been sent in one connection (one request at a time). The result is shown in Fig. 10. In every time slot, a request has been sent to the echo server in IoTcloudServe@TEIN, then the responses from the echo server have been collected and the accumulated responses from each cluster have been plotted in the vertical axis. From this figure, the accumulated responses from IoTcloudServe@TEIN increase by 1 response in every time slot while the accumulated response from GKE remains the same at 0 response. This means that the echo server in IoTcloudServe@TEIN has accepted every request and not redirected any request to GKE. Therefore, we can conclude that our clusters behave as expected, i.e., IoTcloudServe@TEIN accepts every request when the number of parallel connections does not exceed the maximum threshold.

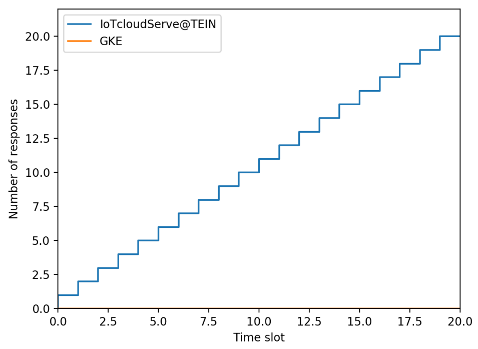

Fig. 10. Accumulated responses from clusters (vertical axis) and time slots (horizontal axis) in a normal state, where the number of parallel connections did not exceed the maximum limit. In every time slot, a request has been sent to the echo server in IoTcloudServe@TEIN (one request at a time).

In the second scenario, we have sent 3 requests at a time (3 parallel connection) to the echo server in IoTcloudServe@TEIN. The result is shown in Fig. 11.

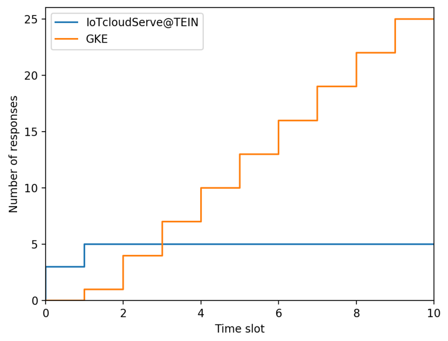

Fig. 11. Accumulated responses from clusters (vertical axis) and time slots (horizontal axis) when the number of parallel connections exceeds the maximum limit. In every time slot, 3 requests have been sent to the echo server in IoTcloudServe@TEIN.

It can be seen that by sending 3 requests in each time slot, the redirecting mechanism is expectedly triggered, and after the first time slot, all the incoming requests have been wholely redirected to the echo server in GKE after the echo server in IoTcloudServe@TEIN cluster has stopped accepting requests. This confirms that the implemented outbursting conditions are functional as initially planned.

In the last scenario, we hope to simulate the real-world use case where IoTcloudServe@TEIN operates in the normal state in the first 4 time slots, then suddenly in the 5th time slot, there has been a spike in demand that exceeds the threshold that has been set (in this case, 3 parallel connections), and IoTcloudServe@TEIN has to redirect all the following requests to GKE. After that, in the 15th time slot, the demand has been back to normal. This scenario is illustrated in Fig. 12.

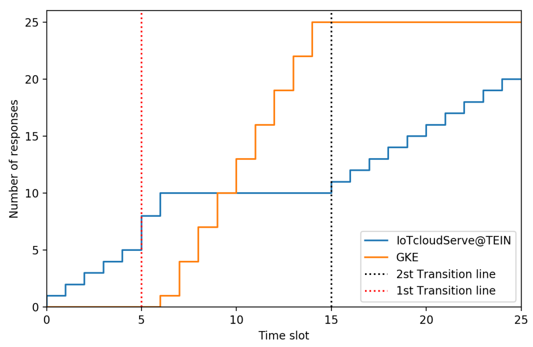

Fig. 12. Accumulated responses from clusters (vertical axis) and time slots (horizontal axis) with 2 transition lines. The first transition line separates the first state (the normal state), where there is 1 request per time slot, and the second state in which there are 3 requests per time slot and IoTcloudServe@TEIN cluster has to redirect the incoming requests to GKE. The second transition line separates the second state and the third state (the normal state) where everything is back to normal.

V. Conclusion

The proposed peer-to-peer federation model has the ability to handle cross-cluster communications as well as traffic policies without tedious setups. However, there are some avenues that we recommend as future works worthy of further investigations. Firstly, since Istio cannot directly control computing resources that services utilize (i.e. developers cannot set the CPU quotas for services by using Istio control plane), all the thresholds that trigger redirecting processes must be abstracted in terms of traffic criteria (e.g. max TCP connections or max HTTP requests). This limitation may prove to be a problem when discussing resource policies between multiple cloud providers. Moreover, this model requires users to create services in both clusters beforehand. Istio can manage only traffic, not Kubernetes resources. Therefore, future works are still needed, in order to create fully automatic cloud federation. Despite these limitations, this model is feasible for multisite cloud providers aiming to develop an interconnected platform whether for new business opportunities or research and development purposes. The capabilities of this model are not limited to only our example scenarios; it can be implemented in an environment with more than 2 cloud clusters or can be configured to distribute traffic in various ways such as round-robin or by percentage, not only for cloud bursting situation.

Acknowledgment

This work was supported by the Asi@Connect's Data-Centric IoT-Cloud Service Platform for Smart Communities (IoTcloudServe@TEIN) project with grant contract ACA 2016/376-562 under the umbrella of Smart-Mobility@Chula demonstration site.

References

[1] C. Lee. Cloud Federation Management and Beyond: Requirements, Relevant Standards, and Gaps. *IEEE Cloud Computing,* vol. 3, pp. 42-49, 01/01 2016.

[2] IoTcloudServe@TEIN. [Online] -- Available from https://www.facebook.com/notes/iotcloudservetein/data-centric-iot-cloud-service-platform-for-smart-communities-iotcloudservetein/331080050821742/ (accessed 10 April 2019).

[3] J. Kim, B. Cha, J. Kim, N. Kim, G. Noh, Y. Jang, H. An, H. Park, J. Hong, D. Jang, T. Ko, W. Song, S. Min, J. Lee, B. Kim, I. Cho, H. Kim, and S. Kang. OF@TEIN: An OpenFlow-Enabled SDN Testbed over International SmartX Rack Sites. *Proceedings of the Asia-Pacific Advanced Network* 2013; 36: 17–22.

[4] V. Massimo, B. Ivona, and T. Francesco. 2012. Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice. IGI Global, Hershey, PA.

[5] C. Lee , R. Bohn, and M. Michel. The NIST Cloud Federation Reference Architecture 5. *NIST Special Publication* 500 (2020): 332.

[6] I. Petri, T. Beach, M. Zou, J. D. Montes, O. Rana and M. Parashar, "Exploring Models and Mechanisms for Exchanging Resources in a Federated Cloud," 2014 IEEE International Conference on Cloud Engineering, Boston, MA, 2014, pp. 215-224.

[7] Kubernetes. [Online] -- Available from https://kubernetes.io (accessed 10 April 2019).

[8] Istio: Connect, Secure, Control, and Observe Services. [Online] -- Available from https://istio.io (accessed 10 April 2020).