

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PMCSN'18, September 2018, Rome, Italy

© 2018 Copyright held by the owner/author(s).

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 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.

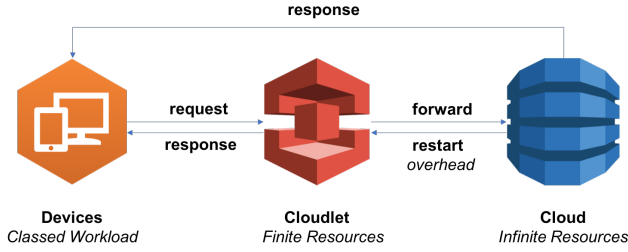


Figure 1: System architecture (high level).

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:

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

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

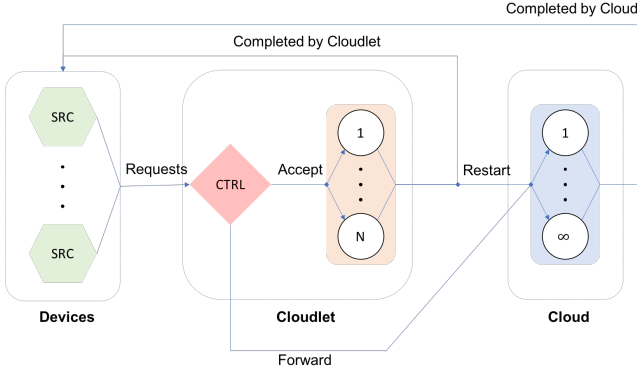


Figure 2: Conceptual model.

- Cloud: $S_{cld} := \{(n_{cld,1}, n_{cld,2}) \in \mathbb{N}^2\}$, 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:

- $A_{cld,j}$: 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.
- $C_{cld,j}$: 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 $\mu_{cld,j}$; 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) $\mu_{cld,i} > \mu_{cld,i} \forall i = 1, 2$ 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.

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

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

```

if task of class 1 then
  if  $n_{cld,1} = N$  then
    | send to the Cloud
  end
  if  $n_{cld,1} + n_{cld,2} < S$  then
    | accept on the Cloudlet
  end
  if  $n_{cld,2} > 0$  then
    | accept on the Cloudlet, interrupt a  $2^{nd}$  class task in
    | the Cloudlet and restart it in the Cloud
  else
    | accept on Cloudlet
  end
end
if arrival of class 2 then
  if  $n_{cld,1} + n_{cld,2} \geq S$  then
    | send to the Cloud
  else
    | accept on the Cloudlet
  end
end

```

Algorithm 1: Off-loading policy.

4.4 Analytical Model

In this Section we will define and solve the *analytical model* of the system. In particular, we will first show the Markov Chain and flow balance equations for a very simple case, in order to introduce the reader to the structure of the chain with its critical states. Then, we will cover the general case showing formulas of routing probabilities and performance metrics. At the end, we will solve the target case explaining how we solve it.

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 $S_{cld,i} \subset S_{cld}$:

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

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

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

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

- $S_{cld,3}$: a task belonging to the 2^{nd} class is interrupted in the Cloudlet and it is restarted in the Cloud.

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

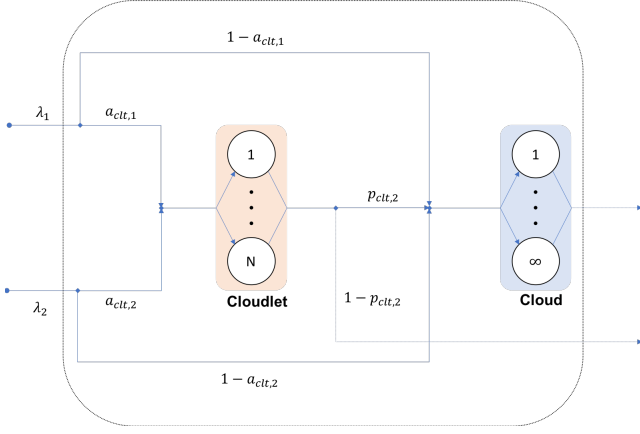


Figure 3: Analytical model.

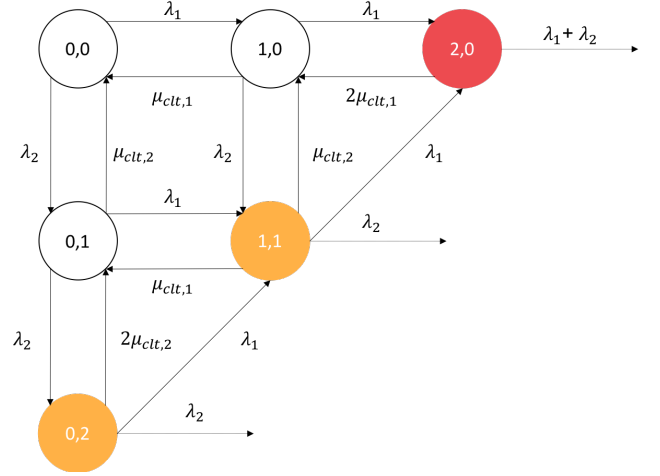
$$\begin{aligned}
 a_{clt,1} &= \sum_{s \in S_{clt,1}} \pi_s \\
 a_{clt,2} &= \sum_{s \in S_{clt,2}} \pi_s \\
 r_{clt,2} &= \sum_{s \in S_{clt,3}} \pi_s \left(\frac{\lambda_2}{\lambda_1 + \lambda_2} \right)
 \end{aligned} \tag{6}$$

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 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 for the Markov Chain and the associated equations would be impractical 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;

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

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

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*:

- 1st class in Cloudlet:

$$\begin{aligned}
 E[T_{clt,1}] &= \frac{1}{\mu_{clt,1}} \\
 E[N_{clt,1}] &= \lambda_{clt,1}E[T_{clt,1}]
 \end{aligned} \tag{11}$$

- 2nd class in Cloudlet:

$$\begin{aligned}
 E[T_{clt,2}] &= \frac{1}{\mu_{clt,2}} \\
 E[N_{clt,2}] &= \lambda_{clt,2}E[T_{clt,2}] - \psi\lambda_pE[T_{clt,2}]
 \end{aligned} \tag{12}$$

- 1st class in Cloud:

$$\begin{aligned} E[T_{cld,1}] &= \frac{1}{\mu_{cld,1}} \\ E[N_{cld,1}] &= \lambda_{cld,1} E[T_{cld,1}] \end{aligned} \quad (13)$$

- 2nd class in Cloud (not preempted):

$$\begin{aligned} E[T_{cld,2}]^{[NP]} &= \frac{1}{\mu_{cld,2}} \\ E[N_{cld,2}]^{[NP]} &= \lambda_{cld,2} E[T_{cld,2}]^{[NP]} \end{aligned} \quad (14)$$

- 2nd class in Cloud (preempted):

$$\begin{aligned} E[T_{cld,2}]^{[P]} &= E[T_{cld,2}]^{[NP]} + \psi E[T_{cld,2}]^{[NP]} \\ E[N_{cld,2}]^{[P]} &= \lambda_p E[T_{cld,2}]^{[P]} \end{aligned} \quad (15)$$

- 2nd class in Cloud (not preempted and preempted):

$$\begin{aligned} E[T_{cld,2}] &= \sum_{m=NP,P} \frac{E[N_{cld,2}]^{[m]}}{E[N_{cld,2}]} E[T_{cld,2}]^{[m]} \\ E[N_{cld,2}] &= \sum_{m=NP,P} E[N_{cld,2}]^{[m]} \end{aligned} \quad (16)$$

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

- Cloudlet:

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

- Cloud:

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

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

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

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

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

$$\rho_{clt} = \frac{\lambda_1 + \lambda_2}{N\mu_{clt,1}} \rightarrow 0 \quad (20)$$

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

$$\rho_{cld} \rightarrow 0 \quad (21)$$

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 simulator has been designed following the *next-event simulation* paradigm and has been implemented as a *Python* application. The full open source code is available in a public Github repository [5]. Representative examples of configuration and outputs are presented in Section 6.

Configuration. The simulation parameters can be fully configured with a YAML file loaded when the simulator starts up. In particular, the following parameters can be configured:

- *arrival process*: the user can configure the statistical distribution law and parameters of arrivals for both task classes. In this work, we consider the Exponential distribution, but any statistical distribution can be set.
- *service process*: the user can configure the statistical distribution law and parameters of services for both task classes. In this work, we consider the Exponential distribution, but any statistical distribution can be set.
- *setup time*: the user can configure the statistical distribution law and parameters of the setup time for both task classes. In this work, we consider a Deterministic setup time set to zero for the 1st class and an Exponential setup time for the 2nd class, but any statistical distribution can be set.
- *Cloudlet dimension*: the user can configure the number of Cloudlet resources.
- *Cloudlet threshold*: the user can configure the threshold for the restart condition of 2nd class tasks in the Cloudlet.
- *server selection policy*: the user can configure the policy adopted to select the 2nd class task to interrupt in the Cloudlet when the threshold has been reached. In this work, we consider the Random selection, but the user can choose among Random, Ordered, Cyclic and Equity selection policies.

Randomization. Random components are ruled by a custom *multi-stream Lehmer generator* to generate pseudo-random events, whose parameters have been described in Section 3 and whose evaluation is presented in Section 5. The *degree of randomization* has been improved by associating the random component the following processes to a dedicated stream of the pseudo-random number generator: arrival process of each task class, service process of each servant and server selection rule. This strategy has been motivated by the fact that in a real case scenario we can assume the independence between (i) inter-classed workload, (ii) computational offer of distinct resources and (iii) selection strategies.

Event management. Events are managed by 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 un-scheduling 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.

Software Classes. We provide here a short description of the main software classes involved in our simulator, in order to help the reader navigate through our code.

- **Simulation:** simulator entry-point, responsible to load and validate configurations, create the calendar of events, the statistics and the target system, show the real-time progress of the simulation, determine the duration of the simulation, write sampling files, reports and output results.
- **Calendar:** calendar of events, implemented as a priority-queue with the ability to both schedule and unschedule events.
- **System:** high level abstraction of the whole system, responsible to initialize the Cloudlet, initialize the Cloud, implement the Controller logic and update system-level classed/global metrics.
- **Cloudlet:** represents the Cloudlet subsystem, responsible to handle events, select the task to interrupt according to the selection policy and update Cloudlet-level classed/global metrics.
- **Cloud:** represents the Cloud subsystem, responsible to handle events and update Cloudlet-level classed/global metrics.
- **RandomComponent:** represents a fully-customizable independent random process. In particular, the calendar and each subsystem receive as input an instance of this object in order to realize their own random logic.

- **RndGen:** instance of a fully-customizable multi-stream Lehmer pseudo-random number generator.
- **Stats:** abstraction encapsulating all metrics involved in the simulation, responsible to make it easy to update metrics during the simulation loop. In particular, each subsystem receives as input a reference to this singleton in order to centralize the management of performance metrics.

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:

$$n_{clt,i} = a_{clt,i} - s_{clt,i} - c_{clt,i} \quad (22)$$

$$n_{cld,i} = a_{cld,i} + s_{cld,i} - c_{cld,i} \quad (23)$$

$$s_{clt,i} = s_{cld,i} \quad (24)$$

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 tasks¹.

- **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:

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

and

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

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

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

model. In Figure 12 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 pretty reliable tool to conduct the performance analysis of the target system. Where the experimental values do not coincide with the theoretical ones, this is to be attributed to the adoption of the random selection of the tasks to be interrupted.

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 applications². 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 $U = U_0, \dots, U_{d-1}$ 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 $(50812, 2^{31} - 1, 256)$ 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

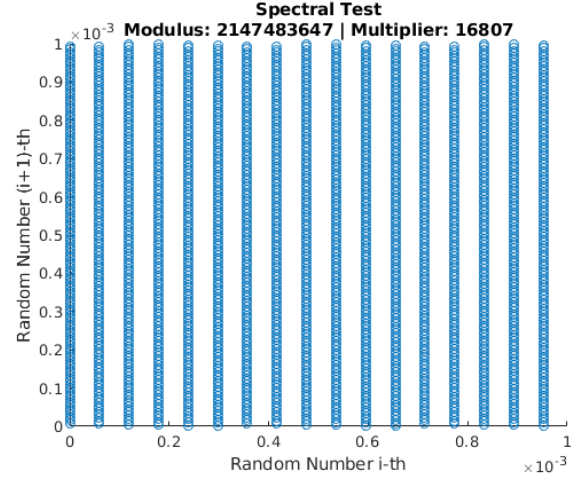


Figure 5: The Spectral Test to evaluate the randomness of the random number generator $(16807, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

fails), that is not far from the theoretical accepted number of fails, i.e. $256 \cdot 0.05 = 13$. 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 9 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 $d = 0.041 < 0.081 = d^*$.

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 with a preemption policy based on random selection.

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 global response time and mean population for the whole system. We assess the transient period of the aforementioned metrics because their behavior

²<https://aws.amazon.com/ec2/instance-types/>

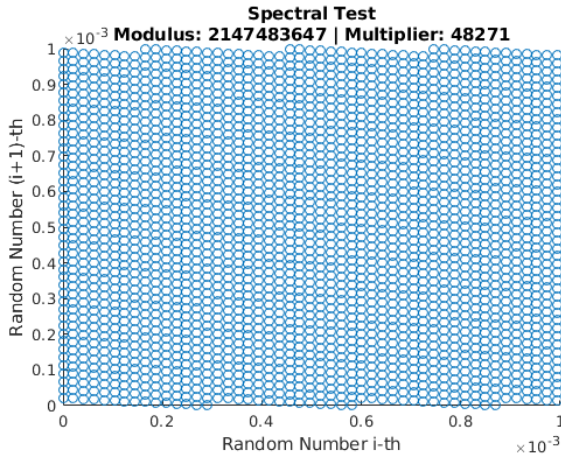


Figure 6: The Spectral Test to evaluate the randomness of the random number generator $(48271, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

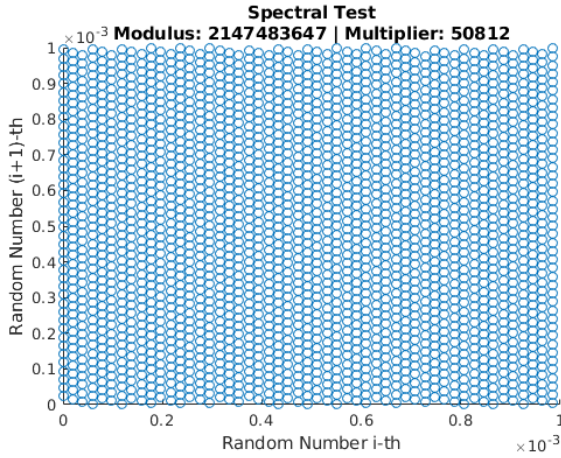


Figure 7: The Spectral Test to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

can be considered a good representation of the dependency of the system from the initial state.

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 10 we show the transient analysis of the global response time in the whole system. In Figure 11 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 τ^* we are sure that all metrics loosed their dependence on starting conditions.

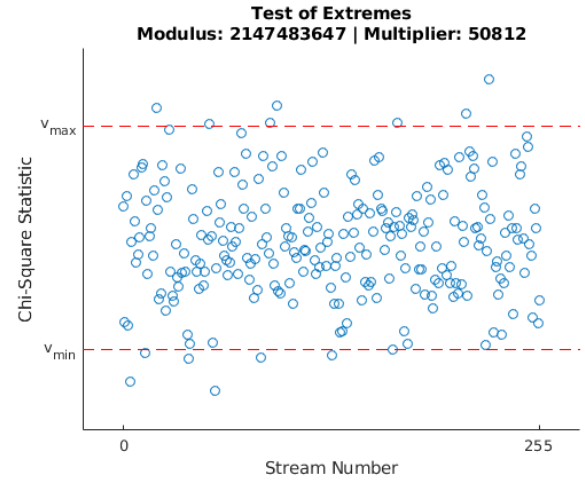


Figure 8: The Test of Extremes with $d = 5$ to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 256)$.

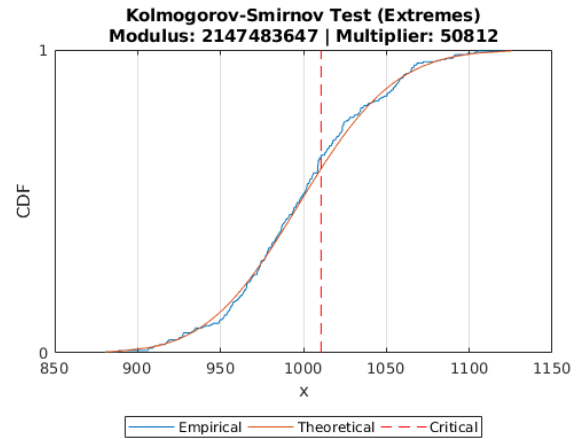


Figure 9: The Kolmogorov-Smirnov Analysis (leveraging the Test of Extremes with $d = 5$) to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 256)$ with 0.95 confidence level.

5.4 Performance Evaluation

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

- (1) *response time* both global and classed, both for the system as a whole and for each subsystem;
- (2) *throughput* both global and classed, both for the system as a whole and for each subsystem;
- (3) *population* both global and classed, both for the system as a whole and for each subsystem;
- (4) *switch ratio* for tasks belonging to the 2^{nd} class.
- (5) *switched response time* for tasks belonging to the 2^{nd} class.

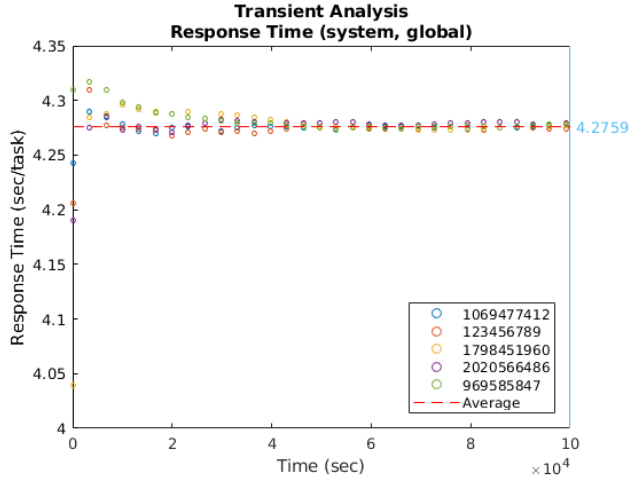


Figure 10: Transient analysis for global response time in the whole system.

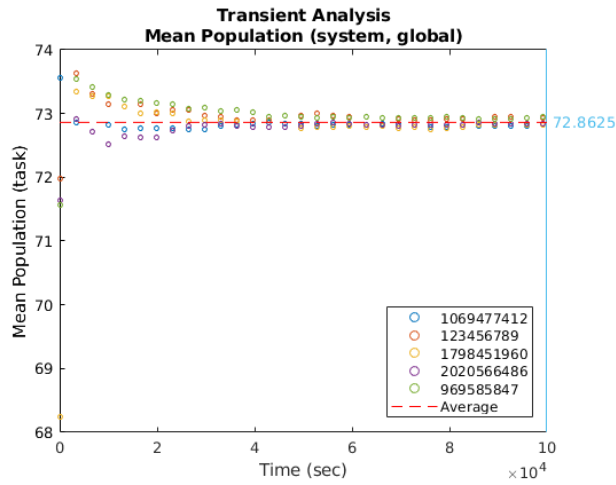


Figure 11: Transient analysis for global mean population in the whole system.

5.5 Distribution Analysis

In this Section we show the distribution analysis of the Cloudlet global throughput. In particular, we focus on the estimation of Probability Density Function (PDF) and the comparison between theoretical and experimental Cumulative Distribution Function (CDF).

In Figure 13 and Figure 14 we show the PDF estimation and the CDF analysis, respectively, for the global Cloudlet throughput when $S = N = 20$, where we adopted the *Freedman-Diaconis Rule* for the binning schema. Results show that the best fitting is the *Normal Distribution* with parameters $A \approx 4.357$ and $B \approx 3546.917$.

Index	Theoretical	Experimental
$E[N_{clt}]$	123456789	123456789 ± 0.00342
$E[N_{1,clt}]$	123456789	123456789 ± 0.00342
$E[N_{2,clt}]$	123456789	123456789 ± 0.00342
$E[T_{clt}]$	123456789	123456789 ± 0.00342
$E[T_{1,clt}]$	123456789	123456789 ± 0.00342
$E[T_{2,clt}]$	123456789	123456789 ± 0.00342
X_{clt}	123456789	123456789 ± 0.00342
$X_{1,clt}$	123456789	123456789 ± 0.00342
$X_{2,clt}$	123456789	123456789 ± 0.00342
$E[N_{cld}]$	123456789	123456789 ± 0.00342
$E[N_{1,cld}]$	123456789	123456789 ± 0.00342
$E[N_{2,cld}]$	123456789	123456789 ± 0.00342
$E[T_{cld}]$	123456789	123456789 ± 0.00342
$E[T_{1,cld}]$	123456789	123456789 ± 0.00342
$E[T_{2,cld}]$	123456789	123456789 ± 0.00342
X_{cld}	123456789	123456789 ± 0.00342
$X_{1,cld}$	123456789	123456789 ± 0.00342
$X_{2,cld}$	123456789	123456789 ± 0.00342
$E[N_{sys}]$	123456789	123456789 ± 0.00342
$E[N_{1,sys}]$	123456789	123456789 ± 0.00342
$E[N_{2,sys}]$	123456789	123456789 ± 0.00342
$E[T_{sys}]$	123456789	123456789 ± 0.00342
$E[T_{1,sys}]$	123456789	123456789 ± 0.00342
$E[T_{2,sys}]$	123456789	123456789 ± 0.00342
X_{sys}	123456789	123456789 ± 0.00342
$X_{1,sys}$	123456789	123456789 ± 0.00342
$X_{2,sys}$	123456789	123456789 ± 0.00342

Figure 12: Evaluation of performance metrics with comparison between analytical results and experimental ones with level of confidence 95%.

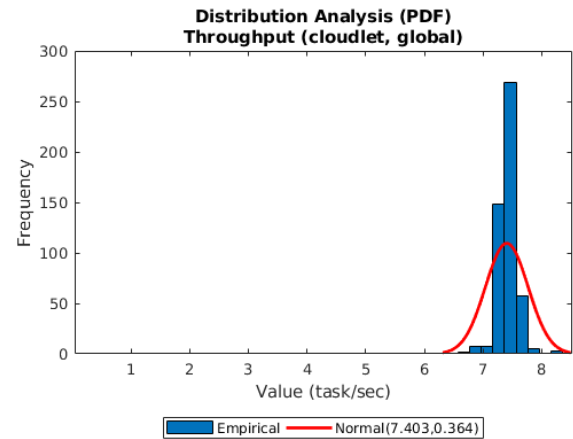


Figure 13: Distribution analysis (Probability Distribution Function) for the Cloudlet global throughput with threshold $S = N = 20$.

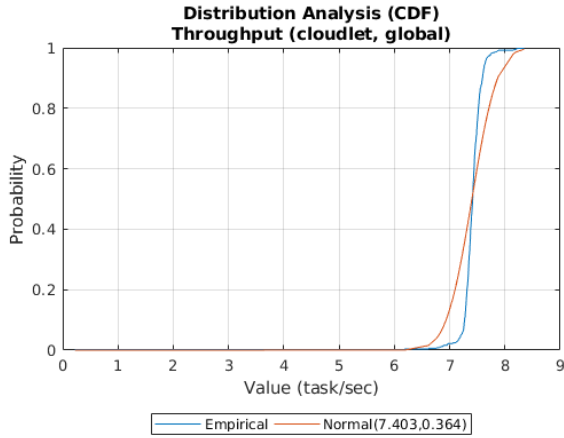


Figure 14: Distribution analysis (Cumulative Distribution Function) for the Cloudlet global throughput with threshold $S = N = 20$.

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 15 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 16 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 17 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 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

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

Generator
Class ..... MarcianiMultiStream
Streams ..... 2 5 6
Modulus ..... 2 1 4 7 4 8 3 6 4 7
Multiplier ..... 5 0 8 1 2
Seed ..... 1 2 3 4 5 6 7 8 9

Test Parameters
Sample Size ..... 1 0 0 0 0
Bins ..... 1 0 0 0
Confidence ..... 9 5 . 0
D ..... 5

Critical Bounds
Lower Bound ..... 9 1 3 . 3 0 0 9 9 8 3 0 9 0 6 4 4
Upper Bound ..... 1 0 8 8 . 4 8 7 0 6 7 5 3 3 8 2 6 1

Error
Theoretical ..... 1 3 (5.078 %)
Empirical ..... 1 4 (5.469 %)
Empirical Lower Bound ..... 7 (2.734 %)
Empirical Upper Bound ..... 7 (2.734 %)

Result
Suggested Confidence ..... 9 4 . 5 3 1
Success ..... False
```

Figure 15: A sample output of the Test of Extremes.

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.

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: Seminumerical 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>

```

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

Generator
Class ..... MarcianiMultiStream
Streams ..... 2 5 6
Modulus ..... 2 1 4 7 4 8 3 6 4 7
Multiplier ..... 5 0 8 1 2
Seed ..... 1 2 3 4 5 6 7 8 9

Test Parameters
Chi-Square Test ..... extremes
Sample Size ..... 1 0 0 0 0
Bins ..... 1 0 0 0
Confidence ..... 9 5 . 0
D ..... 5

KS
KS Statistic ..... 0 . 0 4 1
KS Point X ..... 1 0 1 0 . 6
KS Critical Distance ..... 0 . 0 8 4

Result
Success ..... True

```

Figure 16: 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 17: A sample configuration for a simulation experiment.

[6] Kenneth Reitz and Tanya Schlusser. 2016. *The Hitchhiker's Guide to Python: Best Practices for Development*. O'Reilly Media. <http://amzn.to/2bYCvBD>