IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 6, 2017, accepted May 6, 2017, date of publication May 29, 2017, date of current version June 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2706019

Cloud Resource Management With Turnaround Time Driven Auto-Scaling

XIAOLONG LIU[1], SHYAN-MING YUAN[2], GUO-HENG LUO[2], HAO-YU HUANG[2], AND PAOLO BELLAVISTA[3]

[1]Institute of the Cloud Computing and Big Data for Smart Agriculture, College of Computer andInformation Sciences, Fujian Agriculture and Forestry University, Fuzhou 350002, China
[2]Department of Computer Science, National Chiao Tung University, Hsinchu 300, Taiwan
[3]Department of Computer Science and Engineering, University of Bologna, 40100 Bologna, Italy

Corresponding author: Shyan-Ming Yuan (smyuan@cs.nctu.edu.tw)

The work was supported by the Fund of Cloud Computing and Big Data for Smart Agriculture under Grant 117-612014063 and Grant 177-61201406306 and in part by the MOST of Taiwan under Grant 105-2511-S-009-007-MY3).

ABSTRACT Cloud resource management research and techniques have received relevant attention in the last years. In particular, recently numerous studies have focused on determining the relationship between server-side system information and performance experience for reducing resource wastage. However, the genuine experiences of clients cannot be readily understood only by using the collected server-side information. In this paper, a cloud resource management framework with two novel turnaround time driven auto-scaling mechanisms is proposed for ensuring the stability of service performance. In the first mechanism, turnaround time monitors are deployed in the client-side instead of the more traditional server-side, and the information collected outside the server is used for driving a dynamic auto-scaling operation. In the second mechanism, a schedule-based auto scaling preconfiguration maker is designed to test and identify the amount of resources required in the cloud. The reported experimental results demonstrate that using our original framework for cloud resource management, stable service quality can be ensured and, moreover, a certain amount of quality variation can be handled in order to allow the stability of the service performance to be increased.

INDEX TERMS Network, resource management, big data, turnaround time, service management.

I. INTRODUCTION

The introduction of Software as a service (SaaS) [1] has significantly changed the scenario of IT resource usage. Customers use information services directly through the Internet and no longer have to deploy, manage, and monitor the selected software by themselves. Services are chosen based on considering not only functionality, but also performance, stability, security, and quality. Currently, a service benefits from being provided accompanied by a Service Level Agreement (SLA) [2], which is a contract that addresses factors that customers care about, such as a guaranteed quality level and the specific description of a provided service. In addition, service providers are not required to build their own IT infrastructure in the age of cloud computing [3]; they can allocate the demanded resources rapidly using cloudified and virtualized infrastructures. Application programming interfaces, such as those provided by Amazon Web Service (AWS) or Google Compute Engine (GCE), can be utilized by service providers to create, destroy, and configure, for example, Virtual Machines (VMs), storage, and load balancers [4]. It is widely known and accepted that this approach allows service providers to save substantially on cost when compared with building and maintaining their own computing, storage, and networking resources.

Auto scaling [5] is a key technique used for ensuring that the quality of a service fits the negotiated SLA and for reducing resource wastage. It can be used for automatically increasing or reducing resources according to distinct situations. Auto scaling can be further classified into dynamic auto scaling and schedule-based auto scaling mechanisms [6]. On the one hand, dynamic auto scaling [7], [8], known also as rule-based auto scaling, can be used for scaling out or scaling up service resources based on the rules that managers predefine. The auto-scaling application monitors certain metrics of service measured under genuine workloads and adjusts the system deployment. This is a simple scheme, generally used for assuring a favorable user experience when

workloads are unpredictable, although it is likely to increase latencies until real allocation of additional resources. On the other hand, in schedule-based auto scaling [9], the amount of resources required for specific periods are pre-assigned based on historical workload patterns. In this kind of mechanism, the adverse effects of reaction latency on service performance and stability are avoided. However, the genuine workload patterns of the service must be predictable and regular, and additional methods are required for developing a schedule based on workload history and predictions. These two mechanisms are not mutually exclusive and they can be combined under various scenarios.

Currently, several cloud resource management frameworks are available, such as AWS CloudWatch [10], RightScale [11], and Scalr [12], which supply basic auto-scaling functionality. The mechanisms used in most products involve monitoring system information on the server side in order to trigger system adjustment under certain conditions. The available metrics include CPU Utilization, Disk Read IN/OUT, Network IN/OUT, and other additional custom metrics. Service providers can assign any metric that is currently accessible from the server side on the cloud. The aforementioned core concept is applied in most approaches currently made available by cloud-service providers. However, one drawback is that each manager must determine the indicator metric and the threshold used for the scaling polices of the target service. Conversely, current products also allow schedule-based auto scaling to be applied using a configuration file, but typically with no assistance in developing an appropriate schedule.

In addition, recently numerous studies have focused on determining the relationship between system information and performance experience for the purpose of helping select the metric and the trigger threshold or for estimating the response time of the end user. However, the genuine experiences of clients cannot be readily understood by only using the collected server-side information because of some connected technical challenges. First, the computing resource is provided by virtualization technology, and each physical machine runs numerous VM instances concurrently. Thus, the capacity of each VM instance is uncertain and differences exist in the CPU steal time and the IN/OUT wait time that are decided by neighbors on the same physical machine. Second, the modern system architecture comprises several distinct services and retrieving the details of the capacity of all components might not be possible, which increases the complexity of the estimation method.

This paper introduces a novel cloud resource management framework where the turnaround time of active clients is efficiently monitored directly from outside a service. This framework supports both dynamic and schedule-based auto scaling. In the case of dynamic auto scaling, the framework can deploy one or several monitors on the client-side, which repeatedly send requests for sampling response times over certain durations of service time. A coordinator collects this information in order to decide when the service system must scale out. In the case of schedule-based auto scaling, the framework provides a pre-configuration maker that can conduct a pretest in order to determine the required number of instance nodes over various periods and assist the service manager in developing an effective schedule based on workload history. This framework might be useful for certain services that handle periodic workloads and this mechanism can be used for stabilizing service quality by preparing adequate resources before the arrival of workload peaks. To validate the proposed framework, serial tests were performed for a file-uploading service and the results were compared with those obtained using another mechanism in which the request-arrival rate serves as the target metric. Via extensive experimental evaluation and reported performance indicators, the influences of distinct parameters on the new framework are also discussed as a significant original contribution for the community of researchers in the cloud resource management area.

The remainder of the paper is organized as follows. We briefly describe the previous relevant work in Section II; Section III introduces the designed system and proposed auto scaling mechanisms; the experimental results and discussions are presented in Section IV, while conclusive remarks and directions of future work are given in Section V.

II. RELATED WORKS

The key concerns of cloud-service providers are minimizing costs and satisfying performance requirements. This might involve two sub-problems, job scheduling and resource provision, both not in general terms but in relation to the specific targeted goal of turnaround time-driven auto-scaling. The first problem involves scheduling jobs into suitable VMs in order to optimize performance based on a specific system capacity, whereas the second involves providing adequate resources for satisfying the demanded capacity [13].

For addressing the resource-provision problem, two strategies are available, the schedule-first and scale-first strategies. In numerous approaches, the schedule-first strategy is used [14], [15]. In this strategy, focus is placed on the schedule policy in order to estimate the processing time of submitted jobs and to determine the execution order and the VM worker. In the schedule-first strategy, specific rules based on instance usage and instance type can be applied to develop a schedule in order to minimize costs and satisfy deadlines. Moreover, the system will scale out or scale up if a scheduler cannot determine how certain jobs can be completed before the deadline, given the resources currently available. Therefore, this strategy might be suitable for lengthy jobs or multi-type jobs because they are highly sensitive to the execution sequence and the VM instance characteristics.

The scale-first strategy, which is easier to discuss than schedule-first strategy in relation to the resource-provision problem, focuses on the scaling policy [16]. This strategy can be readily applied using a simple scheduler, which can be merely a load balancer. Most of the current cloud resource-management products support a simple rule-based

auto-scaling functionality. This allows service managers to use certain system-utilization metrics as indicators in order to determine the number of instances. However, selecting the metric and the threshold required for promising a service quality that satisfies the SLA, such as guaranteeing a turnaround time, can be challenging.

Several approaches have been used in order to attempt to identify the mapping relationships between system utilization and performance. In [17]–[19], the measured capacity of VM instance and the request-arrival rate were used for estimating the response time or the cumulative distributions of the response time on a certain number of VM instances. However, these approaches typically cannot be adapted for use in distinct service architectures. Because a system might comprise numerous dissimilar services, obtaining all resource-capacity details might not be possible.

Another approach is to use a direct metric as the indicator when performing the auto scaling [20]–[22]. One SLA-driven system [21] requires only the setting of a request response time between a load balancer and application servers. The load balancer checks the average response time of each server node and the system allocates a new server node when the average response time of any server node is outside a predefined tolerance range. This approach can be used effectively to guarantee stable server-side performance, but the capacity differences of servers or the dispatch policy of the load balancer might lead to excessive scaling out. Thus, this type of scaling cannot be used for guaranteeing the overall performance of a system.

In the approach used in [22], a test was conducted in order to determine the upper bound of requests per second that had an acceptable turnaround response time, and the identified upper bound was used as a base for monitoring the genuine requests per second for the purpose of deciding the amount of resources required for allocating VM instance dynamically over the service time. This is a simple and validated method of auto scaling that allows not only the scale-out timing to be determined, but also enables an estimation of the appropriate amounts of additional resources required. Moreover, this approach can be readily used in systems that feature distinct types of architecture, and the system does not have to be modeled in order to estimate the service quality. This approach can also be effectively applied in schedule-based auto scaling after the workload history is used for preconfiguring the scaling schedule. However, this approach cannot be used for determining the precise amount of resources required. In fact, previous studies have indicated that the performance of the VM instance provided by Infrastructure as a Service (IaaS) varies [23], mainly because IaaS exploits virtualization technologies for providing the resource-supply service. In numerous instances, a single physical machine is shared, and each machine cannot be fully separate from other machines. Distinct numbers of VMs or various jobs running on the physical machine, such as the creation of a new VM instance, might substantially affect the performance of each instance. Thus, the performance of each VM instance is not identical, which means that a limit identified using a specific test cannot fit all VM instances in distinct situations or times. For this motivation, for example, some recent international projects, such as the EU Mobile Cloud Networking [24], have proposed novel methodologies based on experimental characterization of non-functional performance indicators resulting from different traffic patterns (e.g. linearly growing traffic, step-shaped, impulse-shaped) and their linear/non-linear combination in order to enable the realistic prediction of working conditions.

In summary, several problems are associated with the approaches used in studies related to this work; for example, to estimate service quality, detailed capacity information on each component must be obtained; systems cannot effectively cope with services that are deployed on distinct machines; and reaction time demands can cause performance to drop unpredictably. To solve these problems, a system was proposed in our previous work [25], where monitors were deployed outside a service in order to sample turnaround times and analyze the samples for the purpose of driving a rule based auto scaling mechanism. The previous work only focused on presenting a dynamic auto scaling mechanism for ensuring the stability of service performance from the client-side of view. In this paper, we try to propose an integrated cloud resource management framework that can be used across multiple cloud platforms. Different with the work in [25], the proposed framework provides both dynamic and schedule-based auto scaling mechanisms. Such an integrated framework has the relevant advantages of generality and not requiring the knowledge of the capacity details of each component for estimating possible turnaround times. A performance drop caused because of any reason can be detected, and system reaction is based on predefined action in order to provide end users with a stable service quality.

III. OUR FRAMEWORK FOR TURNAROUND TIME DRIVEN AUTO-SCALING

In order to guarantee a stable quality of a service deployed on a cloud platform, we have designed, implemented, and evaluated an original cloud resource management framework with turnaround time driven auto-scaling. Our framework can be regarded as a service deployment and management toolset. The detailed architecture of the proposed framework is described in subsection III-A, while the dynamic auto scaling and schedule-based auto scaling mechanisms are described in subsections III-B and III-C, respectively.

A. ARCHITECTURE

Fig. 1 shows the architecture of the designed framework, which includes three layers, the Cloud Service Provider layer, the Service Management layer, and the Monitor layer. The Cloud Service Provider layer supplies the main resources required for running a service, including computing, storage, and networking resources. This layer currently supports AWS, GCE, and other OpenStack-based [26] IaaS systems.

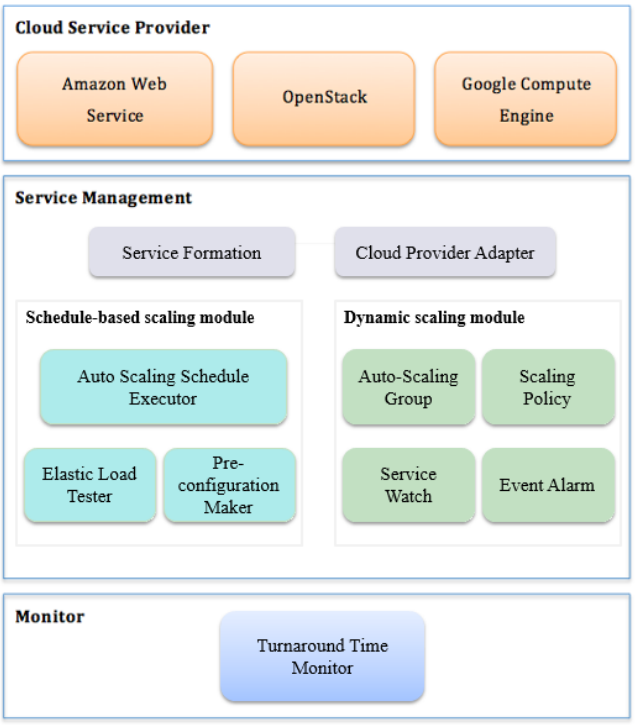

FIGURE 1. Architecture of the designed framework.

The Service Management layer is the main part of the proposed framework: among the other facilities and supports, it originally includes the implementation of our dynamic auto-scaling and schedule-based scaling mechanisms. The flow of operations of this layer mimics the resource-management mechanism of AWS. The Service Formation module can obtain all the resources defined in a configuration file in order to build the architecture of a system automatically. The format of the configuration file is similar to AWS CloudFormation [27], in order to facilitate and leverage rapid application and exploitation in existing deployment environments. The module acquires certain resources such as VMs and load balancers through the Cloud Provider Adapter. Furthermore, the module also obtains custom resources that cloud providers do not support.

As already sketched, the Service Management layer contains both Dynamic scaling module and Schedule-based scaling module. In Dynamic scaling module, the Service Watch is a server that collects all the data from various monitors such as the Turnaround Time Monitors in the Monitor layer, and the Event Alarm can use statistical data obtained from the monitors in order to implement the Scaling Policy and generate the Auto Scaling Group. In Schedule-based scaling module, the Elastic Load Tester is a flexible load-testing tool, and it can build a specific scale-distributed testing framework in order to simulate a large number of clients for the purpose of load testing. The pre-configuration Maker is designed for estimating the amount of service resources demanded in a specific period and for developing a pre-configuration file that is used for managing the resources. Finally, the Auto Scaling Schedule Executor can apply the configurations in order to arrange the resources of the system to be available at a predefined time. The details of the scaling modules are described separately in the following subsections.

The Monitor layer contains the Response Time Monitor, which repeatedly sends requests to the service in order to evaluate the turnaround time on the client end. The monitor can be installed on numerous local computers (i.e. clients) nearby the server to eliminate the effect of network latency. In this way, service manager could observe the genuine user experience of the performance for the purpose of helping maintain stable service quality.

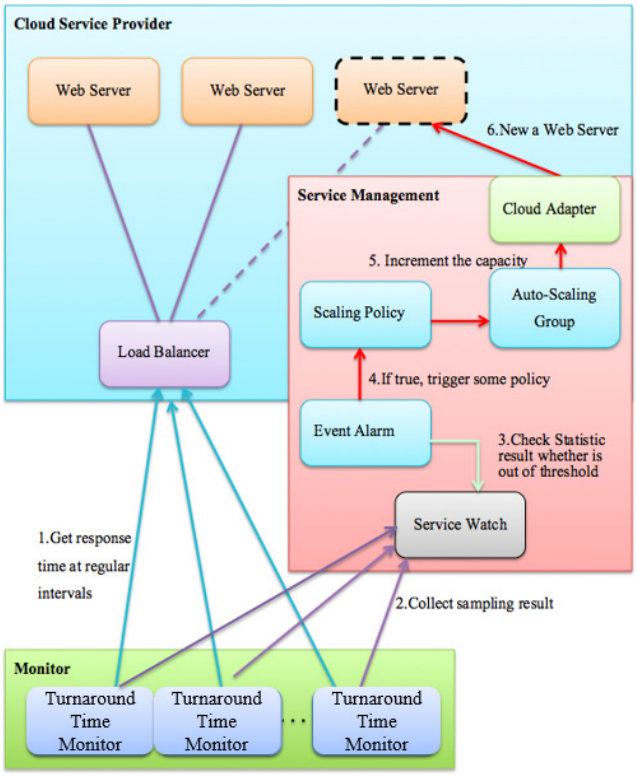

FIGURE 2. Workflow of the proposed dynamic scaling module.

B. DYNAMIC AUTO SCALING

The proposed dynamic auto scaling mechanism is used for dynamically coordinating the resource provisioning of a service. Fig. 2 shows a workflow example of the dynamic scaling module, the details of which are described as following:

1) Turnaround Time Monitors measure the turnaround times at the client side, for example, get the response time by using the GET method in order to load a target webpage at regular predefined intervals.
2) The monitors send the metrics to the Service Watch, which collects and classifies these data.
3) Event Alarm repeatedly checks whether the specific metrics are greater than the threshold or not.
4) If the answer is "true," trigger the relative-scaling policy and execute it. The answer means that the system

might suffer a lack of computing resources, and thus the service must obtain additional resources.

5) All web servers included in the example are organized by Auto Scaling Group, which can use a setting in order to generate numerous identical VMs. The scaling policy increases the capacity of Auto Scaling Group.
6) Auto Scaling Group generates a new web server in the group and allows the service performance to return to the acceptable range.

The workflow is continually repeated while the service is online. Moreover, multiple Event Alarms and Scaling Policies can be defined in order to monitor distinct metrics and adjust various resource deployments. The scenario presented in this example is one of a lack of resources. However, the mechanism can also be applied in situations where resources are in excess in order to eliminate resources and to save costs.

C. SCHEDULE-BASED AUTO SCALING

In the Schedule-based auto scaling module, based on a given workload period, the Pre-configuration Maker can automatically conduct serial testing with the help of Elastic Load Tester, in order to generate a schedule for managing the resources. With the resulted pre-configuration information, the Auto Scaling Schedule Executor can easily apply it to arrange the resources of the system demanded in the specific period. The detailed workflows of Pre-configuration Maker and Elastic Load Tester are described in the following subsections.

1) PRE-CONFIGURATION MAKER

Pre-configuration Maker is designed for estimating the amount of service resources demanded in a specific workload period and for developing a pre-configuration file, which is used for managing the resources. Fig. 3 shows the workflow used for generating the pre-configuration file, which are described as the following steps:

1) Upload the service-resource configuration and the workload configuration. Analyze the workload configuration, and convert it into several 3-min slices and use an increasing order as the test plan. The conversion flow is shown in Fig. 4.
2) Use the Service Builder to construct the target service on the cloud platform according to the service-resource configuration.
3) Choose a slice obtained sequentially in Step 1, and send the command that defines the request-per-second in the current slice to the Elastic Load Tester. The workflow of Elastic Load Tester is described in the next subsection in detail.
4) Repeatedly generate a specific number of requests within a specific duration.
5) Wait for the service to scale out with respect to the turnaround time of all requests and then make it fit the SLA. Every time the executed service scales out, extend the waiting time in the current slice.

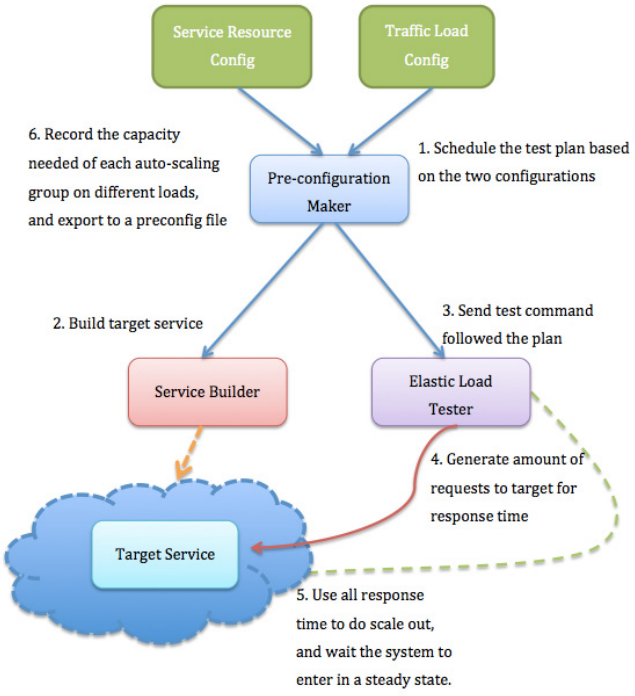

FIGURE 3. Workflow of the Pre-configuration maker.

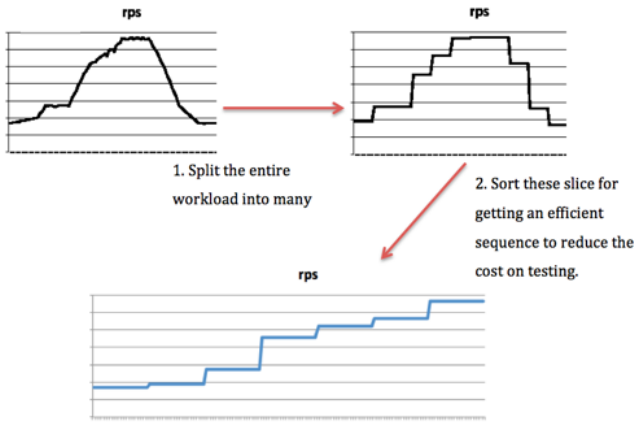

FIGURE 4. Conversion from a workload to a test plan.

6) Record the capacity of each auto-scaling group while the service is entering a steady status, and then return to Step 3 and test the next slice. If all tests are completed, export the service-resource pre-configuration file.

After that, the Pre-configuration file can be used by the Auto Scaling Schedule Executor, for the purpose of deciding the amount of computing resources required and for allowing the quality of service to be of an acceptable level when workloads are predictable.

2) ELASTIC LOAD TESTER

As mentioned above, Elastic Load Tester plays an important role for generating the Pre-configuration file in the workflow of our Pre-configuration maker. It is a flexible load-testing

tool based on Locust [28], which can build a specific scale-distributed testing framework in order to simulate a large number of clients for the purpose of load testing (distributed mode, remote setup of numerous concurrent hosts to emulate a high workload).

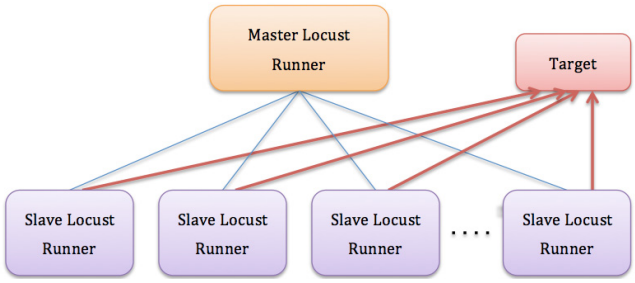

FIGURE 5. Distributed testing mode of Locust.

Fig. 5 shows the basic concept of the mechanisms by which Locust testing works in its distributed mode. Slave Locust Runners are deployed on numerous hosts, and these runners work under a Master Locust Runner. Slave Locust Runners send requests sequentially to the testing target for the purpose of sampling. The Master Locust Runner sends test commands to Slave Locust Runners and collects the testing result obtained from the framework in a specific environment. The Locust framework can be used to effectively simulate a large number of clients and avoid the bottleneck problem that arises on the tester side.

The detailed workflow of the Elastic Load Tester is shown in Fig. 6, which includes two phases. The first phase involves setting up a test in the Locust testing framework. In the second, test commands are continually sent to the testing framework according to the instructions in the predefined workload-definition file. Furthermore, commands from the Workload Generator are all passed through the Test Console, which directly controls the testing framework. The details of the procedure used are the following:

1) Read the Workload Configuration and the Test Environment Configuration. The Workload Configuration declares the number of requests per second that would be present at a given time. The Test Environment Configuration decides the mode of the Locust framework, such as the single-host mode and the distributed mode, and it assigns the number of nodes in the distributed mode.
2) In the distributed mode, the Framework Builder generate a multi-node test environment, shown in Fig. 5. The Framework Builder can also be used to construct the environment manually if VMs are not used as Slave Locust Runners.
3) The Traffic Load Generator splits the load pattern into several slices, and each slice can be used to generate distinct requests per second.
4) The test command is sent according to the schedule prepared in the preceding step.

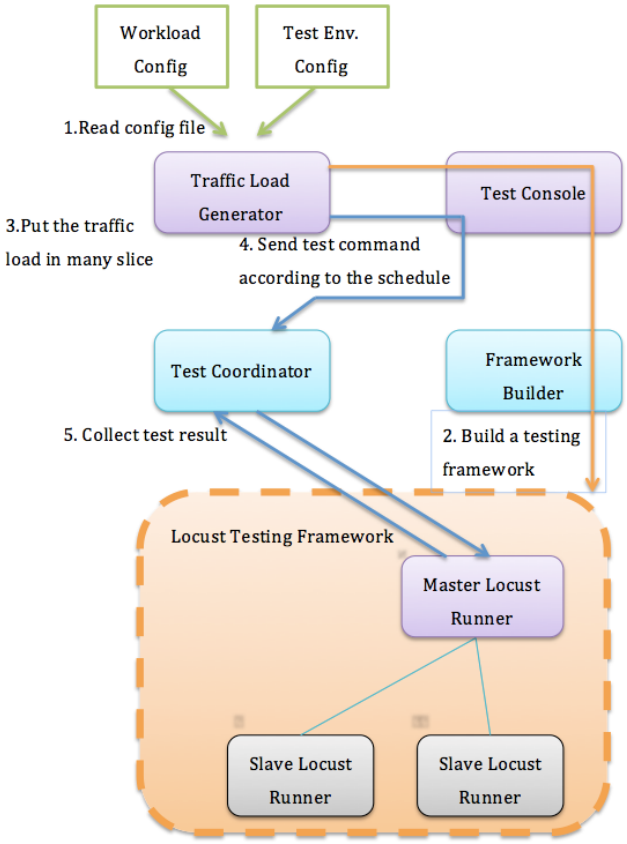

FIGURE 6. Workflow of the elastic load tester.

5) The Tester Coordinator executes the test and collects the results from the Locust testing framework.

IV. SIMULATION AND EXPERIMENTAL RESULTS

This section describes the simulation and evaluation of the proposed framework in the case of both the dynamic and the schedule-based auto-scaling mechanisms and their comparison with the approach proposed by Vasar *et al.* [22]. The testing environment of the experiments is shown in Fig.7. For the sake of simplicity and easy interpretability of the collected results (with no hard-to-understand dependency on application specific peculiarity, the target system that was tested in the experiment was a simple file-uploading web service. Each server node deployed a simple receive server, Droopy [29], on a Google Compute Engine (GCE) n1-standard-1 type instance. The tester nodes were also built on GCE and used the same instance type. The target and the tester were included in Google Asia-East-1 region in Changhua, Taiwan, but on distinct zones (i.e. Asia-East-1a and Asia-East-1b). A zone is an isolated location within a region, and each zone contains the computation, storage, and networking instruments required for assuring the availability of a region. The use of zones can prevent the target and the tester from being deployed on the same physical machine and can provide a suitable distance between the service and the

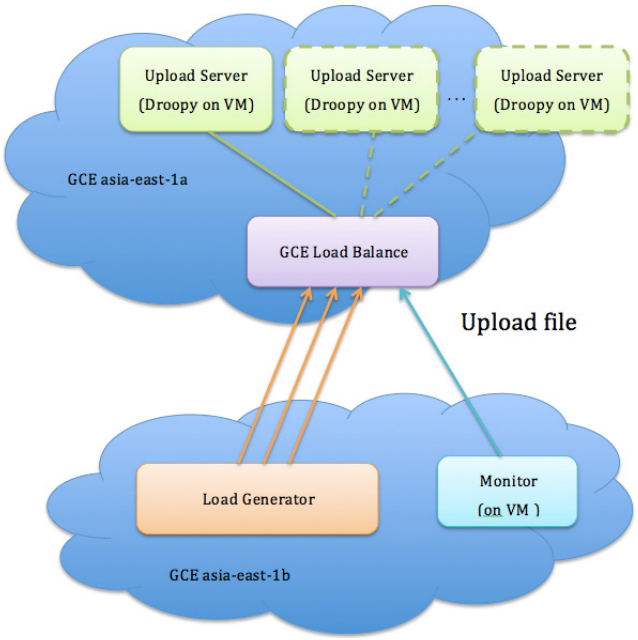

FIGURE 7. Testing environment.

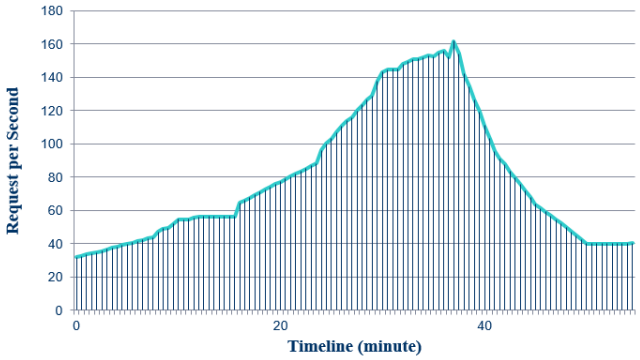

FIGURE 8. Workload configuration.

load generator. Each request uploaded a 100-KB file to the server through the GCE load balancer.

A. DYNAMIC AUTO SCALING RESULTS

To validate the proposed dynamic auto scaling mechanism, three testing with different thresholds (i.e. 500, 700, and 1000 ms) for the scaling policy were performed. The steps used in the experiment are illustrated below:

1) Generate a workload configuration (as shown in Fig. 8) in order to simulate a user workload pattern. In each round of the experiment, the same variable workload was used continually for approximately 1 hour to simulate the workload.
2) Generate a 100-KB file as the uploaded file for use in each request.
3) Prepare a service-deployment configuration file. The initial number of server nodes is one. The scale threshold must be set in this step; the thresholds used individually in the experiments were 500, 700, and 1000 ms.
4) Build a distributed-architecture workload generator featuring four tester nodes. A single monitor was set on a single tester node to measure the turnaround time of each request.
5) Start testing. Three rounds of testing were performed using each threshold setting.

TABLE 1. Results of different threshold on dynamic auto scaling.

Threshold (ms)	500	700	1000
Average of turnaround time (ms)	108.07	116.74	127.45
SD of turnaround time (ms)	76.63	93.05	143.83
CV of turnaround time (ms)	70.91%	79.71%	112.84%
Instance minutes	137	121	112

Table 1 summarizes the results of different threshold during dynamic auto scaling, where the average, standard deviation (SD) and coefficient of variation (CV) for turnaround times of all the requests in the test workload are presented. With respect to the turnaround time performance, the results in Table 1 show that the average turnaround times of the test workload were less than 130ms, regardless of what threshold was used. However, the average turnaround time measured when the threshold was 500ms was roughly 15% lower than that obtained when the threshold was 1000ms. Furthermore, the SD and CV of the turnaround time at a threshold on 500ms were much lower than those measured at threshold on 700ms and 1000ms.

The instance minutes in Table 1 represents how many minutes of instances are used during each testing. The proposed dynamic scaling mechanism can automatically scale the required instance when the threshold was violated. Therefore, a lower threshold may result better turnaround time performance with the cost of more instance minutes. The results in Table 1 indicate that the use of distinct thresholds can lead to dissimilar overall performance. Setting a small threshold on the turnaround time allowed the enhancement of the overall service performance.

TABLE 2. Cumulative percentages of different threshold on dynamic auto scaling.

Threshold (ms) / Turnaround Time (ms)	500	700	1000
100	64.54%	50.01%	48.92%
200	98.88%	97.95%	95.81%
300	99.45%	99.08%	98.23%
400	99.62%	99.33%	98.57%
500	99.73%	99.53%	98.93%

In order to demonstrate the stability of the proposed dynamic scaling mechanism, Table 2 shows the cumulative percentages for turnaround times of different thresholds on dynamic auto scaling. The results in Table 2 present that

almost 99% of the requests' turnaround time at a threshold of 500ms were less than 200ms, and those of threshold at 700ms and 1000ms were 97.95% and 95.81%, respectively. The results indicate that stable service performance can be ensured in the proposed dynamic scaling mechanism. Moreover, these results also demonstrate that the monitoring method used in the proposed framework can be employed for ensuring that turnaround times on client-side of view can serve as a dynamic auto-scaling metric.

B. SCHEDULE-BASED AUTO SCALING RESULTS

This subsection describes the use of the Pre-configuration maker of the proposed framework for generating a pre-configuration file from the workload patterns presented in Fig. 8. To validate the proposed schedule-based mechanism, three testing with different thresholds (i.e. 500, 700, and 1000 ms) of scaling policy were also performed. The following illustrates the steps that were used in the experiment:

1) Generate a workload configuration for simulating user workload patterns. The workload used was the same as that shown in Fig. 8.
2) Generate a 100-KB file as the uploaded file for use in each request.
3) Prepare a service-deployment configuration file. The initial number of server nodes was one. The scale threshold must be set in this step; the thresholds used individually were also 500, 700, and 1000 ms.
4) Build a distributed-architecture workload generator featuring four tester nodes. A single monitor was set on a single tester node to measure the turnaround time of each request.
5) Start pretesting and generate a scaling schedule.
6) Verify the scaling schedule prepared in Step 5 by performing three rounds testing, using the workload patterns whose procedures follow those described in Section IV-A.

TABLE 3. Results of different threshold on schedule-based auto scaling.

Threshold (ms)	500	700	1000
Average of turnaround time (ms)	99.26	105.02	111.18
SD of turnaround time (ms)	43.09	61.07	107.61
CV of turnaround time (ms)	43.41%	58.15%	96.79%
Instance minutes	132	121	104

Table 3 summarizes the results of different thresholds in schedule-based auto scaling. The results of schedule-based auto scaling show the same trend in dynamic auto scaling, where setting a small threshold on the turnaround time allowed the enhancement of the overall service performance. However, compared with the results in Table 1, both the turnaround time performance and instance minutes resulted in schedule-based auto scaling were much better than those in dynamic auto scaling, regardless of what threshold was used.

Table 4 shows the cumulative percentages for turnaround times of different thresholds on schedule-based auto scaling.

TABLE 4. Cumulative percentages of different threshold on schedule-based auto scaling.

Threshold (ms) / Turnaround Time (ms)	500	700	1000
100	74.62%	66.58%	54.79%
200	99.64%	99.36%	99.05%
300	99.84%	99.67%	99.48%
400	99.90%	99.80%	99.68%
500	99.92%	99.87%	99.79%

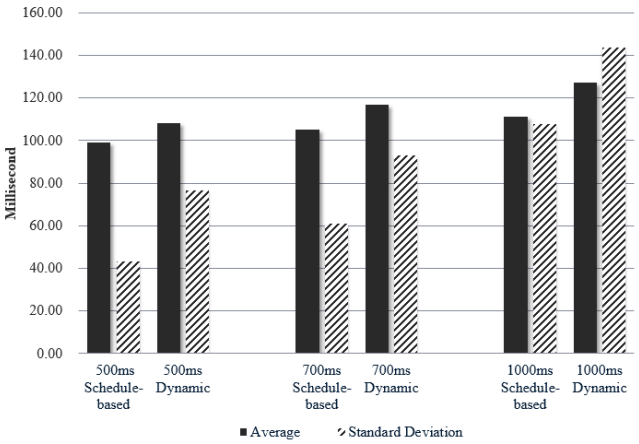

FIGURE 9. Results of schedule-based auto scaling compare to dynamic auto scaling.

The results in Table 4 illustrate that no matter what threshold was set, more than 99% of the requests' turnaround time were less than 200ms. The results show the outstanding stability of the proposed schedule-based scaling mechanism, which is also better than that of the dynamic scaling mechanism. It indicates that the use of schedule-based auto scaling can enhance service quality, and the function used for generating the pre-configuration in the proposed framework was effective.

In order to compare the proposed schedule-based and dynamic auto scaling more clearly, Fig. 9 presents the average and SD of the turnaround time results when selecting different thresholds. It shows that schedule-based auto scaling was superior to dynamic auto scaling in all cases when the same threshold setting was used. In these experiments, the average turnaround time in the case of schedule-based auto scaling was decreased by approximately 10ms and the standard deviation of the turnaround time was decreased by approximately 30ms.

The experimental results indicate that the proposed framework can guarantee the performance in a service within a specific quality range. It also demonstrates that the use of pre-configuration maker in the schedule-based auto scaling mechanism can stabilize the service performance to a greater extent than the use of dynamic auto scaling. Schedule-based auto scaling mechanism is a better choice when workload variation can be predicted, since operational latency would

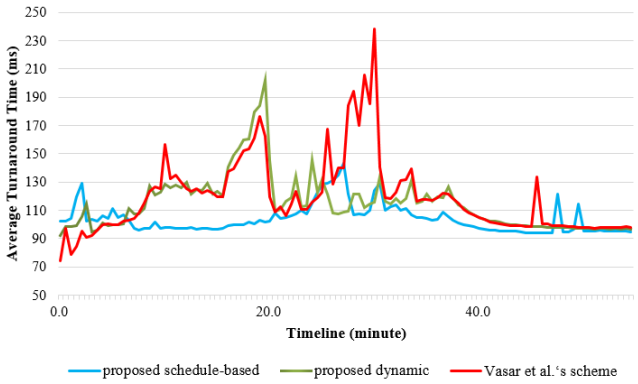

FIGURE 10. Comparison of turnaround time results in different auto scaling schemes.

TABLE 5. Summarized results in different auto scaling schemes.

Schemes	Proposed schedule-based mechanism	Proposed dynamic mechanism	Vasar et al.'s scheme [22]
Average of turnaround time (ms)	105.02	116.74	130.27
SD of turnaround time (ms)	61.07	93.05	141.08
CV of turnaround time (ms)	58.15%	79.71%	108.30%
Instance minutes	121	121	110

be avoided by pre-configuration when resource changes are required.

C. COMPARISONS AND DISCUSSIONS

In this subsection, the results obtained in this study, including the proposed dynamic and schedule-based auto scaling mechanism, are compared with those obtained using the scheme proposed by Vasar *et al.* [22], where arrival-rate was used as the target metric. The comparison is made exactly against Vasar et al.'s scheme is because the arrival-rate (i.e. workload) was also regarded as a critical and significant metric for auto-scaling policy [9]. For performing the comparison, the same workload (as shown in Fig. 8) was generated and tested in each scheme. The arrival-rate threshold in Vasar et al.'s scheme was set as 70 requests per second, and the turnaround-time threshold in the proposed framework was set as 700ms.

Fig. 10 shows the turnaround time variation results in different auto scaling schemes, with serving the same workload (Fig. 8). The results in Figure 10 show that the turnaround times of Vasar et al.'s scheme are very unstable following the timeline. In particular, the variation is very obvious during the timeline from 20 minute to 40 minute. Compared with it, the results of the proposed turnaround time-driven auto scaling mechanisms are more stable, especially for the result of proposed schedule-based auto scaling mechanism.

Table 5 and Table 6 illustrate the comparisons of summarized results and cumulative percentages results of different auto scaling schemes, respectively. The results in Table 5 show that all of the metrics respect to turnaround time

TABLE 6. Cumulative percentages of different auto scaling schemes.

Schemes / Turnaround Time (ms)	Proposed schedule-based mechanism	Proposed dynamic mechanism	Vasar et al.'s scheme [22]
100	66.58%	50.01%	42.58%
200	99.36%	97.95%	95.47%
300	99.67%	99.08%	98.14%
400	99.80%	99.33%	98.54%
500	99.87%	99.53%	98.89%

performance in the two proposed mechanisms are much better than that of Vasar et al.'s scheme, with the cost of a few additional instance minutes. In terms of cumulative percentages results in Table 6, more than 95% of the requests' turnaround time are less than 200ms in all of the auto scaling schemes. However, only 42.58% of the requests' turnaround time in Vasar et al.'s scheme are less than 100ms, which is inferior while comparing with the results in the proposed dynamic or schedule-based mechanism.

Overall, the comparison results demonstrate that the turnaround time driven auto scaling mechanisms in the proposed framework can guarantee a better performance than Vasar et al.'s scheme. The proposed schedule-based auto scaling mechanism is the best choice when workload variation can be predicted. The pretesting technique of our pre-configuration maker in schedule-based auto scaling mechanism can be used to identify relationships between distinct server numbers and turnaround times, for the purpose of obtaining the information required for making decisions in specific situations. If adequate information is gathered, the current status of the system can be addressed and the next action can be determined. This can allow the proposed framework to be useful in diverse architectures, and enable the number of operations to be increased for maintaining stable service quality. In the case of certain services, where the possible workload is not known, the alternative dynamic auto scaling mechanism of the proposed framework is an effective solution. Unlike Vasar et al.'s scheme where the request-arrival rate on the server side is monitored, the dynamic auto scaling mechanism proposed here can be used to determine performance change and handle unpredictable variations of service turnaround times. The overall service quality obtained using the proposed framework is highly stable when proper thresholds and sampling periods are chosen.

The proposed framework is suitable for any virtualized service in the cloud. It can be readily applied to certain services that can be used to increase service system capacity by scaling out in order to meet service performance requirements and improve user experience. For example, increased numbers of web servers or processing servers can be provided in file storage services to maintain stable performance during peak times and avoid service interruption in order to save user time and retain user trust in the service. The Pre-configuration maker used in the proposed framework can guarantee stable quality in cloud service even dealing with different workload

patterns, such as impulses and steps. In addition, the proposed framework only monitored turnaround time as the threshold metric. Integrating additional metrics or connecting to the monitoring service of the IaaS provider in order to obtain additional information might be useful for generating distinct policies for adapting to disparate application scenarios. For instance, the arrival rate could be used for choosing a suitable increment number of instances whenever the turnaround time reaches the threshold in order to handle large numbers of situations. These metrics can be used to complement each other in order to eliminate monitoring blind spots.

V. CONCLUSION

This study proposed a cloud resource management framework that can be used across multiple cloud platforms in order to deploy, monitor, and scale out a service automatically. The framework is driven by turnaround time, and provides both dynamic and schedule-based auto scaling mechanisms for ensuring the stability of service performance. The proposed schedule-based mechanism has demonstrated to be a better choice when workload variation can be predicted. The experimental results shows that the concept of turnaround time driven auto scaling can provide stable service quality, and the function used for generating the pre-configuration in the proposed framework is effective. Compared with Vasar et al.'s scheme [22], both proposed auto-scaling mechanisms could guarantee a better performance and enhance service quality. In future, the proposed mechanisms are suggested to incorporate with some predictive auto-scaling techniques to provide quantitative resource scaling recommendation, which could be very useful for disparate application scenarios.

REFERENCES

[1] M. Cusumano, "Cloud computing and SaaS as new computing platforms," *Commun. ACM*, vol. 53, no. 4, pp. 27–29, Apr. 2010.
[2] A. F. M. Hani, I. V. Paputungan, and M. F. Hassan, "Renegotiation in service level agreement management for a cloud-based system," *ACM Comput. Surv.*, vol. 47, no. 3, Apr. 2015, Art. no. 51.
[3] M. N. O. Sadiku, S. M. Musa, and O. D. Momoh, "Cloud computing: Opportunities and challenges," *IEEE Potentials*, vol. 33, no. 1, pp. 34–36, Jan./Feb. 2014.
[4] S. Mistry, A. Bouguettaya, H. Dong, and A. K. Qin, "Metaheuristic optimization for long-term IaaS service composition," *IEEE Trans. Serv. Comput.*, to be published, doi: 10.1109/TSC.2016.2542068.
[5] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki, "Predictive auto-scaling techniques for clouds subjected to requests with service level agreements," in *Proc. IEEE World Congr. Services*, Jun. 2015, pp. 311–318.
[6] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, Dec. 2014.
[7] Y. W. Ahn, A. M. K. Cheng, J. Baek, M. Jo, and H.-H. Chen, "An auto-scaling mechanism for virtual resources to support mobile, pervasive, real-time healthcare applications in cloud computing," *IEEE Netw.*, vol. 27, no. 5, pp. 62–68, Sep./Oct. 2013.
[8] M. Maurer, I. Brandic, and R. Sakellariou, "Enacting SLAs in clouds using rules," in *Proc. 17th Int. Conf. Parallel Process.*, vol. 1. Berlin, Germany, 2011, pp. 455–466.
[9] J. Yang *et al.*, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Inf. Syst. Frontiers*, vol. 16, no. 1, pp. 7–18, Mar. 2014.
[10] Amazon Web Services, Inc. *AWS CloudWatch—Cloud & Network Monitoring Services*, accessed on Jun. 1, 2016. [Online]. Available: http://aws.amazon.com/cloudwatch/
[11] *RightScale: Cloud Portfolio Management by RightScale*, accessed on Jun. 1, 2016. [Online]. Available: http://www.rightscale.com
[12] *Scalr Enterprise Cloud Management Platform*, accessed on Jun. 1, 2016. [Online]. Available: http://www.scalr.com/
[13] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 67–78.
[14] F. Zhao, J. Si, and J. Wang, "Research on optimal schedule strategy for active distribution network using particle swarm optimization combined with bacterial foraging algorithm," *Int. J. Elect. Power Energy Syst.*, vol. 78, pp. 637–646, Jun. 2016.
[15] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *Proc. 11th IEEE/ACM Int. Conf. Grid Comput. (GRID)*, Oct. 2010, pp. 41–48.
[16] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, New York, NY, USA, Nov. 2011, pp. 49:1–49:12.
[17] Z. Xiao, Q. Chen, and H. Luo, "Automatic scaling of Internet applications for cloud computing services," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1111–1123, May 2014.
[18] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *Proc. World Conf. Services-I*, Jul. 2009, pp. 693–700.
[19] M. Firdhous, O. Ghazali, and S. Hassan, "Modeling of cloud system using Erlang formulas," in *Proc. 17th Asia–Pacific Conf. Commun. (APCC)*, Oct. 2011, pp. 411–416.
[20] T. C. Chieu, A. Mohindra, and A. A. Karve, "Scalability and performance of Web applications in a compute cloud," in *Proc. IEEE 8th Int. Conf. e-Business Eng.*, Oct. 2011, pp. 317–323.
[21] W. Iqbal, M. Dailey, and D. Carrera, "SLA-driven adaptive resource management for Web applications on a heterogeneous compute cloud," in *Cloud Computing*. Berlin, Germany: Springer, 2009, pp. 243–253.
[22] M. Vasar, S. N. Srirama, and M. Dumas, "Framework for monitoring and testing Web application scalability on the cloud," in *Proc. WICSA/ECSA Companion*, New, York, NY, USA, 2012, pp. 53–60.
[23] J. Dejun, G. Pierre, and C.-H. Chi, "EC2 performance analysis for resource provisioning of service-oriented applications," in *Proc. Int. Conf. Service-Oriented Comput.*, Berlin, Germany, 2009, pp. 197–207.
[24] *EU Mobile Cloud Networking*, accessed on Jun. 1, 2016. [Online], Available: http://www.mobile-cloud-networking.eu/site/
[25] X. L. Liu, S. M. Yuan, G. H. Luo, and H. Y. Huang, "Auto-scaling mechanism for cloud resource management based on client-side turnaround time," in *Advances in Intelligent Systems and Computing*, Cham, Switzerland: Springer, vol. 388. 2016, pp. 209–219.
[26] *OpenStack Open Source Cloud Computing Software*, accessed on Jun. 1, 2016. [Online]. Available: http://www.openstack.org
[27] Amazon Web Services, Inc. *AWS CloudFormation—Configuration Management & Cloud Orchestration*. [Online]. Available: https://aws.amazon.com/cloudformation/
[28] GitHub. *Locustio/Locust*. [Online]. Available: https://github.com/locustio/locust
[29] GitHub. *Stackp/Droopy*, accessed on Jun. 1, 2016. [Online]. Available: https://github.com/stackp/Droopyy

XIAOLONG LIU received the B.S. degree from Xiamen University, China, in 2011, the M.S. degree in computer science and information management from Providence University, Taiwan, in 2013, and the Ph.D. degree from the Institute of Computer Science and Engineering, National Chiao Tung University, Taiwan, in 2016.

Since 2016, he has been an Assistant Professor with the College of Computer and Information Sciences, Fujian Agriculture and Forestry University. He has authored over 15 peer-reviewed international journals and conferences. His research interests include internet technologies, multimedia security, and service computing.

Dr. Liu was a recipient of the Taiwan Association for Web Intelligence Consortium Best Dissertation Award in 2016.

SHYAN-MING YUAN received the Ph.D. degree in computer science from the University of Maryland at College Park, College Park, in 1989. In 1990, he was an Associate Professor with the Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan.

Since 1995, he has been a Professor with the Department of Computer Science, National Chiao Tung University, where he is also the Director of the National Chiao Tung University Library. His current research interests include distance learning, internet technologies, and distributed computing.

GUO-HENG LUO received the master's degree from the Institute of Computer Science and Engineering, National Chiao Tung University, in 2009, where he is currently pursuing the Ph.D. degree. His research interests are in Web 2.0, Parallel and Cloud Computing.

HAO-YU HUANG received the master's degree from the Institute of Computer Science and Engineering, National Chiao Tung University, in 2014. He is currently a Researcher with CyberLink, Taiwan. His research interests include parallel and cloud computing.

PAOLO BELLAVISTA is currently an Associate Professor with Alma Mater Studiorum, University of Bologna, Italy. His research activities span from mobile computing to pervasive ubiquitous middleware, from Internet of Things middleware for scalable cloud integration to vehicular sensor networks, from mobile cloud computing to big data adaptive stream processing, and from fog computing to resource management in SDN.

He published over 60 magazine/journal articles and over 80 conference/workshop papers in those fields, reporting results from several national- and EU-funded projects. In 2016, he was a Keynote Speaker of the IEEE Mobile Cloud conference. He serves on the Editorial Board of the IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, the IEEE TRANSACTIONS ON SERVICES COMPUTING, the *IEEE Communications Magazine*, the IEEE TRANSACTIONS ON COMPUTERS, the *Elsevier Pervasive and Mobile Computing Journal*, the *Elsevier Journal on Network and Computer Applications*, the *Elsevier Journal of System Architecture*, the *Springer Wireless Networks*, the *Springer Journal of Network and System Management*, and the *International Journal of Handheld Computing Research*.