Experiences with the ns-2 Network Simulator - Explicitly Setting Seeds Considered Harmful

Martina Umlauft
Women's Postgraduate College for Internet Technologies, Vienna University of Technology, Favoritenstr. 9-11, 1040 Vienna, Austria
umlauft@wit.tuwien.ac.at

Peter Reichl
Telecommunications Research Center Vienna (FTW), Donau-City-Str. 1, 1220 Vienna, Austria
reichl@ftw.at

Abstract

The ns-2 network simulator is one of the most widely used packet network simulators. In version 2.1b9 its old random number generator was replaced by an implementation of MRG32k3a to fix sensitivity to seeds. Due to bad documentation and re-use of old scripts many people still use the old API functions to explicitly set seeds. Unfortunately, this corrupts the correct function of the new generator and can lead to correlated simulation results. This might affect the majority of ns-2 simulation results currently published. We show why this is the case, illustrate possible effects, and how to avoid the problem.

1. Introduction

Over the last couple of years, the ns-2 simulator has become one of the most widely used environments for packet network simulation. Up until version 2.1b8 it used an implementation of the minimal standard multiplicative linear congruential generator by Park and Miller [1] for random number generation. This has been shown to exhibit several weaknesses: apart from the short period length of only $p=2^{31}-2$, which can be a problem for long-running simulations, Entacher and Hechenleitner [2] showed that it is sensitive to the chosen seed. Depending on the choice of seeds, it exhibits correlation between random variables created with these seeds.

In version 2.1b9, the combined multiple recursive generator MRG32k3a proposed by L'Ecuyer was introduced as random number generator to remedy these problems [3, 4, 5]. It is still used in all versions of ns-2 up to and including the current version 2.30.

The old minimal standard generator required to set seeds for random variables explicitly as shown in a typical example below:

```
set rng [new RNG]
$rng seed <n>
set e [new RandomVariable/Exponential]
$e use-rng $rng
```

This creates a new RNG object (line 1) and seeds it (line 2) where <*n*> can be replaced by any positive integer or *0*. In lines 3 and 4 an exponential random variable which uses this RNG object is set up. Typically, the seed would be set once in the simulation script depending on the number of the simulation run. Then, the final result of the simulation would be calculated by averaging over the trace output of several (many) simulation runs. Therefore, if the random numbers created by the RNG using these different seeds are correlated, this results in correlation between the output of those separate simulation runs. This is, of course, undesirable.

In the current MRG32k3a implementation the same approach of setting the seed (hereafter called the "old API" or "old method") wrongly overrides the automatic seed generation of the new generator without giving any error message. Especially experienced users who re-use old simulation scripts containing the old seed setting method have no chance to realize that *even though they are using the new implementation of the random number generator, they can still get correlated results*. Due to this lack of respective error messages and because of bad documentation - the ns Manual [6] still mentions the old API functions without any warning, and it is also still propagated on the ns-user mailing list or in popular lecture notes like [7] (see Sec-

tion 2), this is *completely unnoticable to the unsuspecting user*.

In this paper we demonstrate by experiment (see Section 3.1) that using the old seed setting method compromises the insensitivity of the new RNG to bad seeds and may again result in correlation between the random variables if bad seeds are chosen and show why this is the case (Section 3.2). We illustrate possible effects on network simulation results using a simple wired simulation topology (Section 4.1) and a wireless example (Section 4.2).

As the ns-2 community relies heavily on exchanging hints and scripts between each other, we believe that this might affect up to 80% of all ns-2 simulation results currently published. We show how to avoid the problem in Section 5 and conclude the paper with some thoughts on the impact on currently published simulation results in Section 6.

2. Documentation Issues

The official ns-2 manual [6] can be misunderstood on the issue of correctly seeding the RNG. While it states "You should only set the seed of the default RNG." on p. 218 it still shows the old API functions for seed setting on p. 220, 223, and 226 without any warning that this compromises the seed-insensitivity of MRG32k3a. Searching for "rng seed" on http://www.isi.edu/cgi-bin/nsnam/htsearch (the archive of the ns-2 user mailing list ns-users@isi.edu) gives 73 matches for 2005 and 2006, as shown in Table 1.

Table 1. Postings on the ns-2 user mailing list.

Type of Posting	
Advice or example incorrectly using old method	22
Correct advice in response to seeding question	2
Example containing correct method in other context	7
Ambigous example or advice	4
Advice to use consecutively numbered seeds	2

Also the popular and otherwise excellent lecture notes by Eitan Altman and Tania Jimenez [7] include several examples of old API function usage.

3. Undesirable Behavior of the new ns-2 RNG

The MRG32k3a random number generator is due to L'Ecuyer [3] and belongs to the class of "combined multiple recursive generators". Such generators are defined as normalized linear combinations of J copies of ordinary multiple recursive generators $x_{j,n}$ of order k ($j = 1, \ldots, J$), i.e.

$$x_{j.n} = (a_{j,1}x_{j,n-1} + \ldots + a_{j,k}x_{j,n-k}) \bmod m_j \qquad (1)$$

with distinct primes m_j and $a_{j,l}$ being naturals between 0 and m_j.

More specifically, MRG32k3a has J=2 components of order k=3 and a period length of approx. 3.1×10^{57}, and has been demonstrated to behave well for a broad range of statistical test scenarios. In [5], it is shown how this generator can be further generalized for producing multiple streams and substreams. To this end, it is proposed to cut the resulting (long) sequence of random numbers into adjacent streams of length $Z=2^z$ and then partition each such stream into 2^v substreams (blocks) of length $W=2^{z-v}$. Note that according to [6] this generator provides 1.8×10^{19} independent streams of random numbers, each of which consists of 2.3×10^{15} substreams with a period of 7.6×10^{22} each. In ns-2, each of these substreams corresponds to an individual RNG object, hence on creation of a new RNG object, simply the next substream is used.

In order to start the MRG32k3a, we need initial values for each of the six variables $\{x_{1,0}, x_{1,1}, x_{1,2}; x_{2,0}, x_{2,1}, x_{2,2}\}$ which can conveniently be described as a six-dimensional "seed vector". It is crucial to note that the nearly perfect randomness of the entire (long) sequence is of course maintained approximately also on a stream and substream level and thus for every RNG object, whereas setting explicitly a new seed vector for a newly created RNG object only could destroy this extremely desirable insensitivity property.

3.1. Simple Correlation Experiment

We implemented a simple simulation script where we set up three uniform random variables using the old API functions as follows:

```
for {set i 0} {$i < 3} {
   set rng($i) [new RNG]
   $rng($i) seed $n($i)
   set u($i) [new RandomVariable/Uniform]
   $u($i) use-rng $rng($i)
}
```

Table 2. Sets of Seeds.

Random variable	Set 1 ("good")	Set 2 ("bad")	Set 4 ("bad")
`$u(1)/$n(1)`	1973272912	1	1
`$u(2)/$n(2)`	1822174485	2	634005912
`$u(3)/$n(3)`	1998078925	3	634005911

For the `$n($i)` we use different sets of seeds as shown in Table 2. We then interpret the values drawn for `$u(1)`, `$u(2)`, and `$u(3)` as a vector and plot them as shown in Figure 1 for the new MRG32k3a

RNG and the old Park/Miller RNG. We also plot the results for the new RNG using the new seed setting method and for the old RNG using a set of known "good" seeds. The seed values are taken from [2] with set 1 being a set of known "good" seeds and sets 2 and 4 consisting of known "bad" seeds for the old Park/Miller RNG. While the actual numbers generated are different for MRG32k3a and the Park/Miller RNG, we can see that the behavior is similarly bad for "bad" seed choices (actually, the difference is not noticable at the resolution of the figures).

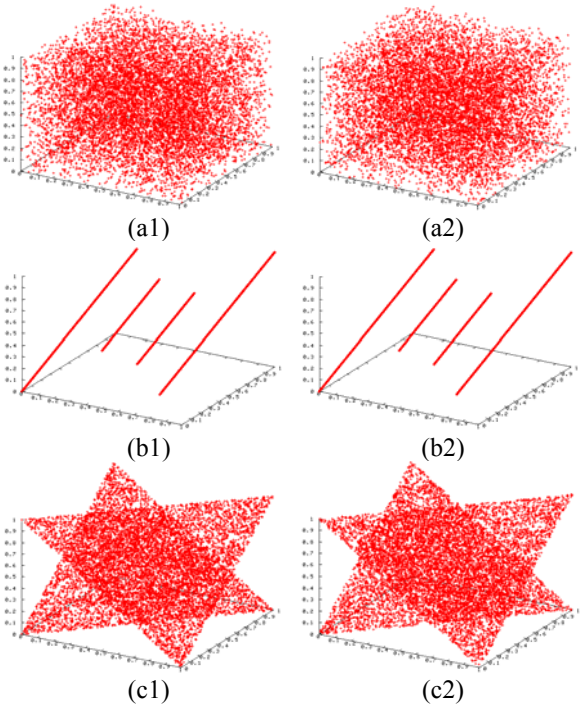

Figure 1. Correlation between three random variables for 10,000 values drawn. Left: MRG32k3a, right: old Park/Miller RNG. (a1) new (correct) seeding method, (a2) Seedset 1 (known "good" seeds), (b1/2) Seedset 2, (c1/2) Seedset 4.

3.2. Source Code Inspection

We inspect the files `rng.cc` (and `rng.h`) in the `ns/tools` directory. The OTcl command `$rng seed <n>` is processed in C++ by the `command` function on line 219 which eventually calls `RNG::set_package_seed()` which passes the seed into each of the 6 members of `next_seed_`, the package-wide 6-dimensional seed vector. It is a static member of the RNG class; iow. it exists only once for *all* objects of type RNG and is shared among them. On creation, each new RNG object will use the seed vector to seed itself and then recalculate `next_seed_` (line 754ff) to set it up for the next RNG object to be created (compare calculation of $x_{j,n}$ above).

Therefore, seeding each new RNG object explicitly overrides the seed vector calculation mechanism of MRG32k3a and hard-seeds the RNG to the values given in the OTcl script! As demonstrated in Section 3.1, this leads to correlation among random variables when bad seeds are chosen.

4. Effects on Network Simulation Results

To illustrate the effect on network simulation results we investigate 2 examples, a simple wired topology and a small wireless example. While these examples are chosen for simplicity and use very simple models, an effect can still be shown.

4.1. Simple Wired Topology Example

Analogous to [2] we generated a simple topology as shown in Figure 2. Node 1 contains a DropTail Queue with a maximum size of 1000 packets. G1 to G5 are exponential traffic generators of type `Application/Traffic/Exponential` generating on/off traffic.

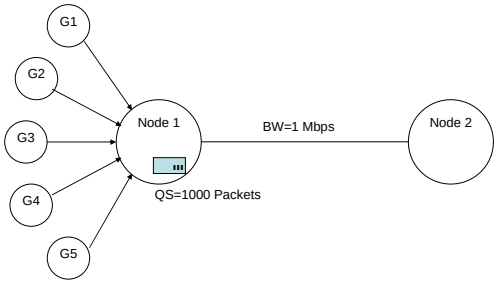

Figure 2. Simple wired simulation topology.

During each on-interval 1 packet of 1000 bytes with an internal "rate" of 1 Gbps is generated (resulting in an on-time of 0.08 µs). The mean of the exponentially distributed off-time is set to 41 ms, resulting in an average arrival rate of $\lambda = 8000\ bits/41\ ms = 0.195\ Mbps$ for each generator and $\Sigma\lambda = 0.976\ Mbps$ total giving a utilization factor of $\rho = \Sigma\lambda/BW = 0.976$. Calculating the mean queue length as

$$\bar{q} = \frac{\rho}{1-\rho} - \frac{\rho^2}{2(1-\rho)} \qquad (2)$$

we expect an average queue length of $\bar{q} = 20.488$ packets.

Table 3 gives the seedsets used for the simulations. Simulation time was 7200 s with a sampling interval of 10 ms. Table 4 shows the average queue lengths measured for the new and old RNGs using the new seeding method vs. a known bad seedset for the new RNG and a known good seedset vs. a known bad seedset for the old RNG. As can be seen, the use of the bad seedset leads to higher values for the average queue length while with the new method or known good seedset the value is close to the theoretical result.

Table 3. Sets of seeds.

Generator	Set 1 ("good")	Set 2 ("bad")
G1	1973272912	1
G2	1822174485	2
G3	1998078925	3
G4	678622600	4
G5	999157082	5

Table 4. Average queue lengths.

RNG	Seedset	Avg. Queue Length
new MRG32k3a	New method	20.2996
new MRG32k3a	2 ("bad")	29.4527
old Park/Miller	1 ("good")	19.4398
old Park/Miller	2 ("bad")	24.2785

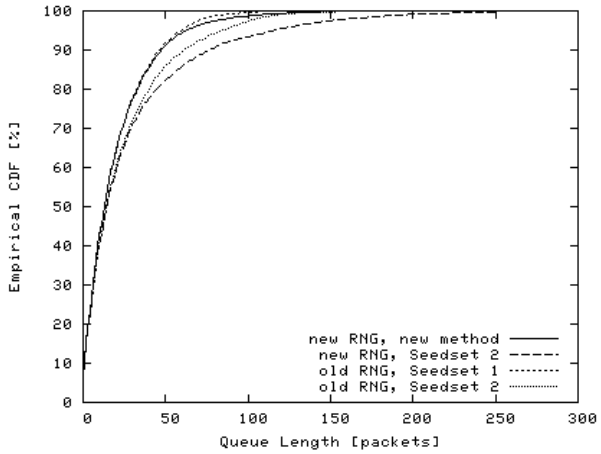

Figure 3. Distribution of queue lengths.

4.2 Simple Wireless Example

As another example to demonstrate the effect we chose a simple wireless model (see Figure 4). In this example, 4 nodes are connected by wireless hops and CBR traffic is sent via UDP from node N1 via intermediate nodes N2 and N3 to the sink N4. For reasons of simplicity, we model the wireless channels with a simple errormodel (`ErrorModel` in ns-2, installed as `lossmodel` on the links between the nodes) which uses a uniform distribution with an error rate of 10^{-3} to randomly drop packets sent on the link. While we are aware that a uniform error model does not reflect the reality of a wireless channel well (as errors are bursty for wireless media) this does not matter as the aim of the example is just to show that the experienced burstyness of a channel is changed when a bad method of seeding the RNG is used.

Each link has a capacity of 1 Mbps and the source generates CBR traffic with a rate of 0.9 Mbps so all drops occur due to losses on the links. Each link has its own errormodel which uses its own RNG which in turn is either seeded via the incorrect old method with different seeds or not seeded at all (using the new method). Simulation time is 600 seconds (a 10 min flow) for each replication.

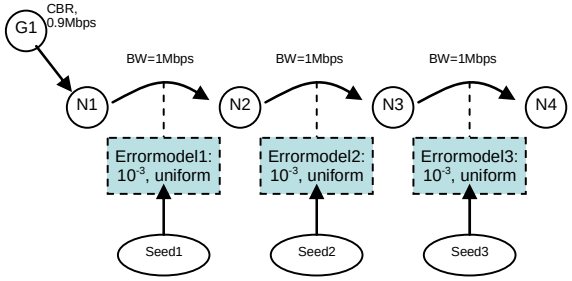

Figure 4. Simple Wireless Topology.

While with the old method, several replication runs of the same simulation were differentiated by setting different seeds for every replication, the API for the new method offers the `next-substream` function to set up the RNG. The code below sets up the 3 RNGs for the 3 errormodels according to the current replication (given in `$rep`).

```
for {set i 1} {$i < $rep} {incr i} {
   $rng1 next-substream;
   $rng2 next-substream;
   $rng3 next-substream;
}
```

We investigate the new MRG32k3a generator and compare the results of 10 replications using the new method with the known bad seedset 2 from Table 3 using the first 3 seeds 1, 2, and 3 for the RNGs of Errormodel 1, 2, and 3 respectively.

Figure 5 shows the ECDF of the lengths of good packet runs. As can be seen, all 10 replications generated with the new method yield quite similar results while the result generated with the bad seedset 2 differs significantly: there are more short and many more medium sized (approx. 220 packets) runs for the bad

seedset, while the number of really long runs (1000 packets or more) is lower than for the replications using the correct new method of setting up the RNG.

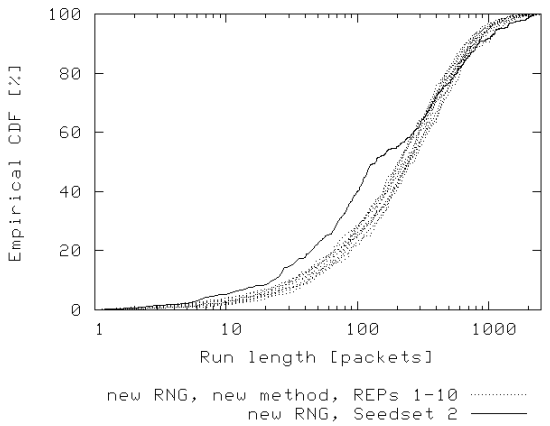

Figure 5. Lengths of good packet runs.

5. How to Avoid this Problem

This problem can be avoided by *only ever using* the new method to seed the RNG as shown in the example code on p. 217f, Section 24.1.1 in [6] and again below using the `next-substream` API function instead of seeding each of the RNG objects explicitly:

```
for {set i 1} {$i < $rep} {incr i} {
   $rng1 next-substream;
   $rng2 next-substream;
   $rng3 next-substream;
}
```

The above sets up 3 RNG objects according to the current replication given in `$rep`. Random numbers drawn from these RNG objects will not be correlated.

Optionally, the `defaultRNG` object (*but **none** of the other RNG objects*) may be seeded.

6. Conclusion: Impact on Currently Published Simulation Results

Using the old method to explicitly set seeds for the current MRG32k3a RNG in ns-2 (version 2.1b9 and above) results in overwriting of the package-wide seed of the generator, thereby confounding the new, automatic seed generation mechanism. If bad seeds are chosen, this leads to correlation between the generated random variables. Not only can the ns-2 manual be misunderstood on this issue, also the majority of postings on the ns-users mailing list for 2005/06 give outdated (and therefore incorrect) or outright harmful advice in this regard (28 incorrect vs. 9 correct). In addition, other popular literature gives incorrect examples and *no* respective error message is thrown. Therefore, we believe that a very large number (maybe even the vast majority) of ns-2 simulation results currently published is based on scripts using an incorrect method to seed the RNG. Out of those, the number of actually affected results is hard to estimate as it depends on several other factors: the choice of seed values (if good seeds are chosen there is no problem) and how the RNG objects are used. Results most prone to correlation are those which use several RNG objects seeded with different seeds within a single simulation run and consecutively numbered seeds.

The problem can be completely avoided by using *only* the new method to seed the RNG which we strongly recommend!

7. Acknowledgements

This research has been partly funded by the Austrian Federal Ministry for Education, Science, and Culture, and the European Social Fund (ESF) under grant 31.963/46-VII/9/2002 and partly by the Austrian Kplus competence center program.

8. References

[1] S.K. Park and R.W. Miller: Random number generation: Good ones are hard to find. Communications of the ACM, 31(10), pages 1192–1201, October 1988.

[2] B. Hechenleitner and K. Entacher: On Shortcomings of the ns-2 Random Number Generator. In T. Znati and B. McDonald, editors, Communication Networks and Distributed Systems Modeling and Simulation (CNDS), 2002.

[3] Pierre L'Ecuyer: Good parameters and implementations for combined multiple recursive random number generators. Operations Research, 47(1), pages 159–164, 1999.

[4] Pierre L'Ecuyer: Software for uniform random number generation: Distinguishing the good and the bad. In Proceedings of the 2001 Winter Simulation Conference, pages 95–105, December 2001.

[5] Pierre L'Ecuyer, et al: An object-oriented random number package with many long streams and substreams. Operations Research, 2001.

[6] K. Fall and K. Varadhan (Eds.): The ns Manual (formerly ns Notes and Documentation), http://www.isi.edu/nsnam/ns/ns-documentation.html, last visited: October 2006.

[7] E. Altman and T. Jimenez: ns-2 for Beginners, lecture notes, Dec. 2003, http://www-sop.inria.fr/maestro/personnel/Eitan.Altman/COURS-NS/n3.pdf, last visited: Oct. 2006.