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, modern software applications and computer infrastructures are getting increasingly complex and require huge financial investments. 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, as it provide them with a competitive advantage, both in terms of technology and investment.

In this technical report we propose a next-event simulation model to analyze the performance of a two-layer Fog-like system that leverages an off-loading policy to minimize response-time. First, we describe how we implemented the multi-stream pseudo-random number generator, that is a fundamental building block of any next-event simulation. Then, we describe the performance model in terms of goals, conceptual, specification, computational model and verification.

The experimental results show (i) the satisfactory randomness degree of the adopted pseudo-random number generator and (ii) the effectiveness of our model to tune the system as 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, January 2018 (PMCSN'18),* 9 pages.
https://doi.org/10.475/123_4

PMCSN'18, January 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, modern software applications and computer infrastructures are getting increasingly complex and require huge financial investments. 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, as it provide them with a competitive advantage, both in terms of technology and investment. In this technical report we propose a next-event simulation model to analyze the performance of a two-layer Fog-like system that leverages an off-loading policy to minimize response-time.

The remainder of the paper is organized as follows. In Section 2 we describe the pseudo-random number generator adopted to generate random variates for the next-event simulation model. In Section 3 we describe the the next-event simulation model in terms of goals, conceptual, specification, computational model and verification. In Section 4 we show the experimental results about the randomness of the adopted pseudo-random number generator and the performances recorded by our model. In Section 5 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 6 we conclude the paper summing up the work that has been done and delineating future improvements.

2 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:

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

where m is the modulus, a is the multiplier, s is the number of streams and $(a^j \bmod m)$ 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 4.

3 MODELING

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

- **workload:** mobile devices send to the system tasks that can be partitioned in two classes C_1 and C_2.
- **system:** a two-tiers system, made of:
  - a remote Cloud server with virtually unlimited servants.
  - a Cloudlet with N servants, having the ability to off-load tasks to the Cloud, accordingly to a certain *off-loading policy* parametrized by a given *threshold*.

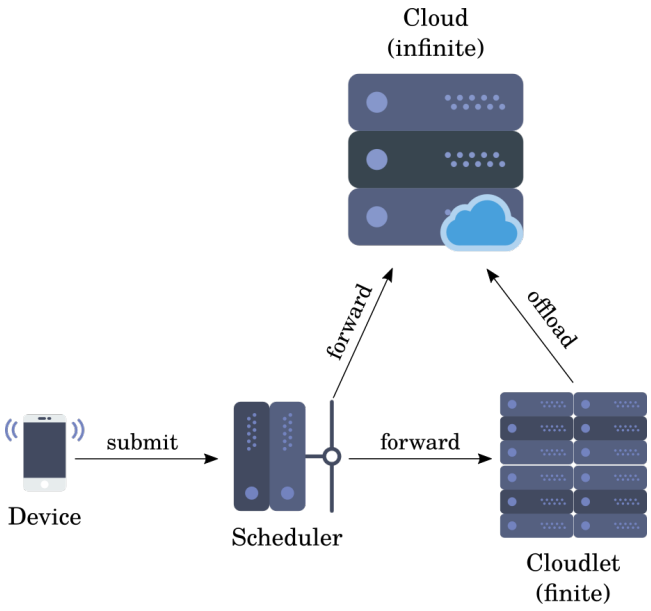

Figure 1: The system sketch.

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*.

Goals and Objectives. The main goals of the 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.

Conceptual Model. The conceptual model is depicted in Figure 2.

Specification Model. The state of the system is represented by the pair $(n_{clt,1}, n_{clt,2}, n_{cld,1}, n_{cld,2})$, where $n_{cld,i}$ is the number of tasks belonging to the i-th class within the Cloudlet and $n_{cld,i}$ is the number of tasks belonging to the i-th class within the Cloud. Tasks belonging to the first class, i.e. $t \in C_1$ arrive to the system with an exponential arrival process with rate λ_1; whilst tasks belonging to the seconds class, i.e. $t \in C_2$ arrive to the system with an exponential arrival process with rate λ_2. The Cloudlet serves tasks belonging to the first class with exponentially distributed service time with rate $\mu_{cld,1}$; whilst the Cloudlet serves tasks belonging to the second class with exponentially distributed service time with rate $\mu_{cld,2}$. The Cloud serves tasks belonging to the first class with exponentially distributed service time with rate $\mu_{clt,1}$; whilst the Cloudlet serves tasks belonging to the second class with exponentially distributed service time with rate $\mu_{clt,2}$. 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 $E[T_{setup}]$.

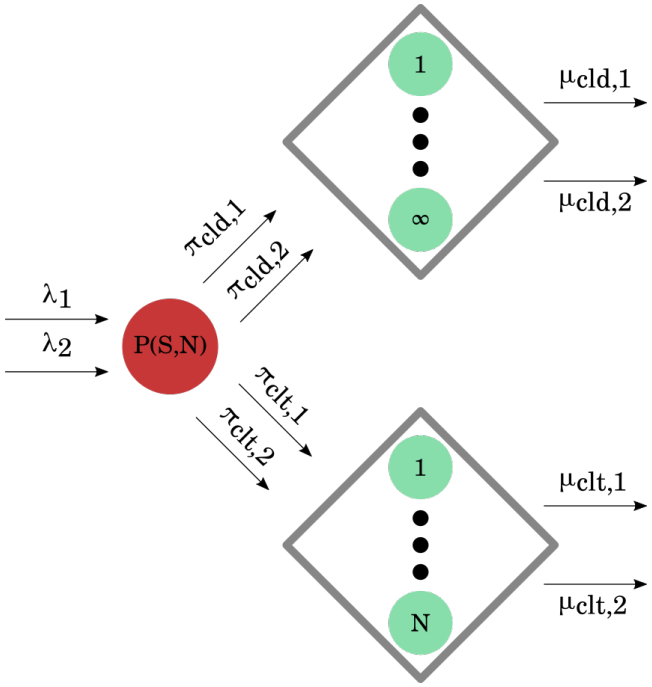

Figure 2: The conceptual model.

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

Algorithm 1: The dispatching policy.

Computational Model. The proposed performance model has been implemented as a Python application. The simulation parameters can be configured with a simple YAML file that can be loaded by the simulation program. The full open source code is available at [5] and some representative configurations and outputs are presented in Section 5.

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 2 and whose evaluation will be presented in Section 4; and (ii) a priority-queue based calendar with the ability both to schedule and un-schedule events.

Verification. The model has been verified by

- **flow consistency:** responsible to verify the correctness of flow trends, such as:

$$n_{cld,i} = a_{cld,i} - c_{cld,i} - s_{cld,i} \tag{2}$$

$$n_{cld,i} = a_{cld,i} - c_{cld,i} + s_{cld,i} \tag{3}$$

$$s_{clt,i} = s_{cld,i} \tag{4}$$

  where $n_{j,i}$ is the population of tasks belonging to i-th class in the j-th subsystem, aj, i is the number of tasks belonging to i-th class arrived to the j-th subsystem, cj, i is the number of tasks belonging to i-th class completed in the j-th subsystem, sj, i is the number of tasks belonging to i-th class switched from/to the j-th subsystem.
- **workload change consistency:** responsible to verify the correctness of metrics variations in response to arrival/service rates variations, such as:

$$\lambda_1' > \lambda_1 \Rightarrow \tag{5}$$

- **stationary check:** responsible to verify the correctness of the model in stationary conditions.

Validation. It is well-known that model development should includes a final validation step. Clearly, we cannot conduct this final step because we cannot compare the performance model with its real counterpart.

4 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[1]. 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].

4.1 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:

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

- **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 $U = U_0, ..., U_{d-1}$ 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.
- **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 7 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 $d = 0.041 < 0.081 = d*$.

4.2 Performance Analysis

Let us now consider the results about the performance recorded during the simulation of the target system. In all the experiments we have considered the following parameters:

- **arrivals:** exponential with rate $\lambda_1 = 6.00\ task/sec$ and $\lambda_2 = 6.25\ task/sec$.
- **services:** exponential with rate $\mu_{clt,1} = 0.45\ task/sec, \mu_{clt,2} = 0.27\ task/sec, \mu_{cld,1} = 0.25\ task/sec$ and $\mu_{cld,2} = 0.22\ task/sec$.
- **setup:** exponential with mean $E[T_{setup}] = 0.8\ sec$.

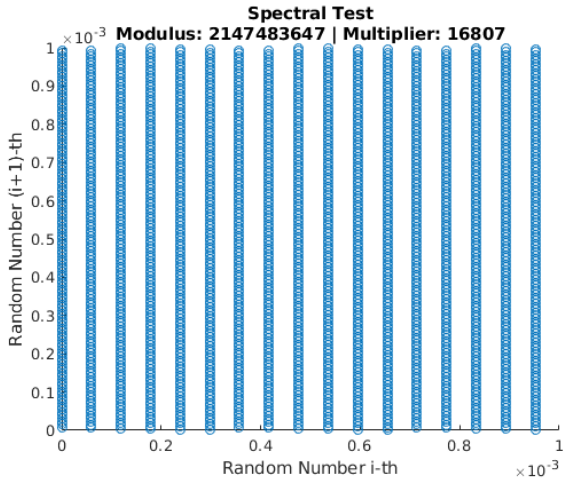

Figure 3: The Spectral Test to evaluate the randomness of the random number generator $(16807, 2^{31}-1, 1)$ in the interval $(0, 10^{-3})$.

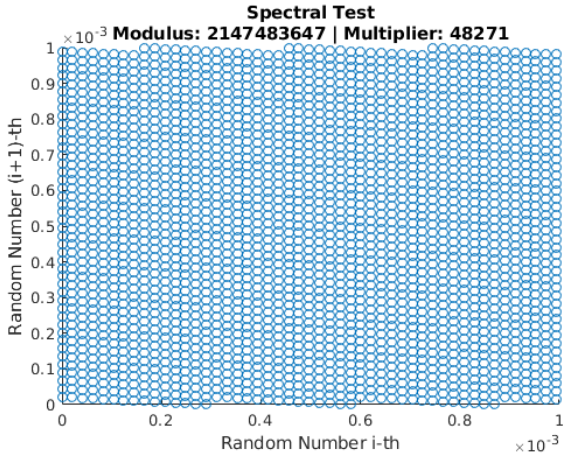

Figure 4: The Spectral Test to evaluate the randomness of the random number generator $(48271, 2^{31}-1, 1)$ in the interval $(0, 10^{-3})$.

4.3 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 global metrics for the whole system: response time, throughout, mean population, ratio of switched tasks, response time for switched tasks. 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.

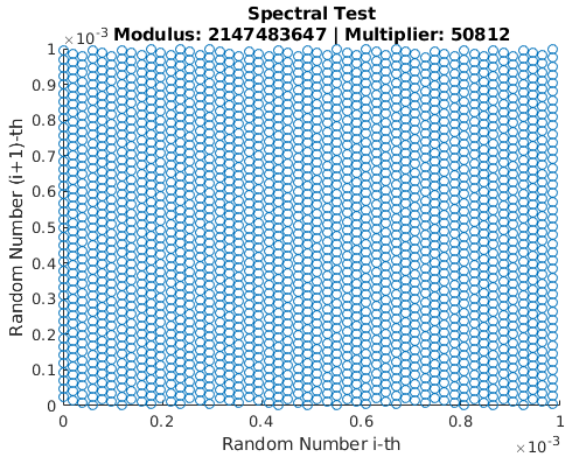

Figure 5: The Spectral Test to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

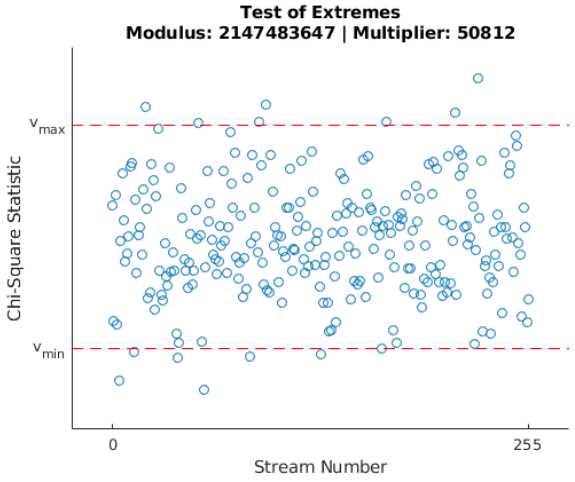

Figure 6: The Test of Extremes with $d = 5$ **to evaluate the randomness of the random number generator** $(50812, 2^{31} - 1, 256)$**.**

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 8 we show the transient analysis of the global response time in the whole system. In Figure 9 we show the transient analysis of the global throughput in the whole system. In Figure 10 we show the transient analysis of the global mean population in the whole system. In Figure 11 we show the transient analysis of the global switch ratio in the whole system. In Figure 12 we show the transient analysis of the response time for switched tasks in the whole system.

The results show that (i) the system is stationary, (ii) the response time, the throughput, the mean population and the ratio of switched tasks loose their dependence on the starting conditions, whilst (iii)

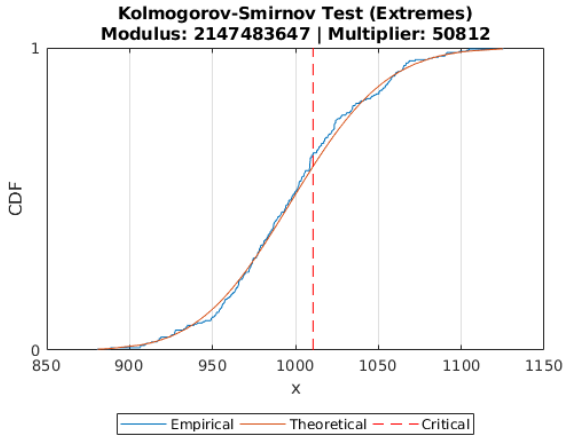

Figure 7: The Kolmogorov-Smirnov Analysis (leveraging the Test of Extremes with $d = 5$**) to evaluate the randomness of the random number generator** $(50812, 2^{31} - 1, 256)$ **with** 0.95 **confidence level.**

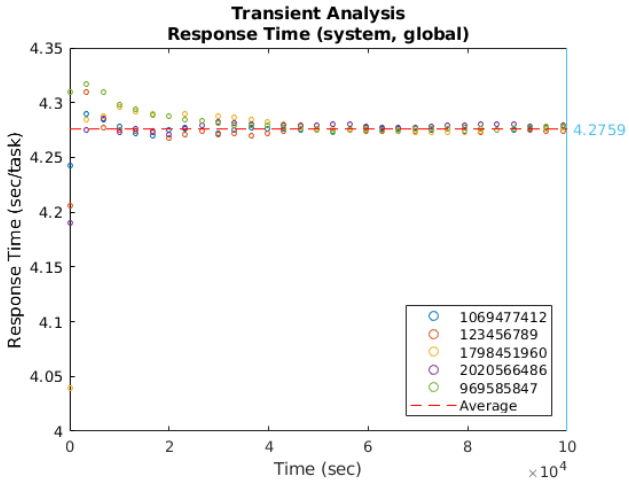

Figure 8: Transient analysis for global response time in the whole system.

the response time for switched tasks maintains its dependence on starting conditions, regardless of the termination of the transient period.

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.

4.4 Performance Evaluation

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

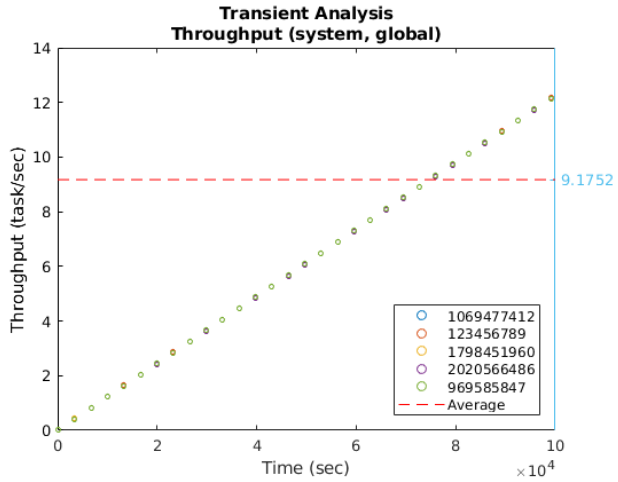

Figure 9: Transient analysis for global throughput in the whole system.

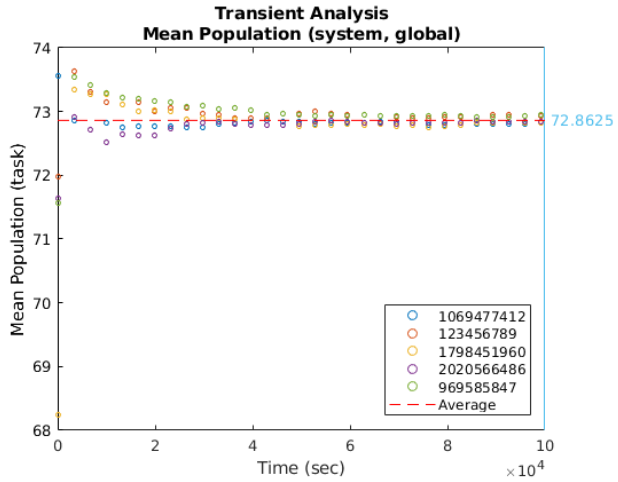

Figure 10: Transient analysis for global mean population in the whole system.

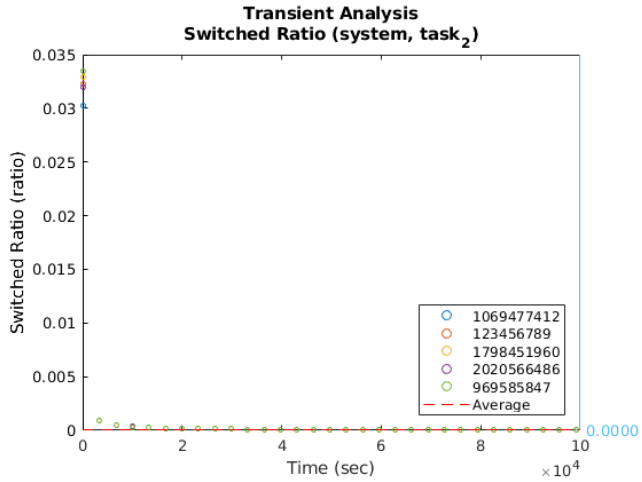

Figure 11: Transient analysis for the ratio of switched tasks of type 2.

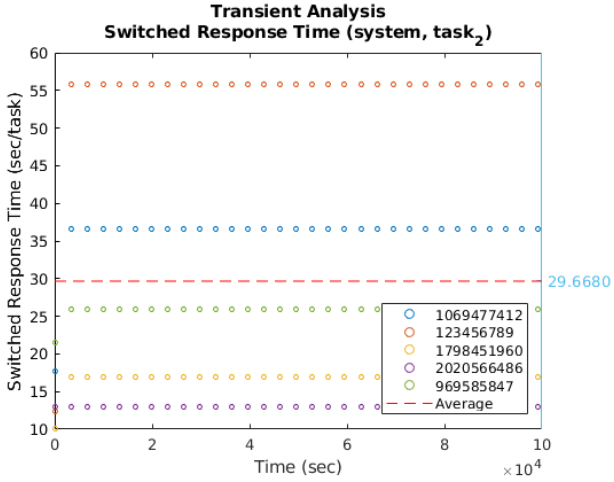

Figure 12: Transient analysis for response time for switched tasks of type 2.

(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.

4.5 Distribution Analysis

5 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
```

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
```

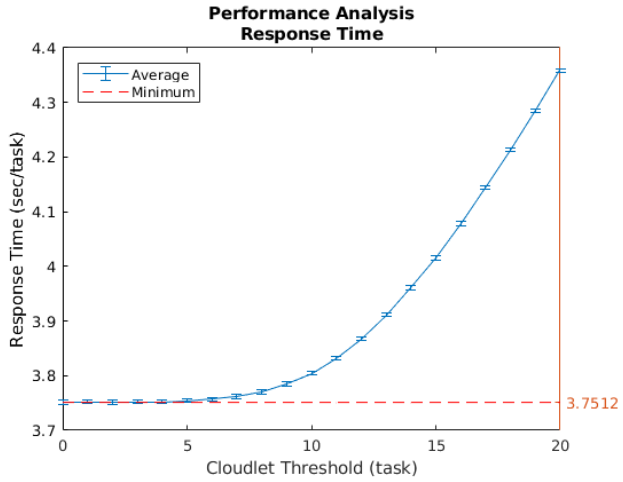

Figure 13: Performance analysis of response time as a function of the threshold S with level of confidence 95%. The threshold that minimizes the response time is $S* = 2$ with mean value $E[R] \approx 3.7512$ sec.

S	$\mu(\mathbf{R}) \pm \delta_{0.05}$	$\sigma(\mathbf{R})$
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%.

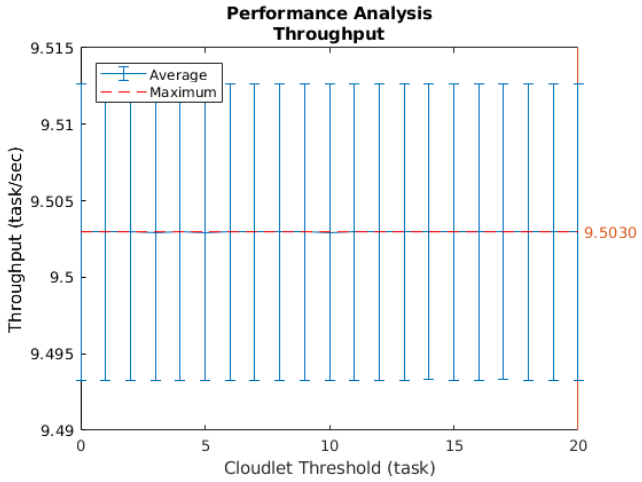

Figure 15: Performance analysis of the throughput as a function of the threshold S with level of confidence 95%. The throughput is clearly threshold insensitive, with a constant mean value $E[X] \approx 9.5030$ $task/sec$. The threshold $S* = 2$ is a good choice

S	$\mu(\mathbf{X}) \pm \delta_{0.05}$	$\sigma(\mathbf{X})$
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%.

6 CONCLUSIONS

In this work we propose a next-event simulation model for a two-layer Cloud system, leveraging a custom multi-stream Lehmer pseudo-random number generator.

Although experimental 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 the simulation faster. From an analysis point of view, the proposed random number generator should be tested more extensively, e.g. taking into account more tests of randomness, and a generator with a 64-bit modulus and less number of streams should be developed. Finally, the simulation model should be extended so as to (i) take into account more performance metrics, such as utilization, (ii) study the influence of different server selection policies, e.g. equity-selection, and (iii) achieve more performance evaluation goals, such as forecasting with respect to the variation of the arrival processes.

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

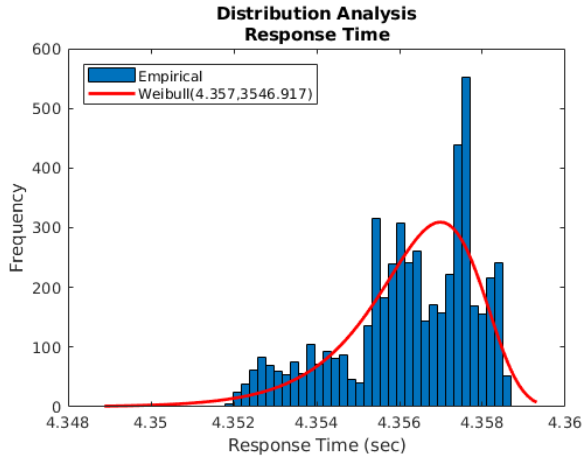

Figure 17: Distribution analysis for response time with threshold $S = 20$. The binnign rule is Freedman-Diaconis Rule. The best fitting is the Weibull with parameters $A \approx 4.357$ and $B \approx 3546.917$.

```
========================================
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.

```
========================================
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.

```
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.