DEVSML 3.0 STACK: RAPID DEPLOYMENT OF DEVS FARM IN DISTRIBUTED CLOUD ENVIRONMENT USING MICROSERVICES AND CONTAINERS

Saurabh Mittal
The MITRE Corporation
7515 Colshire Dr.
McLean, VA, USA
smittal@mitre.org

José L. Risco-Martín
Complutense University of Madrid
C/Prof. José García Santesmases, 9
28040 Madrid, Spain
jlrisco@ucm.es

ABSTRACT

Cloud infrastructure provides rapid resource provision for on-demand computational requirements. Cloud simulation environments are largely client-server architectures with multiple slave nodes solving problems through Monte Carlo methods. However, to implement distributed simulation environment in cloud infrastructure, sufficient automation in the modeling and simulation (M&S) architecture is needed. DEVS/SOA provides an established mechanism to develop distributed Discrete EVent Systems (DEVS) environments using Service Oriented Architectures (SOA), the technology behind Cloud infrastructure. This paper provides a methodology using Microservices and containerization paradigm to rapidly deploy a DEVS Simulation Farm. This work builds on the earlier developed DEVS netcentric Virtual Machine (DEVSVM) within the DEVS Modeling Language (DEVSML) Stack. We propose a new version of the DEVSML Stack, Ver. 3.0 and advance the state-of-the-art in modeling and simulation interoperability and automation. We illustrate the technologies with the help of an example.

Keywords: DEVS/SOA, Microservices, Docker, DEVS/Cloud, DEVS Farm.

1 INTRODUCTION

Cloud infrastructure provide rapid resource provision for on-demand computational requirements. Cloud simulation environments are largely client-server architectures with multiple slave nodes solving problems through Monte Carlo methods. A cloud simulation is not the same as a distributed simulation. A cloud implementation of an M&S application does not ensure that the infrastructure will scale and complexity inherent in distribution simulation is addressed. In addition, implementing distributed M&S services in cloud infrastructure is a non-trivial problem (Cayirci 2013). Having a cloud-based deployment does not guarantee that a distributed simulation infrastructure is a "given" . Both have different architectures. However, cloud computing brings on-demand resources and technologies like virtualization and containerization. Incorporating cloud computing advantages for distributed M&S infrastructure then is a logical thing to get the best of both methods and capitalize on the Moore's law with minimal increase in physical hardware costs.

In this article, we will describe a methodology to deploy a formal discrete event dynamic system simulation infrastructure based on DEVS formalism, known as DEVS/SOA (Mittal, Martin, and Zeigler 2007, Mittal, Martín, and Zeigler 2009) in a distributed cloud environment. A scalable M&S architecture has distinct modeling and simulation layers. In order to deploy in cloud environment, sufficient automation is needed at both the simulation layer and the modeling layer. This can now be achieved by current practices in

SpringSim-TMS/DEVS 2017, April 23-26, Virginia Beach, VA, USA

DevOps implemented using Docker technology. DevOps, a recent buzzword, provides methodologies to automate developer operations, such as compiling, building, releasing, testing through executable scripts. We will integrate Docker with the granular service oriented architecture (SOA) Microservices paradigm and advance the state-of-the-art in model and simulation interoperability using the DEVS Modeling Language (DEVSML) Stack. This automated deployment of various "DEVS nodes" under a single administrative control is defined as a DEVS Farm. We illustrate by an example how such a DEVS Farm can be readily deployed and put to use for distributed simulation.

We make the following contributions in the paper:

1. Incorporate DevOps methodologies using microservices and containerization technologies to develop a cloud-based distributed simulation farm for discrete event dynamic systems specified using DEVS formalism
2. Provide the architecture and advance the state-of-the-art with DEVSML Stack Version 3.0

Section 2 provides an overview of the Microservices, containerization using Docker and symmetrical services architecture as key enablers. We then introduce DEVSML 3.0 Stack in Section 3. We described the new technology based on Microservices (Section 4) and Docker containers(Section 5), followed by the deployment of the classic EFP example in a DEVS Farm in Section 6. Finally, Section 7 presents discussion and conclusions.

2 FOUNDATIONAL TECHNOLOGIES

2.1 Microservices Paradigm

A microservice is a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communication protocol and a set of well-defined Application Programming Interfaces (API) and events, independent of any vendor, product or technology (Karmel, Chandramouli, and Iorga 2016). Microservices architecture is an architectural pattern. The operational architecture has the following components (Larsson 2015, Richardson 2015): Central Configuration server, Authentication and authorization server, monitoring, logging and audit servers, event stores, Edge servers, Load Balancer, Service Discovery servers and many more. These components can be best understood by the Reference Model depicted in Figure 1, adapted from (Larsson 2015). As Figure 1 shows, C-1 and C-2 represent customers. API-1,-2,-3 represent protected published APIs for core and composite microservices. For example, API-1 is fulfilled by microservice MS-3 and MS-1. Microservice MS-3 supports API-1 and API-2. These services communicate with each other through event-driven mechanisms primarily by asynchronous means through dumb message buses. The red arrows in the diagram depict the directed flow of asynchronous messages. Any microservices architecture has to address two fundamental issues: distributed data management, that allow storing a microservice state in local databases and shared event stores, that facilitate information exchange between stateless microservices. To execute the business logic inherent in a microservice, information from local databases is used in conjunction with the event processing inside the microservice.

2.2 Containerization paradigm

A container is a self-contained runtime environment for a software application. To illustrate this paradigm, we looked for open-source containerization solutions, such as Docker. A Docker container wraps a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools,

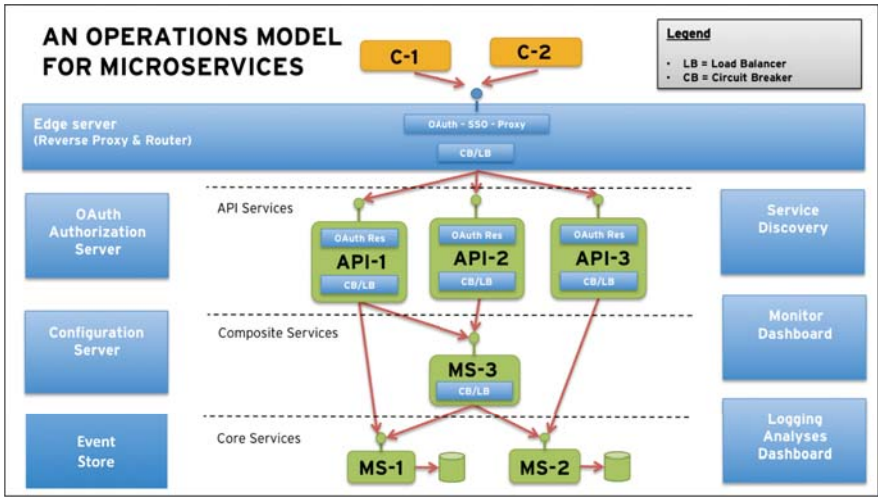

Figure 1: Microservices Reference Model (adapted from Larsson 2015).

system libraries - anything that can be installed on a server, guaranteeing the the software will always run the same, regardless of its hosting environment (Docker 2016). Some of the features of these containers are: (1) lightweight (2) share the same OS kernel (3) share common files (4) shared layering (5) based on open standards (6) run on all major linux distributions (7) run on Microsoft Windows (8) run on top of any infrastructure (9) isolate applications from one another and from the underlying infrastructure (10) composable, and not the last, (11) predictable.

2.2.1 Docker architecture

Typically, computing resources were provided using virtualization, at least until the arrival of Cloud Computing (Joy 2015). The architectural approach of Docker is different from that of virtual machines. Portability is one of the Docker's strong points. Figure 2.a shows how a virtual machine stores the whole operating system (OS), libraries, binaries and applications needed. Thus, a huge memory space is required in the host machine. In contrast, Docker containers can be seen as more flexible tools for packaging, delivery and deployment of software and applications. Figure 2.b shows how a Docker container is composed by libraries, required binaries and applications. But, unlike the virtual machine, all the containers share the same OS kernel. This makes the containers fairly light-weight. In a standard personal computer, while only a few virtual machines can be allocated, more than 100 Docker containers can be launched with ease.

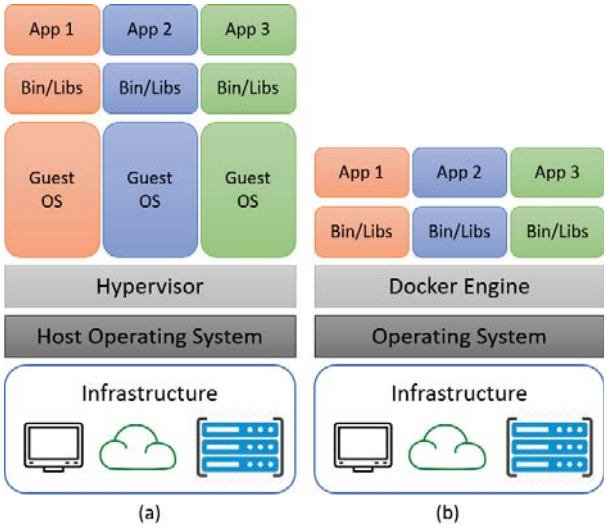

Figure 2: (a) Virtual machine and (b) Docker container architectures (Docker 2016).

Docker follows a client-server architecture. Client communicates with the Docker daemon (which is the server side), and is in charge of building a container, the container's operations and the distribution of containers. Client and daemon can be executed in the same system or in a distributed infrastructure. In this case, communication between client and daemon is performed through a REST API.

Figure 3 shows the Docker architecture. There are three important resources in this architecture: images, registery and containers. Docker images are read-only templates. They are the starting point to create containers. From a Docker image we can create thousands of containers, each one completely isolated and with all the required infrastructure to run our applications. Docker offers an easy mechanism to build new images, update existing ones or download images created by other users.

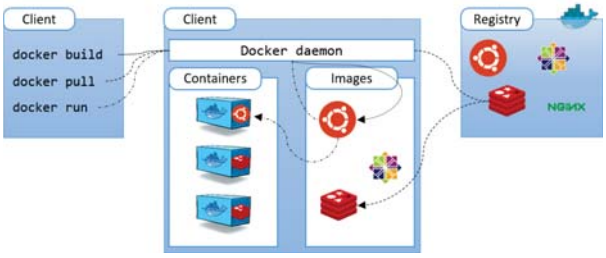

Figure 3: Docker architecture (Docker 2016) .

As Docker images share the OS kernel with the host machine, any Docker image must be based on a compatible host OS. The Docker environment is made available as a daemon after installation. The Docker daemon acts as the central server, which works with both images and containers. The client communicates with the daemon sending execution commands. Docker ecosystem offers the Docker Hub service, which is the official component to distribute registered Docker images. An image can be public or private. If public, any user can access these images. Docker images are collected through official repositories.

2.3 DEVS/SOA Symmetrical Services Architecture

A Web Service framework is essentially a client-server framework wherein a Server provides the requested services to clients. These services are nothing but computational code that is executed at the server's end with a valid return value. The mode of communication between the client and the server must employ standard transport protocols in a netcentric environment. This standardized mode of communication provides interoperability between various services as the data is machine-readable.

The distributed DEVS simulation protocol has two types of components i.e. *coordinator* and *simulator*, that corresponds to a coupled model and an atomic model, respectively. These components need to be deployed at remote nodes in the same session, so that a distributed execution can take place. This web-based DEVS distributed framework was defined in DEVS/SOA architecture (Mittal, Martín, and Zeigler 2009) and later utilized in the netcentric DEVS Virtual Machine (Mittal and Martín 2013).

The DEVS simulation components can be placed anywhere on the network. It is unavoidable that the same Server can act as a provider and a consumer while executing DEVS simulation protocol. Consequently, the SOA that executes the DEVS simulation protocol is constructed such that the servers that provide DEVS Service can play the role of both the coordinator and the simulator.

During the execution of DEVS simulation protocol, each of the simulators make calls to other simulators using SOA. These simulators also communicate with the coordinator using the same transport mechanism. As a result, the same simulator is invoking services from other simulators while providing services to other simulators or coordinator. This results in an architecture that is symmetrical by default i.e. it acts as both a service provider and a service consumer. The temporal role of a remote node is guided by the DEVS simulation protocol.

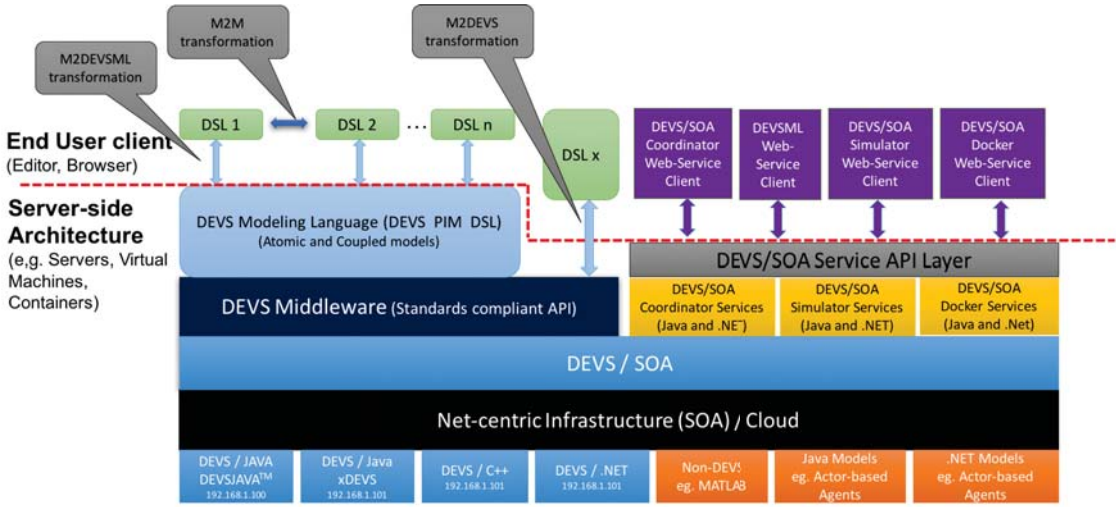

Figure 4: DEVSML 3.0 Stack for DEVS/SOA with Docker containerization.

3 DEVSML 3.0 STACK

The DEVSML Stack Ver. 1.0 (Mittal, Martin, and Zeigler 2007, Mittal, Martín, and Zeigler 2009) and Ver. 2.0 (Mittal and Douglass 2012, Mittal and Martin 2013) incorporated the symmetrical DEVS/SOA architecture i.e. all the services (e.g. coordinator and simulator services) are made available as DEVS nodes. It is used as a foundation to achieve model interoperability when models developed in multiple domain specific languages (DSLs) can be brought together in a DEVS/SOA environment where they subscribe to DEVS simulation protocol and platform neutral message formats in XML or JSON. This is a fundamental concept behind microservices that provides scalability as the "executing code" has now become independent of the infrastructure and the deployment can now be automated in a simplified way using DevOps. Per the Docker parlance, our DEVS/SOA "image" is inspired by this symmetrical services architecture.

We now introduce the next iteration of DEVSML Stack, Version 3.0 (Figure 4). Starting at the bottom (Figure 4), the execution layer of DEVS/SOA is built upon DEVS simulators in native languages (e.g. C++, .NET, Java, etc.) that may get deployed as individual containers. The next layer is the SOA communication layer, where the DEVS simulation protocol is implemented as web-services. Above that is the DEVS/SOA layer that implements Coordinator and Simulator web services to perform distributed simulations along with the required local databases for microservice implementation (as we shall see ahead in Section 4). This layer also incorporates the Docker scripts that build containers. We extend the state-of-the-art on DEVSML Stack with incorporation of Docker-based automation services that work in concert with simulation and coordinator services. Next is the DEVS middleware and the DEVS/SOA service Application Programming Interface (API) layer that makes available the DEVS modeling and DEVS simulation and Docker services for multifarious clients. It is this Service-API/Middleware layer that enables the transparent M&S framework. Finally, at the top, we have the DSLs and various Service clients that utilize the DEVS M&S services. To achieve model interoperability, DEVS models and various web-service clients can be encoded in any given language that conforms to DEVS Modeling and Service-API. Otherwise, DEVS wrappers can wrap the component's behavior as a DEVS model (Mittal et al. 2015). The coupled models are then specified using a platform neutral format, e.g. XML/JSON.

In the following sections, we show how the new additions in DEVSML 3.0 stack are implemented using Docker and microservices technologies.

4 DEVS/SOA MICROSERVICES ARCHITECTURE

Our proposed methodology is to create a DEVS/SOA image, following the scheme of the DEVS Virtual Machine proposed in (Mittal and Martín 2013). Then, using containers, a swarm of DEVS/SOA simulators can be created, ready to perform distributed simulations.

Any microservices architecture is primarily an orchestration of stateless services. In a component-based M&S framework such as DEVS, a component-model has to be transformed to a stateless service and address two fundamental microservice architecture requirements of distributed data management and shared event stores. We shall now discuss how the above aspects are handled at the *modeling* and *simulation* layers such that the model's state and inherent information can be externalized.

4.1 Modeling layer implementation with microservices paradigm

A DEVS model consists of ports (input and output), states and state-variables (including model name, current state, next state, time-of-last-event, time-of-next-event and elapsed-time) and the four characteristic functions ($\delta_{ext}, \delta_{int}, \delta_{con}, \lambda(s)$). In a microservices-based rendition of a DEVS model, we have to partition these elements into the two buckets (of distributed data management and event stores) to achieve scalability through stateless execution of a modular DEVS components. The DEVS state-machine needs to be separated with the operations on the state-variables inside the DEVS atomic model component. The first bucket is the model's state, that is stored in state-variables, which have to be serialized in a local database for that model. This implies that all the operations on these variables through the four characteristic functions will be through the accessor functions i.e. *get* and *set*. The second bucket is of the shared event store that is used for the *input X* and *output Y* sets. The *X* and *Y* sets are transformed into declarative events (that may be implemented as a complex data-type) into a shared Event store. Likewise, the DEVS atomic simulator components are partitioned as well. Table 1 shows the partiontioning of atomic DEVS elements (both modeling and simulation layers). Table 2 shows the coupled DEVS elements partitioning. The above partitioning allows the model to become stateless, which then can be containerized.

4.2 Simulation layer implementation with microservices paradigm

The simulation layer architecture will focus on the simulator and coordinator execution and how they implement the DEVS simulation protocol in an abstract-time manner. The architecture again has to account for the above two requirements: distributed data management and event stores. The DEVS simulation protocol defines the relationship between the model and its underlying simulator. The application of microservices provides resiliency at both the model and the simulator levels, to the effect that a large number of simulator instances with corresponding model instances can now be created when the data-exchange between the model and the simulator is accurately partitioned (as in Tables 1 and 2).

We shall illustrate the microservices-based DEVS/SOA simulation architecture implementation with the help of the classsic EF-P example (Figure 5). EF-P model contains two components: the Experimental Frame (EF) - Processor (P) models (Mittal and Martín 2013). The hierarchical EFP model in Figure 5.a is flattened towards a Generator - Processor - Transducer (GPT) model depicted in Figure 5.b. Next, according to Figure 5.c, each of the three submodels in GPT, is mapped to a separate DEVS/SOA node (container). Although, they all can also belong to a single container. One node can thus contain one or more DEVS models with their corresponding local databases to store model's data (Column 3 in Table 1). The node is selected through the simulation configuration file. For illustration purposes, Figure 5.c shows three nodes each corresponding to a single sub-model in GPT. One simulator is then created for each atomic model. Due to the symmetrical architecture, each node also contains a blueprint of a coordinator. The simulation

Table 1: Partitioning Atomic DEVS M&S Elements for Microservices implementation.

Atomic DEVS model Specification	M&S Element	Data Management	Event Store	Comments
X	Model		x	Input events
Y	Model		x	Output events
name	Model	x		Model name
tl	Simulator	x		Time of last event
tn	Simulator	x		Time of next event
phase	Model	x		Current phase of the model
$v1, v2..vn$	Model	x		Values and their data types

Table 2: Partitioning Coupled DEVS M&S Elements for Microservices implementation.

Coupled DEVS model Specification	M&S Element	Data Management	Event Store	Comments
X	Model		x	Input events
Y	Model		x	Output events
$M1, M2..Mn$	Model	x		Sub-component names
I_Z	Model	x		Set of influencers
Z_{i-d}	Model/Simulator	x		Mapping of influencer outports to influencee's inports

configuration file designates one of the nodes as the root coordinator. At the end, models' state and output events are always managed by their corresponding simulators, using the local databases and global event stores, respectively. Model output events are propagated through synchronous communication between the model-simulator pair, using the aforementioned `set` and `get` accessors. It should be highlighted that if the DEVS model is not flattened in the simulation configuration file, then the coupled information in Table 2 is stored as well.

The event store is implemented using various event cloud technologies such as Esper and provides a means to perform model-integrated systems engineering in which modeling and simulation itself is a part of the systems engineering (Mittal and Martin 2013). The event stores are usually implemented as event clouds which lend themselves to complex event processing (CEP) that can do streaming analytics as well as design monitors that can detect advanced spatiotemporal patterns between the messages flowing between the components. All the containers and global event store interface through a load balancer that divides the load on each container node through various load-balancing policies.

The detailed architecture for coordinator and simulator design will be reported in our extended article. We now look at the automated deployment of a DEVS/SOA node using Docker.

5 DOCKER DEVS/SOA DOCKLET

A Docker Image is a read only template used to instantiate Docker containers. Each image is defined with several layers that compose the final image structure. The Docker registry also called Docker Hub is a Docker Image repository. Images can be downloaded or uploaded. The Docker Hub has a considerable amount of images ready to use. Finally, a Docker Container is the runtime component of the Docker Image.

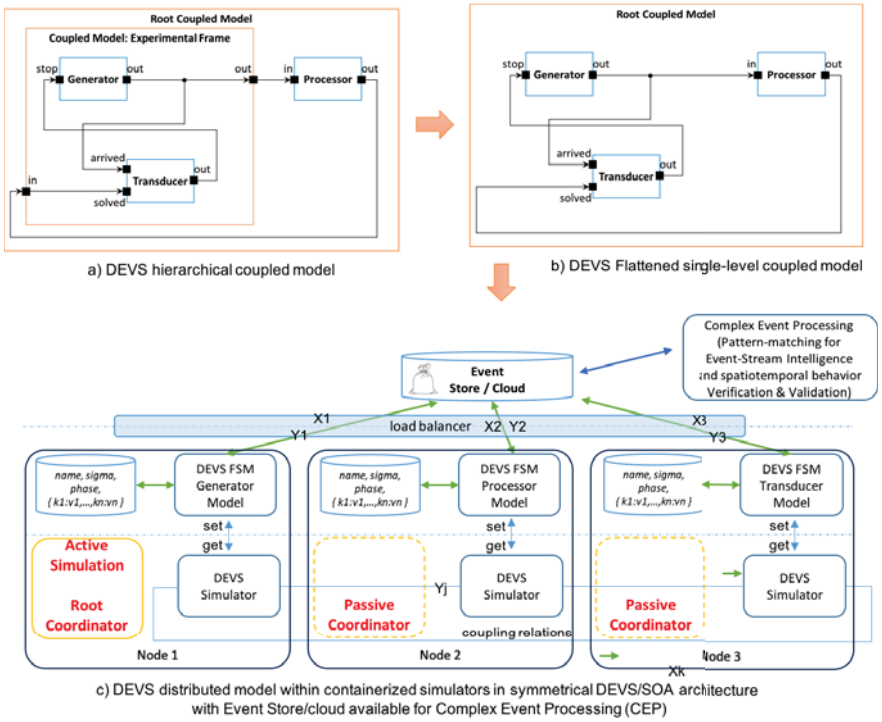

Figure 5: DEVS/SOA container nodes within Microservices paradigm.

Multiple containers can be instantiated from the same Docker Image in an isolated context. Docker container can be run, started, stopped, moved and deleted. To start, Docker must be installed in the host machine. For more information, the reader should be refer to the Docker official web page (Docker 2016).

To build our DEVS/SOA Dockerfile, we start from a DEVS/SOA complete WAR file (Mittal, Martín, and Zeigler 2009), which includes the DEVS simulation engine and several DEVS example models. The corresponding Docker image must then include a minimal runtime environment that contains the DEVS/SOA dependencies: Linux OS, Oracle Java 8 and an application service such as Apache Tomcat. Next, the .war file will be deployed into the webapps directory of Apache Tomcat.

Figure 6 depicts a Dockerfile structure. The source code of this file is shown in Appendix A.1. As Figure 6 shows, the Dockerfile must start with a base image. This is performed using the `FROM` instruction. In our case, the base image will be a Ubuntu. Next, MySQL and Java 8 are added. The final step is to add Apache Tomcat. Finally, as we show in Appendix A.1, more security constraints can be added to the Dockerfile. Once the Dockerfile is completed, we can show and stop container through the `docker ps` and `docker rm` commands on the host OS that has the Docker daemon running.

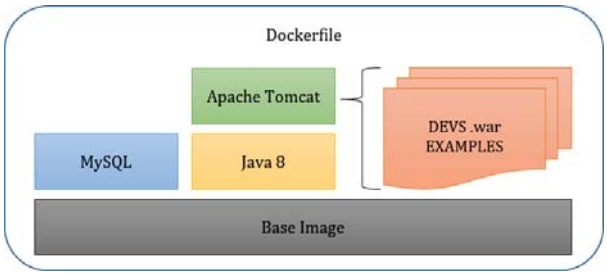

Figure 6: Full Dockerfile structure.

6 EXAMPLE: DEPLOYMENT CONFIGURATION FOR DEVS FARM

We now show how to deploy a set of Docker containers to simulate a distributed GPT DEVS model. The example shown here follows the same scheme as those provided in (Mittal, Martín, and Zeigler 2009).

Figure 7 shows the configuration flow. The source code and scripts needed to perform these steps are detailed in Appendix A.2. It has four steps. First, we must create a bash file, called, for instance, `admin.sh`, to create the Tomcat admin user. This script introduces an admin username and password in the `CATALINA_HOME/conf/tomcat-users.xml` configuration file. The exact content of this file is out of the scope of this paper. Second, the Docker Image specified in the Section A.1 and Appendix A.1 is built, deploying our DEVS/SOA war file into the Tomcat *webapps* directory.

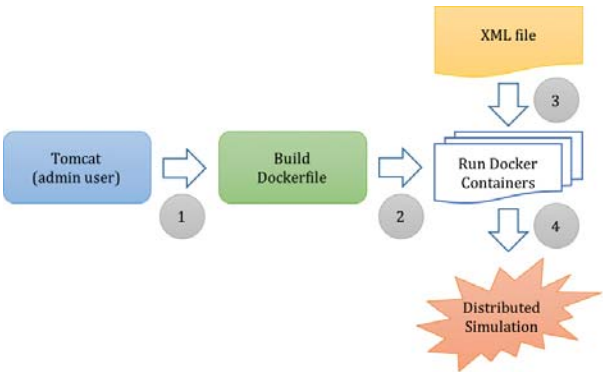

Figure 7: Distributed simulation using DEVS/SOA containers.

As stated earlier, a distributed DEVS/SOA simulation is performed starting with a XML configuration file. This XML file contains the location of each independent sub-model. An excerpt of the XML file in A.2 shows the configuration of a distributed simulation which will use three sub-models located at three different containers. As the third step in the deployment process, three different Docker containers must be created. These three containers can be hosted by the same machine (with the same or different ports) or like in the example (Figure 5) on different machines. These containers can be created manually, or using specialized software like Docker Swarm (Swarm 2017). The first three steps conclude the deployment of both the model and DEVS/SOA runtime in a containerized environment. As a result, the distributed simulation can be executed from any client computer, with the corresponding `devssoa.jar` client application:

```
$ java -jar devssoa.jar devssoa-gpt.xml
```

This completes the automated deployment of a simulation farm, as virtualized containers in a cloud environment, capable of performing a distributed DEVS simulation. To summarize, we are able to run our DEVS/SOA distributed simulations as described in (Mittal, Martín, and Zeigler 2009), using `http://localhost:8080` as a simulation server or client. Of course, this Docker-based methodology is applicable to other distributed simulation engines based on REST, Sockets, or any other distributed mechanism.

7 DISCUSSION AND CONCLUSIONS

DEVS/SOA introduced the execution of DEVS models on SOA and spear-headed the DEVS model and simulation interoperability. This led to the conceptualization of the netcentric DEVS virtual machine that can be deployed as a stand-alone piece of software providing DEVS execution in a platform independent manner. The addition of container using open-source Docker is the next step that automates the deployment of these virtual machines in an automated DevOps manner. This allows the vertical scaling of DEVS VM nodes so that the DEVS nodes can be scaled up on-demand. Finally, the microservices implementation allows horizontal scaling inside a VM at the model level. This is useful when the same model class (i.e. not a model instance) needs to be a part of multiple experimental frames executing a monte-carlo simulation or concurrent execution (albeit separate experiments/sessions).

We reviewed the DEVS/SOA symmetrical services architecture as a precursor to the microservices paradigm, extended the DEVSML stack to employ microservices paradigm and container support using open source container solutions such as Docker, to develop a distributed simulation platform in Cloud environment. This paper has made two contributions. It introduced Microservices and container paradimgs using Docker. We illustrated by an example how the deployment of distributed DEVS Farm in Cloud environment is made possible with these new technologies. And second, advanced the state-of-the-art in transparent simulation infrastructure with DEVSML Stack Version 3.0 that incorporates container services for automated DEVS/SOA deployments. In our extended article, we aim to provide detailed architecture for DEVS Farm and performance evaluation in a distributed Cloud environment.

DISCLAIMER The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author(s). Approved for Public Release: Case Number 17-0320.

REFERENCES

Cayirci, E. 2013. "Modeling and Simulation as a Cloud Service: A Survey". In *Proceedings of Winter Simulation Conference*.

Docker 2016. "What is Docker?". https://www.docker.com/what-docker. Accessed Nov. 08, 2016.

Joy, A. M. 2015, March. "Performance comparison between Linux containers and virtual machines". In *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346. IEEE.

A. Karmel and R. Chandramouli and M. Iorga 2016. "NIST Special Publication 800-180: NIST Definition of Microservices, Application Containers and Virtual Machines". http://csrc.nist.gov/publications/drafts/800-180/sp800-180_draft.pdf. Accessed Nov. 08, 2016.

Larsson, M. 2015. "An Operations model for Microservices". http://callistaenterprise.se/blogg/teknik/2015/03/25/an-operations-model-for-microservices/. Accessed Jan. 08, 2017.

Mittal, S., and S. Douglass. 2012. "DEVSML 2.0: The Language and the Stack". In *DEVS Symposium, Spring Simulation Multiconference*.

Mittal, S., and J. Martin. 2013. "Model-driven systems engineering for netcentric system of systems Engineering with DEVS Unified Process". In *Winter Simulation Conference*.

Mittal, S., J. Martin, and B. Zeigler. 2007. "DEVSML: Automating DEVS simulation over SOA using transparent simulators". In *DEVS Symposium, Spring Simulation Multiconference*.

Mittal, S., and J. L. R. Martín. 2013. *Netcentric System of Systems Engineering with DEVS Unified Process*. CRC Press.

Mittal, S., J. L. R. Martín, and B. P. Zeigler. 2009, July. "DEVS/SOA: A Cross-Platform Framework for Net-centric Modeling and Simulation in DEVS Unified Process". *SIMULATION* vol. 85 (7), pp. 419–450.

Mittal, S., M. Ruth, A. Pratt, D. Krishnamurthy, M. Lunacek, and W. Jones. 2015. "A System of Systems Approach to Integrated Energy Systems Modeling". In *Summer Computer Simulation Conference*. Chicago, IL.

Richardson, C. 2015. "Building Microservices: Using an API Gateway". https://www.nginx.com/blog/building-microservices-using-an-api-gateway/. Accessed Nov. 08, 2016.

Docker Swarm 2017. "Docker Swarm". https://www.docker.com/products/docker-swarm.

AUTHOR BIOGRAPHIES

SAURABH MITTAL is Lead Systems Engineer/Scientist at MITRE Corporation, McLean, VA USA. He is also affiliated with Dunip Technologies, LLC, USA; Society of Computer Simulation (SCS) International; and Enterprise Architecture Body of Knowledge (EABOK) Consortium. He can be reached at smittal@mitre.org.

JOSE L. RISCO MARTIN is an Associate Professor at Complutense University of Madrid, Spain. He is Associate Editor for Transactions of Society of Computer Simulation (SCS) International. His email address is jlrisco@ucm.es.

A APPENDIX: CONFIGURATION SCRIPTS

A.1 Full Dockerfile

```
## Dockerfile
FROM  phusion/baseimage:0.9.17
MAINTAINER  DunipTech <duniptech@xxxxx.com>
RUN echo "deb http://archive.ubuntu.com/ubuntu trusty main universe" \
    > /etc/apt/sources.list
RUN apt-get -y update


## MySQL + Java 8
RUN apt-get -y install mysql-server
EXPOSE 3306
ENV JAVA_VER 8
ENV JAVA_HOME /usr/lib/jvm/java-8-oracle
# Install Java 8
RUN echo 'deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' \
    >> /etc/apt/sources.list && \
    echo 'deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' \
    >> /etc/apt/sources.list && \
    apt-key adv --keyserver keyserver.ubuntu.com --recv-keys C2518248EEA14886 && \
    apt-get update && \
    echo oracle-java${JAVA_VER}-installer shared/accepted-oracle-license-v1-1 \
    select true | sudo /usr/bin/debconf-set-selections && \
    apt-get install -y --force-yes --no-install-recommends \
    oracle-java${JAVA_VER}-installer oracle-java${JAVA_VER}-set-default && \
    apt-get clean && \
    rm -rf /var/cache/oracle-jdk${JAVA_VER}-installer
# Set Java 8 as the default JVM
RUN update-java-alternatives -s java-8-oracle
RUN echo "export JAVA_HOME=/usr/lib/jvm/java-8-oracle" >> ~/.bashrc
# Clean up APT
RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

## Apache Tomcat
# Prepare the environment
RUN apt-get update && \
    apt-get install -yq --no-install-recommends wget pwgen ca-certificates && \
    apt-get clean && \
```

```
    rm -rf /var/lib/apt/lists/*
ENV TOMCAT_MAJOR_VERSION 8
ENV TOMCAT_MINOR_VERSION 8.0.11
ENV CATALINA_HOME /tomcat
# Install Tomcat
RUN wget -q https://archive.../apache-tomcat-${TOMCAT_MINOR_VERSION}.tar.gz && \
        wget -qO- https://archive...tar.gz.md5 | md5sum -c - && \
        tar zxf apache-tomcat-*.tar.gz && \
        rm apache-tomcat-*.tar.gz && \
        mv apache-tomcat* tomcat

ADD admin.sh /admin.sh
RUN mkdir /etc/service/tomcat
ADD run.sh /etc/service/tomcat/run
RUN chmod +x /*.sh
RUN chmod +x /etc/service/tomcat/run

EXPOSE 8080

# Use baseimage-docker's init system
CMD ["/sbin/my_init"]
```

A.2 Deployment configuration

```
// Build a docker image
$ docker build -f Dockerfile -t springsim/devssoa:tomcat-jdk-8 .

// DEVS/SOA war file --> Tomcat webapps directory:
$ docker run -it -d --name devssoa -p 8080:8080 \
  -v "$PWD":/app springsim/devssoa:tomcat-jdk-8 \
  bash -c "cp /app/target/devssoa.war /tomcat/webapps/ \
           & /tomcat/bin/catalina.sh run"

// devssoa-gpt.xml: DEVS/SOA XML configuration file
<coupled id="devssoa-gpt" host="http://192.168.1.105:8080">
  <atomic name="Generator" class="Generator" host="http://192.168.1.101:8080"/>
  <atomic name="Processor" class="Processor" host="http://192.168.1.103:8080"/>
  <atomic name="Transducer" class="Transducer" host="http://192.168.1.105:8080"/>
  <connection atomicFrom="Processor" portFrom="out"
              atomicTo="Transducer" portTo="solved"/>
  <connection atomicFrom="Generator" portFrom="out"
              atomicTo="Processor" portTo="in"/>
  <connection atomicFrom="Generator" portFrom="out"
              atomicTo="Transducer" portTo="arrived"/>
  <connection atomicFrom="Transducer" portFrom="out"
              atomicTo="Generator" portTo="stop"/>
</coupled>
```