

A Performance Modeling Showcase

Giacomo Marciani

University of Rome Tor Vergata

Via del Politecnico 1

Rome, Italy 00133

gmarciani@acm.org

ABSTRACT

As computing is getting more ubiquitous in our lives, modern software applications and computer infrastructures are getting increasingly complex and require huge financial investments. In this context, knowing how to design, redesign and optimize computer systems and networks is one of the most important skills for software engineers because it can guarantee companies a competitive advantage, both in terms of technology and investment.

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

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

CCS CONCEPTS

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

KEYWORDS

performance modeling; simulation tools

ACM Reference format:

Giacomo Marciani. 2018. A Performance Modeling Showcase. In *Proceedings of A Performance Modeling Showcase, Rome, Italy, January 2018 (PMCSN'18)*, 5 pages.

https://doi.org/10.475/123_4

1 INTRODUCTION

As computing is getting more ubiquitous in our lives, modern software applications and computer infrastructures are getting increasingly complex and require huge financial investments. In

this context, knowing how to design, redesign and optimize computer systems and networks is one of the most important skills for software engineers because it can guarantee companies a competitive advantage, both in terms of technology and investment. In this technical report we propose a next-event simulation model to analyze the performance of a two-layer Cloud system that leverages an off-loading policy to minimize response-time.

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

2 RANDOM NUMBER GENERATION

The generation of pseudo-random numbers is a fundamental building-block for next-event simulations. 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 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 *multiple recursive generators*, *composite generators*, and *shift-register generators*. Although all of these produce periods wider than the one produced by a Lehmer generator, they also have a sensible cost in terms of computational complexity.

In this work we adopted a multi-stream Lehmer generator (a, m, s) , which 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 is a de-facto standard, hence it is easy to compare our experimental results with the one 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

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, January 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

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 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 [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 f extremes* and the *analysis of Kolomogorv-Smirnov*. The experimental results are reported in Section 4.

3 MODELING

We consider the system depicted in Figure ??, consisting of:

- mobile devices sending tasks partitioned in two classes, whose arrival process is exponential with rates λ_i ;
- a Cloudlet with N servants, whose service process is exponential with rate $\mu_{clt,i}$;
- a remote Cloud server with virtually unlimited resources, whose service process is exponential with rate $\mu_{cld,i}$;
- a front-dispatcher, responsible to route requests to Cloudlet or Cloud, according to the policy defined in Algorithm ??, that takes into account the number n_{clt} of tasks in Cloudlet, the number n_{cld} of tasks in Cloud and the threshold S .

We assume that service time includes the transmission time and $\mu_{clt,i} > \mu_{cld,i} \forall i = 1, 2$. When a class 2 task is interrupted in the Cloudlet and sent to the cloud, an exponentially distributed setup time T_{setup} has to be considered to restart the task on the Cloud.

The main goal of the simulation is to determine with a 95% level of confidence (i) the response time and the throughput as a function

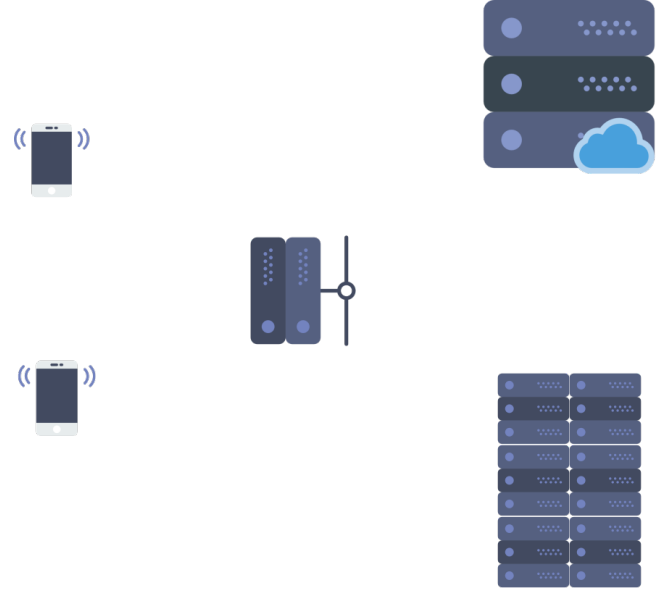


Figure 1: The system sketch.

```

if arrival ∈ C1 then
  if n1 = N then
    | send on the cloud
  end
  if n1 + n2 < S then
    | accept
  end
  if n2 > 0 then
    | accept the task on the cloudlet and send a class 2
    | task on the cloud
  else
    | accept the task on the cloudlet
  end
end
if arrival ∈ C2 then
  if n1 + n2 ≥ S then
    | send on the cloud
  else
    | accept the task on the cloudlet
  end
end

```

Algorithm 1: The dispatching policy.

of the threshold S (ii) the distribution of the response time when $S = N$ and (iii) the threshold S^* that minimizes the response time.

The conceptual model is depicted in Figure ??.

The state of the system is represented by the pair (n_1, n_2) , where n_1 is the number of tasks in Cloudlet and n_2 is the number of tasks in Cloud.

The model has been verified by ...

It is well-known that model development includes a final validation step. Clearly, we cannot conduct such a final process because we cannot compare model results with a real system.

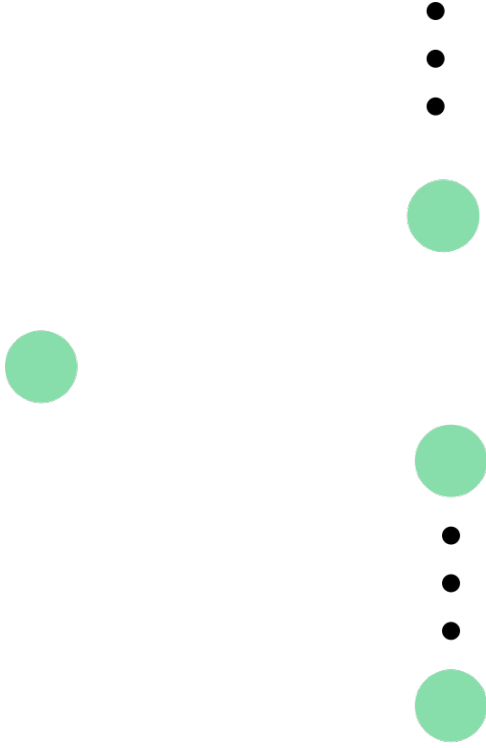


Figure 2: The conceptual model.

4 EVALUATION

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

The experiments have been conducted on an Amazon EC2 c3.8xlarge instance, which is really indicated for high performance science and engineering applications¹. 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, 5].

Let us now consider the results about the randomness 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 $(58012, 2^{31} - 1)$, respectively.

The results show that our generator $(58012, 2^{31} - 1)$ is much better than $(16807, 2^{31} - 1)$, which was a past de-facto standard, and 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 iid 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.

The more the total number of fails is close to the expected value, i.e. $streams \cdot confidence$, the better the generator is. In Figure ?? we show the test results for the proposed generator $(58012, 2^{31} - 1, 256)$ 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. $256 \cdot 0.05 = 13$. The proposed generator successfully passed the test with a 94.531% level of confidence.

- **Kolmogorov-Smirnov Analysis:** the test measures, at a certain confidence level, 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 text of extremes.

In Figure ?? we show the test results for the proposed generator $(58012, 2^{31} - 1, 256)$ with a 95% level of confidence. The proposed generator successfully passed the test, as $d = 0.041 < 0.081 = d^*$.

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

- **arrivals:** exponential with rate $\lambda_1 = 3.25task/s$ and $\lambda_2 = 6.25task/s$.
- **services:** exponential with rate $\mu_{clt,1} = 0.45task/s$, $\mu_{clt,2} = 0.30task/s$, $\mu_{cld,1} = 0.25task/s$ and $\mu_{cld,2} = 0.22task/s$.
- **setup:** exponential with mean $E[T_{setup}] = 0.8s$.

arrival rates, exponential service rates and exponential setup time

5 CONCLUSIONS

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

Although experimental results are pretty satisfactory, the proposed solutions could certainly be improved and be subjected to a more in-depth analysis. From an implementation point of view, the proposed solution should be ported from Python to C and leverage multi-threading to achieve better performances, e.g. to speed-up

¹<https://aws.amazon.com/ec2/instance-types/>

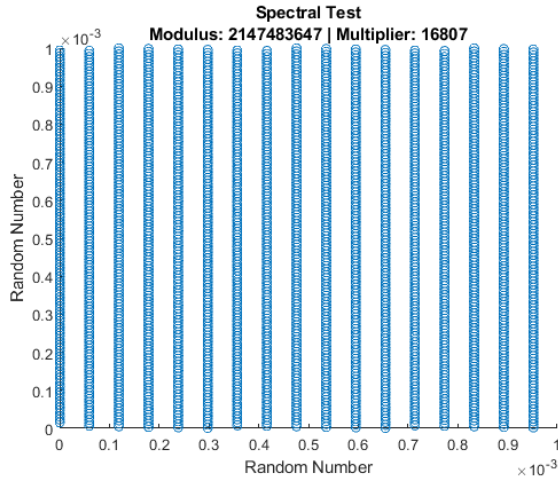


Figure 3: The Spectral Test to evaluate the randomness of the random number generator $(16807, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

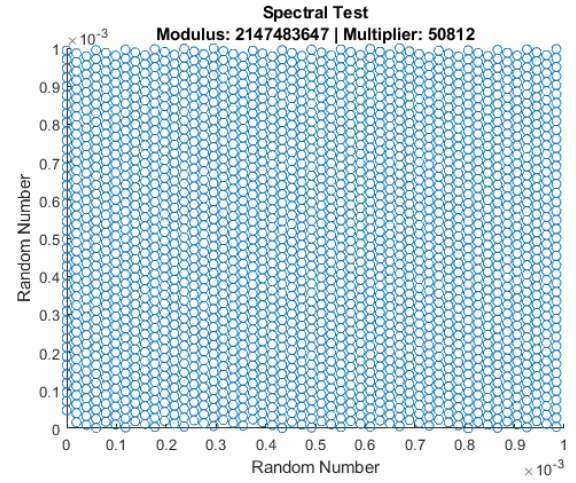


Figure 5: The Spectral Test to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

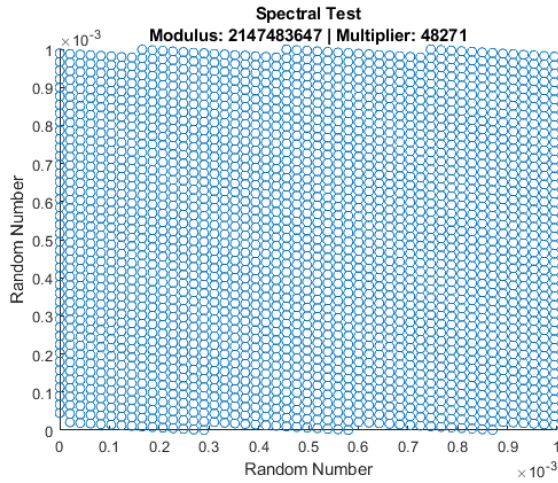


Figure 4: The Spectral Test to evaluate the randomness of the random number generator $(48271, 2^{31} - 1, 1)$ in the interval $(0, 10^{-3})$.

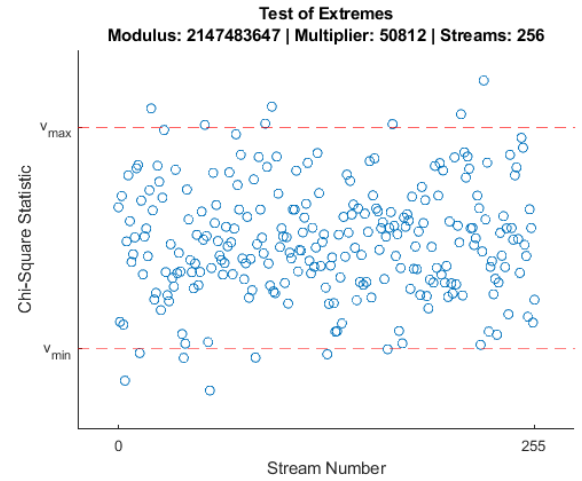


Figure 6: The Test of Extremes with $d = 5$ to evaluate the randomness of the random number generator $(50812, 2^{31} - 1, 256)$.

the algorithms to find suitable multipliers for modulus in 64-bit architectures and make the simulation faster. From an analysis point of view, the proposed random number generator should be tested more extensively, e.g. taking into account more tests of randomness, and a generator for 64-bit architectures and a reduced number of streams should be developed. Finally, the simulation model should be extended to take into account more performance metrics.

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

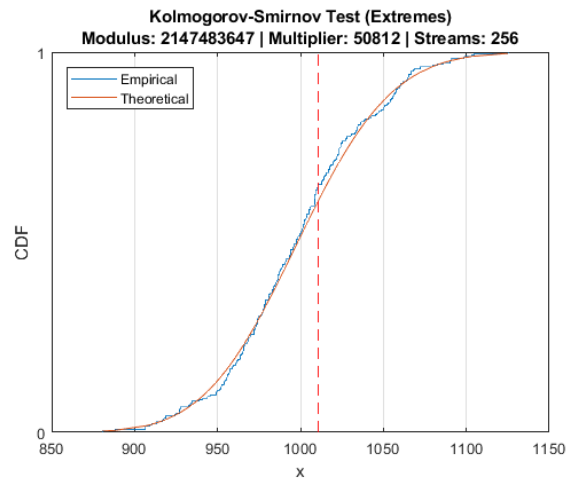


Figure 7: The Kolmogorov-Smirnov Analysis (leveraging the Test of Extremes with $d = 5$) to evaluate the randomness of the random number generator ($50812, 2^{31} - 1, 256$) with 0.95 confidence level.



Figure 8: Response Time Analysis, as a function of N .



Figure 9: Throughput Analysis, as a function of N .



Figure 10: The response time distribution with $S = N$.