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 (i) the system performance, both analytically and experimentally computed, and (ii) the quality of the adopted pseudo-random number generator. 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. Furthermore, we conclude our work delineating possible improvements for our simulator.

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 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 that can be 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.

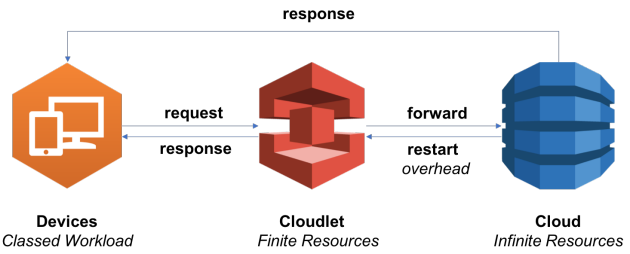

Figure 1: System architecture (high level).

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 the system insights in terms of performance metrics. In particular, considering $S = N$ we propose to evaluate system stationary and the following performance metrics with a 95% level of confidence:

- *system response time* both global and per-class;
- *response time* both global and per-class;
- *Cloud response time* both global and per-class;
- *system throughput* both global and per-class;
- *Cloudlet throughput* both global and per-class;
- *Cloud throughput* both global and per-class;
- *interruption percentage* of the 2^{nd} class tasks;
- *response time of interrupted tasks*;
- *system mean population* both global and per class;
- *Cloudlet mean population* both global and per class;
- *Cloud mean population* both global and per class;
- *distribution of Cloudlet throughput* in terms of distribution fitting and cumulative distribution function.

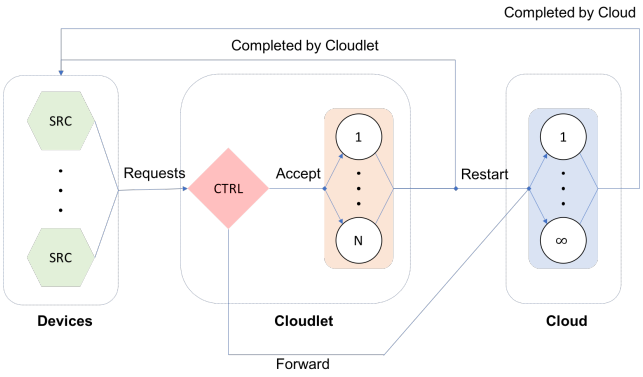

Figure 2: Conceptual model.

4.2 Conceptual Model

The conceptual model of the target system is the middle-level abstraction that makes the high-level architecture closer to the analytical model. We consider the conceptual model 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*: $S_{clt} := \{(n_{clt,1}, n_{clt,2}) \in \mathcal{N}^2 : n_{clt,1}+n_{clt,2} \leq N\}$, where $n_{clt,j}$ is the population of tasks belonging to the j-th class within the Cloudlet.
- *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 interrupted in the Cloudlet and restarted in the Cloud.

4.3 Specification Model

The typical modeling workflow requires to specify (i) the statistical analysis of data collected from the real system in order to determine the input model to drive simulations, (ii) the adopted simulation approach, (ii) the algorithms involved in computations and (iii) the analytical model and equations to determine performance metrics.

Given the importance of the analytical model, we reserve a dedicated section for discussing about it.

Regarding the simulation approach, we decided to adopt the *next-event simulation method*, which is the most effective discrete-event method in terms of algorithmic modeling, time management and computational requirements.

Statistical specifications. In our case we have been provided with the statistical characterization of the target system. 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. From the point of view of algorithms involved in the simulation, we need to specify:

- *off-loading algorithm*, which defines the off-loading policy implemented by the *Cloudlet controller (CTRL)*, as defined in Algorithm 1.
- *simulation algorithm*, which defines the main execution flow of the simulator, as defined in Algorithm 2.

With reference to the simulation algorithm, we need to focus on the following aspects:

- *arrival generation*: a new random arrival is generated every time an arrival is processed. As we need to submit classed arrival, we adopted Algorithm 3 whose statistical correctness relies on the properties of the exponential distribution.
- *event submission*: when a new event is submitted to the system, system components (i) updates both their state (ii) update simulation counters and (iii) generate a random response event, i.e. completion event or an interruption event.
- *stop condition*: when this condition holds true, the simulation is terminated. In particular, the logical definition is twofold: When the simulator is used for transient analysis, the condition holds true when the simulation clock is greater than the stop time. When the simulator is used for performance analysis, the condition holds true when the closed-door condition does and the system has reached the idle state.
- *closed-door condition*: when this condition holds true, no more arrivals will be generated and system will only handle remaining completion until its state reaches the initial idle condition.
- *sampling condition*: when this condition holds true, performance metrics should be sampled.

- *metrics management*: when the system handles an event, it updates simulation counters, e.g. number of arrivals, service time, integral areas and so on. When the sampling condition holds true, those counters are used to opportunely computes a sample of performance metrics. Such a sample is used to update performance metrics collection, which leverages the *one-pass Wellford algorithm, batch means and confidence intervals* using formulas specified in [4].

if *task of class 1* **then**
  if $n_{clt,1} = N$ **then**
    send to the Cloud
  end
  if $n_{clt,1} + n_{clt,2} < S$ **then**
    accept on the Cloudlet
  end
  if $n_{clt,2} > 0$ **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 $n_{clt,1} + n_{clt,2} >= S$ **then**
    send to the Cloud
  else
    accept on the Cloudlet
  end
end

Algorithm 1: Off-loading.

calendar.schedule_arrival();
while $\neg$*stop_condition*() **do**
  e = calendar.next_event();
  if $\neg$*close_door_condition*() $\vee$ *e.type* = *completion* **then**
    e_next = submit_event(e);
    calendar.schedule(e_next);
  end
  if $\neg$*close_door_condition*() $\vee$ *e.type* = *arrival* **then**
    calendar.schedule_arrival();
  end
  if *sampling_condition*() **then**
    sample = sampling();
    update_metrics(sample);
  end
end

Algorithm 2: Simulation.

4.4 Analytical Model

In this Section we 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.

rndgen.stream(ARRIVAL)
$p_1 = \frac{\lambda_1}{\lambda_1+\lambda_2}$
u = rndgen.uniform(0.0,1.0)
if $u \leq p_1$ **then**
  arrival_type = TASK_1
  rndgen.stream(ARRIVAL_TASK_1)
  $t_{inter-arrival}$ = rndgen.exponential(λ_1)
else
  arrival_type = TASK_2
  rndgen.stream(ARRIVAL_TASK_2)
  $t_{inter-arrival}$ = rndgen.exponential(λ_2)
end
$t_{arrival} = t_{last\_arrival} + t_{inter-arrival}$
$t_{last\_arrival} = t_{arrival}$
schedule(arrival_type,$t_{arrival}$)

Algorithm 3: Arrivals generation.

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

- $S_{clt,1}$: 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 interrupted in the Cloudlet and it 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. As the Markovian condition holds true for our system, i.e. Poisson arrivals[1] and Exponential services, we can determine the Markov Chain[2] whose resolution allows us to compute the routing probabilities shown in Equation 6.

[1] same as Exponential inter-arrivals.
[2] If the Markovian condition is not satisfied, Markov Chain solution must be considered only an approximation.

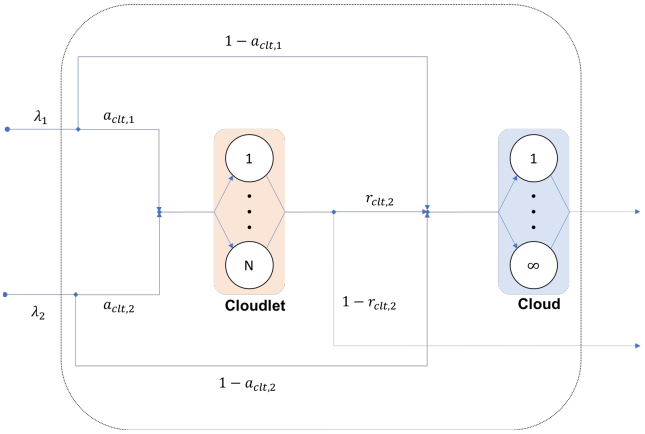

Figure 3: Analytical model.

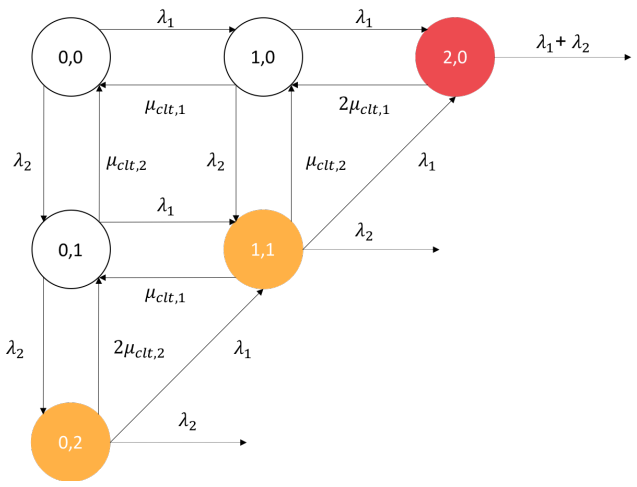

Figure 4: Markov Chain with $N = 2$ and $S = 2$.

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 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*: arrival rate of tasks belonging to j-th class accepted in Cloudlet:

$$\lambda_{clt,j} = a_{clt,j}\lambda_j \tag{8}$$

- *Cloud*: arrival rate of tasks belonging to j-th class forwarded to Cloud:

$$\lambda_{cld,j} = (1 - a_{clt,j})\lambda_j \tag{9}$$

- *Restarts*: rate of tasks belonging to 2^{nd} class interrupted in Cloudlet and restarted in Cloud:

$$\lambda_r = r(\lambda_1 + \lambda_2) \tag{10}$$

Performance metrics. Given the accepted workloads we can determine the following *performance metrics for classed tasks in each subsystem*. We assume here to know the expected time lost in Cloudlet by 2^{nd} class tasks before their interruption. In particular, as we are not able to determine $E[T_{clt,2,lost}]$ from the Markov Chain, we will assume the experimental value computed by the simulator.

- 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:

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

- 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 restarted):

$$
\begin{aligned}
E[T_{cld,2}]^{[NR]} &= \frac{1}{\mu_{cld,2}} \\
E[N_{cld,2}]^{[NR]} &= \lambda_{cld,2}E[T_{cld,2}]^{[NR]}
\end{aligned}
\tag{14}
$$

- 2^{nd} class in Cloud (restarted):

$$
\begin{aligned}
E[T_{cld,2}]^{[R]} &= E[T_{setup}] + E[T_{cld,2}]^{[NR]} \\
E[N_{cld,2}]^{[R]} &= \lambda_r E[T_{cld,2}]^{[R]}
\end{aligned}
\tag{15}
$$

- 2^{nd} class in Cloud (both restarted and not restarted):

$$\begin{aligned} E[T_{cld,2}] &= \sum_{m=NR,R} \frac{E[N_{cld,2}]^{[m]}}{E[N_{cld,2}]} E[T_{cld,2}]^{[m]} \\ E[N_{cld,2}] &= \sum_{m=NR,R} E[N_{cld,2}]^{[m]} \end{aligned} \tag{16}$$

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

- Cloudlet:

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

- Cloud:

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

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 routing probabilities and performance metrics.

We solved the the *analytical model* depicted in Figure 3 leveraging a *Python script* that (i) receives as input the system configuration (ii) generates the Markov Chain representing the Cloudlet (iii) generates the system of equations from the Markov Chain (iv) computes limiting probabilities by solving the system (v) computes routing probabilities (vi) computes performance metrics and (vii) display a report of results.

Analytical results are presented in Figure **??**, along with their experimental counterpart. We preferred to present analytical and experimental results in a unique common view, in order to provide the reader with an idea on how the simulator approximates analytical results.

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].

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 1^{st} class and an Exponential setup time for the 2^{nd} 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 2^{nd} class tasks in the Cloudlet.
- *server selection policy*: the user can configure the policy adopted to select the 2^{nd} 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 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.

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.
- *SimulationMetrics*: abstraction encapsulating all counters and performance 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 decentralize and isolate the updates of metrics leveraging the well-know IoC software design pattern.
- *BatchedMeasure*: an object that represents a measurement with both an instantaneous value and mean, standard deviation and confidence interval leveraging the batch means technique and the confidence interval estimation, as defined in [4].
- *WelfordAccumulator*: an object that is able to return in-place updated mean and standard deviation leveraging the *one-pass Welford algorithm*. This object is widely adopted in our software, as it is used to update batch means.
- *Sample*: an object that represents the instantaneous sample of simulation counters and performance metrics. It is used in our simulator to create a snapshot of metrics during the sampling process.
- *AnalyticalSolver*: an object that is able to compute the analytical solution for the target system. In particular, this solver leverages our Markov Chain generator and an efficient symbolic solver for the resolution of flow-balance equations. The calculus of performance metrics is ruled by the same formulas specified in section dedicated to the analytical model.

Correctness. The correct behavior of mission critical classes, e.g. pseudo-random generator, batch means and others, has been *unit-tested* with the built-in Python test suite.

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[3].
- **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

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

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

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 metrics. 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 **??** 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 applications[4]. 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 $U = U_0, ..., U_{d-1}$ 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 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 $d = 0.041 < 0.081 = d*$.

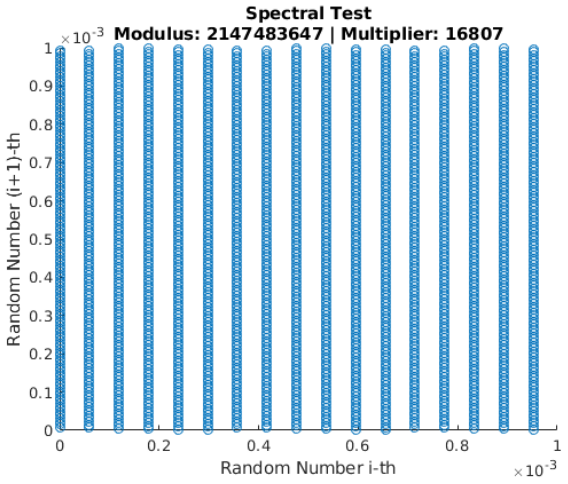

Figure 5: The Spectral Test to evaluate the randomness of the random number generator $(16807, 2^{31}-1, 1)$ in the interval $(0, 10^{-3})$.

5.2 Performance Analysis

Let us now consider the experimental results about system performance recorded by our simulator. In all experiments we considered values stated in Section 4 with a preemption policy based on *random selection*.

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

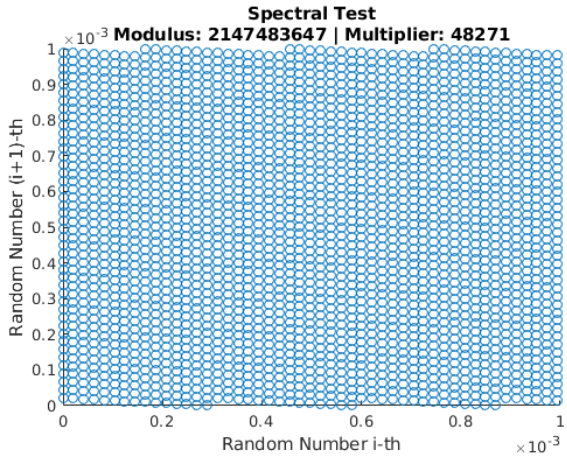

Figure 6: The Spectral Test to evaluate the randomness of the random number generator $(48271, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

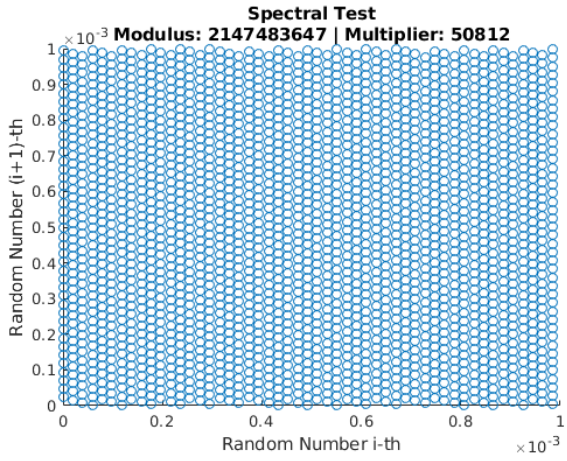

Figure 7: The Spectral Test to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 1)$ **in the interval** $(0, 10^{-3})$**.**

5.3 Transient Analysis

First, we conduct a *transient analysis* to evaluate the system stationary in order to (i) prove its convergence to the steady-state and (ii) 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 system throughput as it 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.

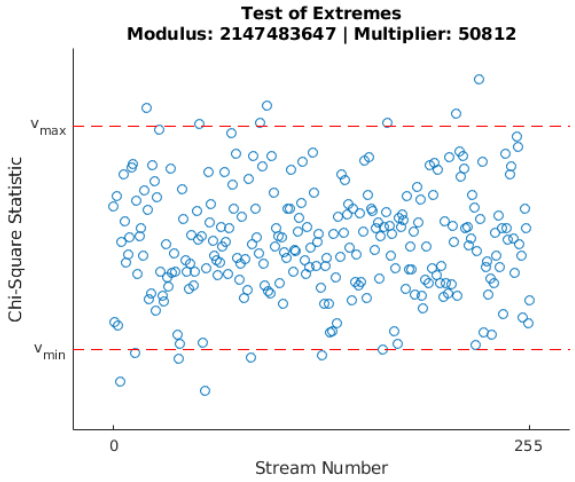

Figure 8: The Test of Extremes with $d = 5$ **to evaluate the randomness of the random number generator** $(50812, 2^{31} - 1, 256)$**.**

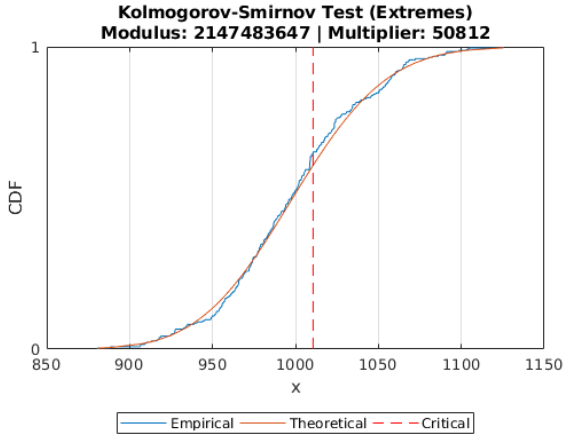

Figure 9: The Kolmogorov-Smirnov Analysis (leveraging the Test of Extremes with $d = 5$**) to evaluate the randomness of the random number generator** $(50812, 2^{31} - 1, 256)$ **with** 0.95 **confidence level.**

In Figure 10 we show the transient analysis of the global throughput in the whole system.

The results show that the system reaches the steady-state. This result is not surprising, because the presence of a stabilizing *infinite-buffer centre*, i.e. the Cloud, largely compensates the possible instability of the *thresholded finite-buffer centre*, i.e. the Cloudlet.

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;

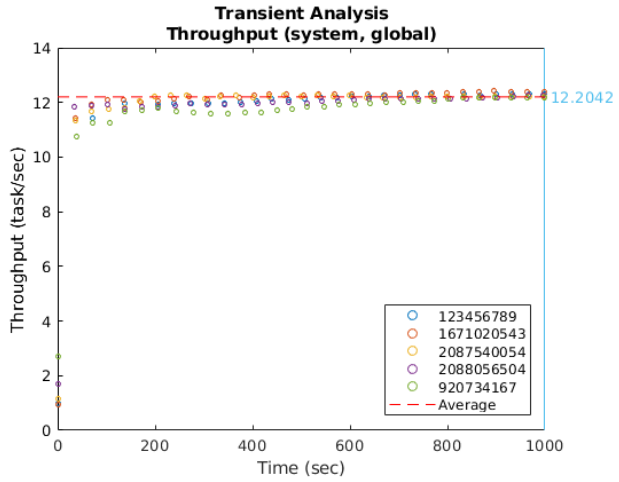

Figure 10: Transient analysis for global throughput in the whole system.

Measure	Theoretical	Experimental
$a_{clt,1}$	0.978326334857105	0.97911346521
a_{clt_2}	0.603529764734761	0.60502880468
r	0.183573830264005	0.15525744148

Figure 11: Routing probabilities: comparison between the theoretical result, computed with the analytical model, and the experimental result, computed leveraging our simulator.

(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) *interrupted ratio* for tasks belonging to the 2^{nd} class.
(5) *interrupted response time* for tasks belonging to the 2^{nd} class.

In our experiment we assume $S = N = 20$, $k = 64$ batches, batch dimension $b = 512$ and initial seed $iseed = 123456789$. Theoretical results have been computed using the formulas presented in Section 4, assuming the experimentally computed value $E[T_{clt,2,lost}] = 1.47445; s.$.

In Table

5.5 Distribution Analysis

In this Section we show the distribution analysis of the Cloudlet global throughput. In particular, we focus on both (i) the Probability Density Function (PDF) estimation leveraging distribution fitting, and (ii) 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 $\mu \approx 7.403$ and $\sigma \approx 0.364$. The Normal behavior shown here can be considered as a further good proof of both the system stationary and the effectiveness of the batch means as a tool to study steady-state statistics.

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

Figure 12: Performance metrics: comparison between the theoretical result, computed with the analytical model, and the experimental result with level of confidence 95%, computed leveraging our simulator.

6 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 multi-threading to achieve better performances, e.g. to speed-up

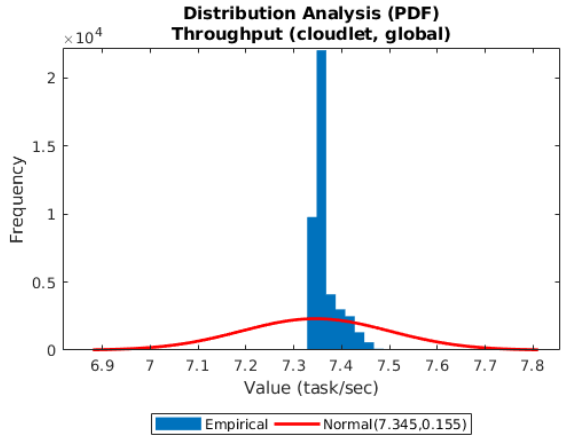

Figure 13: Distribution analysis (Probability Distribution Function)for the Cloudlet global throughput with threshold $S = N = 20$.

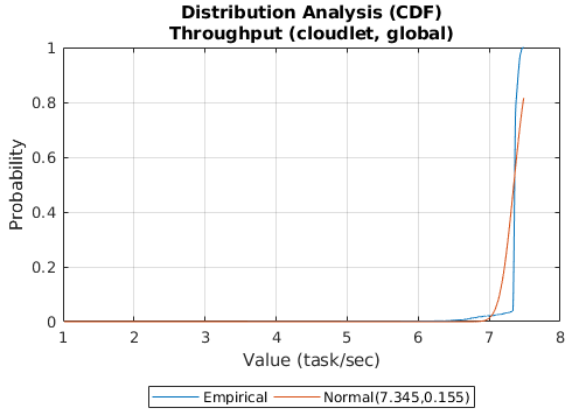

Figure 14: Distribution analysis (Cumulative Distribution Function) for the Cloudlet global throughput with threshold $S = N = 20$.

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: 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