This is the pre-peer reviewed version of the following article: Concurrent simulation in the cloud with the mJADES framework by Antonio Cuomo; Massimiliano Rak; Umberto Villano, which has been published in final form in International Journal of Simulation and Process Modelling and is available at http://www.inderscience.com/info/inarticle.php?artid=59418
DOI: 10.1504/IJSPM.2013.059418

Concurrent simulation in the cloud with the mJADES framework

Antonio Cuomo

Department of Engineering,
University of Sannio,
Benevento 82100, Italy
E-mail: antonio.cuomo@unisannio.it
*Corresponding author

Massimiliano Rak

Department of Engineering,
Second University of Naples,
Aversa 81031, Italy
E-mail: massimiliano.rak@unina2.it

Umberto Villano

Department of Engineering,
University of Sannio,
Benevento 82100, Italy
E-mail: villano@unisannio.it

Abstract: In this paper we discuss the design and implementation of *mJADES*, a new simulation engine that runs on top of an ad-hoc federation of cloud providers and is designed to perform multiple concurrent simulations. These features make mJADES an attractive environment for the simulation of complex systems, in which it is often desiderable to perform many simulation runs, either for statistical validation or to compare the system behavior in several different conditions. Given a set of simulation tasks, *mJADES* is able automatically to acquire the computing resources needed from the cloud and to distribute the simulation runs to be executed.

Keywords: Discrete-event Simulation; Cloud Computing; Concurrency; Platform-as-a-Service; Process-oriented Simulation

Biographical notes: Antonio Cuomo is a Ph.D. student at the University of Sannio, Benevento, Italy. His research activities focus on performance evaluation and prediction of parallel and distributed computer systems, including Cloud Computing, Grid Computing and their integration, with special attention to simulative prediction techniques. He received the laurea specialistica degree cum laude in Computer Science Engineering from the University of Sannio in 2009.

Massimiliano Rak is an assistant professor at Second University of Naples. He got his degree in Computer Science Engineering at the University of Naples Federico II in 1999. In November 2002 he got his Ph.D. in Electronic Engineering at Second University of Naples. His scientific activity is mainly focused on the analysis and design of High Performance System Architectures and on methodologies and techniques for distributed software development.

Umberto Villano is full professor at the University of Sannio at Benevento, Italy, where he is Director of the Department of Engineering. His major research interests concern performance prediction and analysis of parallel and distributed computer architectures, tools and environments for parallel programming and distributed algorithms. He received the Laurea degree in Electronic Engineering cum laude from the University of Naples in 1983.

1 Introduction

In the last few years, cloud computing (Peter Mell and Tim Grance, 2009) has proven to be a convenient paradigm for structuring and deploying applications and infrastructures. The elastic nature of the cloud paradigm, together with the "everything as a service" philosophy mutuated from the Service Oriented Architecture (SOA) methodology, has promoted its use

for on-demand provisioning of applications (Software-as-a-Service, SaaS), development platforms (Platform-as-a-Service, PaaS) or even basic computing resources as processing units, storage and networks (Infrastructure-as-a-Service, IaaS).

The framework described in this paper, mJADES, allows simulation developers to leverage a PaaS platform for developing and providing a SaaS simulation environment. More detailed, mJADES allows its users to execute multiple *concurrent* simulations in the cloud. In order to clarify this definition, below we will point out some key concepts concerned with simulation in parallel or distributed environments. A *simulation* is the evaluation of a model on some input data that will eventually produce output data (typically, performance indicators). A *simulation job* is the direct execution of a single user simulation request. Traditionally, a simulation job is executed through some kind of sequential simulation engine.

It is well known that the discrete-event simulation of complex systems is a time expensive and resource consuming task. These characteristics, along with the availability of multiple processors, led to the development of parallel, distributed and web-based simulation techniques (Fujimoto, 1990), (Perumalla, 2006), (Fujimoto, 2003), (Byrne et al., 2010), whose aim is to exploit all the available processors (in a machine or in a network) to run a simulation. In parallel simulation, the model is decomposed into entities which are allocated on a network of (tightly coupled) processors. Groups of entities are assigned to Logical Processes (LP), which evolve with their own simulation clock. This entails the need to keep the different LPs synchronized, a problem for which different solutions have been proposed, including conservative (Chandy and Misra, 1979) and optimistic (Jefferson and Sowizral, 1985) protocols. However, synchronization issues, along with load balancing and memory management problems, often prevent the attainment of good scalability and computation efficiency. On the other hand, distributed simulation and, more recently, web-based simulation, are targeted to the exploitation of distributed computing resources. In the first case, the objective is to use suitable protocols to organize independent simulation models into a larger, single simulation. In the latter, the stress is on the exploitation of the ubiquitous web-based technology to run simulations through a browser interface (Byrne et al., 2010). Most often, web-based simulation is based on sequential simulation engines (i.e., not parallel), running at the client or at the server side.

The approach followed in this paper is in some sense similar to web-based simulation, in that mJADES aims at running sequential, monolithic discrete-event simulations, adding the possibility to run multiple simulation instances at the same time, leasing automatically a suitable number of computing resources from a cloud. The rationale of this design is that, if a simulation must be executed many times, there is an exploitable source of parallelism in running together multiple simulation instances. This happens when simulations are stochastic and should be repeated to perform statistical validation (Sargent, 2000), or for parameter-sweep simulations, where one or multiple input parameters vary over a range of values. If this is the case, simulation jobs can be decomposed into multiple *simulation tasks*, each of which represents a single sequential simulation run and may be evaluated independently of one another.

The mJADES simulation system founds on a Java-based modular architecture. The mJADES *simulation manager* produces simulation tasks from simulation jobs, and schedules them to be executed concurrently on multiple instances of the *simulation core*. This is a process-oriented discrete-event simulation engine based on the JADES simulation library (Cuomo et al., 2012). The outputs from the runs are handed on to a *simulation analyzer*, whose task is to compute aggregates and to generate reports for the final user. In order to fully exploit cloud facilities and, notably, to be able to run the application on resources dynamically acquired from different cloud providers, mJADES leverages on the mOSAIC platform (Petcu et al., 2011b,a). mOSAIC is a cloud middleware that provides a cloud-enabled, component-based programming interface. In fact, mJADES has been developed as a collection of mOSAIC components. So it can interact easily with other components provided to the developer by the mOSAIC platform. These components make it possible, among other things, to build up custom REST APIs and web interfaces, to provide SLA management and to monitor the cloud providers.

The remainder of this paper is structured as follows: the next section presents background and related work on simulation, pointing out the original features of our approach. Section 3 details the design and implementation of the JADES simulation library. Section 4 provides an overview of the mOSAIC platform. The architecture and workflow of the mJADES system is described in section 5. A simulation case study is described in section 6. Section 7 draws the conclusions and sketches our future research directions.

2 Background and Related Work

mJADES aims at providing a management framework for the distribution of simulation tasks on cloud resources. Even if the system can be made independent of the simulation engine used, the focus of this paper will be on Java-based engines for process-oriented discrete event simulation. In the following, we will briefly review the literature dealing with these research topics.

2.1 Process-oriented Discrete-event Simulation: a Java perspective

The *process-oriented* (or process interaction) view hinges on the well-known concept of process. It enables the

modeler to clearly grasp a model structure, since each object can be represented through a single, coherent process rather than multiple event routing. Very often the process-oriented view is internally implemented on the top of an event-oriented kernel, due to the efficiency of the last approach. But the simulator design is not trivial: to implement through a sequential program the concurrent execution of simulation processes evolving in discrete time, these should take turns in their execution on a sequential machine (the use of parallel machines introduces further problems). This requirement is strictly linked to the concept of *coroutines*, a generalization of subroutines, introduced in the sixties by Conway. As described in (Marlin, 1980), the two must important features of coroutines are that:

- the values of data local to a coroutine persist between successive calls;
- the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine later.

In the Java language, as there is no direct support for them, coroutines can be implemented through the built-in thread system: it is sufficient to associate a thread to each coroutine, and to perform yields through the basic synchronization primitives generally available with threads (e.g., wait() and notify()). However, many researchers have recognized that the use of threads just to hold some computation state is overkill, and can possibly lead to large overheads.

The simulation engine used in mJADES is built using the JADES (JAva Discrete Event Simulator) library. JADES, described in section 3, uses continuations on the top of a standard JVM to provide the process-oriented view. This gives several performance advantages, due to the use of a single simulation thread with fast switching capabilities. A performance comparison of JADES to several other simulation libraries is available in (Cuomo et al., 2012).

Other Java-based process-oriented discrete-event simulators/libraries have been developed and described in the literature. Traditional, thread-based systems include SimJava(Howell and McNab, 1998) and JSim (Miller et al., 1997), DesmoJ (Lechler and Page, 1999) and SSJ (L'Ecuyer and Buist, 2005; L'Ecuyer et al., 2002). In recent years, different approaches have been followed to provide a better implementation of the process interaction view. D-SOL (Jacobs et al., 2002; Jacobs and Verbraeck, 2004) is based on the use of a process interpreter. This can be thought of as a virtual machine implemented in Java that executes the code of the process class. The interpreter takes care of pausable methods and is able to save their execution context. Tortuga (Weatherly and Page, 2004; Weiland et al., 2005) provides an implementation of coroutines in Java which is not based on threads. This is based on a modification of a non-standard Java virtual machine, the Jikes RVM, able to provide the state-saving mechanism. JSL (Rossetti, 2008) initially supported process interaction with Java threads, but the implementation is now deprecated: the library supports an approach called process description, in which the process interaction is mapped to a state transition diagram based on event scheduling. The MyTimeWarp simulator (Kunert, 2008) hinges on the JavaFlow continuation library (which is also adopted by the JADES simulator presented in this paper) and focuses on process-oriented time warp optimistic parallel simulation.

2.2 Distributed Simulation

As it has been discussed in the introduction, mJADES is designed in such a way that parallel or distributed hardware is exploited by running multiple concurrent instances of a traditional sequential simulator. This makes it fairly different from the majority of the systems developed in main body of research on parallel and distributed simulation (PADS). (Fujimoto, 1990), (Perumalla, 2006), (Fujimoto, 2003). In these systems, the model is decomposed into entities which are allocated on a network of processors. Some simulators are able to exploit SMP (symmetric multiprocessing), as ComplexSim (Messina et al., 2012) while others as ns-3 (Pelkey and Riley, 2011) are able, using message passing, to operate on processors located on different nodes interconnected by a local or wide area network. In PADS systems, groups of entities are assigned to Logical Processes (LP), which evolve with their own simulation clock. This entails the need to keep the different LPs synchronized, a problem for which different solutions have been proposed, including conservative (Chandy and Misra, 1979) and optimistic (Jefferson and Sowizral, 1985) protocols. A recent survey on Java - based parallel simulators is (Castilla et al., 2011).

The basic idea behind mJADES, that is, to speed-up simulations by launching multiple concurrent runs ("replications") is not new, and was explored in a seminal paper by Heidelberger in the eighties (Heidelberger, 1986), where the Authors compared the obtained performance to the one attained by a more conventional parallel simulator. While in the past two decades the dominance of tightly coupled, network-based platforms like clusters and grid systems led to focus on parallel simulation engines, the cloud approach, which founds on web transactions managed by independent tasks, makes the idea of concurrent runs again attractive.

2.3 Simulation-as-a-Service

The "as-a-service" methodology originates in the world of Service-Oriented Architecture (SOA). In the literature there are several contributions on the "as-a-service" provisioning of simulation software. The paper (Marr et al., 2000) presents a Java-based architecture for replicated simulations on networks of processors, called

"alliance". Our work relies on the more recent cloud technologies and exploits an open-source simulator engine instead of Silk (Healy and Kilgore, 1998). Some architectural aspects (providing web services interfaces to simulations, translating simulation requests to XML) are tackled in (Seo et al., 2010). (Guo et al., 2011) discusses issues related to the design of simulation web services and the composition of those into simulation workflows. (Tsai et al., 2011) introduces SimSaaS (Simulation Software-as-a-Service), a simulation framework which mainly focuses on the issues related to the management of multi-tenancy.

2.4 Simulation and Cloud Computing

Papers on the use of cloud resources for computer simulation are just beginning to appear in the literature. Most of them try deal with the porting of "traditional" parallel/distributed simulation schemes to a public or private cloud. Park and Fujimoto (Park and Fujimoto, 2009) propose Aurora, an adaption of optimistic parallel discrete-event simulator based on a master-worker approach. Aurora is more suitable to be used in utility computing systems than traditional ones based on optimistic synchronization.(Fujimoto et al., 2010). D'Angelo (D'Angelo, 2011) argues that the high variability in performance, typical of cloud environments, must be taken explicitly into account. His research team is developing Artis/GAIA+, a middleware for parallel and distributed simulations that aims at balancing adaptively the simulation by model decomposition, dynamic partitioning and bottleneck removal. In (Taylor et al., 2011) the authors present an application of a master-worker volunteer-computing based platform to solve simulation models requiring many runs. However, unlike in mJADES, the generation of tasks and the analysis of results is not performed automatically.

2.5 Open-source cloud middlewares

Nowadays, cloud computing is still a hot research topic. Public providers are growing in number, and open-source middleware designed for hosting private clouds have reached sufficient maturity to be deployed in commercial and research environments. (von Laszewski et al., 2012) is an up-to-date comparison of the most used IaaS frameworks, namely Eucalyptus, Nimbus, OpenNebula and OpenStack. Research prototypes providing additional features include PerfCloud (Mancini et al., 2009), which aims at adding performance-oriented services to the customary IaaS offering, and CLEVER (Tusa et al., 2010), which focuses on scalability and fault-tolerance.

Recently, attention is shifting to higher level architectural layers, like Platform-as-a-Service. In this context, currently dominated by proprietary solutions, open-source frameworks are beginning to spread. They will allow the development of cloud applications portable among different providers. (Petcu et al., 2012) discusses the emerging open-source offerings in the context of mOSAIC, the platform of choice for mJADES. An overview of mOSAIC is provided in section 4.

3 Process-oriented discrete event simulation with JADES

JADES is an engine for the simulation of discrete-event simulation models. It supports the process-oriented view, allowing the modeler to describe his model in terms of interacting processes. It is not a complete stand-alone simulation language, but a Java library that exploits continuations to implement the simulated processes. It is not a simulator, as it only provides the simulation building blocks but leaves model design entirely to its users. The aim of JADES is to provide modelers with an implementation of the process-oriented paradigm which is both *effective* (as it offers a wide range of building blocks to compose models) and *efficient* (allowing the creation of large, complex models and their fast evaluation).

A JADES simulation is composed of processes, which are active components able to act upon (and interact through) passive objects, or resources. The JADES simulation engine is based on the use of *continuations*, which enable it to run all the simulation processes in a single Java thread. This has significant effects as far as performance is concerned (Cuomo et al., 2012), due to the absence of thread switch overheads. Furthermore, it helps the integration with mOSAIC cloudlets, which are completely asynchronous and not thread-safe. Subsection 3.1 describes the design of the library, while 3.2 presents the implementation details focusing on the management of the continuations.

3.1 JADES Design

While the implementation is quite innovative, the design of the JADES programming interface is inspired by the popular simulator CSIM (Schwetman, 2001), which provides one of the most complete process-oriented simulation APIs for C and C++. There exists a Java version of CSIM, which, unlike the coroutine-based C/C++ version, uses Java threads to implement the simulation processes. This design choice introduces all the drawbacks discussed in the background section 2.1. Due to the similarity between the two programming interfaces, the wide literature on the development of CSIM simulation models (Schwetman, 1988; Edwards and Sankar, 1992; Panda et al., 1997; Doğan, 2009) can also be used as reference by JADES users.

Figure 1 shows the class diagram of the simulator. Modelers create their own simulation model as a subclass of `Model`: they have only to implement the `run` method, providing all the model-specific logic to create the initial simulation processes and the resources. Besides defining the overall model, developers have also to specify the

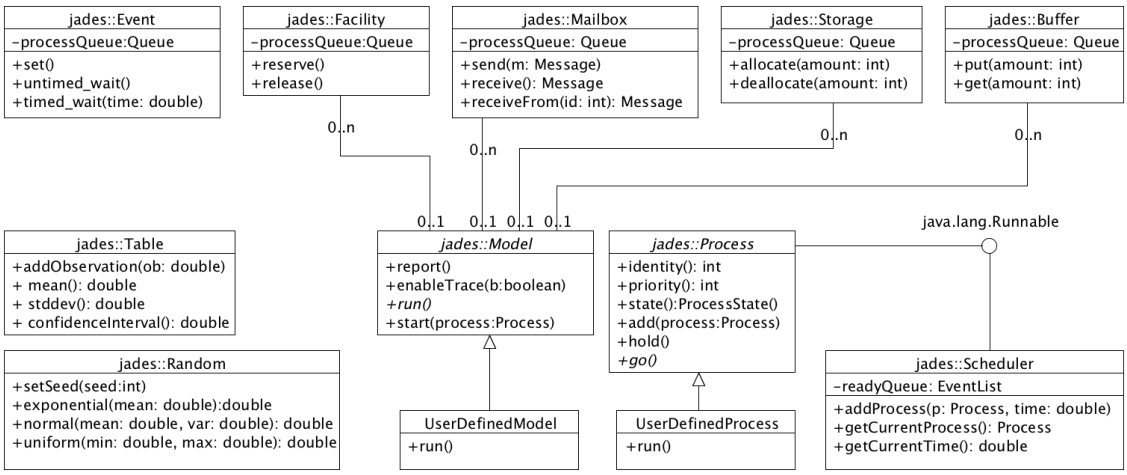

Figure 1: The JADES Simulation Library

behavior of their processes. As for the model, this is done by subclassing a predefined `Process` class, which provides all the common basic functionality a process needs. These include: *a*) *holding*, that is, waiting for the passing of a certain amount of simulation time; *b*) adding other processes to the simulation (at current simulation time, or later); *c*) making use of resources. The resources are passive objects that provide higher-level, more useful abstractions than simple process queues. Predefined class of resources provided with the simulator are:

- `Facility`, which models all-or-none resources, as servers;
- `Storage`, which models resources that can be partially allocated as memories, disks, ...;
- `Buffer`, a resource through which processes can communicate and synchronize by producing and consuming items;
- `Mailbox`, through which processes can interact by exchanging messages;
- `Event`, for conditional process synchronization. Process can wait for events to occur and declare events as occurred.

Every resource is instrumented to gather statistics about its usage during the simulation (average length of the associated queue, time spent waiting, number of processes served). Additional statistics can be gathered programmatically using `Table`s, which allow to add modeler-defined observations of values at given points in the program.

To obtain further details about the execution, it is possible to instruct the model to generate detailed traces of execution. Almost all the events that occur during the execution can be singularly traced, together with the process to which they refer and the simulation time when the event occurred.

The `Random` class allows the generation of pseudo-random numbers. Modelers can make use of a variable number of random streams by instantiating the desired quantity of Random objects, for which different seeds are provided (with the possibility to provide directly the seed).

3.2 JADES Implementation

This subsection will discuss some of the internals of JADES, with particular regard to its distinguishing feature, the use of continuations to implement the simulation processes. The following presentation purposely omits the description of the more "conventional" parts of the simulation library. Since continuations are not directly provided by the standard Java platform, JADES must resort to an external system which we call *continuation provider*. Several existing continuation providers have been evaluated for use in JADES. As one of JADES objectives was the use of a *standard* JVM (to allow easy integration of the simulator in other applications), the implementation did not consider approaches based on modified VMs. Instead, the simulator relies on Javaflow (Ortega-Ruiz et al., 2004), a component of the Apache Jakarta Commons Sandbox. Javaflow provides *asymmetric continuations*, in that it forces the programmer to specify the continuation to which he wants to pass control. The core JavaFlow API is found in the static methods of the `Continuation` class:

- `Continuation startWith(Runnable r)` makes it possible to construct a continuation from a `Runnable` object and execute its `run` method. Control passes to the continuation and goes back to the caller if `run` ends or if the `suspend()` method is invoked (in which case a valid continuation is returned).

- `Continuation continueWith(Continuation c)` resumes the execution of the continuation passed as parameter from where it left off.
- `void Continuation suspend()` stops the running continuation, creating a resumption point and giving control back to the method that called `startWith` or `continueWith`.

JavaFlow implements the continuations functionalities through bytecode rewriting. The bytecode of all the classes of the system is scanned for the invocation of the `suspend` method. When a method is found that contains such invocation, it (and all its potential invokers, recursively up to the start of the continuation), are instrumented to add the continuation-management code. This includes: *a*) code which must be executed when a suspension occurs, which includes switches that represent the intermediate points of the method being executed and calls to a library-managed stack that maintains the content of the stack frame; *b*) code for resuming the execution at the point of the suspension (following from the start the chain of switches) with the associated stack contents (popping the stack managed by the library). Examples of this process are shown in (Ortega-Ruiz et al., 2004; Kunert, 2008).

Let us now describe how the JavaFlow library is used in different part of JADES. When the first process is added to the simulation, it is added to the ready queue and a continuation is created for the scheduler code:

```
public void start (Process p){
    scheduler.addProcess(p, scheduler.getCurrentTime());
    Continuation.startWith(scheduler);}
```

When a process wants to add a new process to the simulation, it tells the scheduler to enqueue the new process and gives control to the scheduler continuation by suspending itself.

```
public void add (Process p, double delay){
    double currentTime = scheduler.getCurrentTime();
    scheduler.addProcess(p, currentTime+delay);
    scheduler.addToReadyProcesses(
        scheduler.getCurrentProcess(), currentTime);
    Continuation.suspend();}
```

The code for a process hold is shown below. If the current process will sleep past the wake-up time of the process at the head of the queue, the process is added to the ready queue, giving control to the scheduler continuation. Otherwise, the current process has to hold just to be made active immediately next: we can just advance simulation time to its wake-up time, avoid unnecessary rescheduling.

```
public void hold(double time){
    double wakeupTime = scheduler.getCurrentTime() + time;
    double nextWakeupTime = scheduler.peekNextTime();
    if (nextWakeupTime > wakeupTime)
        scheduler.setCurrentTime(wakeupTime);
    else{
        scheduler.addToReadyProcesses(this, wakeUpTime);
        Continuation.suspend();}}
```

The scheduler continuation executes the scheduling loop in which:

1. The process with smallest wakeup time is extracted from the ready queue. If there is no process available, simulation is finished. Otherwise:

2(a). if the extracted process is at its first schedule, a continuation is created for it and started through the `Continuation.startWith()` method;

2(b). else if the extracted process is not at its first schedule, its continuation is resumed through the `Continuation.continueWith()` method.

4 mOSAIC: Development of Cloud Applications

As mentioned in the introduction, mJADES is a cloud application built on the top of the mOSAIC middleware. To give more insight on the structure of the simulation environment, below we will briefly sketch the design of the mOSAIC framework. Further details can be found in (Petcu et al., 2011b,a).

mOSAIC provides an API to develop cloud applications, which are successively executed in a leased environment provisioned and controlled by the mOSAIC run-time. Hence, the target user for the mOSAIC solution is the application developer (*mOSAIC user*). In mOSAIC, a cloud application is structured as a set of components running on cloud resources (i.e., on resources leased by a cloud provider) and able to communicate with each other. Cloud applications are often provided in the form of Software-as-a-Service, and can also be accessed/used by other users than the mOSAIC developer (i.e., by *final users*). In this case, the mOSAIC user acts as service provider for final users.

The mOSAIC framework is composed of several subsystems. The key roles are played by the *Platform* and the *Cloud Agency*. The Platform enables the execution of applications developed using the mOSAIC API. The Cloud Agency acts as a provisioning system, brokering resources from a cloud provider, or even from a federation of cloud providers.

mOSAIC can be useful in three different scenarios:

- when a developer wishes to develop an application not tied to a particular cloud provider;
- when an infrastructure provider aims at offering "enhanced" cloud services in the form of SaaS;
- when a final user (e.g., a scientist) wishes to execute his own application in the cloud because he needs processing power.

In this paper the focus will be on the first and second scenario. In fact, the objective of mJADES is to provide a simulation service, whether deployed by a user or offered by a provider in SaaS mode, on the top of an existing infrastructure (of federation of infrastructures).

A mOSAIC application is built up as a collection of interconnected *mOSAIC components*. Components

may be (*i*) core components, i.e., predefined helper tools offered by the mOSAIC platform for performing common tasks, (*ii*) COTS (commercial off-the-shelf) solutions embedded in a mOSAIC component, or (*iii*) *cloudlets* developed using the mOSAIC API and running in a *Cloudlet Container*. All the mOSAIC components run on a dedicated virtual machine, named mOS (mOSAIC Operating System), which is based on a minimal Linux distribution. The mOS is enriched with a special mOSAIC component, the *Platform Manager*, which makes it possible to manage a set of virtual machines hosting the mOS as a virtual cluster, on which the mOSAIC components are independently managed. It is possible to increase or to decrease the number of virtual machines dedicated to the mOSAIC Application, which will scale in and out automatically.

A cloud application is described as a whole in a file named *Application Descriptor*, which lists all the components and the cloud resources needed to enable their communication. A mOSAIC developer has the role both of developing new components and of writing application descriptors that connect them. The mOSAIC API, based on Java, enables the development of new components (in the form of *cloudlets*) through a wrapping system. Cloudlets self-scale and consume any kind of cloud resource, independently of the technologies and the API it adopts.

4.1 Autoscaling in mOSAIC

One of the advantages of developing applications using the mOSAIC framework is that mOSAIC enables an easy way to reconfigure and auto-scale the application. First of all, the mOSAIC platform offers ready-to-use components to create scalable architectures, such as queuing systems (*RabbitMQ* and *zeroMQ*), which are used for component communications, an HTTP gateway, which accepts HTTP requests and forwards them to application queues, and NO-SQL storage systems (as KV store and columnar databases like Riak, Membase and Redis). Moreover, the *Cloudlets*, i.e., the components developed through mOSAIC API, are developed following an EDA (Event-Driven Asynchronous) approach, so they execute code (and consume computational resources) in response to events generated by other components. The Cloudlets are stateless, maintaining state information only in the shared cloud storage systems. This implies that they can be started in multiple instances without varying their behavior, and can automatically perform *load balancing* by dividing among the cloudlet instances the events generated by communication resources (Petcu et al., 2011b,a).

At a lower level, cloud resources like Virtual Machines are managed by the mOSAIC Platform, which transparently deploys component instances on the VMs. For example, if all the VMs shows constantly high CPU load, the Platform may invoke the Cloud Agency in order to buy new VMs and add them to the pool of available resources. Moreover, when the Platform detects that a single component is overloaded (e.g., when the queue of messages associated with the component is too long), it may choose to start a new component instance. The decisions about the number of component instances and their mapping to the acquired resources are automatically made by the Platform. The Platform autoscale policies are defined by the application developer, using the SLA mOSAIC Framework components (Rak et al., 2011).

5 mJADES: Architecture and Workflow

The previous sections have described the JADES simulation library and the mOSAIC platform. mJADES, the integration of these two technologies, is a system able to execute JADES simulation tasks concurrently, by means of multiple instances of the simulation engine, which are executed as mOSAIC cloudlets. The mJADES architecture is represented in Figure 2. The application is composed of three types of cloudlets: `mJADES Manager`, `mJADES Simulation Core` and `mJADES Simulation Analyzer`. To support the *as-a-service* approach, final users can interact with the application through the HTTP interface made available by the mOSAIC HTTP gateway component. In the following, we outline the general workflow of the system to allow an easy understanding of its purpose and capabilities. Then each mJADES component is described in a dedicate subsection.

The complete workflow is described in Figure 3. mJADES users can interact with the web interface to specify simulation job requests. Each Simulation Job describes a whole simulation that may require the evaluation of different Simulation Tasks. The request is translated into a Simulation Job Description and sent to the mJADES Manager. This component is able to translate the Job into the basic Simulation Tasks, and to produce specifications for their execution. Task specifications are distributed to Simulation Core instances launched on cloud resources. Each Simulation Core hosts an implementation of the simulation model developed with the JADES library. The distribution and deployment of the whole simulation is greatly simplified, thanks to the capabilities of the mOSAIC framework. Each simulation result produced by the Core instances is stored in a dedicated cloud storage resource provided by mOSAIC. Upon completion of all the simulation tasks, the output data are aggregated as required by the Analyzer, and presented to the user.

5.1 The mJADES Manager cloudlet

The `mJADES Manager` cloudlet coordinates the execution of the distributed simulation. It receives simulation job requests from its users in the form of messages, which may come from the HTTP interface or from other components. Requests are translated into jobs, which

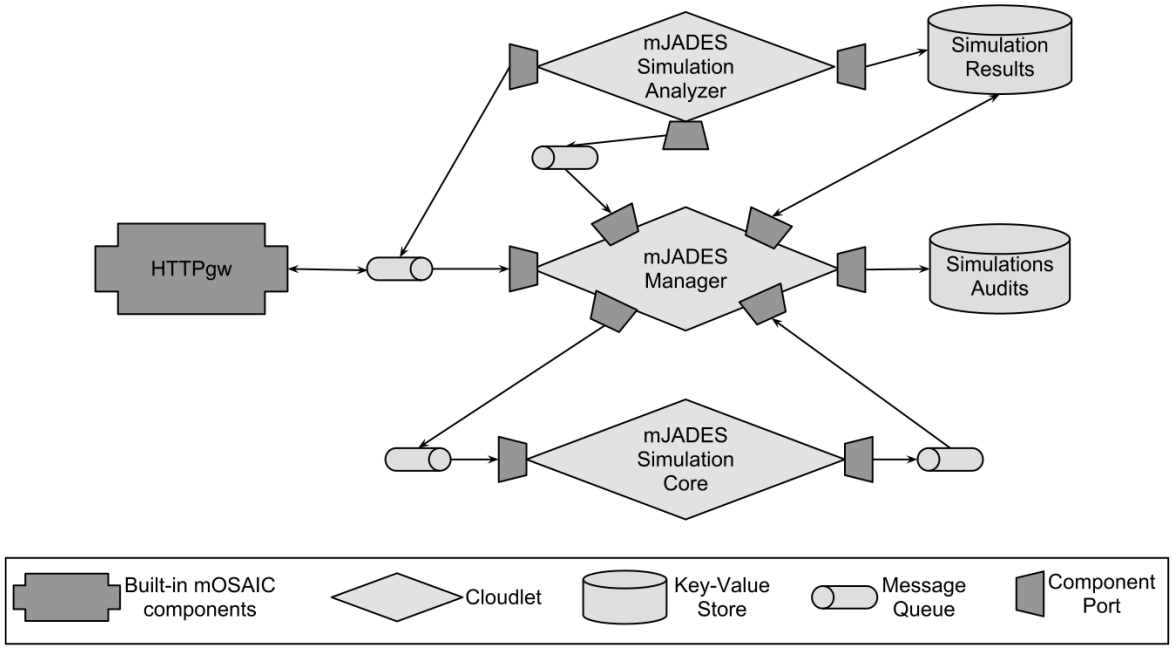

Figure 2: mJADES Architecture

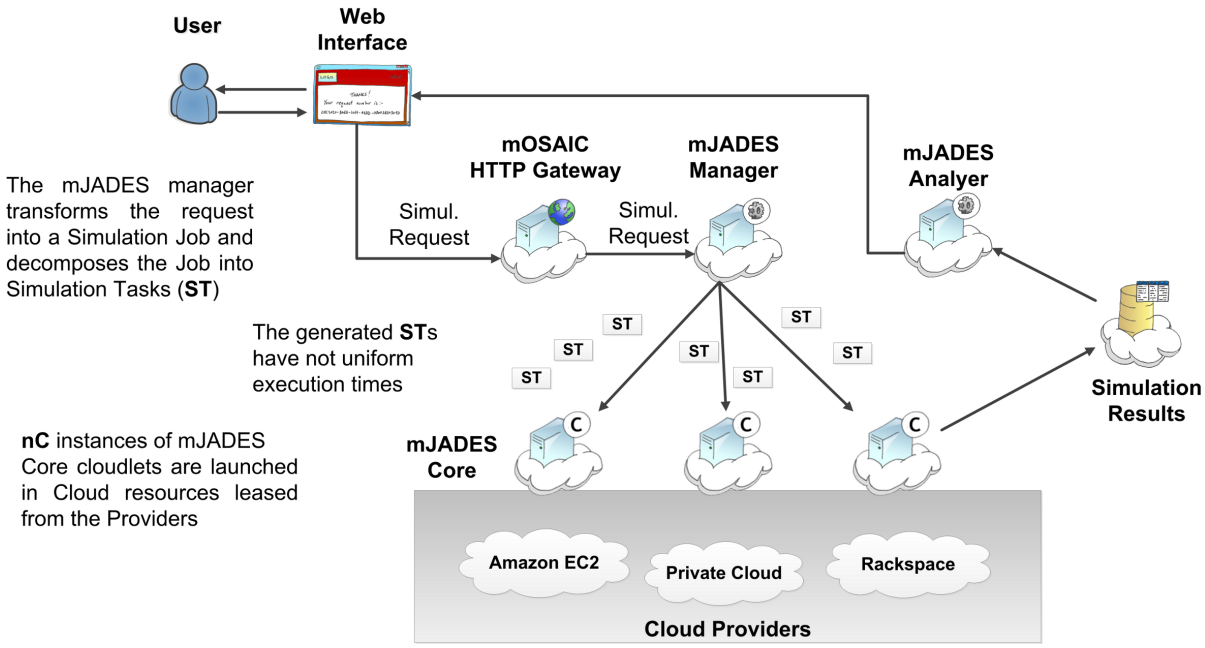

Figure 3: mJADES Workflow

receive a global identification key to keep track of the job metadata. Metadata include job status (maintained in the Simulation Audits Key-Value Store) and its outputs (that will be grouped in the Simulation Results). Then the simulation jobs are decomposed into the set of simulations tasks that must be performed. These are dispatched to multiple instances of the Simulation Core (described next), which perform the actual model evaluation required. The key function of the manager is the decomposition of the job into simulation tasks. Currently, the manager supports two configuration modes to carry out this decomposition:

- the job can specify a *number of iterations* of the same simulation task to be performed, until some statistical property is met. Iterations are transformed into *replications*, in that multiple instances of the task are launched in parallel;
- the job can require a *parameter sweep* over one or several simulation parameters. A simulation task is generated for each distinct parameter tuple.

In the current implementation, the simulation tasks are managed statically, in that all the tasks launched are completed independently of the simulation results (which may show, for example, that further simulations are not necessary because the process has already converged). Smarter task management will be added in the future versions of mJADES. We are also studying algorithms to speed up the parameter sweep, by cutting the solution space to be explored through the use of meta-heuristics, including classical algorithms like tabu search, simulated annealing and more recent bio-inspired algorithms as the one described in (Di Stefano and Morana, 2012).

As tasks are generated, they become part of the job metadata and receive an identification key which, together with the job identifier, allows all the cloudlets to maintain and to update the global state through the dedicate Key-Value Store. The task status is fed to the Simulation Audits, while the task output goes to the Simulation Results store.

When all the tasks have been completed, the Manager launches the Simulation Analyzer cloudlet. This operates on the raw simulation data and computes the final output to be presented to users. The simulation analyzer accepts as input a message containing the job key, so that it can retrieve the list of all the associated tasks.

5.2 The mJADES Simulation Core cloudlet

The `mJADES Simulation Core` cloudlet builds up a simulation engine exploiting the JADES (JAva Discrete Event Simulator) library described in Section 3. The Core executes a single simulation task and generates a simulation report, which is sent out on a dedicated output queue. It should be noted that, thanks to the mOSAIC approach, the process of launching multiple simulation tasks is made totally transparent for the user. The cloudlet container adapts automatically the number of cloudlet instances to the number of simulation requests, in order to attain high performance, possibly acquiring new resources from the cloud. As shown in the bottom of Figure 2, the Simulation Core cloudlet has just two connectors to the input and output queues. These connectors allow the cloudlet to be awaken on message arrivals, realizing the asynchronous behaviour typical of mOSAIC. Messages contain the simulation description in JavaScript Object Notation (JSON), a lightweight, text-based, language-independent data interchange format (Crockford, 2006). The Core cloudlet decodes the message content and starts up the requested simulation model. The simulation output, in the form of simulation logs, is returned as a JSON object, which is sent on the output message queue to be handled by the Manager, stored in the Simulation Results store and processed by the Analyzer.

5.3 The JADES Simulation Analyzer cloudlet

The third cloudlet, `Simulation Analyzer`, is dedicated to the analysis of the simulation results. It receives analysis requests from the Manager containing the ID of the job to be analyzed. The Analyzer retrieves the simulation task results for the job from the Simulation Results Key-Store and performs the required analysis on them. In the current version, the Analyzer is able to perform statistical analysis and to generate graphical reports. Statistical analysis of the results include the calculation of key indicators, sensitivity analysis and summary statistics. The graphical reports make it possible to choose between several kinds of charts and graphs, like bar charts, histograms and XY Plot. Simulation results can be plotted against the input parameters of the model, the different simulation runs or model specific variables. The simulation analyzer is able to send out messages to the HTTP gateway queue in order to provide the analysis results to the final users.

6 Example: the Simulation of MPI Applications

To show the complete workflow of the simulation engine, this section will present a case study that has been developed on the platform. This is the evaluation of a trace-driven simulation model that predicts the execution time of a message-passing application running on a distributed system, using the MPI programming interface (Gropp et al., 1999) for inter-task communications. An MPI application is typically structured as an SPMD (Single Program, Multiple Data) program. Given a single executable program, MPI spawns a number of tasks on multiple Processing Elements (CPUs or cores) that reside on the nodes composing the system. Every task executes the same program, but typically tasks take different

execution paths. Tasks are able to communicate with each other through the exchange of messages.

In trace-driven simulation, *traces* obtained from previous experimentation are fed as input to the simulator. In the case of MPI applications, traces recording relevant computation and communication events (as measured in previous runs) can be used to predict how the application will behave under different settings. The process-oriented paradigm promoted by JADES (and its execution through mJADES) is a natural match for the description (and evaluation) of simulation models of MPI applications.

The case study presented below studies the run times of a popular benchmark application, the NAS Parallel Benchmark LU (Bailey et al., 1995) varying the total number of processes. The program implementing the simulation model was developed in Java, using the JADES constructs presented in Section 3. The code skeleton is shown in Listing 1. Firstly the file containing the configuration of the execution platform and the trace files collected from an execution on another platform are read (excerpts of the files are visible in Figure 4, presented later). Then a "launcher" process is started, which creates a TraceDrivenProcess for every MPI process to be simulated and waits for the end of the simulation. The TraceDrivenProcess, shown in Listing 2, models a real MPI process by reading computation and communication events from the trace, and simulating their execution, as follows:

- computation is modeled by use of JADES *facilities*, which represent processing elements (CPUs);
- communication is modeled through JADES *mailboxes*, and communication time is computed by means of a simple linear model based on latency and bandwidth.

Listing 1: Simulation Code Skeleton

```
import jades.Model;
public class TraceDrivenModel extends Model{
  private Trace[] traces;
  public void run(){
    ... //read traces from trace file
    ... //read platform configuration from
        //file, creating cpus and mailboxes
    start(new LauncherProcess());

  }

  private class LauncherProcess{

  public void run(){
    for (int i = 0; i < numProcesses; i++){
      TraceDrivenProcess p = new
          TraceDrivenProcess(traces[i]);
      add(p) //add process to simulation
    }
    //wait on the predefined event that
    //signals the end of the simulation
    this.scheduler.getEventListEmpty().
        untimed_wait();
  }
}
```

Listing 2: Trace-driven Simulation Process

```
import jades.Process;
import jades.Facility;
import jades.Mailbox;
public class TraceDrivenProcess extends
    Process{
  private Trace t;
  private Facility cpu;
  private Mailbox incomingMsgs;

  public void run(){
    while(t.hasMoreEvents){
      Event e = t.nextEvent();
      if (e instanceof ComputeEvent){
        ComputeEvent ce = (ComputeEvent) e;
        cpu.reserve(ce.getDuration());
      }
      else if (e instanceof SendEvent){
        SendEvent se = (SendEvent) e;
        Mailbox m = launcher.
            getMailboxForProcess(se.
            getMessage().getDestination());
        m.send(se.getMessage());
        this.hold(se.getMessage().getSize*
            NET_BANDWIDTH + NET_LATENCY);
      }
      ...//code goes on with a branch for
         //every kind of event. Each MPI
         //primitive has an associated event
    }
  }
}
```

After "replaying" the process execution, the simulation code will output statistics on the simulated program execution. These include CPU usage (for each available CPU), time spent in communication and the total execution time of the simulated application. The inputs to the model include the number of processes to be simulated, the description of machines and networks, a machinefile which specifies the mapping processes-to-machines and the computation/communication traces mentioned above. In this context, simulation can be very useful to predict the effect of variations of run-time settings on application performance figures. This involves running multiple simulations under different settings.

The user can issue a simulation request through the web interface shown in Figure 4, which has fields for every configuration item needed. In the case of our experiment, the number of processes is the sweep parameter that varied over the range from 2 to 32. Through the graphical interface, the user can supply the hosts/machine/trace files, and ask the analyzer to produce a plot of total execution time vs. number of processes (the swept parameter).

When the user submits the simulation job request, an XML-based request message is sent to the manager (Listing 3). This message contains information on the sweep parameter(s), the simulation configuration and asks for the production of the final plot. The `sweep` element is used by the Manager to generate multiple simulation configuration objects, each of which is enriched with a specific value of `paramName`. Each

generated configuration object is translated into JSON (Listing 4), and sent to the input queue of the Core, to be evaluated as an independent simulation task.

In our case study, `numProcesses` is the parameter that will be set in the configuration object. This determines the number of MPI processes that will be simulated by each specific task. As simulation tasks are executed, their output is sent to the input queue of the Analyzer in the JSON format shown in Listing 5. Once the Analyzer has received all the results, it produces the required plot and sends it to the web interface. The output of the LU application simulation is shown in Figure 5. The Analyzer can optionally output raw simulation data. These make it possible to evaluate the simulation error for our example test, by comparing them to actual application execution times measured on the real hardware. Table 1 shows the simulation errors we obtained for the LU benchmark.

Let us now consider the advantages linked to the use of the mJADES platform. Table 2 shows the time needed to run a single (monolithic) simulation of 2 to 32 MPI processes. It should be noted that simulation time grows very rapidly as the number of processes rises. As response times are highly variable, the assignment of multiple simulation tasks to multiple cloudlets is not trivial, as the possible load unbalance may reduce the positive effect on performance of concurrent simulation.

Table 3 presents the response times obtained running the same set of simulations, using mJADES in parameter-sweep mode, *nProcesses* = 2 to 32. There are significant response time reductions, even if the simulation time cannot be less than the response time of the biggest task (*nProcesses* = 32, 342 s.). As can be seen in the table, using a single Core Cloudlet (CC), i.e., a single simulation execution unit, there is a performance loss of 8.57% as compared to the "traditional" execution of a sequential simulation. This is due to the mOSAIC overhead, introduced by the task distribution mechanism. Using two Core Cloudlets (third row of Table 3), we can observe a reduction in execution time of about 25%. Adding more cloudlets brings no additional benefits, as we are very close to the lower bound given by the execution time of the 32-processes simulation task.

Let us suppose now that every simulation task must be evaluated a multiple number of times. This is not uncommon, as multiple simulation runs are required for statistical validation. A higher number of tasks to be distributed to the cloudlets may indeed improve load balance. In Table 4, we see the response times when the parameter sweep over the number of MPI processes (in this case, just for *nProcesses* = 2 to 16, due to memory constraints) is repeated five times. As was to be expected, the execution with one Core Cloudlet always brings a performance loss, because of the overhead for task distribution. Increasing the number of Core Cloudlets leads to more significant performance gains, up to a reduction of over 70% of the sequential execution time, using six Core Cloudlets.

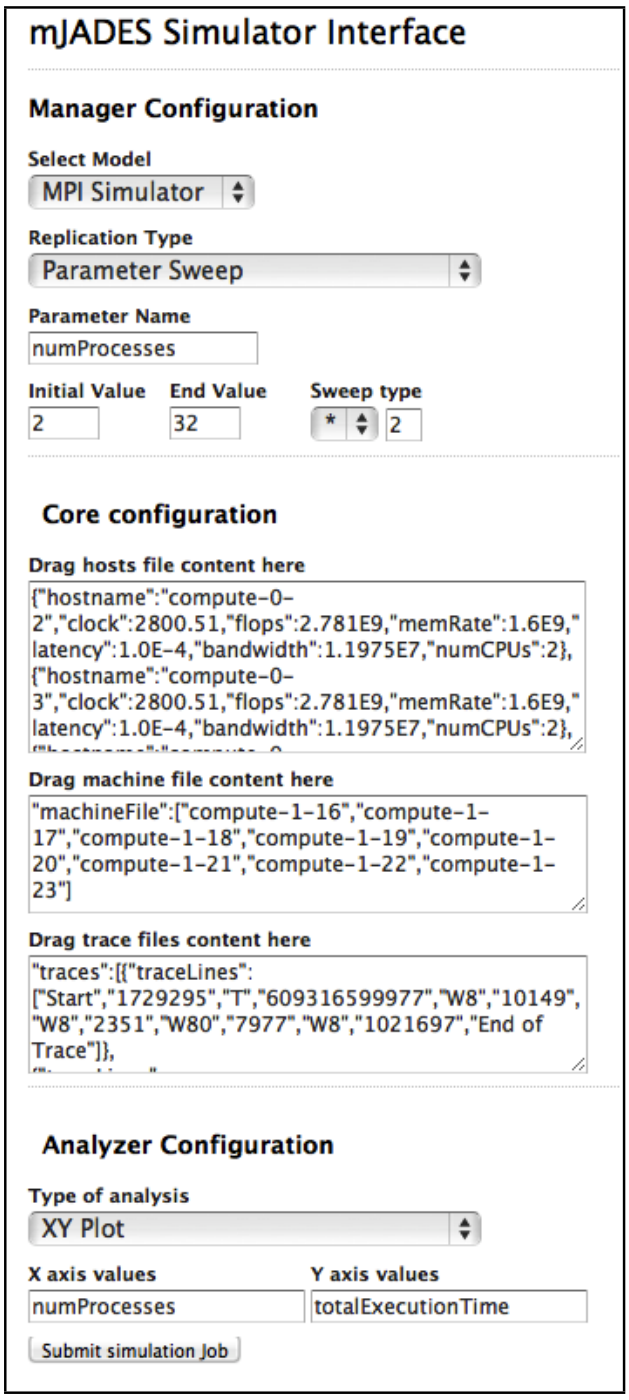

Figure 4: Screenshot of the mJADES web interface

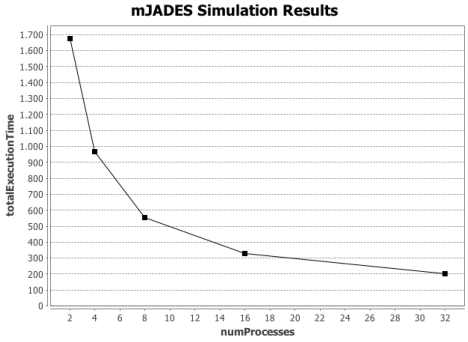

Figure 5: Output of the LU Benchmark mJADES simulation

Listing 3: XML Simulation Job Request

```
<sweep>
  <paramName> numProcesses</paramName>
  <start>2</start>
  <end>32</end>
  <sweepOp>mult</sweepOp>
  <factor>2</factor>
</sweep>

<configuration>
  ...see JSON Listing 2
</configuration>
<plot>
  <type>lineChart</type>
  <xSeries>numProcesses</xSeries>
  <ySeries>totalExecutionTime</ySeries>
</plot>
```

Listing 4: JSON Configuration Message

```
'config':
{
  'numProcesses': <number>,
  'machinefile': ['<string>],
  'hosts' : [ {
      'hostname': <string>,
      'nCPUs': <number>,
      'clockFreqMHZ': <number>,
      'FLOPS' : <number>,
      'memBandwidth' : <number>,
      'netLatency' : <number>,
      'netBandwidth'  : <number>
    }*]
  'traces': [{
      'traceName': <string>,
      'traceString': <string>
    }]
}
```

Listing 5: JSON Result Message

```
{
  'simulationJobID': <string>
  'simulationTaskID': <string>
  'CPUUsage': [<number>]
  'commTime': [<number>]
  'totalExecutionTime': <number>
}
```

Table 1 Simulation error for the LU Benchmark

Benchmark Execution time			
numProcesses	Measured (s)	Predicted (s)	Error
2	1729.31	1673.60	-3.22%
4	928.30	965.79	4.03%
8	544.68	572.85	5.17%
16	300.14	328.45	9.43 %
32	189.08	203.20	7.47 %

Table 2 Simulation time vs. number of simulated processes

nProcesses	Simulation Time
2	1.41
4	7.30
8	31.94
16	132.66
32	342.00

Table 3 Simulation times for simple parameter sweep

Parameter Sweep (nProcesses = 2...32)		
Exec. Platform	Time (s)	Perf. Gain (Loss)
Sequential Execution	516.01	N.A.
mJADES - 1 CC	560.23	(8.57%)
mJADES - 2 CC	386.25	25.15%
mJADES - 3 CC	387.16	24.97%

Table 4 Simulation times for parameter sweep plus replication

Parameter Sweep + Replication (nProcesses = 2...16, nReps=5)		
Exec. Platform	Time (s)	Perf. Gain (Loss)
Sequential Execution	906.53	N.A.
mJADES - 1 CC	1103.25	(21.70%)
mJADES - 2 CC	592.38	34.65%
mJADES - 3 CC	425.54	53.09%
mJADES - 4 CC	366.43	59.58%
mJADES - 5 CC	297.82	67.15%
mJADES - 6 CC	269.91	70.23%
mJADES - 7 CC	265.97	70.66%

7 Conclusions and Future Work

In this paper we have presented a software architecture that makes it possible to deploy concurrent simulations in the cloud. It relies on two existing technologies, mOSAIC and JADES, for the cloud and discrete-event simulation support, respectively. Their integration can be very useful for performing complex, repetitive simulation tasks. Initial results on real simulation models show performance benefits with respect to sequential evaluation. Work is ongoing to evaluate in detail the performance and the scalability of the system under variable load. The proposed architecture opens up a wide field of research opportunities. For example, the simulation manager, used by mJADES to manage the set of simulations to be launched, is a versatile and extensible component. We plan to make it smarter by adding intelligent behavior and optimization capabilities that can reduce the number of simulation tasks that must be evaluated without compromising the accuracy of the results. Another ongoing research objective is the automatic generation of user interfaces from the model specification.

References

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. (1995). The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center.

Byrne, J., Heavey, C., and Byrne, P. (2010). A review of web-based simulation and supporting tools. *Simulation modelling practice and theory*, 18(3):253–276.

Castilla, I., Aguilar, R. M., and Callero, Y. (2011). Java for parallel discrete event simulation: a survey. *International Journal of Simulation and Process Modelling*, 6(4):250–260.

Chandy, K. and Misra, J. (1979). Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452.

Crockford, D. (2006). RFC 4627: The application/json media type for javascript object notation (json). Technical report, IETF.

Cuomo, A., Rak, M., and Villano, U. (2012). Process-oriented discrete-event simulation in Java with continuations:quantitative performance evaluation. In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH) - Rome, 28-31 July 2012*. To appear.

D'Angelo, G. (2011). Parallel and distributed simulation from many cores to the public cloud. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 14–23. IEEE.

Di Stefano, A. and Morana, G. (2012). A bio-inspired distributed algorithm to improve scheduling performance of multi-broker grids. *Natural Computing*, pages 1–14. 10.1007/s11047-012-9319-8.

Doğan, A. (2009). A study on performance of dynamic file replication algorithms for real-time file access in data grids. *Future Generation Computer Systems*, 25(8):829–839.

Edwards, G. and Sankar, R. (1992). Modeling and simulation of networks using csim. *Simulation*, 58(2):131–136.

Fujimoto, R., Malik, A., and Park, A. (2010). Parallel and distributed simulation in the cloud. *SCS M&S Magazine*, 3.

Fujimoto, R. M. (1990). Parallel discrete event simulation. *Commun. ACM*, 33:30–53.

Fujimoto, R. M. (2003). Distributed simulation systems. In *Proceedings of the 35th conference on Winter simulation: driving innovation*, WSC '03, pages 124–134. Winter Simulation Conference.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message passing interface*, volume 1. MIT press.

Guo, S., Bai, F., and Hu, X. (2011). Simulation software as a service and Service-Oriented simulation experiment. In *Information Reuse and Integration (IRI), 2011 IEEE International Conference on*, pages 113–116. IEEE.

Healy, K. J. and Kilgore, R. A. (1998). Introduction to SILK and Java-based simulation. In *Proceedings of the 30th conference on Winter simulation*, WSC '98, pages 327–334, Los Alamitos, CA, USA. IEEE Computer Society Press.

Heidelberger, P. (1986). Statistical analysis of parallel simulations. In *Proceedings of the 18th conference on Winter simulation*, WSC '86, pages 290–295, New York, NY, USA. ACM.

Howell, F. and McNab, R. (1998). simjava: a discrete event simulation package for java with applications in computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*.

Jacobs, P., Lang, N., and Verbraeck, A. (2002). D-SOL; a distributed Java based discrete event simulation architecture. In *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, pages 793–800. Winter Simulation Conference.

Jacobs, P. and Verbraeck, A. (2004). Single-threaded specification of process-interaction formalism in Java. In *Proceedings of the 36th conference on Winter simulation*, pages 1548–1555. Winter Simulation Conference.

Jefferson, D. and Sowizral, H. (1985). Fast concurrent simulation using the Time Warp mechanism. In *SCS Conf. Distributed Simulation*, pages 63–69.

Kunert, A. (2008). Optimistic parallel Process-Oriented DES in Java using Bytecode Rewriting. In *Proc. of MESM 2008*, pages 15–21.

Lechler, T. and Page, B. (1999). *DESMO-J: An object oriented discrete simulation framework in Java*.

L'Ecuyer, P. and Buist, E. (2005). Simulation in Java with SSJ. In *Proceedings of the 37th conference on Winter simulation*, pages 611–620. Winter Simulation Conference.

L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002). SSJ: a framework for stochastic simulation in Java. In *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, pages 234–242. Winter Simulation Conference.

Mancini, E., Rak, M., and Villano, U. (2009). Perfcloud: GRID services for performance-oriented development of cloud computing applications. In *Proceedings of the 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*, pages 201–206. IEEE.

Marlin, C. (1980). *Coroutines: A Programming Methodology, a Language Design and an Implementation*, volume 95 of *Lecture notes in computer science*. Springer.

Marr, C., Storey, C., Biles, W., and Kleijnen, J. (2000). A Java-based simulation manager for Web-based simulation. In *Simulation Conference, 2000. Proceedings. Winter*, volume 2, pages 1815 –1822 vol.2.

Messina, F., Pappalardo, G., and Santoro, C. (2012). ComplexSim: an SMP-aware complex network simulation framework. In *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 861–866. IEEE.

Miller, J., Nair, R., Zhang, Z., and Zhao, H. (1997). JSIM: A Java-based simulation and animation environment. In *Simulation Symposium, 1997. Proceedings. 30th Annual*, pages 31–42. IEEE.

Ortega-Ruiz, J., Curdt, T., and Ametller-Esquerra, J. (2004). Continuation-based mobile agent migration. Retrieved 10 October 2011.

Panda, D., Basak, D., Dai, D., Kesavan, R., Sivaram, R., Banikazemi, M., and Moorthy, V. (1997). Simulation of modern parallel systems: a csim-based approach. In *Proceedings of the 29th conference on Winter simulation*, pages 1013–1020. IEEE Computer Society.

Park, A. and Fujimoto, R. M. (2009). Parallel discrete event simulation on desktop grid computing infrastructures. *International Journal of Simulation and Process Modelling*, 5(2):157–171.

Pelkey, J. and Riley, G. (2011). Distributed Simulation with MPI in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, pages 410–414. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Perumalla, K. (2006). Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 38th conference on Winter simulation*, pages 84–95. Winter Simulation Conference.

Petcu, D., Craciun, C., Neagul, M., Lazcanotegui, I., and Rak, M. (2011a). Building an interoperability API for sky computing. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 405–411. IEEE.

Petcu, D., Craciun, C., and Rak, M. (2011b). Towards a cross platform Cloud API. Components for Cloud Federation. In *Procs. 1st International Conference on Cloud Computing and Services Science, SciTePress–Science and Technology Publications, Portugal*, pages 166–169.

Petcu, D., Macariu, G., Panica, S., and Craciun, C. (2012). Portable cloud applicationsfrom theory to practice. *Future Generation Computer Systems*, (0):–.

Peter Mell and Tim Grance (2009). The NIST Definition of Cloud Computing.

Rak, M., Aversa, R., Venticinque, S., and Martino, B. D. (2011). User Centric Service Level Management in mOSAIC Applications. In Alexander, M., D'Ambra, P., Belloum, A., Bosilca, G., Cannataro, M., Danelutto, M., Martino, B. D., Gerndt, M., Jeannot, E., Namyst, R., Roman, J., Scott, S. L., Träff, J. L., Vallée, G., and Weidendorfer, J., editors, *Euro-Par Workshops (2)*, volume 7156 of *Lecture Notes in Computer Science*, pages 106–115. Springer.

Rossetti, M. D. (2008). Java simulation library (JSL): an open-source object-oriented library for discrete-event simulation in Java. *International Journal of Simulation and Process Modelling*, 4(1):69–87.

Sargent, R. (2000). Verification, validation, and accreditation: verification, validation, and accreditation of simulation models. In *Proceedings of the 32nd conference on Winter simulation*, pages 50–59. Society for Computer Simulation International.

Schwetman, H. (1988). Using csim to model complex systems. In *Proceedings of the 20th conference on Winter simulation*, pages 246–253. ACM.

Schwetman, H. (2001). CSIM19: a powerful tool for building system models. In *Proc. of the 33nd Winter Simulation Conference*, pages 250–255. IEEE.

Seo, C., Han, Y., Lee, H., Jung, J., and Lee, C. (2010). Implementation of Cloud Computing Environment for Discrete Event System Simulation using Service Oriented Architecture. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 359–362. IEEE.

Taylor, S., Ghorbani, M., Kiss, T., Farkas, D., Mustafee, N., Kite, S., Turner, S., and Strassburger, S. (2011). Distributed Computing and Modeling & Simulation: Speeding up Simulations and Creating Large Models. In *Proceedings of the 2011 Winter Simulation Conference*.

Tsai, W., Li, W., Sarjoughian, H., and Shao, Q. (2011). SimSaaS: simulation software-as-a-service. In *Proceedings of the 44th Annual Simulation Symposium*, pages 77–86. Society for Computer Simulation International.

Tusa, F., Paone, M., Villari, M., and Puliafito, A. (2010). CLEVER: A cloud-enabled virtual environment. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 477 –482.

von Laszewski, G., Diaz, J., Wang, F., and Fox, G. (2012). Comparison of multiple cloud frameworks. *Science*, 15:3.

Weatherly, R. and Page, E. (2004). Efficient process interaction simulation in Java: Implementing co-routines within a single Java thread. In *Proceedings of the 36th conference on Winter simulation*, pages 1437–1443. Winter Simulation Conference.

Weiland, W., Weatherly, R., Ring, K., Page, E., Mikula, R., and Kuhl, F. (2005). Simplified Concurrency: A Java Simulation Framework.