PyDES, a performance modeling showcase

Giacomo Marciani
University of Rome Tor Vergata
Via del Politecnico 1
Rome, Italy 00133
gmarciani@acm.org

ABSTRACT

As computing is getting more ubiquitous in our lives, computer infrastructures are getting increasingly complex and software applications are required to meet high level performance.

In this context, knowing how to design and optimize computer systems and networks is one of the most important skills for software engineers and a strategic asset for companies, both in terms of technology and investments.

In this technical report we propose a next-event simulator to analyze the performance of a two-layers Fog-like system that serves classed workloads and leverages an off-loading policy between its layers. First, we describe how we implemented the multi-stream pseudo-random number generator, that is the fundamental building block to provide any next-event simulator with random components. Then, we describe the performance model in terms of (i) goals, (ii) conceptual model, (iii) specification model, (iv) computational model, (v) verification and (vi) validation. At the end, we evaluate the quality of randomization and conduct the performance analysis of the target system leveraging our simulator.

The experimental results show (i) the satisfactory randomness degree of the adopted pseudo-random number generator and (ii) the effectiveness of our model to study the system so as to, for example, tune it in order to achieve better performance. Although the promising results, we conclude our work delineating possible improvements for our model.

CCS CONCEPTS

• **Networks** → **Network simulations**; **Network performance analysis**; • **Theory of computation** → **Random walks and Markov chains**;

KEYWORDS

performance modeling; simulation tools

ACM Reference format:
Giacomo Marciani. 2018. PyDES, a performance modeling showcase. In *Proceedings of PyDES, a performance modeling showcase, Rome, Italy, September 2018 (PMCSN'18),* 11 pages.
https://doi.org/10.475/123_4

PMCSN'18, September 2018, Rome, Italy

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

1 INTRODUCTION

As computing is getting more ubiquitous in our lives, computer infrastructures are getting increasingly complex and software applications are required to meet high level performance.

In this context, knowing how to design and optimize computer systems and networks is one of the most important skills for software engineers and a strategic asset for companies, both in terms of technology and investments. In this technical report we propose a next-event simulator to analyze the performance of a two-layers Fog-like system that serves classed workloads and leverages an off-loading policy between its layers.

The remainder of the paper is organized as follows. In Section 2 we give an high level description of the target system. In Section 3 we describe the pseudo-random number generator adopted to generate random variates for the next-event simulation model. In Section 4 we describe the the next-event simulation model in terms of goals, conceptual model, specification model, computational model, verification and validation. In Section 5 we show the experimental results about both the randomness of the adopted pseudo-random number generator and the performance analysis of the target system conducted leveraging our simulator. In Section 6 we show how to configure and run experiments and give some sample outputs to provide a better idea of what has been created. In Section 7 we conclude the paper summing up the work that has been done and delineating future improvements.

2 SYSTEM

In this section we give an high level description of the target system.

We consider the environment in Figure 1, which is characterized by:

- **workload:** mobile devices send to the system tasks partitioned in two classes.
- **system:** a two-layers Fog-like system, made of:
  - **Cloudlet:** upfront layer made of one-hop finite resources, having the ability to off-load tasks to the Cloud server, accordingly to an *off-loading policy* based on the occupancy state of the Cloudlet. In particular, the Cloudlet may *forward* incoming tasks to Cloud or *restart* preempted tasks in Cloud with some *overhead.*
  - **Cloud:** backfront layer made of a remote Cloud server with virtually unlimited resources.

We assume that (i) the Cloudlet provides tasks with higher service rate than the Cloud, (ii) when a task is interrupted in the Cloudlet and it is sent to the Cloud, the restart process comes with a *setup time overhead.*

Such a system can be considered very actual nowadays. In fact, it sketches the typical asset of a simple Fog Computing solution.

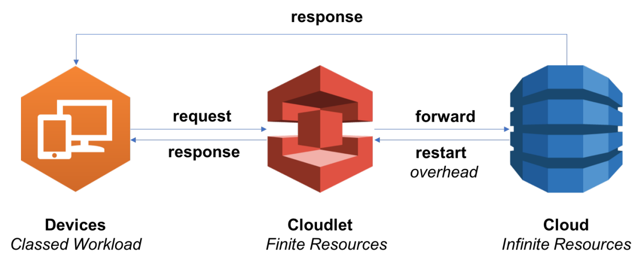

Figure 1: System architecture (high level).

3 RANDOM NUMBER GENERATION

The generation of pseudo-random numbers is a fundamental building-block in any next-event simulation. In fact, a sequence of pseudo-random numbers uniformly distributed in $(0, 1)$ can be used to generate stochastic variates, e.g. the exponential distribution, that can be leveraged to generate streams of random events, e.g. requests to the system with random occurrence time and computational demand. There exist many techniques for random number generation, a lot of which are comprehensively presented in [3]. The most notable algorithmic generators are *linear congruential generators*, *multiple recursive generators*, *composite generators*, and *shift-register generators*.

In this work we adopted a custom implementation of a multi-stream Lehmer generator (a, m, s), which belongs to the family of linear congruential generators and it is defined by the following equation:

$$x_{i+1} = (a^j \mathrm{mod} m) x_i \mathrm{mod} m \qquad \forall j = 0, \ldots, s-1 \tag{1}$$

where m is the modulus, a is the multiplier, s is the number of streams and $(a^j \mathrm{mod} m)$ is the jump multiplier.

We have chosen this solution because (i) it provides a great degree of randomness with the appropriate parameters (ii) the multi-streaming is required by simulations with multiple stochastic components, (iii) it has a simple implementation and a smaller computational complexity with respect to others, and (iv) it is a de-facto standard, hence it is easy to compare our experimental results with the ones provided in literature.

We propose a generator with the following parameters:

- **modulus $2^{31} - 1$:** the modulus should be the maximum prime number that can be represented in the target system. Although all modern computers have a 64-bit architecture, we considered a 32-bit one because the algorithm to find the right multiplier for a 64-bit modulus can be very slow. For this reason we have chosen $2^{31} - 1$ as our modulus.
- **multiplier 50812:** the multiplier should be *full-period modulus-compatible* with respect to the chosen modulus. The chosen modulus has 23093 of such multipliers. Among these there are also multipliers such 16807, widely used in the past, and 48271, that is currently the most widely adopted. We have chosen 50812 as our multiplier because we wanted to study a suitable multiplier that is different from the de-facto standard.
- **256 streams:** the original periodic random sequence can be partitioned in different disjoint periodic random subsequences, one for each stream. The number of streams should be no more than the number of required disjoint subsequences, because streams come with the cost of reducing the size of the random sequence. We have chosen 256 streams, that is a lot more than the strictly required for our simulations, because it is a de-facto standard hence it is useful for comparisons between our evaluation and the one proposed in literature [4].
- **jump multiplier 29872:** the jump multiplier is used to partition the random sequence in disjoint subsequences, one for each stream, whose length is often called jump size. The jump multiplier should be *modulus compatible* with the chosen modulus. We have chosen 29872 as our jump multiplier because it is the value that maximizes the jump size.
- **initial seed 123456789:** the initial seed is the starting point of the finite sequence of generated values. Even if the initial seed does not impact the randomness degree of a generator in a single run (it only has to be changed in different replication of the same ensemble), we decide to indicate it here for completeness.

The randomness degree of such a generator has been assessed by the usage of *spectral test*, *test of extremes* and the *analysis of Kolomogorv-Smirnov*. The experimental results are reported in Section 5.

4 PERFORMANCE MODELING

In this section we describe the performance model used to analyze the target system. We will follow the widely adopted modeling approach suggested in [4], which consists in (i) goals and objectives (ii) conceptual model (iii) specification model (iv) computational model (v) verification and (vi) validation.

4.1 Goals and Objectives

The main goals of simulation are about system tuning. In particular, we propose to determine with a 95% level of confidence

- the response time as a function of the threshold S,
- the throughput as a function of the threshold S,
- the distribution of the response time when $S = N$ and
- the threshold of the off-loading policy that minimizes the response time.

4.2 Conceptual Model

The conceptual model of the target system is depicted in Figure 2.

State space. The state space S of a system is a comprehensive characterization of the system. Each state $s \in S$ is a comprehensive characterization of the system in a given instant of time. The state space of the whole system is represented by the state space of its subsystems:

- Cloudlet: $S_{clt} := \{(n_{clt,1}, n_{clt,2}) \in \mathcal{N}^2 : n_{clt,1} + n_{clt,2} < N\}$, where $n_{clt,j}$ is the population of tasks belonging to the j-th class within the Cloudlet.

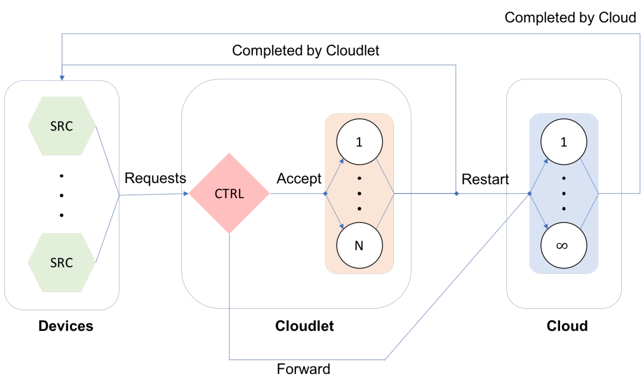

Figure 2: Conceptual model.

- Cloud: $S_{cld} := \{(n_{cld,1}, n_{cld,2}) \in \mathcal{N}^2\}$, where $n_{cld,j}$ is the population of tasks belonging to the j-th class within the Cloud.

Events space. An event is an occurrence that could change the state of the system at the event time, according to the event type. We consider the following events:

- $A_{clt,j}$: a task belonging to the j-th class arrives to the Cloudlet.
- $A_{cld,j}$: a task belonging to the j-th class arrives to the Cloud.
- $C_{clt,j}$: a task belonging to the j-th class is completed by the Cloudlet.
- $C_{cld,j}$: a task belonging to the j-th class is completed by the Cloud.
- R_2: a task belonging to the 2^{nd} class is stopped in the Cloudlet and restarted in the Cloud.

4.3 Specification Model

Statistical specifications. Tasks belonging to the j-th class arrive to the system according to an exponential arrival process with rate λ_j. The Cloudlet serves tasks belonging to the j-th class according to an exponential service process with rate $\mu_{clt,j}$; the Cloud serves tasks belonging to the j-th class according to an exponential service process with rate $\mu_{cld,j}$. We assume that (i) $\mu_{clt,i} > \mu_{cld,i}\ \forall i = 1,2$ and (ii) the setup time T_{setup} is exponentially distributed with expected value $E[T_{setup}]$.

In particular, we consider values shown in Equations 2.

$$\begin{aligned} \lambda_1 &= 6.00\ tasks/sec \\ \lambda_2 &= 6.25\ tasks/sec \\ \mu_{clt,1} &= 0.45\ tasks/sec \\ \mu_{clt,2} &= 0.27\ tasks/sec \\ \mu_{cld,1} &= 0.25\ tasks/sec \\ \mu_{cld,2} &= 0.22\ tasks/sec \\ E[T_{setup}] &= 0.8\ sec \end{aligned} \tag{2}$$

Algorithmic specifications. The *off-loading policy* implemented by the *Cloudlet controller (CTRL)* is defined in Algorithm 1

```
if task of class 1 then
    if n_clt = N then
        forward to Cloud
    end
    if n_clt + n_cld < S then
        accept
    end
    if n_cld > 0 then
        accept on Cloudlet and restart a class 2 task to Cloud
    else
        accept on Cloudlet
    end
end
if arrival of class 2 then
    if n_clt + n_cld >= S then
        forward to Cloud
    else
        accept on Cloudlet
    end
end
```

Algorithm 1: Off-loading policy.

4.4 Analytical Model

The *analytical model* is depicted in Figure 3, whose *routing probabilities* are defined in Equation 6. The definition of routing probabilities relies on the following subsets of states $S_{clt,i} \subset S_{cld}$:

- $S_{clt,1}$: a task belonging to the 1^{st} class is accepted in the Cloudlet.

$$S_{clt,1} := \{(n_{clt,1}, n_{clt,2}) \in S_{clt} : n_{clt,1} + n_{clt,2} < N \vee n_{clt,2} > 0\} \tag{3}$$

- $S_{clt,2}$: a task belonging to the 2^{nd} class is accepted in the Cloudlet.

$$S_{clt,2} := \{(n_{clt,1}, n_{clt,2}) \in S_{clt} : n_{clt,1} + n_{clt,2} < N \wedge n_{clt,2} < S\} \tag{4}$$

- $S_{clt,3}$: a task belonging to the 2^{nd} class is restarted in the Cloud.

$$S_{clt,3} := \{(n_{clt,1}, n_{clt,2}) \in S_{clt} : n_{clt,1} + n_{clt,2} = N \wedge n_{clt,2} > 0\} \tag{5}$$

$$\begin{aligned} a_{clt,1} &= \sum_{s \in S_{clt,1}} \pi_s \\ a_{clt,2} &= \sum_{s \in S_{clt,2}} \pi_s \\ r_{clt,2} &= \sum_{s \in S_{clt,3}} \pi_s \left(\frac{\lambda_2}{\lambda_1 + \lambda_2} \right) \end{aligned} \tag{6}$$

Markov Chain. Assuming Poisson arrivals and exponential services, we can determine the Markov Chain whose resolution allows us to compute the routing probabilities shown in Equation 6.

In Figure 7 we show the Markov Chain with the associated flow balance equations listed in Equation 7. For sake of simplicity, we

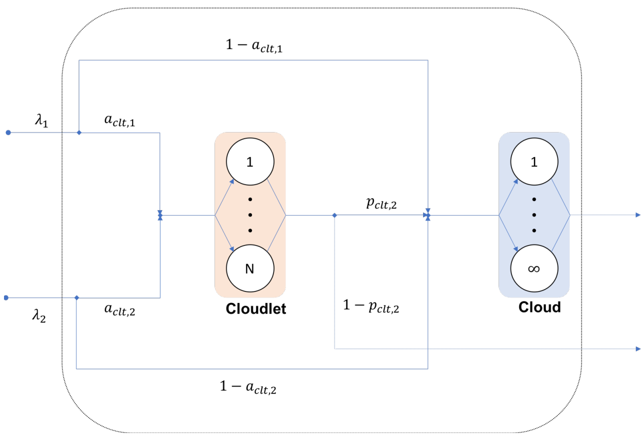

Figure 3: Analytical model.

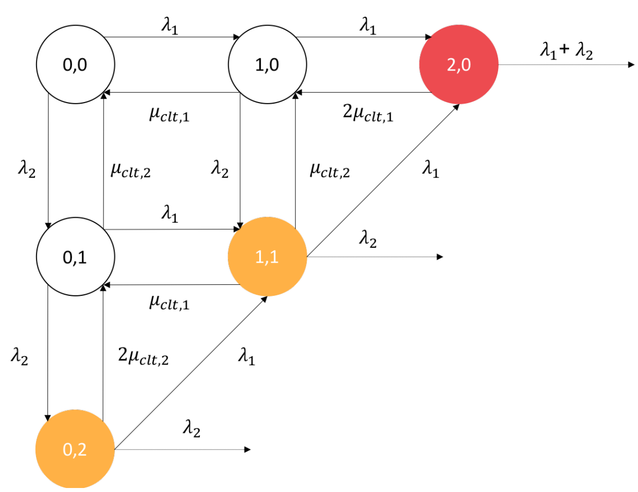

Figure 4: Markov Chain with $N = 2$ and $S = 2$.

consider here the simple case with $N = S = 2$ in order to (i) give an idea of the system of equations to be solved and (ii) graphically recognize the critical states. In fact, the representation fo the Markov Chain and the associated equations would be inpractical for the case $N = S = 20$, due to the combinatorial explosion of the state space.

In the considered simple case, the critical states are:

- (2, 0): every arrival is forwarded to the Cloud;
- (1, 1): every arrival belonging to class 1 is accepted in Cloudlet, causing the restart in Cloud of the serving task belonging to class 2; whilst every arrival belonging to class 2 is forwarded to Cloud;
- (0, 2): every arrival belonging to class 1 is accepted in Cloudlet, causing the restart in Cloud of a random serving task of Class 2; whilst every arrival belonging to class 2 is forwarded to Cloud;

$$
\begin{aligned}
\pi_{0,0}(\lambda_1 + \lambda_2) &= \pi_{1,0}\mu_{clt,1} + \pi_{0,1}\mu_{clt,2} \\
\pi_{0,1}(\lambda_1 + \lambda_2 + \mu_{clt,2}) &= \pi_{0,0}\lambda_2 + \pi_{1,1}\mu_{clt,1} + \pi_{0,2}2\mu_{clt,2} \\
\pi_{1,0}(\lambda_1 + \lambda_2 + \mu_{clt,1}) &= \pi_{0,0}\lambda_1 + \pi_{1,1}\mu_{clt,2} + \pi_{2,0}2\mu_{clt,1} \\
\pi_{1,1}(\lambda_1 + \mu_{clt,1} + \mu_{clt,2}) &= \pi_{0,1}\lambda_1 + \pi_{1,0}\lambda_2 + \pi_{0,2}\lambda_1 \\
\pi_{0,2}(\lambda_1 + 2\mu_{clt,2}) &= \pi_{0,1}\lambda_2 \\
\pi_{2,0}2\mu_{clt,1} &= \pi_{1,0}\lambda_1 + \pi_{1,1}\lambda_1 \\
1 &= \pi_{0,0} + \pi_{0,1} + \pi_{1,0} + \pi_{1,1} + \pi_{0,2} + \pi_{2,0}
\end{aligned} \tag{7}
$$

Accepted Workload. Given the routing probabilities we can determine the following *accepted workloads*:

- Cloudlet: arrivals of tasks belonging to j-th class accepted in Cloudlet:
$$\lambda_{clt,j} = a_{clt,j}\lambda_j \tag{8}$$
- Cloud: arrivals of tasks belonging to j-th class forwarded to Cloud:
$$\lambda_{cld,j} = (1 - a_{clt,j})\lambda_j \tag{9}$$
- Preemption: tasks belonging to 2-nd class preempted in Cloudlet and restarted in Cloud:
$$\lambda_p = r(\lambda_1 + \lambda_2) \tag{10}$$

Performance metrics. Given the accepted workloads we can determine the following *performance metrics for classed tasks in each subsystem*:

- 1^{st} class in Cloudlet:
$$
\begin{aligned}
E[T_{clt,1}] &= \frac{1}{\mu_{clt,1}} \\
E[N_{clt,1}] &= \lambda_{clt,1}E[T_{clt,1}]
\end{aligned} \tag{11}
$$
- 2^{nd} class in Cloudlet:
$$
\begin{aligned}
E[T_{clt,2}] &= \frac{1}{\mu_{clt,2}} \\
E[N_{clt,2}] &= \lambda_{clt,2}E[T_{clt,2}] - \psi\lambda_p E[T_{clt,2}]
\end{aligned} \tag{12}
$$
- 1^{st} class in Cloud:
$$
\begin{aligned}
E[T_{cld,1}] &= \frac{1}{\mu_{cld,1}} \\
E[N_{cld,1}] &= \lambda_{cld,1}E[T_{cld,1}]
\end{aligned} \tag{13}
$$
- 2^{nd} class in Cloud (not preempted):
$$
\begin{aligned}
E[T_{cld,2}]^{[NP]} &= \frac{1}{\mu_{cld,2}} \\
E[N_{cld,2}]^{[NP]} &= \lambda_{cld,2}E[T_{cld,2}]^{[NP]}
\end{aligned} \tag{14}
$$
- 2^{nd} class in Cloud (preempted):
$$
\begin{aligned}
E[T_{cld,2}]^{[P]} &= E[T_{clt,2}] + E[T_{setup}] + \psi E[T_{cld,2}]^{[NP]} \\
E[N_{cld,2}]^{[P]} &= \lambda_p E[T_{cld,2}]^{[P]}
\end{aligned} \tag{15}
$$
- 2^{nd} class in Cloud (not preempted and preempted):
$$
\begin{aligned}
E[T_{cld,2}] &= \sum_{m=NP,P} \frac{E[N_{cld,2}]^{[m]}}{E[N_{cld,2}]} E[T_{cld,2}]^{[m]} \\
E[N_{cld,2}] &= \sum_{m=NP,P} E[N_{cld,2}]^{[m]}
\end{aligned} \tag{16}
$$

Then we can determine the following *performance metrics for each subsystem*:

- Cloudlet:

$$
\begin{aligned}
E[T_{clt}] &= \sum_{j=1,2} \frac{E[N_{clt,j}]}{E[N_{clt}]} E[T_{clt,j}] \\
E[N_{clt}] &= \sum_{j=1,2} E[N_{clt,j}] \\
E[X_{clt}] &= \sum_{j=1,2} \lambda_{clt,j} - \lambda_p
\end{aligned} \tag{17}
$$

- Cloud:

$$
\begin{aligned}
E[T_{cld}] &= \sum_{j=1,2} \frac{E[N_{cld,j}]}{E[N_{cld}]} E[T_{cld,j}] \\
E[N_{cld}] &= \sum_{j=1,2} E[N_{cld,j}] \\
E[X_{cld}] &= \sum_{j=1,2} \lambda_{cld,j} + \lambda_p
\end{aligned} \tag{18}
$$

Finally, we can determine the following *performance metrics for the whole system*:

$$
\begin{aligned}
E[T] &= \sum_{i=cld,clt} \frac{E[N_i]}{E[N]} E[T_i] \\
E[N] &= \sum_{i=cld,clt} E[N_i] \\
E[X] &= \sum_{i=cld,clt} E[X_i]
\end{aligned} \tag{19}
$$

Thinking about the *utilization of each subsystem*, the following hold true:

- Cloudlet: simplifying our argument by assuming the whole incoming traffic belonging to the 1^{st} class served at the maximum rate (the best case), we can state that

$$\rho_{clt} = \frac{\lambda_1 + \lambda_2}{N\mu_{clt,1}} \to 0 \tag{20}$$

  That is, the Cloudlet is not able to serve all traffic as it saturates.
- Cloud: as a queue with infinite resources, we can conclude that

$$\rho_{cld} \to 0 \tag{21}$$

  That is, the Cloud can handle all requests as it never saturates.

Resolution. Given the *analytical model* depicted in Figure 3, the resolution of the Markov Chain for the case $S = N = 20$ allows us to determine the routing probabilities and performance metrics shown at the end of this paragraph.

We solved the the *analytical model* depicted in Figure 3 leveraging (i) a *Python script* that determines the transition matrix and the notable subsets of states, i.e. $S_{clt,1}$, $S_{clt,2}$ and $S_{clt,3}$, and (ii) a *Matlab Live Script* that takes them as input and computes routing probabilities, accepted/preempted traffic and performance metrics.

4.5 Computational Model

The proposed performance model has been implemented as a Python application. The simulation parameters can be configured with a YAML file loaded by the simulator when it starts up. The full open source code is available in a public repository [5] and representative examples of configuration and outputs are presented in Section 6.

We adopted the next-event simulation paradigm, using (i) a custom multi-stream Lehmer generator to generate random events, whose parameters have been described in Section 3 and whose evaluation is presented in Section 5; and (ii) a priority-queue based calendar with the ability both to schedule and un-schedule events.

Even if both the initial and terminal state can have any possible value, we adopted the convention of initializing and terminating the system in the idle state $(0, 0, 0, 0)$. In particular, the terminal state is reached via the well-known closed door technique driven by a stop time condition.

The calendar is initialized by scheduling the first arrival in the initialization phase. The submission of an arrival a to the system could induce (i) the scheduling of the corresponding completion event, (ii) the scheduling of a new arrival, or (iii) the unscheduling of a previously scheduled completion, i.e. interruption in Cloudlet.

The next-event calendar is implemented as priority queue, appropriately extended to manage scheduling/unscheduling of events and exclusion of impossible events, i.e. arrivals with occurrence time greater than the stop time. The impossibility of events is managed by letting the calendar contain possible events only, which is the best approach when the event list is assumed to be very long.

4.6 Verification

The main goal of verification is to assess the consistency of the computational model with the specification model. The verification has been carried out by evaluating the following consistency checks based on simulator logs and outputs:

- **state consistency:** verifies the correctness of the system state evolution, i.e. state transitions;
- **arrival consistency:** verifies the correctness of arrivals ordering, i.e. tasks arrived before are served before;
- **service consistency:** verifies the correctness of service ordering, i.e. tasks with less service time leave the system before;
- **flow consistency:** verifies the correctness of flow trends, such as:

$$n_{clt,i} = a_{clt,i} - s_{clt,i} - c_{clt,i} \tag{22}$$

$$n_{cld,i} = a_{cld,i} + s_{cld,i} - c_{cld,i} \tag{23}$$

$$s_{clt,i} = s_{cld,i} \tag{24}$$

  where $n_{j,i}$ is the population in the j-th subsystem belonging to i-th class of tasks, $a_{j,i}$ is the number of arrivals to the j-th subsystem belonging to i-th class of tasks, $c_{j,i}$ is the number of completions in the j-th subsystem belonging to i-th class of tasks $s_{j,i}$ is the number of switches from/to the j-th subsystem belonging to i-th class of tasks[1].

[1] notice that $s_{j,1} = 0 \forall j = 1, 2$, as tasks belonging to class $C1$ cannot be switched from Cloudlet to Cloud.

- **workload change consistency:** verifies the correctness of performance metrics variations in response to arrival/service rates variations. For example, we verified that the following hold true:

$$\mu_{cld,2}^{new} > \mu_{cld,2}^{old} \Rightarrow E[T_{sys,2}]^{new} > E[T_{sys,2}]^{old} \tag{25}$$

and

$$S^{new} > S^{old} \Rightarrow E[N_{cld,2}]^{new} < E[N_{cld,2}]^{old} \tag{26}$$

4.7 Validation

It is well-known that model development should include a final validation step in order to assess the consistency of the model with the real system. As the simulation main purpose is insight, a widely adopted Turing-like technique is to place the computational model alongside with the real system and assess the consistency of performance indices. Clearly, we cannot adopt this technique as we cannot compare the model with its real counterpart. For this reason, we totally rely on the validation with respect to the analytical model. In Figure 5 we show the comparison between theoretical performance results, taken from the analytical model, and their experimental counterpart, taken from the simulator. The obtained results demonstrate that our simulator is a reliable tool to conduct the performance analysis of the target system.

Index	Theoretical	Experimental
$E[N_{clt}]$	123456789	123456789
$E[N_{1,clt}]$	123456789	123456789
$E[N_{2,clt}]$	123456789	123456789
$E[T_{clt}]$	123456789	123456789
$E[T_{1,clt}]$	123456789	123456789
$E[T_{2,clt}]$	123456789	123456789
X_{clt}	123456789	123456789
$X_{1,clt}$	123456789	123456789
$X_{2,clt}$	123456789	123456789
$E[N_{cld}]$	123456789	123456789
$E[N_{1,cld}]$	123456789	123456789
$E[N_{2,cld}]$	123456789	123456789
$E[T_{cld}]$	123456789	123456789
$E[T_{1,cld}]$	123456789	123456789
$E[T_{2,cld}]$	123456789	123456789
X_{cld}	123456789	123456789
$X_{1,cld}$	123456789	123456789
$X_{2,cld}$	123456789	123456789
$E[N_{sys}]$	123456789	123456789
$E[N_{1,sys}]$	123456789	123456789
$E[N_{2,sys}]$	123456789	123456789
$E[T_{sys}]$	123456789	123456789
$E[T_{1,sys}]$	123456789	123456789
$E[T_{2,sys}]$	123456789	123456789
X_{sys}	123456789	123456789
$X_{1,sys}$	123456789	123456789
$X_{2,sys}$	123456789	123456789

Figure 5: Validation: comparison between analytical results and experimental results.

5 EVALUATION

In this Section, we present our experimental results. First, we show the results about the randomness degree of the adopted pseudo-random number generator. Then, we show the results about the performance recorded by the simulation of the target system.

The experiments have been conducted on an Amazon EC2 c3.8xlarge instance, which is really indicated for high performance science and engineering applications[2]. The instance is equipped with 32 vCPU based on an Intel Xeon E5-2680 v2 (Ivy Bridge) processor, 30 GB of RAM and SSD with 900 IOPS. It runs Debian 8.3 (Jessie), Python 3.5.2, and the Python-ported version of the official Leemis library for discrete-event simulation, indicated in [4]. Our solution has been developed in Python, following the de-facto standard best-practices, stated in [1, 6].

5.1 Randomness Analysis

Let us now consider the results about the randomness degree of the adopted generator. The randomness has been assessed by the following tests:

- **Spectral Test:** this test is considered one of the most powerful tests to assess the quality of linear congruential generators [2]. It relies on the fact that the output of such generators form lines or hyperplanes when plotted on 2 or more dimensions. The less the distance between these lines or planes, the better the generator is. In fact, a smaller distance between lines or planes highlights a better uniform distribution.
  In Figure **??** we show the test results for generators $(16807, 2^{31}-1)$, $(48271, 2^{31}-1)$ and $(50812, 2^{31}-1)$, respectively.
  The results show that our generator $(50812, 2^{31}-1)$ is much better than $(16807, 2^{31}-1)$, which was a past de-facto standard, and it is really similar to $(48271, 2^{31}-1)$, which is the current de-facto standard, according to [4].
- **Test of Extremes:** this test relies on the fact that if $U = U_0, \ldots, U_{d-1}$ is an independent identically distributed sequence of $Uniform(0, 1)$ random variables, then $\max(U)^d$ is also a $Uniform(0, 1)$. The test leverages this property to measures, for every stream, how much the generated random values differ from the theoretical uniform distribution.
  Given a number of streams s and a level of confidence $c = 1 - \alpha$, the more the total number of fails is close to the expected value, i.e. $s \cdot c$, the better the generator is.
  In Figure **??** we show the test results for the proposed generator $(508012, 2^{31}-1, 256)$ with sample size $n = 10000$, $k = 1000$ bins, sequence size $d = 5$ and 95% level of confidence. The proposed generator shows critical values $v_{min} = 913$ and $v_{max} = 1088$ and 14 total fails (7 lower fails and 7 upper fails), that is not far from the theoretical accepted number of fails, i.e. $256 * 0.05 = 13$. The proposed generator successfully passed the test with a 94.531% level of confidence.

[2] https://aws.amazon.com/ec2/instance-types/

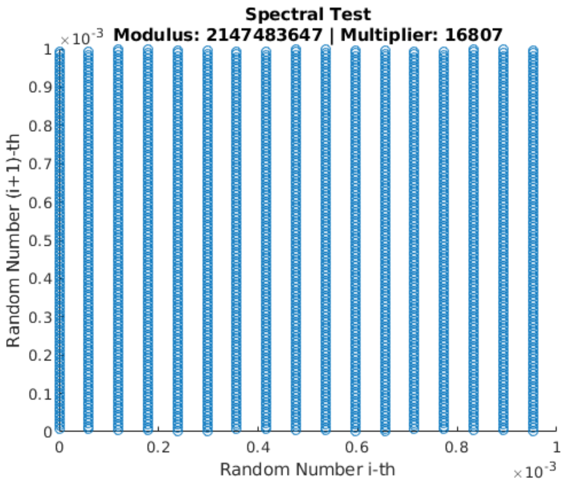

Figure 6: The Spectral Test to evaluate the randomness of the random number generator $(16807, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

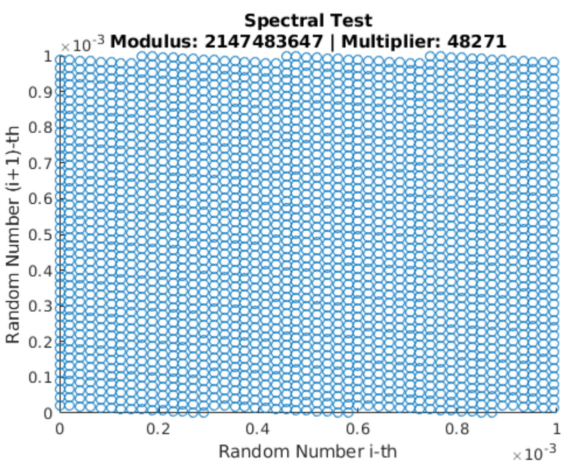

Figure 7: The Spectral Test to evaluate the randomness of the random number generator $(48271, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

- **Kolmogorov-Smirnov Analysis:** the test measures, at a given level of confidence, the biggest vertical distance between the theoretical cumulative distribution function and the empirical cumulative distribution function. The more the recorded distance d is less than the critical value $d*$ for the considered level of confidence, the better the generator is. As the Kolmogorov-Smirnov analysis relies on pre-calculated randomness statistics, we have chosen to take into account the statistics obtained by the previous test.
  In Figure 10 we show the test results for the proposed generator $(50812, 2^{31} - 1, 256)$ with a 95% level of confidence. The proposed generator successfully passed the test, as $d = 0.041 < 0.081 = d*$.

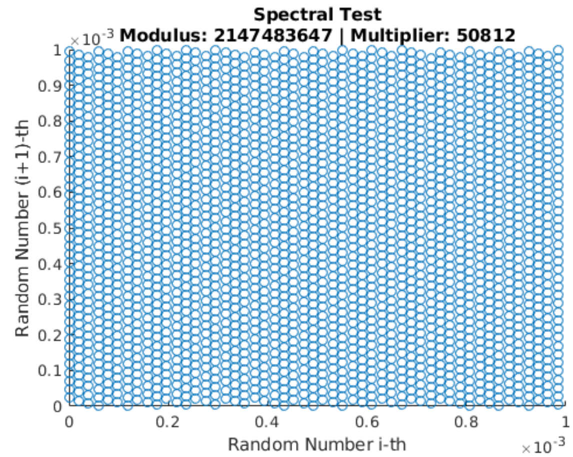

Figure 8: The Spectral Test to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

Figure 9: The Test of Extremes with $d = 5$ **to evaluate the randomness of the random number generator** $(50812, 2^{31} - 1, 256)$**.**

5.2 Performance Analysis

Let us now consider the results about the performance recorded during the simulation of the target system. In all experiments we considered values stated in Section 4.

5.3 Transient Analysis

First, we conduct a *transient analysis* to evaluate the stationary of the system and to estimate the duration of the transient period. In fact, given a system that converges to stationary, the knowledge of the duration of the transient period is really important to conduct an effective performance evaluation. In particular, it allows the analyst to focus performance evaluation on a system in its stationary conditions. In the transient analysis we focus on the following

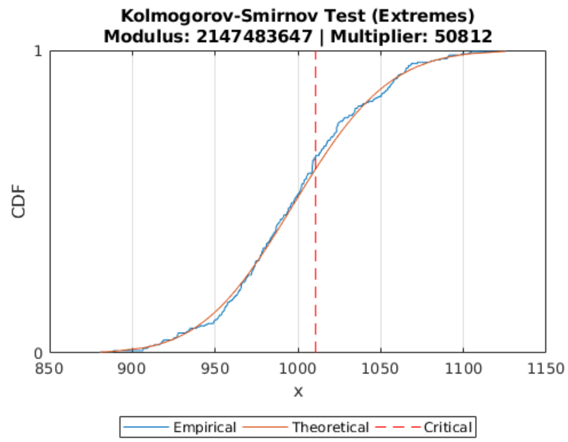

Figure 10: The Kolmogorov-Smirnov Analysis (leveraging the Test of Extremes with $d = 5$) to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 256)$ with 0.95 confidence level.

global metrics for the whole system: response time and mean population. We assess the transient period of the aforementioned metrics because they are also the performance metrics that will be taken into account in the final performance evaluation, thus it is really important to study their stationary.

The following results have been produced by considering an ensemble of 5 replications, where the i+1-th replication is initialized with the last seed of the i-th replication, so as to achieve the best decoupling between random sequences of different replications.

In Figure 11 we show the transient analysis of the global response time in the whole system. In Figure 12 we show the transient analysis of the global mean population in the whole system.

The results show that the system is stationary.

As we could image, each metric exposes a distinct transient period, e.g. the ratio of switched tasks converges faster than the mean population. Thus, we consider $\tau* = 8 \cdot 10^4$ *sec* as the final instant of the transient period, as in $\tau*$ we are sure that all metrics loosed their dependence on starting conditions.

5.4 Performance Evaluation

Let us now focus on the *performance evaluation*, taking into account the following metrics:

(1) response time both global and per-class, both for the system as a whole and for each subsystem;
(2) throughput both global and per-class, both for the system as a whole and for each subsystem;
(3) mean population both global and per-class, both for the system as a whole and for each subsystem;
(4) ratio of switched tasks of type 2;
(5) response time for switched tasks of type 2.

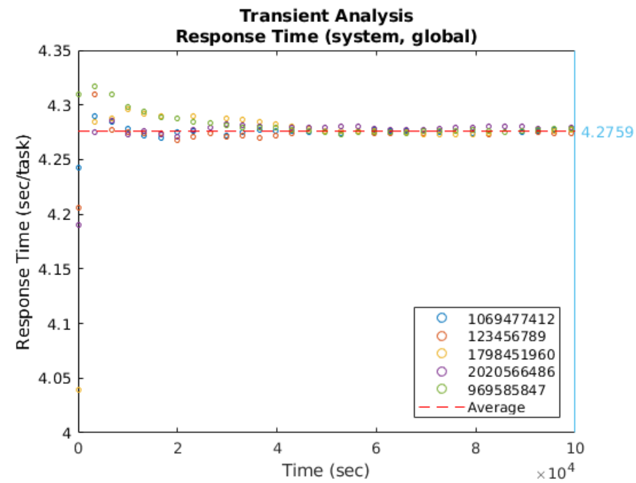

Figure 11: Transient analysis for global response time in the whole system.

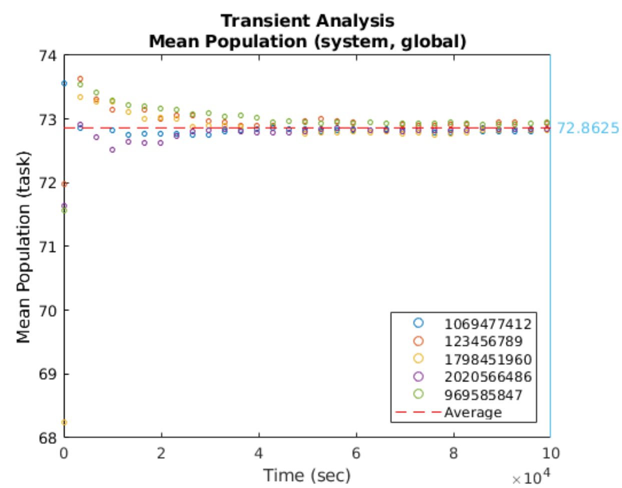

Figure 12: Transient analysis for global mean population in the whole system.

5.5 Distribution Analysis

In Figure **??** we show the distribution analysis for the response time when $S = N = 20$, where we adopted the *Freedman-Diaconis Rule* for the binning schema. Results show that the best fitting is the *Weibull* with parameters $A \approx 4.357$ and $B \approx 3546.917$.

6 USAGE

In this Section we show how to configure and run experiments and some sample outputs to provide a better idea of what has been created.

The test of extremes for a custom random number generator produces the output shown in Figure 18 and can be executed with default configuration by running the script

```
exp/random/randomness/extremes/main.py
```

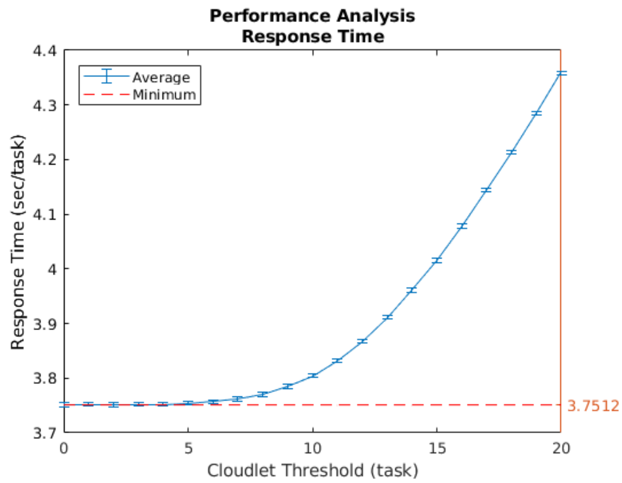

Figure 13: Performance analysis of response time as a function of the threshold S with level of confidence 95%. **The threshold that minimizes the response time is $S* = 2$ with mean value $E[R] \approx 3.7512$ sec.**

S	μ(R) $\pm$ $\delta_{0.05}$	σ(R)
0	3.75149 ± 0.00342	0.01346
1	3.75172 ± 0.00335	0.01322
2	**3.75120 ± 0.00338**	**0.01330**
3	3.75198 ± 0.00334	0.01315
4	3.75200 ± 0.00328	0.01292
5	3.75430 ± 0.00324	0.01275
10	3.80394 ± 0.00330	0.01299
15	4.01623 ± 0.00406	0.01599
20	4.35885 ± 0.00342	0.01349

Figure 14: Performance analysis of the response time as a function of the threshold S with level of confidence 95%.

The test of Kolmogorov-Smirnov for a custom random number generator produces the output shown in Figure 19 and can be executed with default configuration by running the script

```
exp/random/randomness/kolmogorov-smirnov/main.py
```

The simulation is configured providing a configuration YAML file as the one shown in Figure 20, produces the output shown in Figure 21 and can be executed by running the script

```
exp/simulation/performance/main.py
```

7 CONCLUSIONS

In this work we propose a next-event simulator for a two-layer Cloud system with off-loading policy on class-partitioned workload, whose random components leverage a multi-stream Lehmer pseudo-random number generator.

We may conclude that (i) our simulator returns experimental results that are consistent with the theoretical ones, (ii) the system can achieve the steady-state (iii) the choice of the threshold S is

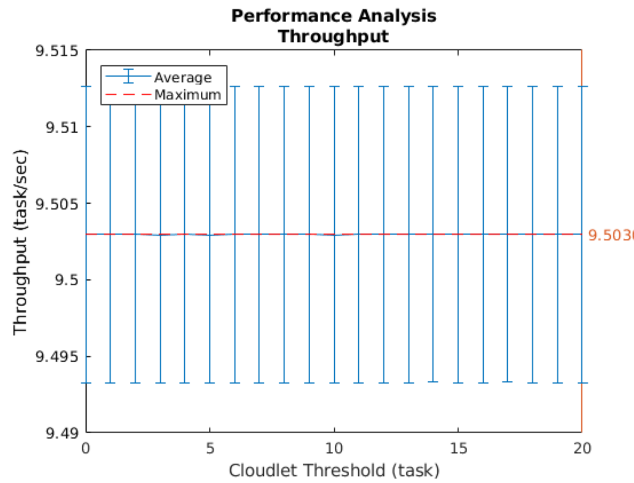

Figure 15: Performance analysis of the throughput as a function of the threshold S with level of confidence 95%. **The throughput is clearly threshold insensitive, with a constant mean value $E[X] \approx 9.5030$ $task/sec$. The threshold $S* = 2$ is a good choice**

S	μ(X) $\pm$ $\delta_{0.05}$	σ(X)
0	9.50296 ± 0.00969	0.03818
1	9.50295 ± 0.00969	0.03818
2	**9.50296 ± 0.00969**	**0.03816**
3	9.50294 ± 0.00967	0.03808
4	9.50296 ± 0.00969	0.03816
5	9.50294 ± 0.00969	0.03818
10	9.50295 ± 0.00969	0.03818
15	9.50295 ± 0.00968	0.03813
20	9.50296 ± 0.00970	0.03821

Figure 16: Performance analysis of the throughput as a function of the threshold S with level of confidence 95%.

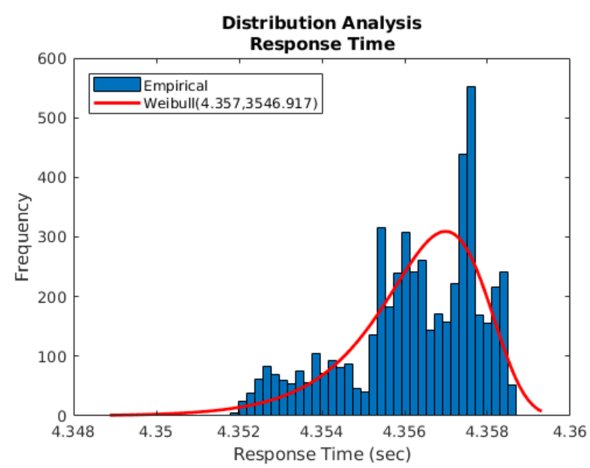

Figure 17: Distribution analysis for response time with threshold $S = N = 20$.

```
========================================
TEST OF EXTREMES
========================================

Generator
Class................MarcianiMultiStream
Streams.............................256
Modulus......................2147483647
Multiplier.......................50812
Seed.........................123456789

Test Parameters
Sample Size......................10000
Bins..............................1000
Confidence........................95.0
D....................................5

Critical Bounds
Lower Bound...........913.3009983090644
Upper Bound..........1088.4870675338261

Error
Theoretical.................13 (5.078 %)
Empirical...................14 (5.469 %)
Empirical Lower Bound........7 (2.734 %)
Empirical Upper Bound........7 (2.734 %)

Result
Suggested Confidence............94.531
Success..........................False
```

Figure 18: A sample output of the Test of Extremes.

critical for system performances and (iv) the adopted preemption policy allows to balance response time for classes of tasks with different service rates.

Although our results are pretty satisfactory, the proposed solutions could certainly be improved and be subjected to a more in-depth analysis. From an implementation point of view, the proposed solution should be ported from Python to C and leverage multi-threading to achieve better performances, e.g. to speed-up the algorithms to find suitable multipliers for modulus in 64-bit architectures and make faster simulations. From an analysis point of view, the proposed random number generator should be tested more extensively. For example, we may (i) take into account more tests of randomness (ii) use a pseudo-random number generator with a 64-bit modulus and less number of streams. Finally, the simulation model should be extended in order to (i) study the influence of different server selection policies, e.g. equity-selection, and (ii) achieve more performance evaluation goals, such as forecasting with respect to the variation of the arrival processes.

```
========================================
TEST OF KOLMOGOROV-SMIRNOV
========================================

Generator
Class................MarcianiMultiStream
Streams.............................256
Modulus......................2147483647
Multiplier.......................50812
Seed.........................123456789

Test Parameters
Chi-Square Test...............extremes
Sample Size......................10000
Bins..............................1000
Confidence........................95.0
D....................................5

KS
KS Statistic.....................0.041
KS Point X......................1010.6
KS Critical Distance.............0.084

Result
Success...........................True
```

Figure 19: A sample output of the Test of Kolmogorov-Smirnov.

REFERENCES

[1] Google. 2016. The Google's Python Styleguide. (sep 2016). http://bit.ly/2d4M9UN
[2] Donald E Knuth. 1981. *The Art of Computer Programming; Volume 2: Seminumeral Algorithms.*
[3] Pierre L'Ecuyer. 1994. Uniform random number generation. *Annals of Operations Research* 53, 1 (1994), 77–120.
[4] Lawrence M Leemis and Stephen Keith Park. 2006. *Discrete-event simulation: A first course.* Pearson Prentice Hall Upper Saddle River.
[5] Giacomo Marciani. 2018. pyDES. (jan 2018). http://bit.ly/2BFhqwi
[6] Kenneth Reitz and Tanya Schlusser. 2016. *The Hitchhiker's Guide to Python: Best Practices for Development.* O'Reilly Media. http://amzn.to/2bYCvBD

```
general:
  t_stop: 604800
  t_tran: 80000
  n_batch: 64
  t_sample: 100
  confidence: 0.95

tasks:
  arrival_rate_1: 3.25
  arrival_rate_2: 6.25

system:
  cloudlet:
    n_servers: 20
    service_rate_1: 0.45
    service_rate_2: 0.30
    threshold: 20
    server_selection: "ORDER"

  cloud:
    service_rate_1: 0.25
    service_rate_2: 0.22
    t_setup_mean: 0.8
```

Figure 20: A sample configuration for a simulation experiment.

```
========================================
         SIMULATION-THRESHOLD-20
========================================

                general
t_stop ...........................604800
t_tran ............................80000
n_batch ..............................64
t_batch .........................8200.0
rndgen ............... MarcianiMultiStream
rndseed .......................123456789

                 tasks
arrival_rate_1 ......................3.25
arrival_rate_2 ......................6.25
n_generated_1 ...................1965888
n_generated_2 ...................3781874

            system/cloudlet
service_rate_1 ......................0.45
service_rate_2 .......................0.3
n_servers ............................20
threshold ............................20

             system/cloud
service_rate_1 ......................0.25
service_rate_2 ......................0.22
setup_mean ...........................0.8

              statistics
population_mean ................61.92085
population_sdev .................0.21395
population_cint .................0.05431
response_mean ...................4.35885
response_sdev ...................0.01347
response_cint ...................0.00342
throughput_mean .................9.50297
throughput_sdev .................0.03823
throughput_cint .................0.00971
```

Figure 21: A sample output of a simulation experiment.