Simulation Modelling Practice and Theory 101 (2020) 102032

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Fog and cloud collaboration to perform virtual simulation experiments

Khaldoon Al-Zoubi[a,*], Gabriel Wainer[b]

[a] Faculty of Computer & Information Technology, Jordan University of Science & Technology (JUST), Jordan
[b] Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

ARTICLE INFO

Keywords:
Modeling & Simulation (M&S)
Edge computing
Fog computing
Cloud computing
Simulation experiments

ABSTRACT

Fog Computing can enhance users' quality of service, particularly when countless users spread around the globe to access the same Cloud resources. We present a set of collaboration mechanisms between Fog and Cloud computing resources to conduct simulation experiments. A user creates an experiment on a Fog server (with a *model* attached to it) without worrying where and how the simulation will be executed. Once a simulation starts, the experiment reach servers with simulation environments that can execute the *model* and then needs to select the best servers to perform the actual simulation. We introduce the concept of *Virtual Experiments (VE)* to decouple M&S environments specifics from the general experiment framework, providing interplay processing units between users and simulation servers. In addition, we present a Fog/Cloud scalable architecture, and discuss how the M&S capabilities are advertised, structured in pools, dynamically discovered, and selected to simulate. As a proof of concept, we built our concrete private Clouds and Fogs based on OpenStack to demonstrate the proposed ideas using various Fog and Cloud physical, deployment and computing capabilities.

1. Introduction

Cisco introduced the concept of *Fog computing* in 2012 [6]. It is defined as a "Cloud close to the ground" [6], meaning that we want to move part of the Cloud computation closer to the users. In fact, as suggested by surveys like [18,20,32], Fog computing utilize users nearby resources that are usually idle off peak hours. Fog computing has also been linked to the concept of the Internet of Things (IoT) [6,32]. As IoT protocols provide means to connect all kind of devices, Fog is a promising idea to place some computation resources near these countless devices, allowing Fog servers to handle some of the processing. Users may be served by their Fog local resources, if possible; otherwise, the Fog servers need to get those services from somewhere else (for instance, the Cloud).

The *collaboration* mechanisms between Fog and Cloud servers play a key role on the overall services provisions, and we thus present advanced collaboration mechanisms between Fog/Cloud servers to perform simulation experiments, which can enhance Modelling and Simulation (M&S) technology. We propose mechanisms to enhance the Fog/Cloud collaboration so that users can create and manipulate their simulation experiments via Fog servers. We have the following practical requirements:

1 The software architecture should be *scalable*. The collaboration mechanisms of the overall architecture should be independent of any increasing number of Fog or Cloud computing resources. This includes the exchanged messages between servers. Our solution to this requirement is to organize all the servers hierarchically. We place Fog servers (i.e. users' entry points) on the top of the

* Corresponding author.
E-mail addresses: ktalzoubi@just.edu.jo (K. Al-Zoubi), gwainer@sce.carleton.ca (G. Wainer).

https://doi.org/10.1016/j.simpat.2019.102032
Received 30 June 2019; Received in revised form 22 October 2019; Accepted 26 November 2019
Available online 26 November 2019
1569-190X/

hierarchy; consequently, they can reach all information via their children servers.

2 Fog servers should detect and discover M&S resources change dynamically (i.e. knowledge discovery). This needs to be done with the assumption that servers can join or leave the overall architecture at any time.
Our solution is that when a server joins the architecture, it *advertises* its capabilities to its parents. The resource pools are then organized along the way in the hierarchy until they reach the Fog servers on top. Servers leaving the architecture are also managed in the same way, but in top-down direction instead.

3 Simulation servers are not equal and may support different M&S capabilities. This means that some simulation models may not be able to execute on some servers. The Fog servers should know how to reach compatible resources. To do this, we group similar M&S resources together in pools. This makes matching models to simulation environments a trivial task by Fog servers.

In our previous research, we introduced the RESTful Interoperability Simulation Environment (RISE) [4] middleware. RISE provides a RESTful API for integrated simulation environments, particularly DEVS-based tools [28]. RISE currently supports different versions of CD++ like the distributed CD++ [3], which allows different distributed partitions to execute the simulation. We reused two RISE components (1) the RESTful WS framework API to be able to communicate through the Web using REST style and (2) The access to the DEVS tool via the REST API to run simulation experiments.

As a proof of concept, we built Fogs and Clouds from privately owned computing resources. We used OpenStack [19], which provides Computing, Storage and Networking virtualization services. OpenStack services are interoperated using RESTful WS.

To this end, while meeting the previously stated requirements, the main contributions of the presented research here (with respect to other existing platforms) can be summarized as follows:

(1) *Development of algorithms and a full-fledged Fog/Cloud infrastructure to conduct simulation experiments*. This includes Fog/Cloud collaboration mechanisms to allow client devices to create and manipulate their experiments on nearby Fog nodes. When a client starts the simulation, the experiment **(i)** needs to *discover* the simulation environments that can execute the simulation, and then **(ii)** out of those discovered resources, *select* the best resources (in our case, the least loaded servers that advertised the M&S resources). To enable experiments to perform these two steps, the heterogeneous M&S resources are dynamically discovered and organized in form of homogeneous *resources Pools*, making resources discovery a trivial task by experiments. Further, those Pools are dynamically being updated with information (e.g. load) to improve the selection of resources to execute the simulation. Once the simulation starts execution, the type of simulation (e.g. parallel, distributed, real-time, etc.) depends on the used simulation environment and experiment settings.
This workflow to conduct simulation experiments over Fog/Cloud is a completely novel approach. In contrast, existing M&S platforms perform simulation by directly connecting the client devices with the VMs that are previously known to be able to run the simulation. This makes client devices aware of the centralized cloud VMs capabilities and locations. Further, this prevents clients from taking advantage of organizations' local mini-clouds (i.e. Fogs).

(2) *New Virtualization layers at the level of experiments and M&S resources*. Having this new layer is important to hide cloud details (e.g. VMs) from simulation experiments, decoupling M&S resources and experiments from Fog/Cloud details. The following layers are defined: (a) experiments layer (to setup and execute simulations), (b) M&S resources layer (to discover and select M&S resources while hiding VMs details from experiments), (c) VMs layer (to host M&S resources and hide the hardware details), and (d) physical hardware. Once a VM is launched, it advertises its M&S capabilities to the upper layer, allowing M&S resources to be organized and made available to experiments. In turn, when an experiment wants to execute a simulation, it goes through the M&S resources to discover and select suitable M&S resources to carry out the simulation. Once this is done, the VMs that host those resources is indirectly discovered and selected (but this is hidden from the experiments). Decoupling M&S resources from VMs resembles decoupling OS from hardware within a VM. This virtualization of M&S resources is important because VMs are not equal since they may advertise different M&S resources. Comparing to existing approaches, discovery, selection and managing in our approach happens at the level of M&S resources rather than at the level of VMs. In existing approaches, the highest layer is the VMs management layer (i.e. cloud resources). In this case, VMs are treated as computing powers to execute simulation jobs, that need then to be scheduled over available VMs with respect to some parameters to balance the load. As can be seen that those approaches assume that all VMs have the same M&S capabilities, hence there is no need for the discovery stage (as in our case) since all VMs have the same M&S capabilities. However, in the Cloud, VMs normally support heterogeneous M&S resources that need to be discovered before being matched to the simulation jobs that they can execute.

(3) *Virtual Experiments (VEs)*: clients devices can view Fog nodes (i.e. that belong to local mini-clouds) as the magic servers that can setup and execute experiments using various simulation environments. Because these experiments give clients the impression of conducting simulation over different simulation environments, they are called *Virtual Experiments*.

The rest of the paper is organized as follows: Section 2 presents relevant related work and compares them to our presented contributions. Section 3 discusses the Virtual Experiments framework, interactions and API. It also describes how users would use their experiments over the Fog/Cloud resources. Section 4 describes the M&S resources management. This means how M&S resources are discovered and organized so that experiments (Section 3) can select compatible M&S resources to execute simulation. Section 5 demonstrates proof of concept cases using privately built Clouds and Fogs using OpenStack [19]. Conclusions are presented in Section 6.

2. Related work

Simulation has been used for decades, and one of the starting points was the effort by the U.S Department of Defense (DoD), to use it for war games purposes in 1950s [17]. As networking technologies developed, simulations have also advanced to reach each other across geographical areas to *collaborate* and *reuse* each other services to solve common problems. Examples of such platforms in the defense sector are SIMulator NETworking (SIMNET) in 1983 [7], Distributed Interactive Simulation (DIS) standards in 1990s [13], and the High-Level Architecture (HLA) standard in 1999 [14]. Distributed simulation also progressed outside the defense sector. Examples of such works are based on the Common Object Request Broker Architecture (CORBA) (e.g. [9]) during 1990s, SOAP-based Web-services (e.g. [29]), and RESTful-based Web-services (e.g. [4,3]). As REST is the interoperability style that the Web itself uses, and it addresses resources with URIs and exchanges messages between those URIs via uniform interface (i.e. HTTP methods), interoperability of models and simulations is also improved. Since then, RESTful WS have become popular in various applications, particularly with the emergence of Cloud computing technology (e.g. [19,23]). Nowadays, Clouds typically exposes their services via RESTful WS APIs to ease the interoperability of software systems with Cloud services via the Internet.

Despite of the success of Cloud computing for providing and managing computing resources, there are a few issues not solved. Cloud computing tends to be in centralized datacenters; however, users need to access the Cloud from virtually everywhere. Clouds need to consider mobility support, geographical locations awareness, service latency, resources allocation, and geographical-location awareness [6,18,20,32]. To overcome such issues, Cisco introduced the concept of *Fog computing* in 2012 [6]. It is defined as a "cloud close to the ground" [6], meaning that we want to move part of the cloud computation closer to the end users. In fact, as suggested by surveys like [18,20,32], Fog computing should utilize users nearby resources that are usually idle off peak hours. Because of this, we preferred building our own private clouds based on OpenStack [19].

OpenStack [19] is an open-source cloud operating system software tools that was originally developed by NASA and Rackspace. The major OpenStack services are *Storage, Compute*, and *Networking* each with its own RESTful API. OpenStack is currently supported by a long list of companies such as AT&T, Ericsson, Huawei, Intel, Rackspace.

2.1. Cloud-based simulation related work

Research in [16,31] showed how to deploy HLA-based simulations on the Cloud as SaaS-oriented frameworks. Others, like [22,23], deployed HLA-based simulations on the Cloud using containers rather than using virtual machines (VMs), as containers on-demand provisions are lightweight compared to VMs. This is true if VMs or containers need to be started upon simulation start, because VMs virtualize the actual computer hardware, but containers only virtualize the operating system.

We also deploy Simulation as Service middleware (servers) on the Cloud; however, our approach is different as we assume that the M&S capabilities may not be the same on all servers. Before simulating, our Fog servers find the correct server(s) with the capability to execute the simulation based on end user's configuration. Another difference is that in our case, VMs are viewed as dedicated servers with M&S capabilities and they need to advertise their M&S capabilities, allowing Fog servers to discover them.

Further, cloud-based scheduling algorithms have been studied by different researchers. For example [12,17] proposes energy aware scheduling (based on the VMs temperatures) to schedule virtual machines and minimize energy consumption in data centers. The work in [21] uses Formal Concept Analysis to schedule tasks on VMs, and [23] uses different parameters to schedule containers on the Cloud (e.g., simulation deadlines) to enhancing quality of service. All these works view VMs (servers) as a *pool* of processing units with the same M&S capabilities. In contrast, we structure M&S capabilities in pools (rather than VMs). When a new VM server is launched, its M&S capability is dynamically detected and added to the M&S pools. Therefore, users can setup experiments on Fog servers without the need to know how and where the Fog servers will execute the simulation. We call these *Virtual Experiments*. In other above related work experiments like [16,31], users are aware of the M&S capabilities locations (and they need to create experiments on servers with the required M&S capabilities to run their simulations). Further, once the experiment finds the servers with the required M&S capabilities, they need to select the servers with the least load.

Furthermore, other research like [12,17] use first-fit scheduling, while [8] is based on best-fit scheduling. These scheduling algorithms try to fit the jobs based on their start/end times. Based on this they find the computing resources to do so. However, in practice, it is hard to estimate the simulation jobs completion time or the load they add onto the servers, because the simulation will continue if there are events to execute (which may further generate other events). Thus, this depends on the model that is being executed and the simulation events generated. For this reason, our approach is to select the least loaded resources at the time of starting a simulation execution.

2.2. Fog-based simulation related work

Several simulators have been developed with the purpose of modeling Fog computing environments. For example, EdgeCloudsim [24] simulates Fog servers' computation and networking capabilities. SimpleIoTSimulator [26] creates Fog test environments with thousands of sensors and Fog servers. iFogSim [11] simulates Fog computing environments with large number of Fog nodes and IoT devices.

Other related Fog-based simulations have targeted specific issues. For example, Assila et al. [5] uses simulation to study caching in the Fog layer for many-to-one matching games between a set of client devices and a set of Fog devices. Further, Wang et al. [30] uses simulation to study military operations latency and effectiveness when adding a Fog layer between users and the Cloud backbone. These treat the Cloud as storage resources, which is different from our case, where the collaboration between Fog and Cloud servers

Table 1
Examples of Fog/Cloud Related work.

Related Work	Application Services	Fog Functions	Cloud Functions
Dubey et al. [10]	Collect data from various wearable sensors used for tele health applications	Processes the collected raw data from monitoring sensors and converts it into patterns. Also, zip & unzip files	Database
Stantchev et al. [27]	Collect data from health monitoring sensors	Check data and contact help if an intervention is needed. Otherwise, forward data to the Cloud	check data and contact help if an intervention is needed
Zamfir et al. [33]	provides a platform for IoT prototyping health assistive and monitoring applications	Partial data processing	Large data processing
Zhang et al. [34]	Home automation	Moved some controls from the Cloud to be processed at local home gateways to enhance users' privacy	Backups

starts when the simulation starts in an experiment. The experiment then finds the servers with the required M&S capabilities and the collaboration ends when the simulation is completed. If information is needed about previous simulation runs, they only communicate with the Fog servers.

Others have built simulators for Big Data collection and processing (motivated by the IoT). For example, Abdelhafidh et al. [1] proposes a platform to collect and process large numbers of data collected from sensors within a water pipeline monitoring system. BigDataNetSim [2] models the main components of the data movements in Big Data platforms such as network topologies and switching/routing protocols. MRSG [15], which is implemented on top of SimGrid [25], is a MapReduce parallel simulator that produces a large set of data to mimic users' devices in IoT computing environments. This research could be combined with ours. Big Data could input substantial amounts of data into experiments when simulating IoT. Our *Virtual Experiments* could then be used to prepare and filter data (on the Fog server) before inputting to the actual simulation engine. Likewise, they could process simulation outputs before sending it to users' devices. The *Virtual Experiments* can be viewed as interplay logical units between the user and the simulation. This interplay data processing is outside the scope of this article, and part of our future research plans.

2.3. Fog/cloud collaboration related work

Table 1 shows a sample of the use of Fog computing as a middle-processing layer between users and Clouds. They all have in common that Fog nodes do some local processing, and, based on some decision they may forward data to the Cloud. Our work is different from those as we have simulation experiments that know how to find the appropriate M&S capabilities to run the simulation. Further, our solution can detect and discover when M&S capabilities are added (or removed).

3. Virtual experiments

Before presenting our collaboration mechanisms, we need to remember that M&S is based on two aspects: the *model* and the *simulation*. A *Model* is the representation of a real system of interest that captures the necessary information in form of equations or other non-formal mechanism. The *simulation* environment (or engine/tool) executes the *model*. In practice, *models* can be expressed as files of some format that a simulation engine knows how to parse and execute. Further, models can also exist as source code of some programming language that might be compiled with the simulation engine source code before being executed. Regardless how a model is expressed, the assumption here is that modeling is separated from simulation. Therefore, experiments, in our case, are created by users with configuration settings, including models that can be executed by compatible simulation environments.

Based on these concepts, this section describes how users conduct their Virtual Experiments over Fog/Cloud resources. Fog Servers are the entry points for users to setup and configure Virtual Experiments. Virtual Experiments indicate that experiments are decoupled from M&S environments specifics. This means that users are not concerned with how or where the experiment is going to find server(s) equipped with the required M&S capabilities that can execute the simulation. Experiments find these servers with the help of a local component (on the same Fog server), called *Scheduler*. Schedulers (discussed in Section 4) are responsible for the overall M&S resources management, discovery, and scheduling.

3.1. Experiments organization and interactions

Fig. 1 shows example of how users run simulations within their experiments. This example shows that users would only access services via the Fog servers. For example, when User1 starts simulation on experiment E1 (on Server-A), E1 requests the Scheduler (on Server-A) to find a server. In this case, it obtained Server-B (on the other Fog). Consequently, experiment E1 creates the experiment E3 (on Server-B) with all required data and starts the simulation. Now, E1 is an interplay unit between the user and the simulation on E3. This means that it would handle all interactions between user and the running simulation. However, when the simulation ends, E3 is deleted while all previous simulation results are saved in experiment E1. However, when User2 starts a simulation on experiment E2, E2 requests the Scheduler (on Server-A) to find the servers. In this case, it obtained Server-B and Server-C (on the Cloud). Accordingly, E6 experiment is created on Server-C and Server-D (on the Cloud). This is a distributed simulation

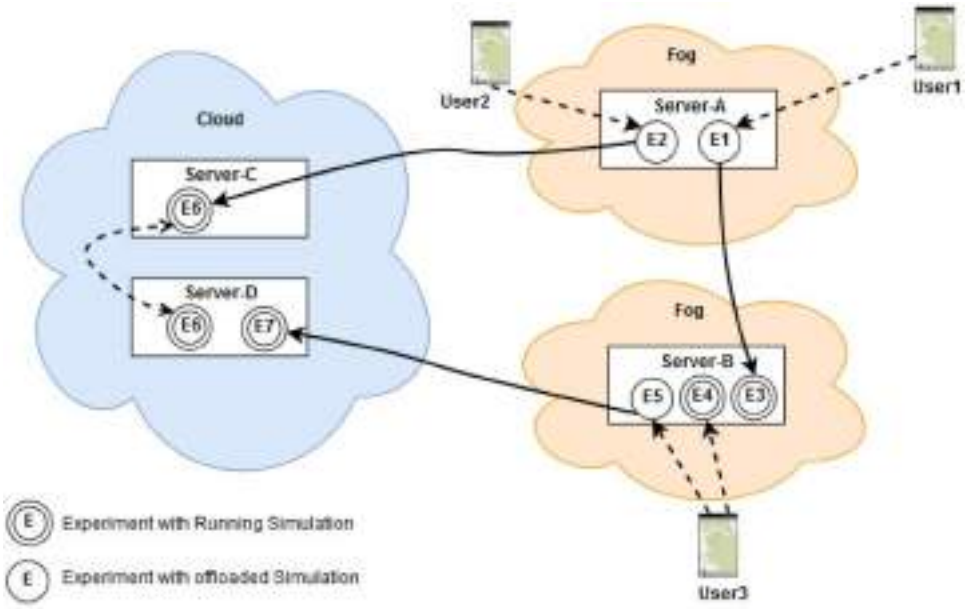


Fig. 1. Example of Users Experiments behavior over Fog/Cloud Servers.

partitioned over two servers. In our case, we use the distributed version of CD++, which has been presented in [3]. This shows how the Virtual Experiment concept can hide the simulation environments details from users. As far as User2 concern, E2 is running on Server-A. For User3, two experiments are running: E4 on the local Fog server itself, and E5, which is offloaded to Server-D and executed within experiment E7.

Fig. 2 shows the major steps for starting a simulation on an experiment instance. The figure assumes that the experiment has already been created with all the required settings, and it is ready to simulate. As the figure shows, in *Step #1* the user starts the simulation by creating the URI …*/simulation* via HTTP PUT. In *Step #2*, the experiment asks the *Scheduler* (on a local server) *where* to execute the simulation. As discussed in later sections, the *Scheduler* knows all M&S resource pools, and knows how to find best middleware/server to execute the simulation. *Step #2* shows that the experiment can request more than one server, based on configuration settings (for example to provide fault tolerance, or to run the simulation in distributed fashion, in which case each portion

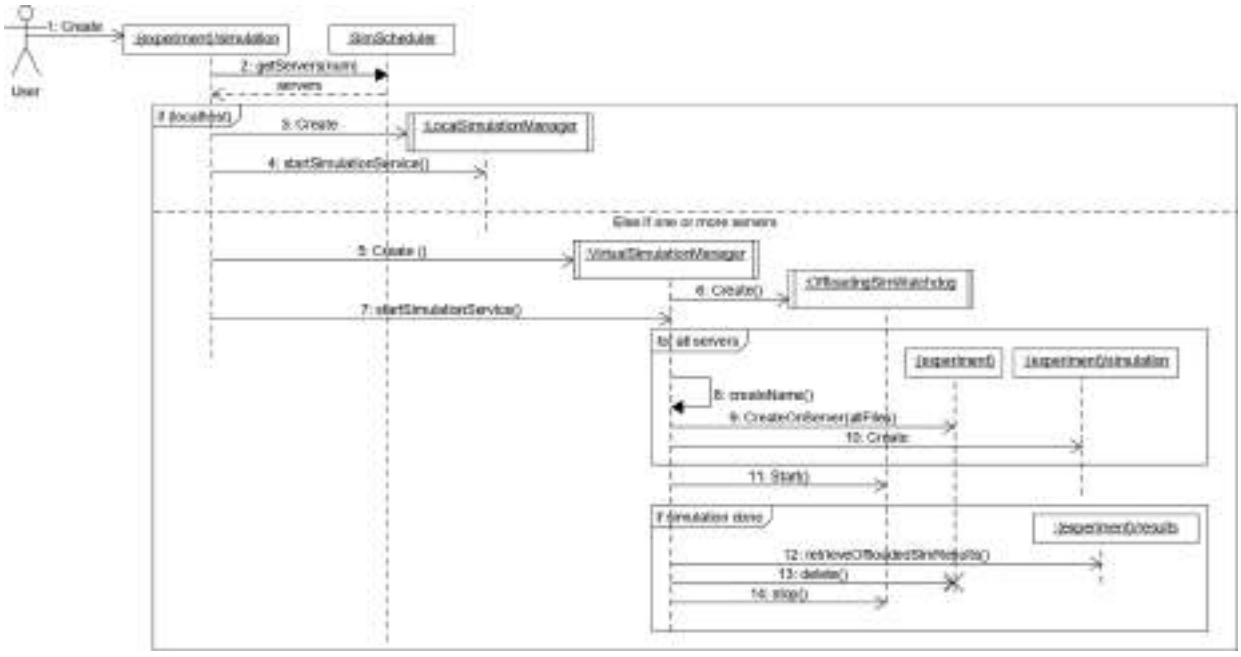


Fig. 2. Starting Simulation on Virtual Experiment (VE).

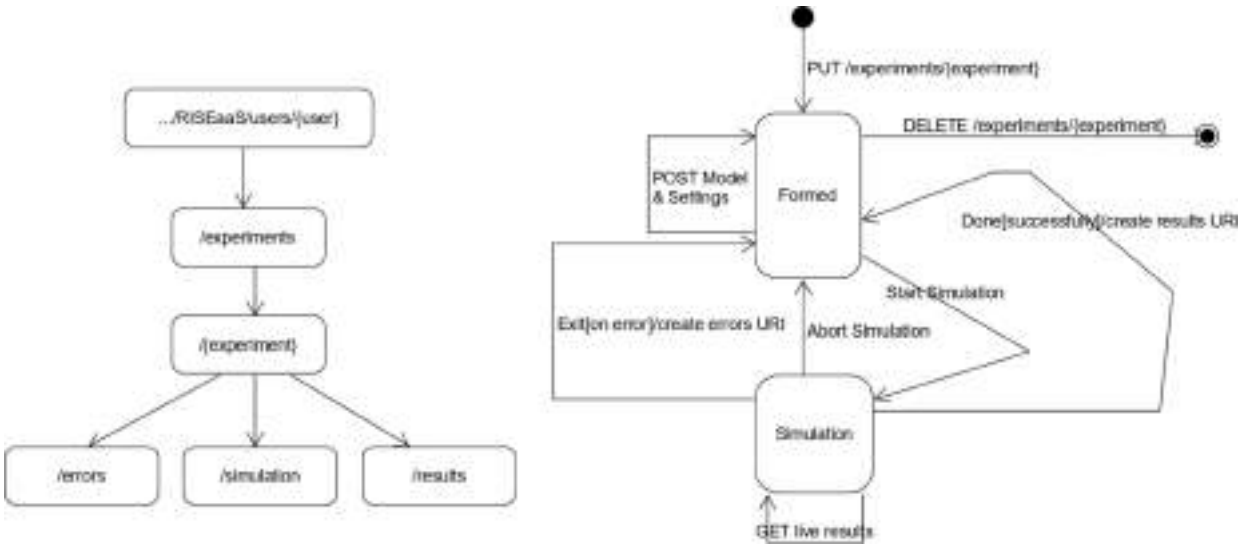


Fig. 3. Virtual Experiment RESTful API and State Diagram.

of the model is simulated on a separate server [3]). Now, if the *Scheduler* returns the localhost, then the simulation is going to be executed on the local Fog middleware as a local experiment. In this case, the local server creates a local simulation manager, as shown in *Step #3*. Then, in *Step #4*, the local server starts the simulation, which in turn starts the required simulation environment to execute the model. However, if the *Scheduler* returns other servers, the simulation needs to be offloaded to them. In this case, in *Step #5*, the local server creates a Virtual Simulation Manager to wrap the simulation, and then in *Step #6*, it creates an offloading simulation watchdog that will be checking the simulation execution. After that, in *Step #7*, the local server starts the simulation by creating temporary experiments on all those servers. To do so, in *Step #8*, an experiment unique UUID name is created. *Step #9* creates a remote experiment with that unique name, and it submits all required files and settings. *Step #10* starts the simulation on all the remote experiments. In *Step #11* the offloading simulation watchdog starts. Finally, when the simulation is completed, the local experiment retrieves the results (*Step #12*), deletes other experiments (*Step #13*), and conducts local cleanup (*Step #14*).

3.2. Experiments framework and implementation

The experiment is built as a container that wraps all things related to executing simulation for a model. The experiment exposes certain URIs to allow users to control and manipulate the experiment at all its phases. Fig. 3 shows the experiment's RESTful API and the states that it usually goes through.

The Experiment API is shown in Fig. 3-left. Users send HTTP requests using the PUT method on the URI (*…/RISEaaS/users/{user}/experiments/{experiment}*). This is the main URI for an experiment instance. For instance, URI (*…/RISEaaS/users/**John**/experiments/**FireModel***) is the URI used for experiment instance *FireModel* for user *John*. As a result, the experiment instance advances to the *Formed* state (Fig. 3-right). In this state, a user can submit models and other settings via HTTP POST. Further, the user can delete the experiment instance via HTTP DELETE on the experiment main URI. The user starts the simulation via PUT on URI (*…/{experiment}/simulation*) (Fig. 3-left). This would create the active simulation URI and put the experiment in *Simulation* state (Fig. 3-right). In this state, the user can retrieve live results via HTTP GET using URI *…/simulation*. Further, an active simulation can be aborted via DELETE on this URI. If the simulation completes successfully, the *…/{experiment}/results* is created, allowing results to be downloaded and replayed later. Users can retrieve results during running simulation via HTTP GET using URI *…/simulation*. If the simulation ends with an error, the URI *…/{experiment}/errors* is created.

Fig. 4 shows the design of Java implementation of the VE. The experiment RESTful API discussed in Fig. 3-1 is implemented by four classes (in Fig. 4): (1) *ExperimentRestfulAPI,* which handles the HTTP requests to the URI *…/experiments/{experiment}*, (2) *SimulationRestfulAPI*, which handles the HTTP requests to URI *…/{experiment}/simulation*, (3) *ErrorsRestfulAPI*, which handles the HTTP requests to URI *…/{experiment}/errors*, and (4) *ResultsRestfulAPI*, which handles the HTTP requests to URI *…/{experiment}/results*. Each of those classes contains the methods required to be invoked by the RESTful framework based on the HTTP method in the HTTP request, as follows: (1) *represent()* handles the GET method, (2) *acceptRepresentation()* handles the POST method, (3) *removeRepresentations()* handles the DELETE method, and (4) *storeRepresentation()* handles the PUT method. When the simulation starts (Fig. 2), an instance of *SimulationRestfulAPI* is created and *storeRepresentation()* is called. Then, an instance of class *VirtualSimulationManager* is created, which represents an active simulation. The simulation starts via *startSimulationService()*, which in turn creates experiments on other servers and runs the simulations on them as previously discussed.

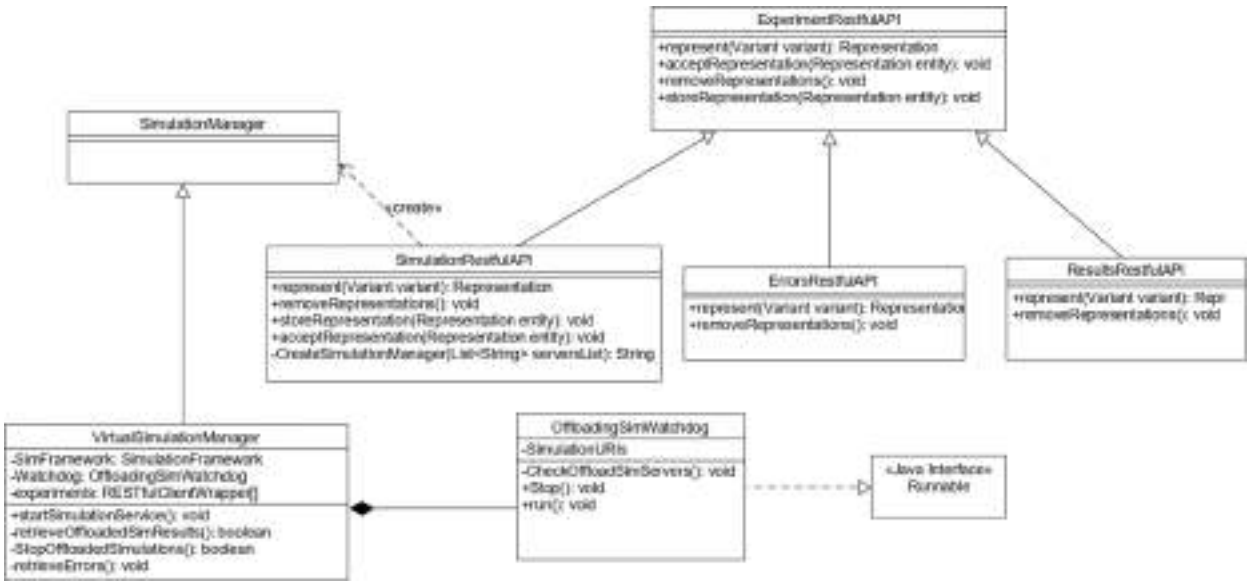


Fig. 4. Virtual Experiment (VE) Context Implementation.

4. M&S resources management

As discussed in Section 3, experiments use the local *Scheduler* to find servers to execute the simulation. Schedulers need to find servers such that: (1) those servers can execute the simulation, since they are not equal and may support different M&S capabilities, and (2) servers load should be taken into consideration for performance reasons. To do so, Schedulers manage M&S resources in two stages: *Stage #1*: Discover dynamically M&S capabilities and organize them in such way to ease matching experiments to compatible M&S capabilities. In this stage, similar M&S capabilities are organized in identical pools and made available to Fog servers, easing the matching process. This *Discovery* stage is discussed in Section 4.2. *Stage #2*: Based on the structured resource pools formed in Stage #1, load-parameters are periodically collected about each pool, and made available to Fog servers, easing the servers' selection process. This *Selection* stage is discussed in Section 4.3.

The overall system architecture must be independent of increasing number of Fog and Cloud servers, hence *scalable*. Further, message interactions between Schedulers must also be independent of enlarging the number of participant Fog/Cloud servers. To solve this problem, Schedulers are structured in hierarchy fashion. Further, resources management interaction messages are *only* passed between a Scheduler and its parents/children Schedulers (if any). Root Schedulers (i.e. that are without parents) are the Fog servers, which are users' entry points. This Schedulers hierarchical architecture is discussed next.

4.1. Hierarchical architecture and deployment

Each server (middleware) communicates management control messages using their local running components, called Schedulers. Thus, we could logically consider servers are also structured hierarchically. A typical deployment example is shown in Fig. 5, where M&S resources are virtualized between two tenants. Tenants are the way to let different organizations to share same physical resources but with complete separate virtual networks. This gives each tenant the impression of owning all physical resources. *Tenant A* owns resources on Fog1, Fog2, and Cloud servers 1, 2 and 3. *Tenant B* owns Fog 3 and Cloud servers 3, 4 and 5. *Tenant B* users can access the M&S resources servers on the Cloud via *F3-Server1*. *Tenant A* users can access M&S resources via *F1-Server1* and *F1-Server2* (on Fog 1), and via *F2-Server1* (on Fog 2). Both tenants' servers are completely separated. To keep tenants separated, each tenant resources are kept in separate XML configurations as described later. For example, Fig. 6 shows two possible deployment configuration for the Tenant A resources in Fig. 5.

Fig. 6-1 shows the hierarchy in two levels (for Tenant A resources in Fig. 5): The Fog servers on the top and all other servers with M&S capabilities at the second level. Even though this configuration is correct, placing Fog servers on top of all resources have some issues, like communicating management control messages with too many servers (particularly when those children servers exist in different Fogs and Clouds). In practice, administrators would place some intermediate nodes within the hierarchy to avoid having such issues. Fig. 6-2 shows an equivalent structure with a new node to manage the three Cloud servers, called *Cloud-Node*. It further added a new node to manage the Fog 1 two servers, called *Fog1-Node*. The three Fog servers in Fig. 6-2 can still discover M&S capabilities as in Fig. 6-1, but with using fewer control messages. As discussed in Sections 4.2 and 4.3, Schedulers only exchange management control messages with their parents and children for better scalability. For example, if we want to give some server access to Fog 1 resources, we then make it parent for *Fog1-Node*. This is done by sending the XML *deployment* message (Fig. 7) via POST (Table 2) to *Schedulers* to configure them with their parents and children. As previously mentioned, each server has this

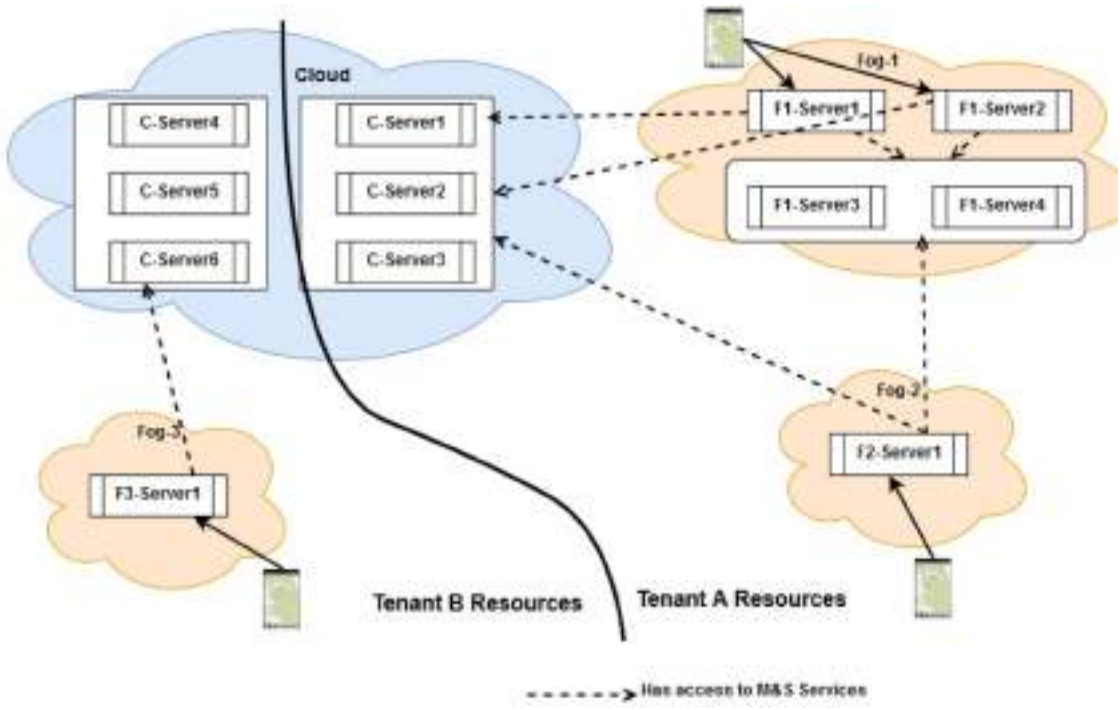


Fig. 5. Deployment Example of Fog/Cloud Servers with Two Tenants.

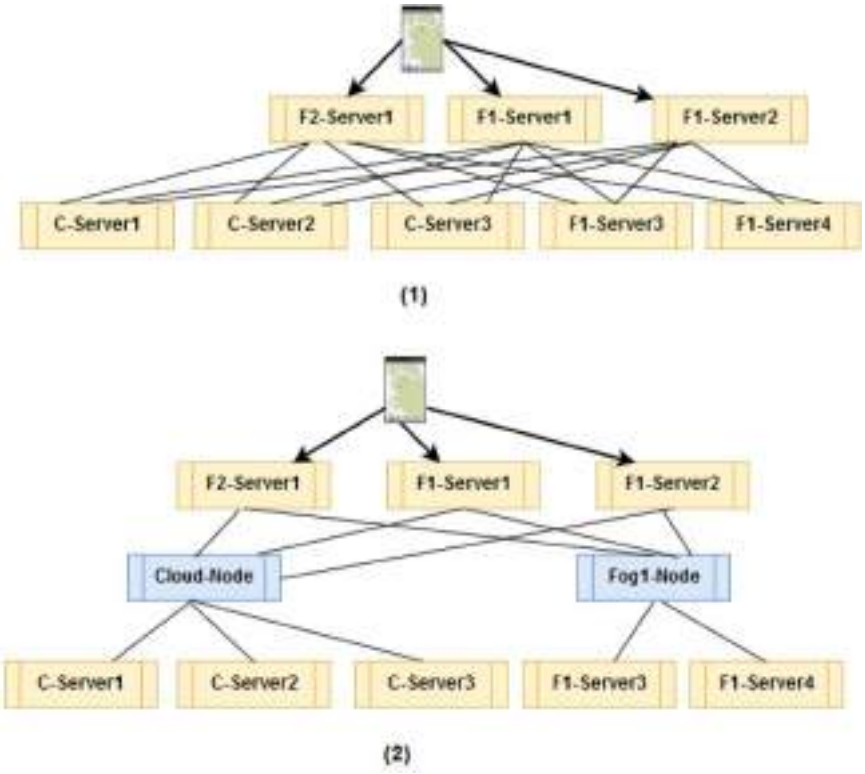


Fig. 6. Hierarchical Structure for Tenant-A Resources deployed in Fig. 5.

```
<deployment>
    <servers>
        <server>
            <URI>…/ Scheduler-1</URI>
            <children>
                    <URI>…/ Scheduler-2</URI>
                    <URI>…/ Scheduler-3</URI>
            </children>
        </server>
        <server>
            <URI>…/ Scheduler-2</URI>
            <parents >
                    <URI>…/ Scheduler-1</URI>
            </parents>
            <children>
                    <URI>…/ Scheduler-4</URI>
                    <URI>…/ Scheduler-5</URI>
            </children>
        </server>
    </servers>
</deployment>
```

Fig. 7. Deployment Configuration Example.

Table 2
Simulation Scheduler RESTful API and Operations.

URI	…/RISE/manage/Scheduler	
Operation	HTTP Method	Description
Deployment	POST	Hierarchical configuration description (Fig. 7). It is Sent to Schedulers to inform them with their children (if any) and their parents (if any).
Check for Loops	GET	Checks for loops in the configuration structure. Query variable "?operation = Checkloop" is attached to the URI.
Advertise Resources	POST	Sent from a child to all parents to advertise its M&S resources capabilities (discussed in Section 4.2 - Fig. 8).
Compute Load	GET	Sent from parents to children to compute Processing Power for all advertised resources (discussed in Section 4.3 – Fig. 9). Query variable "?operation = compute" is attached to the URI.
Get Servers	GET	Find best possible RISE middleware to offload simulation onto them. Query variable "?operation = getServer" is attached to the URI. This always returns one server. However, to return more than one server, it needs to be specified with variable server as follows "?operation = getServer&server = ##"

Scheduler component.

The XML description in Fig. 7 shows a snippet of a typical deployment configuration for two servers. *Scheduler-1* has only two children *Scheduler-2* and *Scheduler-3*. *Scheduler-2* has two children *Scheduler-4* and *Scheduler-5*, and one parent *Scheduler-1*. This message typically passed to the root node, which then passes it throughout the hierarchy. However, this message can also be sent to configure single node. To make sure there are no loops, each node uses operation "checks for loops", in which each parent in the hierarchy sends a message with a unique UUID through their children via HTTP GET. After that, if it gets this message back from one of its parents, it reports to the admin that it sits in a loop with that parent.

Table 2 describes the RESTful API and operations for *Schedulers*. As discussed earlier, each server has a single Scheduler. We have already discussed the first two operations in this subsection. The last three operations will be discussed in more details in Sections 4.2 and 4.3.

4.2. M&S resources discovery

The *Advertise Resources* operation (API in Table 2) is used to organize M&S resources in pools, allowing Fog servers to know how to reach those resources. This starts when a server with M&S capabilities is added to the hierarchy or its M&S capabilities change. The Scheduler on that server advertises its resources and passes them up in the hierarchy. The parents merge advertised resources with similar resource pools, if any. Otherwise, they create new resource pools, and advertise the newly created pools to all parents. This is repeated until it reaches the root servers, which are the Fog servers where users access services, as seen in Fig. 8.

Fig. 8 example shows how to build resource pool channels. The figure shows a hierarchy of seven servers. We only show the *Schedulers* components, since they do the actual work. In this example, the leaf *Schedulers* have M&S capabilities: *Scheduler #4* and *#5* can run simulations on environments *SimA* and *SimB, Scheduler 6* is only enabled with *SimA*, and *Scheduler #7* is only enabled with *SimB*. The intermediate servers with *Scheduler #2* and *#3* are used for resource management organization for better scalability as discussed in Section 4.1. It is worth noting that intermediate nodes could also be used as Fog servers to allow user devices to access the M&S resources. The root server (*Scheduler #1*) is a Fog server, allowing users to access the M&S resources. When a leaf Scheduler is added or its M&S resources change, it sends an advertisement message via POST method to all its parents. For example, leaf *Scheduler #4* (in Fig. 8) sends parent *Scheduler #2* advertisement message like the following:

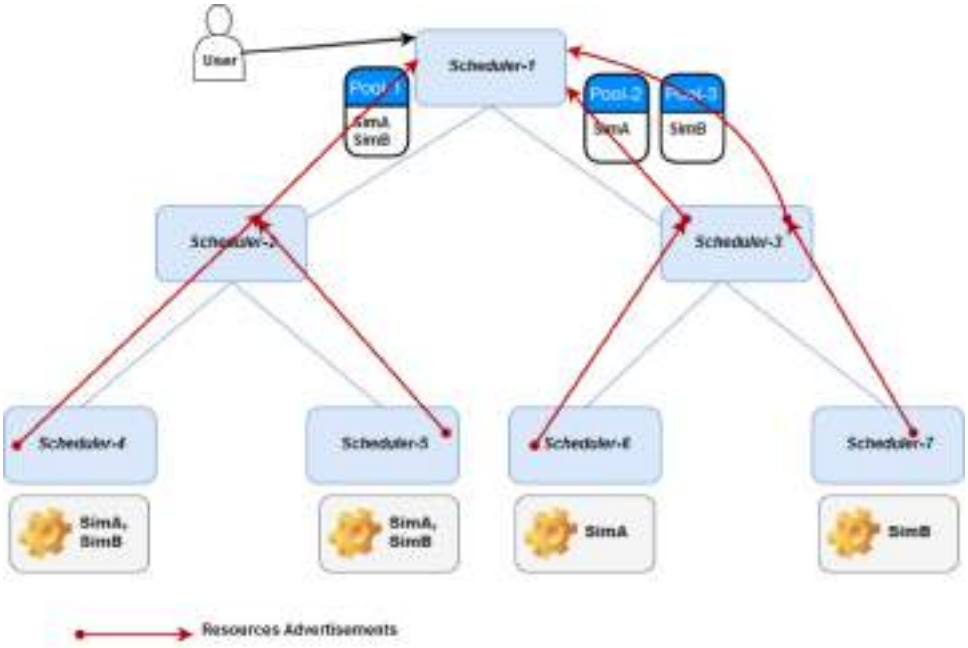


Fig. 8. M&S Resource pools and Advertisements.

```
<advertisement><resources>
<resource><name>SimA</name></resource>
<resource><name>SimB</name></resource>
</resources></advertisement>
```

When parent *Scheduler #2* (in Fig. 8) receives this message, it creates a resource pool object with resources *SimA* and *SimB*. It then passes it to parent *Scheduler #1*, which also creates a resource pool object in a similar way. Consequently, a channel is created between root *Scheduler #1* and the actual resources location at *Scheduler #4*. When leaf *Scheduler #5* advertises its resources *{SimA, SimB}* to *Scheduler #1*, it simply connects *Scheduler #5* to the existing resource pool. This is because the advertised resources match an existing pool channel at *Scheduler #1*. In this case, nothing is passed up in the hierarchy. Now, root *Scheduler #1* views resources located at *Scheduler #4* and *#5* as one pool. In contrast, *Scheduler #2* creates two resource pools when receiving children advertisements and passes both to parent *Scheduler #1*. This is because children advertisements do not match, as they are heterogeneous. Finally, the *Scheduler #1* has three resource pools: (1) *{SimA, SimB}*, (2) *{SimA}*, (3) *{SimB}*. Now *Scheduler #1* knows how to discover and match resources. For example, if a model can only execute on *SimA*, then the first and second pools will only be considered for server selections.

4.3. M&S resources selection stage

The discovery stage discussed in Section 4.2 allows Schedulers to match experiments to compatible M&S resources. In this section, we discuss the selection process from those found matched resources. For example, the root *Scheduler #1* (Fig. 8) needs to select the best pool out of the matched pools. This is needed to keep the load balanced between M&S resource. To do so, root *Schedulers* need to know more information about each resource pool channel load. This information is collected via the Scheduler "Compute Load" operation.

The "Compute Load" operation is summarized as follows: the *Load Compute* message (Fig. 9) is periodically sent from root Schedulers to all children until it reaches leaf nodes. These are usually the ones with M&S capabilities and can run simulations. Once the compute message reaches the leaf Schedulers, they compute their local processing power and utilization and return this information back to the parents. We can consider many factors to compute the processing power and utilization: CPUs available, Memory, and Disk storage. In our case, we made it as a policy that can be configured by system administrators. However, by default we considered the number of logical processors and their utilization.

Consider the example in Fig. 9. As a precondition, suppose resource pools channels are already structured as previously discussed (in Fig. 8 in Section 4.2). Now, let us assume that leaf nodes in Fig. 9 are as follows: *Scheduler #4* (cpu = 2, utilization = 20%), *Scheduler #5* (cpu = 4, utilization = 25%), *Scheduler #6* (cpu = 8, utilization = 75%), and *Scheduler #7* (cpu = 8, utilization = 50%). Thus, when *Scheduler #5* gets the *compute* operation, it responds back to parent *Scheduler #2* {cpu = 4, utilization = 25%} as shown in Fig. 10.

However, because *Scheduler #2* merges its children resources in one pool channel, it needs to merge both of its children responses before responding back to its parent *Scheduler #1*. This is because both of children channels are mapped by *Scheduler #2* to one

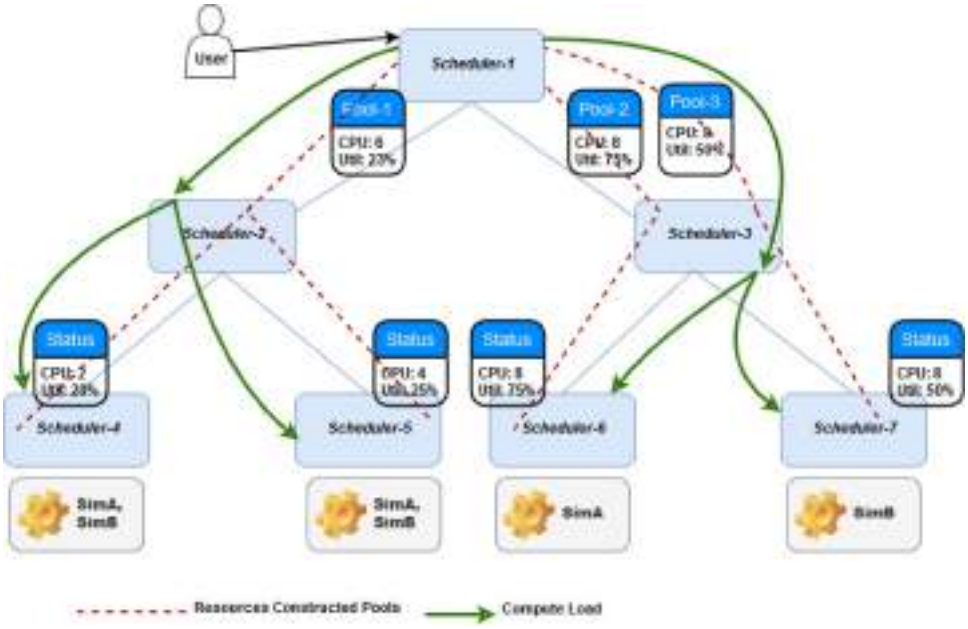


Fig. 9. M&S Resources Load Computations.

```
<resourcePools>
     <resourcePool>
          <uuid>4186c6e6-71ce-11e9-a923-1681be663d3e</uuid>
          <power><cpu>4</cpu></power>
         <utilization><cpu>0.25</cpu></utilization>
    </resourcePool>
 </resourcePools>
```

Fig. 10. XML Description for Resource Pool Channel Computation.

channel. The utilization of the merged channel is calculated as follows:

$$U = \left(1 - \left(\sum_{pool=1}^{n} cpu - u{*}cpu\right)/Tcpu\right){*}100\%$$

Where U is the utilization of all merged channels, *cpu* is the number of CPUs in a channel, *u* is a single channel utilization reported from a child. T*cpu* is the total of CPUs for all channels. Based on the above, *Scheduler #2* calculates the utilization for the merged channel as follows: $U = (1- ((2 – 0.2 * 2) + (4 - 0.25 * 4))/6) *100\% = 23\%$. It then responds back to parent *Scheduler #1* with information {cpu =6, utilization = 23%} with the XML message format in Fig. 10. Further, Schedulers calculate the processing power for all of resource pools and keep them sorted according to their processing power. Resource pool processing power is calculated as follows:

$$ProcessingPower = cpu - cpu{*}u$$

Where *cpu* is the CPU count of the resource pool, *u* is the resource pool current utilization.

For example, *Scheduler #1* in Fig. 9 has three pools with: (1st pool) {cpu: 6, utilization: 23%}, (2nd pool) {cpu: 8, utilization: 75%}, and (3rd pool) {cpu: 8, utilization: 50%}. Thus, their processing powers are: (1st pool = 4.62), (2nd pool = 2), and (3rd pool = 4). This means first pool is with the least load, then third pool, then second pool.

At this point root/Fog servers (like *Scheduler-1*) have all needed information to discover matched servers with least loads. To do so, the Fog server Scheduler (*Scheduler #1* in our example), invokes *Get Server* operation (see API in Table 2). As discussed previously, VEs (in Fog servers) need to find servers with the simulation environments that can execute the subject model. Thus, each resource pool processing power and utilization information always needs to be available for Fog servers (Fig. 9). This is done as follows.

When *Get Server* is invoked by the Fog/root Scheduler, it performs the following (1) Get set of the resource pools with the required M&S capabilities, (2) Select from this matched resource pools, the pool with least load (i.e. highest processing power), (3) Send *Get Server* message down the hierarchy through the child with least loaded resource pool. This will eventually reach the actual servers and return the required information. Once servers are found, the connection is directly established between those servers and the Fog server so that its subject experiment can use them to run simulation.

Therefore, the above three steps will be repeated downward the hierarchy until a leaf node is reached. This leaf node is the

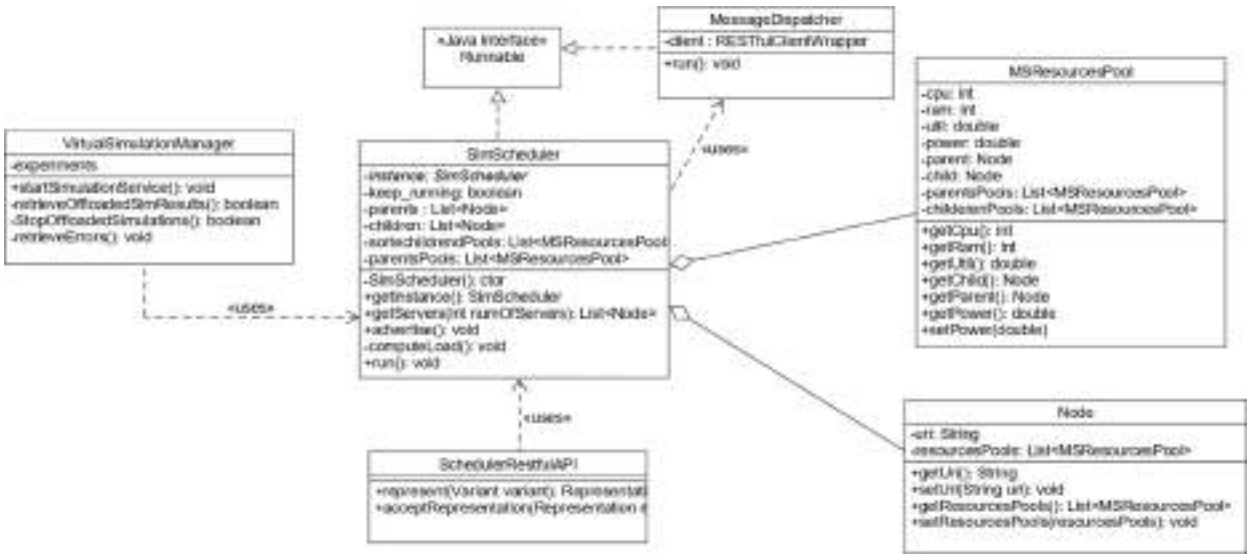


Fig. 11. Snippet of Simulation Scheduler Context Implementation.

selected server with the required capability. Therefore, it will respond with its information in XML format to its parent. The parent will then push this message to its parent until it reaches the Fog server (on the top), which then passes this information to the VE. Note that the leaf server might reject the request, if it is heavy loaded (in our case, utilized more than 90%). In this case, the Fog server may launch another VM with the suitable server image, or simply reject the simulation start request. This is based on system admins configuration. It is worth noting that load balancing is not our main objective; however, this selection stage is a necessary stage in the overall Fog/collaboration workflow cycle.

4.4. Implementation

Fig. 11 shows the simulation Scheduler context implementation, which is the component responsible for the above-discussed M&S resources management. This Scheduler component exists in every server. Those Schedulers exchange control messages via RESTful API to discover and collect information about those resources.

Upon HTTP request receipt with the *Scheduler* URI, the server allocates a thread for the incoming request and creates an instance of class *SchedulerRestfulAPI* (Fig. 11). Scheduler only supports the GET and POST HTTP methods. Thus, if it is a GET request, the HTTP framework invokes method *represent()*. On the other hand, if it is a POST request, method *acceptRepresentation()* is then invoked. Those two methods will then invoke the proper operation from class *SimScheduler* based on the API previously described in Table 2. Class *SimScheduler* is singleton where only one instance can only exist in a single middleware server. This class contains the Scheduler operations that we explained above throughout Section 4. Method *computeLoad()* is invoked periodically within a thread to go downward in the hierarchy to get resource pools current utilizations (see Fig. 9). Method *advertise()* is used to pass M&S resources description upward in the hierarchy, which causes resource pools channels to be constructed along the way (see Fig. 8). Method *getServers()* allows experiment to find a server (and optionally redundant servers) to run the simulation, which could be on the localhost or a remote server to offload simulation onto it (see Fig. 2). Class *SimScheduler* also keeps track of all of children resource pools and their corresponding parents' children pools. Class *MSResourcesPool* defines a resource pool while class Node defines a server (i.e. that is another Scheduler on that server). Class *MessageDispatcher* is used to dispatch HTTP messages (within separate threads) to other servers. Class *VirtualSimulationManager* manages active simulation within a VE, as previously discussed in Fig. 4.

5. Case study

This section demonstrates the proposed ideas using a real Fog and Cloud system with various physical and configuration setups. We will take the following points into consideration:

- Deploy different Fog/Cloud physical setups with homogeneous/heterogeneous hardware, various VMs computing capabilities, and various M&S capabilities.
- Deploy the Fog/Cloud with different deployment configurations. This shows how M&S resources discovered, organized, virtualized and separated. This kind of change reconstructs the discovered resources onto pools so that the Fog servers can find and use them, as previously discussed in Section 4.1.
- Ensure that experiments load will be spread over compatible servers. This also means that the servers that cannot execute experiments should not get those simulation jobs. We will change some servers M&S capabilities to show how the resources are

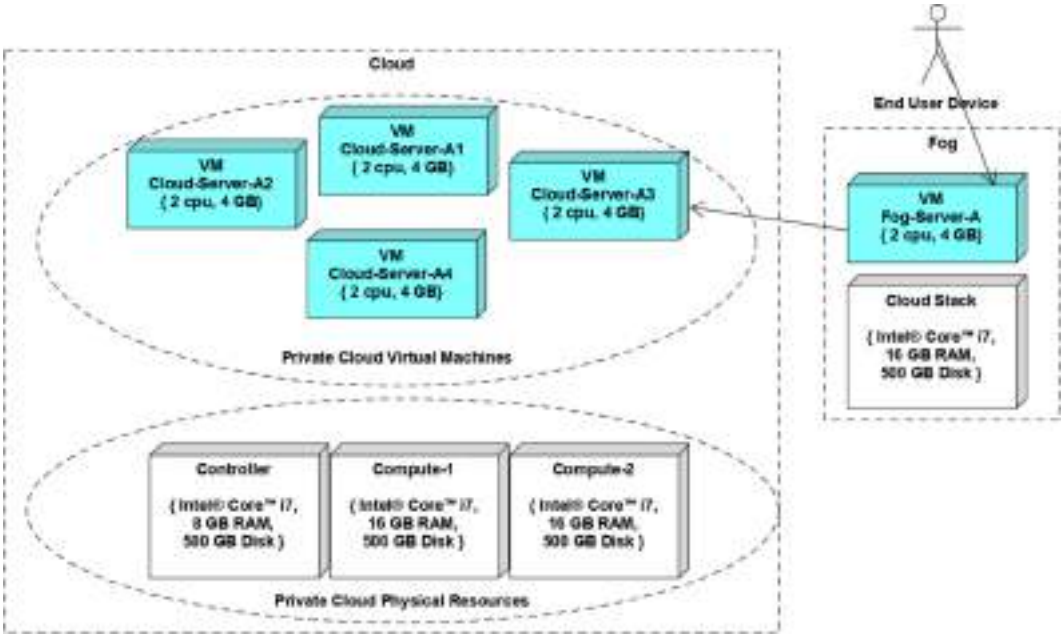


Fig. 12. *First* Physical Fog/Cloud Setup — Deployed VMs alongside their Hardware Configuration.

reconstructed by observing where the simulations were executed.

- Explore performance, particularly when users can run their simulation on their local Fog servers comparing to the Cloud.

We build three different Fog/Cloud physical setups, as shown in Figs. 12–14. In the *first physical setup* (shown in Fig. 12), we built two physical Clouds: one represents the Fog micro Cloud, built over one physical machine with one virtual machine (VM). The other represents the Cloud backbone, built over three physical machines with four allocated VMs called, *Cloud-Server-A1, Cloud-Server-A2, Cloud-Server-A3*, and *Cloud-Server-A4*. The *second physical setup* (shown in Fig. 13) is similar to the *first physical setup* (Fig. 12), but with different hardware capabilities. However, the *third physical setup* (shown in Fig. 14) is built with two Fogs and one Cloud, which built with more heterogeneous VMs and hardware capabilities, comparing to the first and second setups. Further, the user device is a laptop communicating via a typical household Wi-Fi with the Fog server. Finally, the Cloud resources are placed about 30 km from

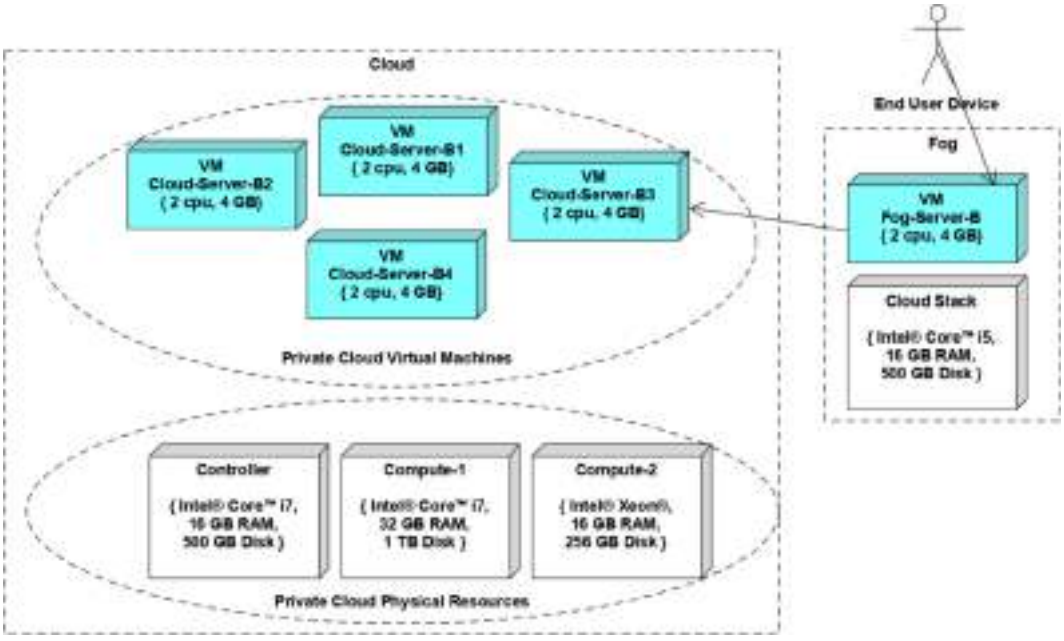


Fig. 13. *Second* Physical Fog/Cloud Setup — Deployed VMs alongside their Hardware Configuration.

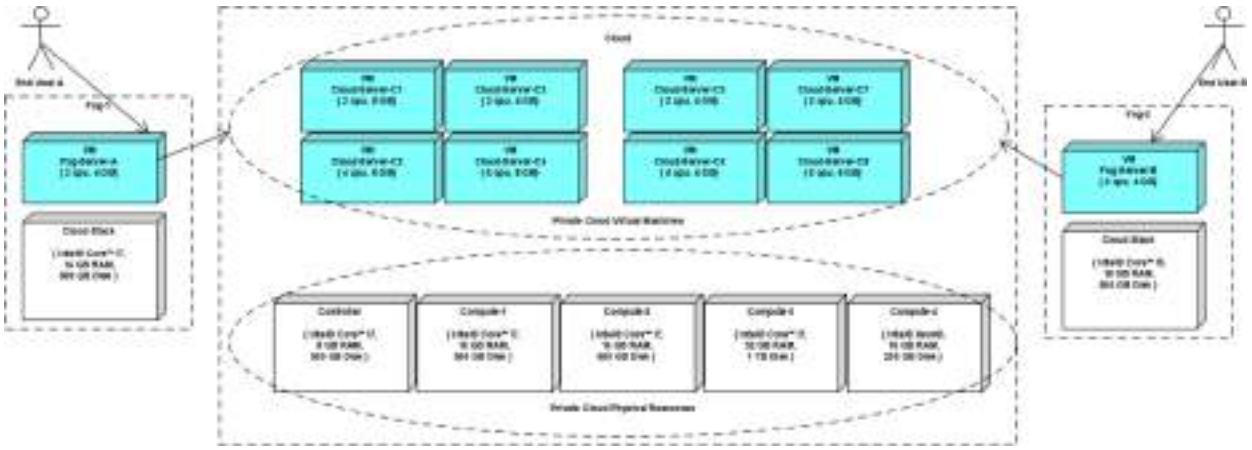


Fig. 14. *Third* Physical Fog/Cloud Setup — Heterogenous VMs and Hardware.

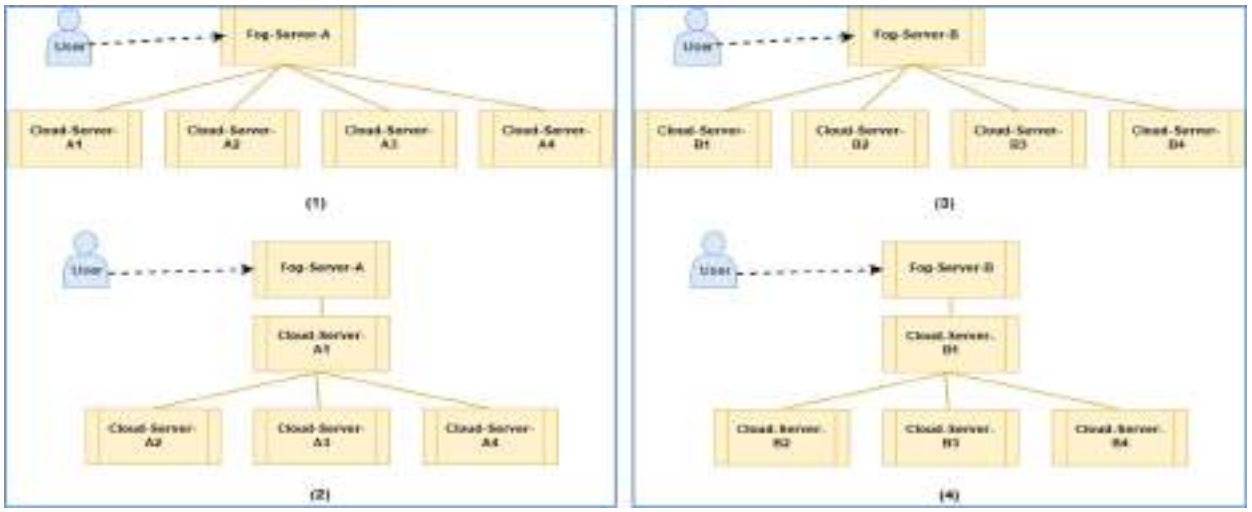


Fig. 15. Used Deployment Configuration for the First & Second Physical Setups in Fig. 12 & Fig. 13.

the Fog within the same city.

As discussed in Section 4.1, M&S resources are deployed in hierarchal structure. As a result, Figs. 15 and 16 show the used deployment configurations for the above three physical setups (shown in Figs. 12–14). Fig. 15-1 and 15-2 are the used deployment configurations for the *first physical setup* in Fig. 12. Fig. 15-1 shows that the Fog server *Fog-Server-A* as the parent node to all Cloud servers. The second configuration in Fig. 15-2 is more realistic, which uses *Cloud-Server-A1* as Cloud gateway for the Fog. Similarly, Fig. 15-3 and 15-4 provide the deployment configuration on top of the *second physical setup* (in Fig. 13). Thus, the only difference is the underlying physical hardware and computing power, but with similar deployment configuration. Fig. 16 shows three configurations over the *third physical setup* (in Fig. 14). This physical setup is different from the previous two setups for having two Fogs with heterogeneous VMs capabilities and heterogeneous computing hardware. Fig. 16-1 shows how M&S resources can be *split* between two tenants. Each tenant (e.g. organization) gets the impression of using all the capabilities by itself while sharing those resources with other tenants. Fig. 16-2 shows how to combine the two tenants' resources (in Fig. 16-1) together so that they become one tenant. This done by making *Fog-Server-B* and *Fog-Server-A* parents for *Server-C1* and *Cloud-Server-C5* respectively, as discussed in Section 4.1.

Table 3 presents different use cases to experiment with the proposed concepts, using different combinations of Fog/Cloud physical setups and deployment configurations. Those uses cases mainly focus on enabling the servers with different M&S capabilities combinations. In this way, we can run use cases over different deployments with different M&S capabilities. This would show how the system is able to manage resources discovery and selection.

It is worth noting that since our use cases focus on what happens after starting the simulation within experiments, the user created and prepared 60 equivalent experiments on each of the Fog servers with the complete settings. They all have same settings, use the same model, and being executed on CD++. This is needed to make the comparison between simulation runs as fair as possible. All these experiments used a forest fire model, which studies the fire propagation in forests. As previously stated, the client side is not in

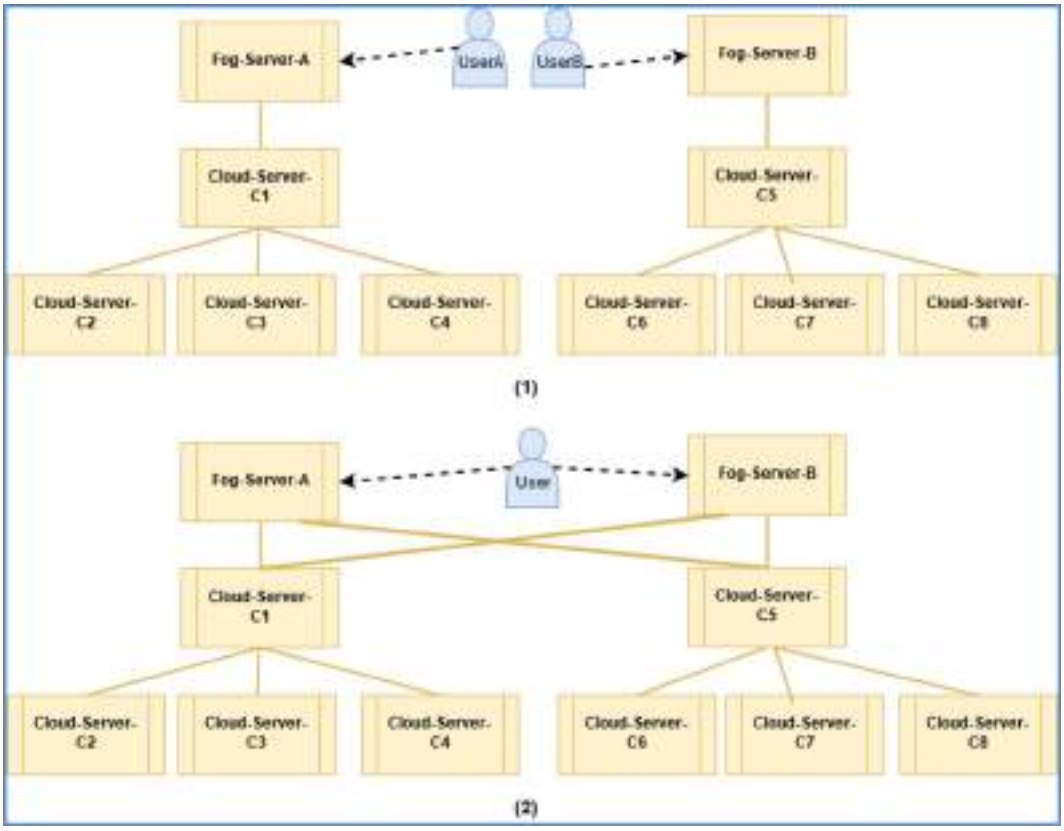


Fig. 16. Used Deployment Configurations for the Third Physical Setup in Fig. 14.

our focus in this presented work. However, client devices need to retrieve simulation results (e.g. XML, text, etc.) from experiments (via Fog servers) to be visualized locally. Fig. 17 shows an example of how clients would view fire model retrieved simulation results.

Fig. 18 shows a case where the simulation of 60 experiments started randomly over a period of ten seconds. Fig. 18-1 applies *Use Case #1* while Fig. 18-2 applies *Use Case #2* in Table 3. The only difference between these two use cases is the underlying hardware (i.e. Use Case #1 uses the physical setup in Fig. 12 while Use Case #2 uses the physical setup in Fig. 13). In this case, the Fog servers cannot run the simulation locally, since they did not advertise CD++. Further, the simulation for 60 experiments are expected to spread over the four Cloud servers. This is because all those servers can execute the CD++ simulation, as they advertised this capability earlier. This case was repeated over 100 rounds, though, each run, in Fig. 18, is an average of 10 actual runs. As results show in Fig. 18, the experiments simulations were reasonably spread over the four servers. Further, Fig. 18-1 and 18-2 showed similar results regardless of the difference of the underlying hardware. This is because VMs computing power and M&S advertised capabilities are the same, hence hiding the hardware heterogeneity. Furthermore, the variation on the number of experiments executed on servers is mainly because of the load status at the time of servers' selection. Further, servers load may not always be related to simulation, since they also need to handle other background processes. The main point of this situation is that the Fog servers' experiments were able to find and select the compatible servers. This means the resources pools were structured correctly and correctly made available to the Fog server.

The above discussed cases conditions in Fig. 18 are repeated in Fig. 19, but with different hierarchical configuration deployment and M&S capabilities. Fig. 19-1 applies *Use Case #3* while Fig. 19-2 applies *Use Case #4* in Table 3. In this new changed configuration, Cloud-*Server-A1* (Fig. 15-2) and Cloud-*Server-B1* (Fig. 15-4) are now made intermediate Schedulers without any M&S capabilities (i.e. cannot run simulation). Consequently, new resource pools will be constructed. In this configuration, the experiments simulations are expected to be spread over the three cloud servers (i.e. that means no simulation will be executed on the Fog and intermediate servers). As the results show in Fig. 19, the experiments simulations were executed over the expected servers. This case findings, in Fig. 19, support the previous case findings in Fig. 18. This is because they are same, but with different deployment and M&S capabilities configuration. Further, Fig. 19-1 and 19-2 showed similar results regardless of the difference of the underlying hardware, hence above layers hid this hardware heterogeneity. Furthermore, this case results show that the overall Fog/Cloud architecture reorganized itself dynamically based on the configuration change. This is important to prove because in practice configuration changes dynamically by system administrators.

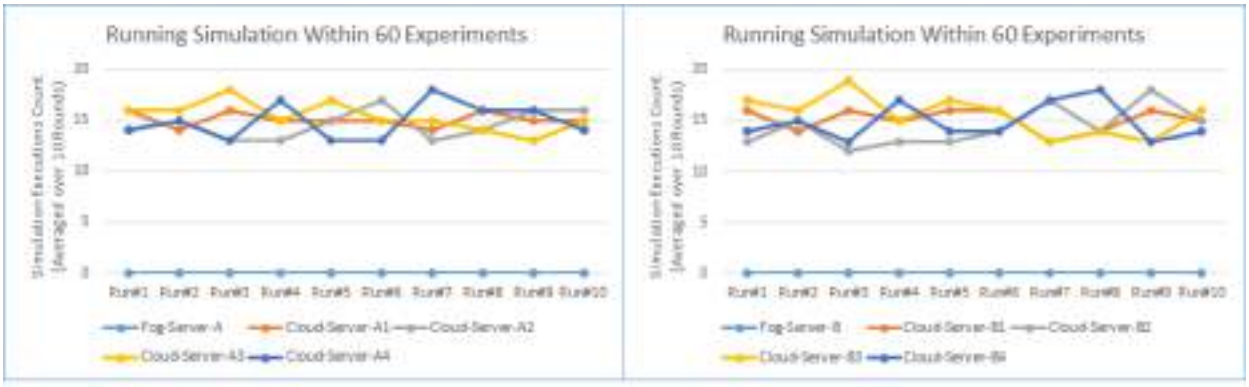
To further explore the overall system reaction to dynamic changes, the case in Fig. 20 explores the case when servers advertise heterogenous M&S capabilities. This means experiments need to find the servers that able to execute their simulations. To do so, since

Table 3
Different M&S Capabilities over Different Deployment Configuration.

Use Case #	Servers (VMs) M&S Capabilities		Physical Setup	Deployment Configuration	Results
1	Fog-Server-A	None	First Setup (Fig. 12)	Fig. 15-1	Fig. 18-1
	Cloud-Server-A1, Cloud-Server-A2, Cloud-Server-A3, Cloud-Server-A4	CD++			
2	Fog-Server-B	None	Second Setup (Fig. 13)	Fig. 15-3	Fig. 18-2
	Cloud-Server-B1, Cloud-Server-B2, Cloud-Server-B3, Cloud-Server-B4	CD++			
3	Fog-Server-A	None	First Setup (Fig. 12)	Fig. 15-2	Fig. 19-1
	Cloud-Server-A1	None (i.e. used as gateway to M&S cloud resources)			
	Cloud-Server-A2, Cloud-Server-A3, Cloud-Server-A4	CD++			
4	Fog-Server-B	None	Second Setup (Fig. 13)	Fig. 15-4	Fig. 19-2
	Cloud-Server-B1	None (i.e. used as gateway to M&S cloud resources)			
	Cloud-Server-B2, Cloud-Server-B3, Cloud-Server-B4	CD++			
5	Fog-Server-A	None	First Setup (Fig. 12)	Fig. 15-2	Fig. 20-1
	Cloud-Server-A1	None			
	Cloud-Server-A2, Cloud-Server-A3	CD++			
	Cloud-Server-A4	Dummy (i.e. no CD++ simulation can run on it)			
6	Fog-Server-B	None	Second Setup (Fig. 13)	Fig. 15-4	Fig. 20-2
	Cloud-Server-B1	None			
	Cloud-Server-B2, Cloud-Server-B3	CD++			
	Cloud-Server-B4	Dummy (i.e. no CD++ simulation can run on it)			
7	Fog-Server-A, Fog-Server-B	None	Third Setup (Fig. 14)	Fig. 16-1, & Fig. 16-2	
	Cloud-Server-C1, Cloud-Server-C5	None (i.e. used as gateway to M&S cloud resources)			
	Cloud-Server-C2, Cloud-Server-C3, Cloud-Server-C4, Cloud-Server-C6, Cloud-Server-C7, Cloud-Server-C8	CD++			

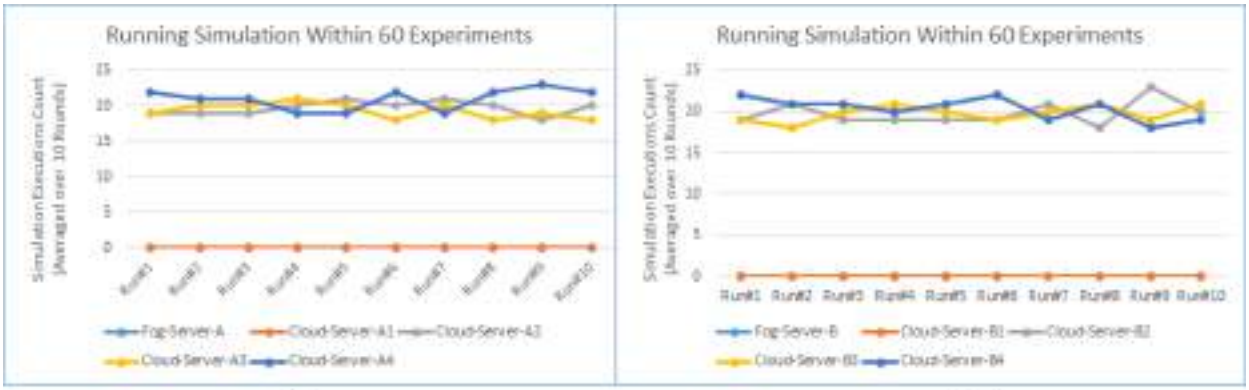
Fig. 17. Example of Client-Side Simulation Snapshots for Fire Propagation on top of Google Map.

our experiments require CD++ to run their simulation, we made *Cloud-Server-A4* (*Use Case #5* in Table 3) and *Cloud-Server-B4* (*Use Case #6* in Table 3) advertise dummy M&S capability (to be different from CD++). This means that no CD++ models can be executed on these two servers. On the other hand, simulation should be able to execute on the other servers that had advertised CD++ capability. As expected, Fig. 20-1 (i.e. that applies *Use Case #5* in Table 3) shows how experiments were spread over *Cloud-Server-A2* and *Cloud-Server-A3*. However, no experiments have run on *Cloud-Server-A4*, since it does not match experiments requirements (to execute on CD++). This case findings support the same findings of the previous cases in a sense of spreading experiments simulation



(1) (2)

Fig. 18. Running 60 Experiments Over Four Compatible Servers (Use Case #1 & #2 in Table 3).



(1) (2)

Fig. 19. Running 60 Experiments Over Three Compatible Servers (Use Case #3 & #4 in Table 3).

(1) (2)

Fig. 20. Running 60 Experiments Using Heterogeneous M&S Resources (Use Case #5 & #6 in Table 3).

over the expected servers with CD++ capabilities. Fig. 20-2, which applies *Use Case #6* in Table 3, also shows similar findings as Fig. 20-1 since both use the same software setups but with different underlying hardware. In addition, the results (in Fig. 20) show that M&S resources were dynamically and correctly restructured to allow the Fog server to find the compatible servers. This also shows that *Cloud-Server-A4* and *Cloud-Server-B4* were excluded from the selection process, hence no simulation was executed over there. This is important in a sense that incompatible servers are not even considered in the servers' selection process.

Use Case #7 in Table 3 presents more realistic setups comparing to the above discussed cases thus far. The physical setup (Fig. 14) is built from two Fogs and a Cloud that is constructed from VMs with mixed computing capabilities on top of heterogeneous

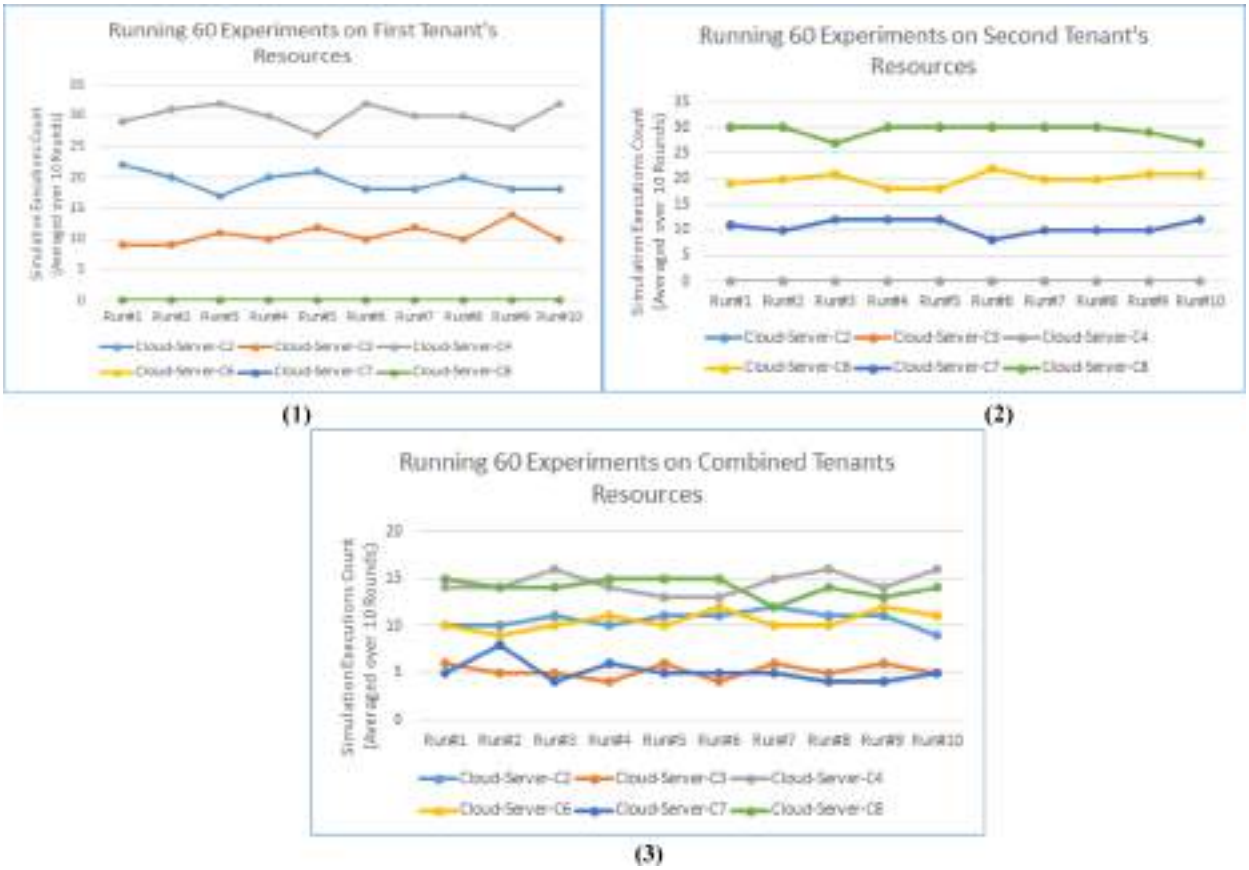


Fig. 21. Tenants M&S Resources Separation and Merging (Use Case #7 in Table 3).

hardware. The *first* real life scenario to discuss is the case of separating tenants (e.g. organizations) resources from each other while they still share the underlying clouds capabilities. To do so, Fig. 16-1 deployment configuration split M&S resources over two tenants. The first tenant uses the deployment in Fig. 16-1(left) and access the resource via *Fog-A*, while the second tenant uses Fig. 16-1(right) and access resources via *Fog-B*. Based on this, *Fog-A* will only see the first tenant resources in Fig. 16-1(left), hence can only execute simulation over those resources. Similarly, *Fog-B* can only access resources in Fig. 16-1(right). This is shown in Fig. 21.

Fig. 21-1 shows the simulation execution of experiments that were sent via *Fog-A*; hence they will only be executed over the first tenant servers (i.e. *Cloud-Server-C2, Cloud-Server-C3*, and *Cloud-Server-C4*). As also can be seen (in Fig. 21-1) that the second tenant's servers are not being used since they are not owned by the first tenant. In contrast, Fig. 21-2 shows that simulation is executed over the second tenant servers (i.e. *Cloud-Server-C6, Cloud-Server-C7, Cloud-Server-C8*) since the requests came via *Fog-B*. Yet again, in this case the first tenant's servers are not being used (in Fig. 21-2) since they are not owned by the second tenant. On the other hand, when combine both tenants' resources in a single combined tenant (as explained earlier in Fig. 16-2), all of the resources will then be used regardless of their original access (from *Fog-A* or *Fog-B*), as shown in Fig. 21-3. Further, the results show that some servers have been put to work more than others. This is because the servers computing capabilities are different from each other, hence the work is almost proportionate to their computing capabilities. For example, in Fig. 21-2 *Cloud-Server-C8* executed the simulation of around 50% (i.e. 30 out of 60) of all experiments comparing to the *Cloud-Server-C7* that executed around 17% (10 out of 60) of all experiments. This argument applies to all results in Fig. 21. The variation between different runs of same servers are usually related to servers' background load (that may not be related to simulation) at the time of their selection, as discussed previously.

Thus far we used fog servers as only access points to the cloud resources. However, fog servers can also advertise M&S resources, hence can run simulation locally (like any other servers) without even going to the cloud. Fig. 22 compares between a simulation running on the fog and on the cloud in terms of the user response time, which is measured by the Round-Trip-Time (RTT). RTT is the time it takes a client device to retrieve results (or read values) from an active simulation (within an experiment). To do so, we repeated *Use Case #1* in Table 3, but while enabling the fog server to advertise the CD++. This means the fog server in the physical setup in Fig. 12 can now run CD++ simulation as well. We then ran two simulations in parallel one on the fog server and the other on the cloud so that we can compare between their response times (with respect to the client device). The message was to make the client device to send a simple message to ongoing simulation to read the current simulation time. The sending of this message was repeated 100 times. Each Round Trip Time (RTT) shown on the figure is an average of ten times of actual sending. The results are shown in Fig. 22. According to our system settings and conditions, the cloud response time to users was about two to four times larger

Fig. 22. Fog/Cloud User Response Time Comparison to the Client Device.

than the Fog response (Fig. 22). The difference is mainly due to networking latency since both of those simulations are equal in terms of computing power and experiment settings. However, in practice, we believe users latency is expected to be much better than our findings here. This is because the cloud server was not running under heavy load (i.e. it was running single simulation like the Fog server). Further, cloud resources may not be sitting close to each other within the same city as in our case. Therefore, many factors can play a role in affecting users response time.

6. Conclusions

We proposed complete Fog/Cloud collaboration architecture/mechanisms to conduct M&S experiments. Complete in a sense of having all required steps to allow Fog/Cloud resources to cooperate with each other to conduct simulation experiments. The proposed experiment is decoupled from M&S resources specifics, hence called Virtual Experiments (VEs). This is because in practice Cloud resources may advertise different M&S resources. Therefore, a VE can virtually simulate any model, as soon as it can find a simulation environment that can execute that model. VEs decoupled experiments framework from M&S specifics. Thus, VEs took virtualization to the level of the M&S experiment itself. This can be powerful concept toward enhancing M&S within the Fog/Cloud computing technology where resources heterogeneity is a fact rather than an assumption. Further, our VE proposal is still in compliance with the Cisco Fog Computing architecture. In this architecture, end users enter deployed services via Fog servers, hence users create their experiments on the Fog servers. This means Fog servers should have global knowledge about the deployed M&S capabilities. This knowledge should enable VEs (on Fog servers) to find M&S environments that can simulate the model. Further, this knowledge should enable VEs to select the best servers (with those equipped capabilities) to run the simulation. To enable VEs with these requirements, we further proposed full set of M&S resources management algorithms. This allowed M&S resources to be discovered and organized dynamically in form of resource pools, since in practice events happens dynamically. We then allowed computing status information to be collected about those discovered resources. This allows VEs to select best discovered resources servers. To prove presented concepts, we have built our private Fog and Clouds to put the presented ideas into demonstration. We further presented several cases with different physical setups, deployment configurations and M&S capabilities.

References

[1] M. Abdelhafidh, M. Fourati, L. Fourati, A. Mnaouer, Z. Mokhtar, Cognitive internet of things for smart water pipeline monitoring system, Proceedings of the IEEE/ACM 22nd International Symposium On Distributed Simulation and Real Time Applications (DS-RT), (2018).
[2] L. Almeida, E. Almeida, J. Murphy, R. Grande, A. Ventresque, BigDataNetSim: a simulator for data and process placement in large big data platforms, The Proceedings of the IEEE/ACM 22nd International Symposium On Distributed Simulation and Real Time Applications (DS-RT), (2018).
[3] K. Al-Zoubi, G. Wainer, Distributed simulation of DEVS and Cell-DEVS models using the rise middleware, Simulation Model. Pract. Theory Elsevier 55 (June 2015) 27–45.
[4] K. Al-Zoubi, G. Wainer, RISE: a general simulation interoperability middleware container, J. Parallel Distrib. Comput. Elsevier. 73 (5) (May 2013) 580–594.
[5] B. Assila, Abdellatif Kobbane, Mohammed El Koutbi, A many-to-one matching game approach to achieve low-latency exploiting fogs and caching", The Proceedings of the IEEE of the 9th IFIP International Conference On New Technologies, Mobility and Security (NTMS), (2018).
[6] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, The Proceedings of the First MCC Workshop Mobile Cloud Comput, (2012), pp. 13–16.
[7] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, D.;. Owen, The SIMNET virtual world architecture, Proceedings of Virtual Reality Annual International Symposium (IEEE VRAIS 1993). Los Alamitos, CA, USA, (1993).
[8] C. Castillo, G.N. Rouskas, K. Harfoush, Efficient implementation of best-fit scheduling for advance reservations QoS in grid, Proc. of the 1st IEEE/IFIP Intl. Workshop on End-to-end Virtualization and Grid Management (EVGM), (2007) October.

[9] Y.K. Cho, B.P. Zeigler, H.S.;. Sarjoughian, Design and implementation of distributed real-time DEVS/CORBA, Proceedings of IEEE International Conference on Systems, Man and Cybernetics (ICSMC2001). Tucson, AZ, USA, (2001).
[10] H. Dubey, J. Yang, N. Constant, A.M. Amiri, Q. Yang, K. Makodiya, Fog data: enhancing telehealth big data through fog computing, the ACM Proceedings of the ASE Big Data & Social-Informatics, (2015).
[11] H. Gupta, A. Dastjerdi, S. Ghosh, R. Buyya, iFogSim: a toolkit for modeling and simulation of resource management techniques in internet of things, Edge and Fog Computing Environments, Softw. Pract. Exper. (SPE) 47 (9) (2017) 1275–1296.
[12] S. Hafiz, H.X. Tan, A. Ishfaq, R. Sanjay, "Energy and performance-aware scheduling of tasks on parallel and distributed systems, ACM J. Emerg. Technol. Comput. Syst. Vol.4 (No.8) (2013) 1–37.
[13] IEEE Std 1278.1a-1998 - "IEEE Standard for Distributed Interactive Simulation—Application Protocols". https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=729408 . Accessed November 2019.
[14] Technical Report 1516, (2000) IEEE.
[15] W. Kolberg, P.D.B. Marcos, J.C.S. Anjos, A.K.S. Miyazaki, C.R. Geyer, L.B. Arantes, Mrsg - a MapReduce simulator over simgrid, Parallel Comput. 39 (4–5) (2013) 233–244.
[16] X. Koutsoukos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, J. Sztipanovits, SURE: a modeling and simulation integration platform for evaluation of secure and resilient cyber–physical systems, The Proceedings of the IEEE 106.1, (2018), pp. 93–112.
[17] X. Li, P. Garraghan, X. Jiang, Z. Wu, J. Xu, "Holistic virtual machine scheduling in cloud datacenters towards minimizing total energy, IEEE Trans. Parallel Distrib. Syst. Vol.29 (6) (2018) 1317–1331.
[18] R. Naha, S. Garg, D. Georgakopoulos, P. Jayaraman, L. Gao, Y. Xiang, R. Ranjan, Fog computing: survey of trends, architectures, requirements, and research directions, IEEE Access. 6 (2018) 47980–48009.
[19] B. Silverman, M. Solberg, OpenStack for Architects: Design production-ready private cloudinfrastructure, 2nd edition, Michael Solberg, Birmingham, UK, 2018.
[20] D.S. Hu, H. Ning, T. Qiu, Survey on fog computing: architecture key technologies applications and open issues, J. Netw. Comput. Appl 98 (Nov. 2017) 27–42.
[21] F. Hao, G. Pang, Z. Pei, K.Y. Qin, Y. Zhang, X. Wang, Virtual machines scheduling in mobile edge computing: a formal concept analysis approach, Proceedings of the IEEE Transactions on Sustainable Computing, (2019).
[22] K. Rehman, O. Kipouridis, S. Karnouskos, O. Frendo, H. Dickel, J. Lipps, N. Verzano, A cloud-based development environment using HLA and Kubernetes for the co-simulation of a corporate electric vehicle fleet, The proceedings of the IEEE/SICE International Symposium on System Integration (SII), 2019.
[23] S. Shekhar, H. Abdel-Aziz, M. Walker, F. Caglar, A. Gokhale, X. Koutsoukos, A simulation as a service cloud middleware, Ann. Telecommun. 71 (3) (2016) 93–108.
[24] C. Sonmez, A. Ozgovde, C. Ersoy, EdgeCloudSim: an environment for performance evaluation of edge computing systems, The Proceedings of the IEEE Second International Conference on Fog and Mobile Edge Computing (FMEC), (2017).
[25] H. Casanova, A. Giersch, A. Legrand, M. Quinson, F. Suter, Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms, J. Parallel Distrib. Comput. Elsevier 74 (10) (2014) 2899–2917.
[26] SimpleSoft SimpleIoTSimulator. Online: https://www.smplsft.com/SimpleIoTSimulator.html. Accessed June 2019.
[27] V. Stantchev, A. Barnawi, S. Ghulam, J. Schubert, G. Tamm, Smart items, fog and cloud computing as enablers of servitization in in healthcare, Sens. Transd. 185 (2) (2015) 121.
[28] G. Wainer, Discrete-event Modeling and Simulation: A Practitioner's Approach, CRC/Taylor & Francis, UK, 2009.
[29] G. Wainer, R. Madhoun, K. Al-Zoubi, Distributed simulation of DEVS and Cell-DEVS models in CD++ using web services, Simul. Modell. Pract. Theory 16 (9) (October 2008) 1266–1292.
[30] Y. Wang, R. Zhiyuan, Z. Hailin, H. Xiangwang, X. Yao, "Combat cloud-fog" network architecture for internet of battlefield things and load balancing technology, Proceedings of the IEEE International Conference on Smart Internet of Things (SmartIoT), (2018).
[31] Wei Xiong, Wei-Tek Tsai, "HLA-based saas-oriented simulation frameworks, The Proceedings of the IEEE 8th International Symposium on Service Oriented System Engineering, (2014).
[32] S. Yi, C. Li, Q. Li, A survey of fog computing: concepts applications and issues, Proc. Workshop Mobile Big Data (2015) 37–42.
[33] M. Zamfir, V. Florian, A. Stanciu, G. Neagu, S.,. Preda, G. Militaru, Towards a platform for prototyping IoT health monitoring services, Proceedings of the 2016 Springer International Conference on Exploring Services Science. Springer, (2016), pp. 522–533.
[34] B. Zhang, W. Li, B.B. Zhou, A.Y. Zomaya, Home fog server: taking back control from the cloud, Proceedings of the IEEE Globecom Workshops, (2016).