Pitfalls when using parallel streams in OMNeT++ simulations

Bernhard Hechenleitner and Karl Entacher*
Salzburg University of Applied Sciences and Technologies
School of Telecommunications Engineering
5020 Salzburg, Austria
{Bernhard.Hechenleitner, Karl.Entacher}@fh-sbg.ac.at

Abstract

By means of a simple OMNeT++ simulation scenario, which uses parallel streams of random numbers, we want to point out several technical pitfalls when applying different random number generators and using different initialization values for them. We describe shortcomings of the built-in random number generator (RNG) and describe traps when using modern RNGs. Quantitative and qualitative analyses of simulation results expose the danger of a careless simulation setup with regard to random components.

1. Introduction

OMNeT++ [24, 25] is an object-oriented discrete event simulation system based on C++. It is primarily designed to simulate computer networks, multi-processors and other distributed systems. The basic development of OMNeT++ began at the Technical University of Budapest (BME) in 1992 by András Varga. Currently, OMNeT++ is being used by dozens of universities and companies as a research tool, for validating hardware and protocol designs, and for performance evaluations. OMNeT++ is a non-commercial, open-source project. It is easy to integrate new simulation modules or alter current implementations of modules within its object-oriented architecture. Further information on OMNeT++, its fields of application and projects it is used in, can be found at the OMNeT++ home-page [24].

Like many other simulation systems, OMNeT++ currently implements the well-known "minimal standard" generator of Lewis, Goodman and Miller [18]. We will denote this RNG as `ran0`. As was already pointed out in [7], in certain simulations a wrong usage of this RNG can lead to severely wrong simulation results.

This paper is mainly aimed at showing the OMNeT++ community possible pitfalls with regard to parallel streams of random numbers, which are often very likely to be overseen but can have dramatically bad effects on the simulations. It does not only cover the problems when using `ran0` as a source for random numbers, it also describes issues concerning the usage of modern RNGs like Mersenne Twister [20] or the object-oriented RNG package of L'Ecuyer et al. [16], in case the simulation setup has not carefully been thought through. In the following, the Mersenne Twister RNG will be denoted as `MT` and the object-oriented RNG of L'Ecuyer et al. will be denoted as `RandU01`. The descriptions in this paper are all related to the current official version of OMNeT++, which is OMNeT++ 2.2 with applied patch 3 (`omnetpp-2.2p3`).

The paper is structured as follows. Section 2 describes the simulation setup, which was the basis for all simulations and tests. Section 3 outlines possible problems when using OMNeT++ with its built-in RNGs and describes the basic properties of this kind of RNGs. In Section 4 we show some problems which can occur when `RandU01` is used wrongly and describe how to correctly use `RandU01` in an OMNeT++ simulation. Section 5 briefly describes the `MT` RNG and how it is used within OMNeT++. In Section 6 we depict methods for finding good initialization vectors for RNGs. Section 7 compares the execution times of simulations using the different types of RNGs and Section 8 concludes the paper.

2. Simulation setup

Figure 1 shows the topology which was chosen for the simulations. Job streams of 5 exponential generators (`Expo 1` to `Expo 5`) are aggregated at `FIFO`, a simple First In First Out buffer with a buffer size of 1000 jobs. `FIFO` is connected to `Sink`, which does nothing else than absorbing the jobs, which are handed over by the service entity of `FIFO`. The 5 exponential generators produce streams of jobs with exponential inter-arrival times and `FIFO` offers an exponentially distributed service time for each job. Therefore, the resulting simulation topology constitutes a

*This work was supported by the Austrian Science Fund (FWF) Grant S8311-MAT.

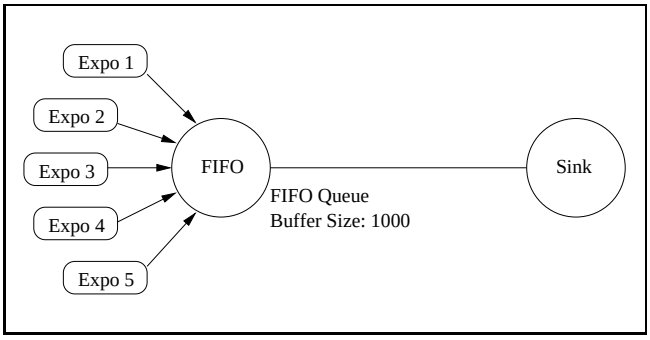

Figure 1. Simulation topology.

M/M/1/1001 queuing system.

For the simulations done, each of the exponential traffic generators `Expo 1` to `Expo 5` had the same parameter settings. The mean of the exponentially distributed time between the generation of two successive jobs (inter-arrival time) was set to $41\,ms$. Each of the 5 job streams produced by the generators `Expo 1` to `Expo 5` can be seen as a Markov Process with an average arrival rate of $\lambda = \frac{1\,job}{41\,ms}$. When aggregating Markov Processes, their average arrival rates may simply be added to get the average arrival rate of the aggregated Markov Process. Therefore the aggregated Markov Process, which arrives at the buffer of `FIFO`, has an average arrival rate of $\lambda_{sum} = 5 \cdot \lambda = \frac{1\,job}{8.2\,ms}$. The service times of `FIFO` are exponentially distributed with a mean of $8\,ms$, thus the mean service rate of `FIFO` is $\mu = \frac{1\,job}{8\,ms}$. Summing up, `FIFO` can be seen as a M/M/1/1001 queuing system with a utilization factor of $\rho = \frac{\lambda_{sum}}{\mu} = 0.97561$.

Referring to the theoretical M/M/1 model in [10, 12], the mean number of jobs in the system $\bar{N}$ calculates as $\bar{N} = \rho/(1-\rho)$. Considering a M/M/1 queuing system with a utilization factor of $\rho = 0.97561$, the average number of jobs in such a system would therefore be $\bar{N} = 40.0004\,jobs$. The distribution of the number n of jobs in the system is given by the geometric distribution $p(n) = (1-\rho)\rho^n$.

The exponential traffic generators and the service component of `FIFO` are fed by independent RNGs. Therefore, this simulation setup uses six parallel streams of random numbers.

3. Using OMNeT++ with its built-in RNGs

OMNeT++ currently implements the well-known "minimal standard" generator `ran0`, which was originally suggested for the IBM System/360 by Lewis, Goodman and Miller in 1969 [18]. It was examined in more detail in Park and Miller [22] and further on in several other studies on random number generation.

3.1. Properties of `ran0`

This generator is a multiplicative linear congruential type [6, 9, 12, 13, 21] which produces pseudorandom integers via the recursion

$$x_n \equiv a \cdot x_{n-1} \pmod{m}, \quad n \geq 1, \tag{1}$$

with multiplier $a = 7^5 = 16807$, modulus $m = 2^{31} - 1 = 2147483647$, and seed $1 \leq x_0 < m$. The period length of this recursion equals $p = m - 1$. Uniform pseudorandom numbers in $[0, 1)$ are derived by transformation $u_n = x_n/m$, non-uniform distributions by different transformation methods [3].

This particular generator has widely been used and actual implementations are available from the Internet. See [1, 6, 9, 12, 13, 14, 16, 17, 22] for references, empirical tests and implementations in free and commercial software. The following online resources contain related material: Resampling Stats (`www.resample.com`), Numerical Recipes (`www.nr.com`), the mathematical software MATLAB (`www. mathworks.com`), the IMSL Libraries, or the simulation software ACSL (`www. acslsim.com`), SIMAN/Arena, Slam II, AweSim (`www.pritsker.com`) and the network simulation software ns-2 (`www.isi.edu/nsnam/`).

A first problem of `ran0` is the period length $p = 2^{31} - 2$ which is far too short for actual simulations, especially when several parallel streams of random numbers are applied.

One also has to be very careful when manually seeding parallel streams. For example, using the seeds $x_{i,0} = i, i = 1, 2, \ldots k$ would result in the following k random number streams which are heavily correlated

$$x_{i,n} \equiv a^n \cdot x_{i,0} \pmod{m} \quad n \geq 0, \quad 1 \leq i \leq k. \tag{2}$$

These correlations can easily be shown from the vectors

$$\vec{x}_n := (x_{1,n}, x_{2,n}, \ldots, x_{j,n}) \tag{3}$$
$$\equiv a^n \cdot (1, 2, \ldots, j) \pmod{m}, \tag{4}$$

for $j \leq k,\ n \geq 0$. Since $a^n \pmod{m}, n \geq 1$ cycles all numbers in $\{1, 2, \ldots m-1\}$, the vectors above are contained in the set

$$\{n \cdot (1, 2, \ldots, j) \pmod{m} \ : \ 1 \leq n \leq m-1\}. \tag{5}$$

Therefore the normalized vectors $\vec{u}_n := \vec{x}_n/m, n \geq 1$ are situated in a lattice structure in the unit square $[0, 1)^2$ consisting of a few lines only, see Fig. 2 for $j = 2, 3$.

In Section 3.3 we will show that such correlations can result in completely biased results even in simple OMNeT++ simulation examples.

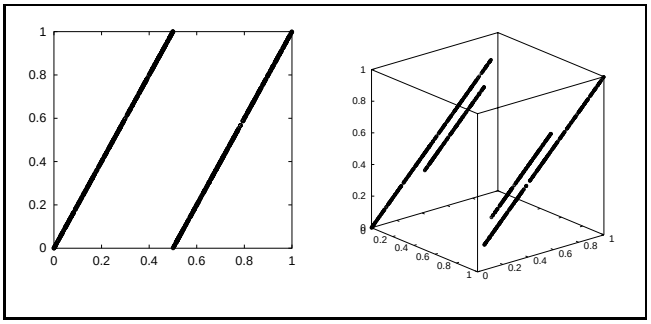

Figure 2. Correlations between two and three different streams of random numbers generated by the OMNeT++ RNG `ran0`.

Unfortunately not only simple seeding procedures produce strongly correlated streams of random numbers. There are many possible seed combinations where the corresponding streams are heavily correlated. Further examples are seeds $x_{i,0} = k \cdot i$, $i \geq 1$, $k \in \mathbf{Z}$, or all combinations of seeds consisting of very small numbers.

If special seeds are used, so that the full period of the RNG is divided into large blocks, and if each of these blocks will be used as a single stream of random numbers, then strong correlations will appear as well. This property is well known as long-range correlations[1] of linear random numbers, see [2, 4, 5] and the references given there. Such a situation may for example occur when using predefined seeds of the OMNeT++ seeding procedure in a wrong manner, see Sect. 3.2.

The quality of linear random number generators and their parallelization has theoretically and empirically been studied in detail. For an overview and references see the surveys contained in [6, 8, 13, 14, 21].

3.2. OMNeT++ seeding mechanisms

By default, all random variates used by OMNeT++ simulations are generated by the standard RNG object of OMNeT++ (`RNG0`). The default starting seed of `RNG0` is 1227283347. For the generation of parallel streams of random numbers, OMNeT++ offers up to 32 independent RNGs. The starting seeds of the RNGs are the first 32 values of the array `starting_seeds[]`, which is defined in the OMNeT++ source file `seeds.cc`. The starting seeds of the RNGs can be manually modified either by setting the corresponding parameters in the configuration file `omnetpp.ini`, or within simulation modules by directly calling the appropriate functions. Further details on the usage of the built-in RNGs can be found in the online documentation of OMNeT++ at [24].

In addition to manually setting certain seed values, it is also possible to use the automatic seed selection mechanism of OMNeT++. For this purpose, an array of 256 seed values, which are each 1 million values apart within the original `RNG0` sequence with the starting value $x_0 = 1$, is predefined in the OMNeT++ source file `seeds.cc`. The automatic seeding mechanism uses these predefined values as starting seeds for the RNGs in sequential order. However, this automatic seed selection has to be used with caution, especially if several simulation runs are to be executed.

The first issue which has to be thought of is the block size of 1 million values. If parallel random number streams with more than one million values are needed, these precalculated seed values lead to overlapping streams and this can involve correlations. Hence, if more than one million values per stream are needed, it is not recommended to use the automatic seeding procedure.

The second problem arises, if the different simulation runs are executed via the `-r` command line option of OMNeT++. In the OMNeT++ documentation, the following statement can be found concerning the automatic seed selection:

> "If you have several runs, each run is started with a fresh set of seeds that are 1,000,000 values apart from the seeds used for previous runs. Since the generation of new seed values is costly, OMNeT++ has a table of pre-calculated seeds (256 values); if they are all used up, OMNeT++ starts from the beginning of the table again."

This statement is only true if the different simulation runs are executed "at once" by having appropriate entries for the `runs-to-execute` parameter in the `[Cmdenv]` section of `omnetpp.ini`, thus executing the simulation runs by only one invocation of the simulation executable. If the simulation runs are executed via the `-r` command line option – which can be very useful when embedding the simulation invocation in a shell script – the automatic seed selection always starts anew for each run, hence always producing the same random number streams for all simulation runs.

A third problem may arise if wrong seeds are applied from the predefined seeding array. As already mentioned in the previous section, large blocks of linear random numbers may suffer from long-range correlations. This is not the case when adjacent numbers from the seeding table are used. The block length in this case is one million. We applied a spectral test [5, 11, 15], a standard test from the field of random number generation, which showed that no suspicious correlations arise between blocks of random numbers from such pairs or triples. But if one uses seeds from this table which are further apart, which means that the block

[1] The term long-range correlations may be slightly misleading since it also appears in the theory of stochastic processes. In our context it refers to a geometric property of linear random number generators.

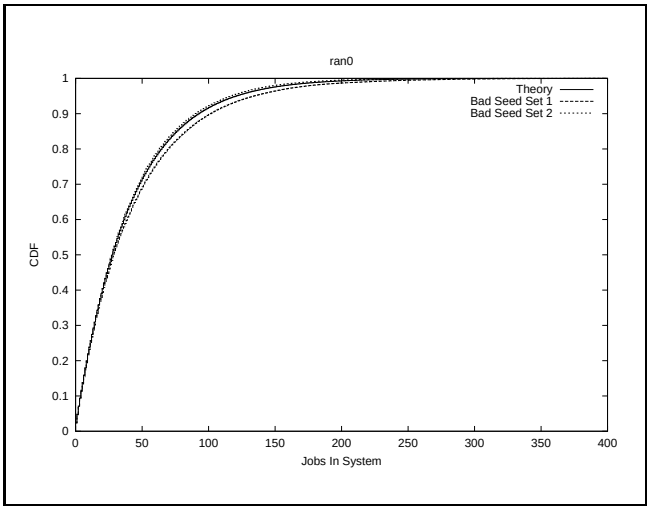

Figure 3. Using `ran0` with bad seeds.

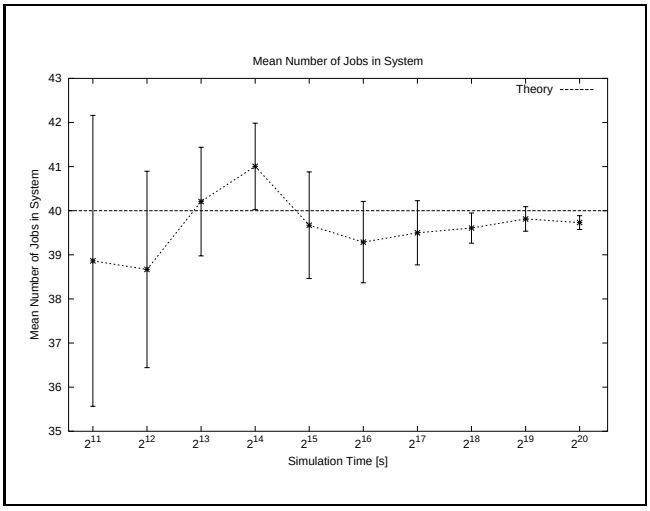

Figure 4. Simulation results when using `ran0` correctly.

length increases, then heavy correlations may appear. A random search using the spectral test showed many of such seed combinations from the seeding array with bad spectral test results.

3.3. Simulation results

Figures 3 and 4 show some simulation results when using `ran0` as sources for all required random streams in our simulation setup.

Figure 3 shows the empirical CDFs of the number of jobs in the system for a simulation time of $2^{18}s$, when "bad" seeds are applied in the simulation. Seed Set 1 corresponds to the worst case seeding-scenario described in Sect. 3.1, and Seed Set 2 consists of seeds, which divide the full cycle of `ran0` into 6 large blocks. The streams from both sets are strongly correlated. The applied seeds and the resulting mean values for the number of jobs in the system are listed in Table 1.

RNG Object	Seed Set 1	Seed Set 2
`Expo 1`	1	1
`Expo 2`	2	634005912
`Expo 3`	3	634005911
`Expo 4`	4	2147483646
`Expo 5`	5	1513477735
`FIFO`	6	1513477736
$\overline{N}$	43.3752186	38.9098305

Table 1. Simulation results from `ran0` and "bad" seeds.

Figure 4 shows results of simulations with increasing simulation times. The figure shows the mean values of the number of jobs in the system together with the corresponding 95% confidence intervals for 10 different simulation runs at each simulation time t_i, with $t_i = 2^i\ s, 11 \leq i \leq 20$. For each simulation run, the queue was initialized with an appropriately distributed number of jobs. Since the large simulation times consume more than 10^6 numbers, we used the OMNeT++ utility `seedtool` to generate appropriate seeds which guarantee random streams with sufficient lengths. The block length for the simulations with $2^{20}s$ time was about 50 million.

As can be seen from Fig. 4, with increasing simulation time the sizes of the confidence intervals shrink, but all results of the longer simulation times are (sometimes significantly) below the theoretic value of $\bar{N} = 40.0004$. The reason therefore may be that certain long-range correlations occur for the huge block-sizes used.

4. Using `RandU01`

A modern object-oriented RNG which supports well-tested parallel streams of random numbers was implemented by L'Ecuyer et al. The source-code of this RNG package is available at [16]. The provided C++ files, which implement the class `RngStream`, have to be placed in the source tree of all the other simulation files. In order to compile the RNG files using the provided OMNeT++ scripts, the file `RngStream.cpp` has to be renamed to `RngStream.cc`[2]. In the simulations described in the following sub-sections, all streams were produced by `RandU01` RNG objects.

[2]When executing the OMNeT++ scripts `opp_makemake -f` and `make` in the simulation directory, the RNG package is compiled together with all the other simulation components.

4.1. Basic properties of `RandU01`

The basic algorithm behind `RandU01` is a *combined multiple recursive generator* with a period length of about 2^{191}. The generator itself performs well with respect to the spectral test up to 45 dimensions. The mechanism to produce independent streams of random numbers is based on partitioning the full cycle into sub-streams with the length 2^{76}. The package provides a method which automatically distributes these parallel streams, and from a spectral test it is guaranteed that there are no suspicious correlations between the applied streams.

The `RandU01` RNG package uses vectors of size six for seeding. It is recommended to use the function `RngStream::SetPackageSeed(seed_vector)` for initializing the whole RNG package before instantiating the first RNG object. The first RNG object belonging to this class will use this seed vector, and all following RNG objects will be seeded automatically by the package. However, it is also possible to seed single RNG objects without influencing the others by using the member function `SetSeed(seed_vector)` of the according RNG object. If no explicit initialization of the RNG package has been performed, the default seed vector of the first instantiated RNG object is $\{12345, 12345, 12345, 12345, 12345, 12345\}$ – all following objects are seeded automatically.

4.2. Traps when using `RandU01`

Although the `RandU01` RNG package is well documented, it can't be concluded that all users follow the recommendations of the documentation. Especially a wrong usage of the `SetSeed` function can lead to false simulation results. We want to show two possible scenarios, which obviously produce wrong outputs.

For the first scenario, all used RNG objects are initialized by the user by means of the `SetSeed` function, and the user chooses bad seed vectors for initializing the RNG objects. As an example, we manually applied the seed vectors $\{j, j, j, j, j, j\}$, $1 \leq j \leq 6$ for the initialization of the six streams for `Expo 1` to `Expo 5` and `FIFO`. This is one of the worst seeding-scenarios for a linear generator like `RandU01`. The parallel random number streams obtained from these special seeds are highly correlated and the simulation yields poor results. A theoretical verification for this forecast may be done in the same way as for `ran0` in Section 3.1. Similar as for `ran0` there are also many other seed combinations for `RandU01` which produce correlated streams. Therefore it is heavily recommended to use the well tested automatic seeding method of `RandU01`. Using our simulation topology and a simulation time of $2^{18}s$, the mean number of jobs in the system would be 45.76, which is far away of the theoretic value of $\bar{N} = 40.0004$. Figure 5 shows the resulting CDF of the number of jobs in the system, compared with the theoretic distribution function.

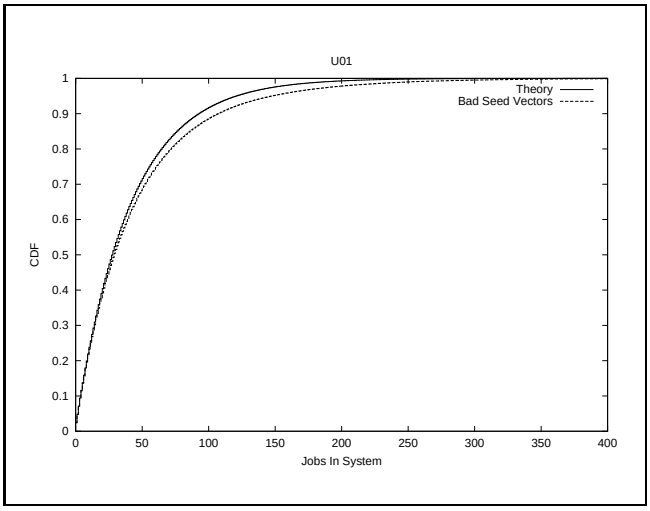

Figure 5. Using `RandU01` with bad seed vectors and without recommended seeding method.

In the second hazardous scenario, the unaware user applies the `SetSeed` function instead of `SetPackageSeed` for initializing the first RNG object. Furthermore, he uses the `-r` option of OMNeT++ for the invocation of different simulation runs. Therefore, only the first RNG stream is different for all executed simulations, for OMNeT++ specific reasons similar to those described in Section 3.2. For each simulation run, the simulation environment is started anew, and therefore all RNG streams except the first one (which was seeded by the user applying the "wrong" seeding function) produce identical number sequences in all simulation runs, leading to strongly biased simulation results. As an example, Fig. 6 shows the simulation results of a series of simulations with increasing simulation time. For each simulation time, ten simulation runs with different seed vectors for the first RNG object were executed using the `-r` option of OMNeT++. In conjunction with using the "wrong" seeding function `SetSeed` for initializing the first RNG object, the simulation results, represented by mean values and 95% confidence intervals for the mean number of jobs in the system, are strongly biased. The reason for this is that in this context, the remaining four traffic generators `Expo 2` to `Expo 5` as well as the RNG object for the exponential service times of `FIFO` always produce the same streams of random numbers. These remaining five streams are different from each other, but each RNG object always produces the same sequence of random numbers for all simulation runs!

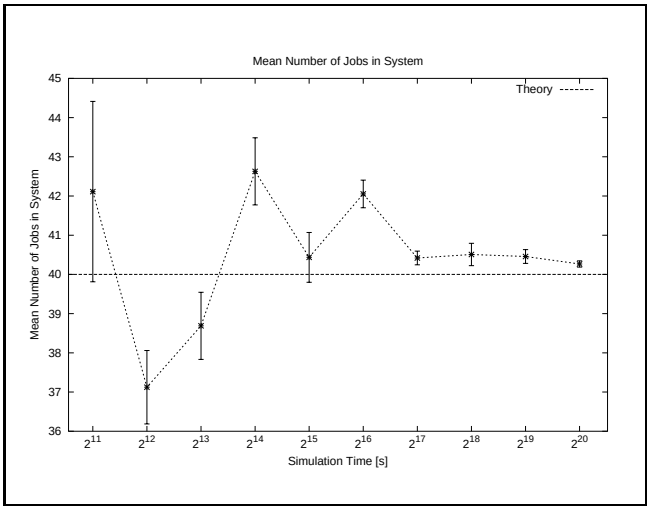

Figure 6. Using `RandU01` without recommended seeding method and with the `-r` option of OMNeT++.

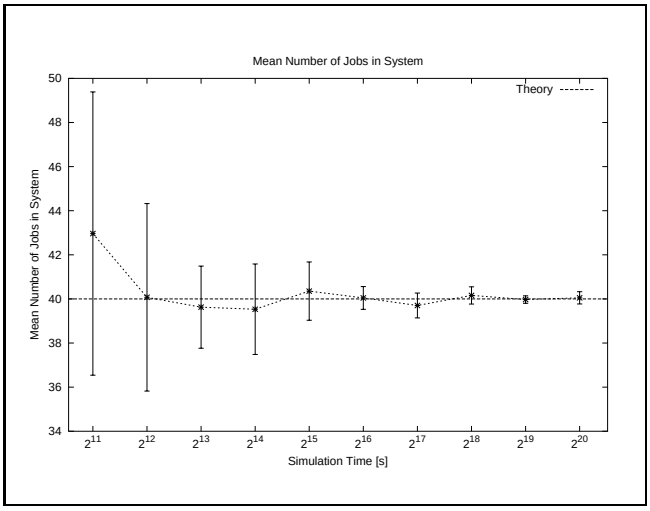

Figure 7. Using `RandU01` correctly.

4.3. Tips for a correct usage of `RandU01`

Looking at the current architecture of OMNeT++, a correct usage of `RandU01` is potentially only possible with some workarounds.

If all simulation runs are executed "at once" by having corresponding entries for the `runs-to-execute` parameter in the `[Cmdenv]` section of the `omnetpp.ini` file, everything should be fine. The first RNG object of the first simulation run is initialized with the default seed vector (see Section 4.1). All following RNG objects (the remaining five of the first run as well as all following objects of the further runs) are seeded automatically by the package, therefore always producing different streams of random numbers.

If, for some reason, the simulation runs are called by the `-r` option, things are getting more complicated. To avoid equal streams for different simulation runs, manually seeding the RNG package is inescapable. As the `RandU01` RNG package shall be initialized *before* the first RNG object is instantiated, a possibility for doing this has to be created. In an email conversation with András Varga concerning this problem, he suggested adding a C++ class, which calls the initialization function of the RNG package (`RngStream::SetPackageSeed(seed_vector)`) before the actual simulation starts. This first suggestion is fine if the seed vector is going to be coded into the simulation executable. If the seeds are to be handed over to the simulation dynamically by means of the configuration file `omnetpp.ini`, this trick has to be refined. We suggest the creation of a simulation module `RNGInit` with the seed vector as input parameter. In the initialization function of this module (`RNGInit::initialize()`) the seed vector can be read from `omnetpp.ini` and the RNG package can be seeded by calling `RngStream::SetPackageSeed(seed_vector)`. This method works fine, but has one restriction: any RNG objects, which are used by other simulation modules must not be defined as members of these modules, because otherwise these embedded RNG objects are created *before* the initialization of the `RNGInit` object. This would lead to similar problems as described before: those RNG objects, which were instantiated before the initialization of the RNG package always produce the same streams.

Figure 7 shows simulation results when the `RandU01` package is correctly used. For these simulation runs, the simulation module `RNGInit` was defined for initializing the RNG package before any RNG object has been created. The starting vectors have been randomly chosen. The single simulation runs were executed by using the `-r` option, and different seed vectors for the package were fed into the runs by appropriate entries for the `RNGInit` module in the `[Run]` sections of `omnetpp.ini`. The figure once again shows the mean values of the number of jobs in the system together with the corresponding 95% confidence intervals for 10 different simulation runs at each simulation time t_i, with $t_i = 2^i\ s, 11 \leq i \leq 20$. For each simulation run, the queue was initialized with an appropriately distributed number of jobs. As can be seen, with increasing simulation time the sizes of the confidence intervals shrink and the mean values closely match the theoretic value of $\bar{N} = 40.0004$.

Figure 8 shows the CDFs of the number of jobs in the system for 10 simulations with a simulation time of $2^{18}s$, when `RandU01` is correctly used. For each simulation, the corresponding seeding vector for the RNG package consisted of six values of the array `starting_seeds[]` within the OMNeT++ source file `seeds.cc`. As can be

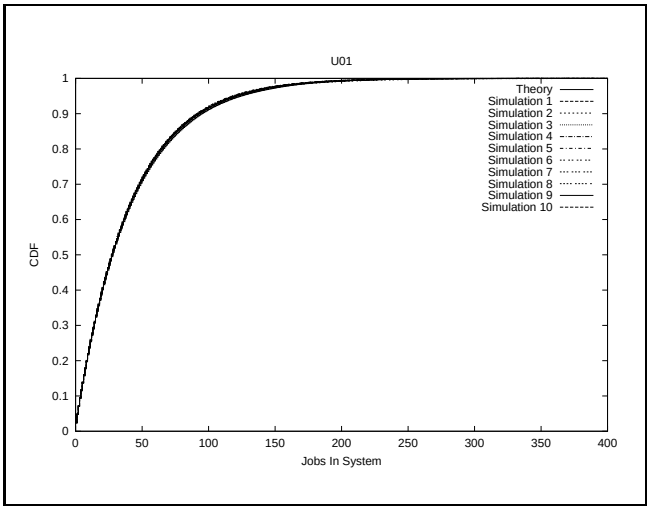

Figure 8. Using `RandU01` correctly.

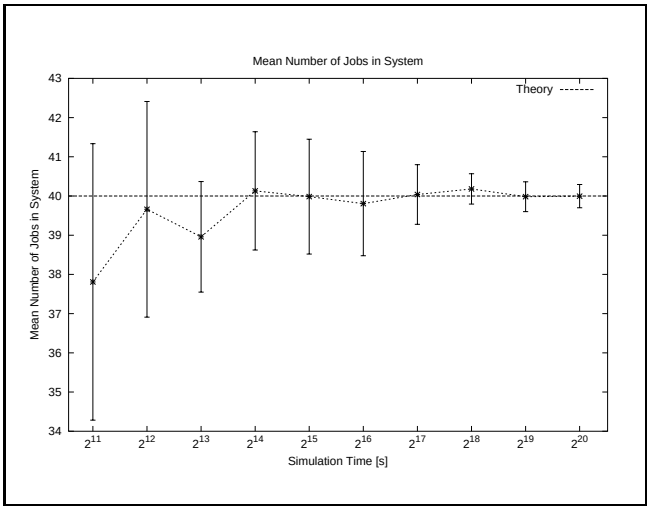

Figure 9. Simulation results when using `MT` correctly.

seen, the resulting empirical CDFs closely match the theoretic CDF. The resulting mean values of the number of jobs in the system are listed in Table 2.

Simulation	$\overline{N}$ from `RandU01`	$\overline{N}$ from `MT`
1	39.9008995	39.9007757
2	39.4299781	40.0820813
3	39.9336464	39.3793289
4	39.4803352	39.6598721
5	39.4676439	40.140133
6	40.0930451	40.7477009
7	39.7217354	40.5483123
8	39.7909931	40.2387971
9	40.7838613	40.2055123
10	39.7184333	39.1911062

Table 2. Simulations results from `RandU01` and `MT`.

5. The `MT` RNG

The Mersenne Twister (`MT`) [20] is another up-to-date random number generator. The basic algorithm is a variant of a *twisted generalized feedback shift register generator*. This RNG is very fast, has a huge period ($2^{19937} - 1$) and is known to be theoretically and empirically well tested. Source code of `MT` for several programming languages is freely available at the Mersenne Twister home page [19]. For the integration of `MT` as an OMNeT++ component, the latest `MT` C-code (`mt19937ar-cok.c`) was transformed into a corresponding C++ class.

In comparison to `RandU01`, the Mersenne Twister provides no equivalent method for independent well tested parallel streams. It is often recommended to use randomly chosen seeds for `MT` since because of the large period of the generator it is very unlikely that such streams will overlap. Hence, we will recommend to use `MT` as an additional OMNeT++ RNG for simulation verification.

We applied the method of randomly chosen seeds for our simulations. The seeds have been generated using the old OMNeT++ RNG `ran0`.

5.1. Simulation results using `MT`

Figure 9 shows simulation results, when all used RNG objects are of type `MT` and each RNG object of each simulation run has obtained its own unique seed value. The single simulations were executed by using the `-r` option, and different seeds for `Expo 1` to `Expo 5` and `FIFO` were applied by appropriate entries in the `[Run]` sections of `omnetpp.ini`. The figure once again shows the mean values of the number of jobs in the system together with the corresponding 95% confidence intervals for 10 different simulation runs at each simulation time t_i, with $t_i = 2^i\ s, 11 \leq i \leq 20$. For each simulation run, the queue was initialized appropriately. Fig. 9 shows that the Mersenne Twister behaved well in our simulations.

Additionally, Fig. 10 shows empirical CDFs of the number of jobs in the system for 10 simulations with a simulation time of $2^{18}s$, when `MT` is correctly applied. The resulting mean values of the number of jobs in the system are listed in Table 2.

6. Choosing good initialization values

As already mentioned in the previous sections, one should be very careful in choosing seed values for `ran0`. Even if the seeds are chosen from the predefined table

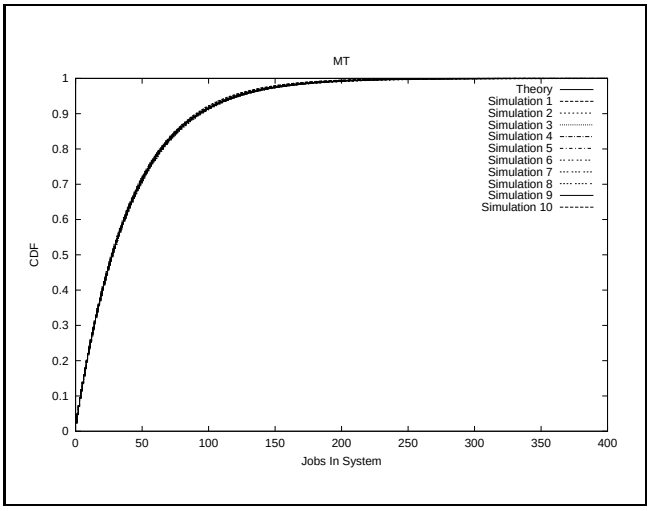

Figure 10. Simulation results when using `MT` correctly.

Simulation Time t_i [s]	`ran0` Execution Time [s]	`RandU01` Execution Time [s]	`MT` Execution Time [s]
2^{11}	3.3	3.3	3.7
2^{12}	6.6	6.6	7.3
2^{13}	13.1	13.8	14.6
2^{14}	26.0	26.6	29.1
2^{15}	52.1	53.1	57.7
2^{16}	103.7	105.0	115.8
2^{17}	207.7	209.3	231.9
2^{18}	416.2	422.9	468.7
2^{19}	831.3	832.9	927.3
2^{20}	1673.4	1710.5	1879.6

Table 3. Performance comparison of `ran0`, `RandU01` and `MT` when used within a simulation.

of OMNeT++, correlations may appear. Nevertheless, the OMNeT++ utility `seedtool` or the table with the predefined seeds or even the basic generator `ran0` may be used to produce seeds for the modern generators `RandU01` and `MT`.

In case of `RandU01` one has to apply integer vectors of length six and use these for initializing the whole RNG package using `RngStream::SetPackageSeed(seed_vector)` *before* instantiating the first RNG object of this type. For `MT` one simply has to apply a `ran0` number or seed for the `MT` initialization function `init_genrand(seed)` for each RNG object of this type. Again, note that for `RandU01` the seeding mechanism for several independent streams is well tested, and for `MT` the latter seeding procedure generates arbitrarily initialized streams within the `MT` full period cycle.

7. Performance comparison

In order to see the influence of the modern RNGs with regard to the speed of random number generation within a simulation setup, a performance test between `ran0`, `RandU01` and `MT` was executed.

The simulation scenario with increasing simulation time was used to determine the impact of the different RNGs on the durations of the executions of the simulations. For each simulation time t_i, 10 simulation runs were performed and the mean execution time of the simulations was calculated. Table 3 shows the results of the performance comparison. The table shows that with increasing simulation time `RandU01` and especially `MT` have a slightly more decelerating impact on the execution of the simulations than `ran0`.

The tests were performed on an unstressed PC working with SuSE Linux 8.0 operating system in console mode, an Intel Pentium 4 1.7 GHz CPU and 256 MB of RAM.

8. Conclusions

Simulations are one of the most important tools for research and investigation in the area of telecommunications. However, the credibility of simulation results deeply depends on how carefully the simulation setup has been thought through and how critically the simulation results have been called into question. For a critical study on this topic, see [23].

One influence coefficient, which may lead to bad effects on simulation results, is the incautious usage of parallel streams of random numbers. In this paper, we show possible pitfalls in the context of using parallel streams and when the simulation tool OMNeT++ is used. We describe and demonstrate the inherent entrapments of the widely used `ran0` RNG. We discuss two modern RNGs (`RandU01` and `MT`), which can be used as an alternative for `ran0`, and show that these current RNGs have to be used with caution as well. This doesn't refer to a lack of functionality of these RNGs (they are well-tested), but concerns their application within a simulation environment like OMNeT++, which has its own basic conditions of how to embed and invoke a new RNG component.

Looking at the considerations in this paper, the following conclusions may be drawn. One should never blindly rely on simulation results. A careful walk-through of the simulation setup and the verification of expected values should be the basic principle of performing simulations, e.g. the verification that random streams of different simulation runs *are* different if they ought to be. Furthermore, even if modern

RNGs are being used, their correct usage has to be carefully examined in order to avoid unnoticed falsification of simulation results. Further examinations on the basis of the simulation setup, which was used for all simulations in this paper, showed that mixing different types of RNGs within one simulation can lead to biased simulation results. Hence, we recommend using only *one* type of RNGs within the same simulation, and then use *another* type of RNGs for verifying the results. Concerning the "quality" of RNGs, the simulation results for our setup showed that `RandU01` and `MT` produce results, which are closer to the expected values than those of `ran0`, provided that they are used correctly and the simulation time is long enough.

Many of the problems described in the paper are related to the current architecture of OMNeT++, which can lead to the oversight of dangerous pitfalls in the context of parallel streams (`-r` command line option). The architecture for generating random streams and random variates within OMNeT++ is being discussed and will most likely be different in future releases.

Nevertheless, as some of the described pitfalls are easily overlooked and most of them produce severely wrong simulation results, it is always important to keep a careful eye on all random number components of a simulation setup.

References

[1] R. Boisvert, M. McClain, and M. B. GAMS The Guide to Available Mathematical Software. National Institute of Standards and Technology, Gaithersburg, MD, USA, 1998. `http://math.nist.gov/gams/`.

[2] A. DeMatteis and S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Comput.*, **14**:207–210, 1990.

[3] L. Devroye. *Non-Uniform Random Variate Generation*. Springer, New York, 1986.

[4] M. Durst. Using linear congruential generators for parallel random number generation. In E. MacNair, K. Musselman, and P. Heidelberger, editors, *Proceedings of the 1989 Winter Simulation Conference*, pages 462–466, 1989.

[5] K. Entacher. Parallel streams of linear random numbers in the spectral test. *ACM Transactions on Modeling and ComputerSimulation*, **9**(1):31–44, 1999.

[6] G. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*, volume 1 of *Springer Series in Operations Research*. Springer, New York, 1996.

[7] 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)*, pages 71–77. The Society for Modeling and Simulation International, 2002.

[8] P. Hellekalek and G. Larcher (eds.). *Random and Quasi-Random Point Sets*, volume **138** of *Lecture Notes in Statistics*. Springer, Berlin, 1998.

[9] R. Jain. *The art of computer systems performance analysis*. John Wiley & Sons, New York, 1991.

[10] L. Kleinrock. *Queueing Systems. Volume I: Theory*. John Wiley & Sons, New York, 1975.

[11] D. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[12] A. Law and W. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, 2 edition, 1991.

[13] P. L'Ecuyer. Uniform random number generation. *Ann. Oper. Res.*, **53**:77–120, 1994.

[14] P. L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, 2001.

[15] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, **9**(2):209–217, 1997.

[16] P. L'Ecuyer, R. Simard, E. Chen, and K. W.D. An object-oriented random-number package with many long streams and substreams. *Operations Research*, **50**(6):1073–1075, 2002. Previous extended version of the Manuscript and source-code from `http://www.iro.umontreal.ca/~lecuyer/`.

[17] H. Leeb and S. Wegenkittl. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM Trans. Modeling and Computer Simulation*, **7**(2):272–286, 1997.

[18] P. Lewis, A. Goodman, and J. Miller. A pseudo-random number generator for the System/360. *IBM Syst. J.*, **8** :136–146, 1969.

[19] M. Matsumoto. Mersenne twister home page. `http://www.math.keio.ac.jp/~matumoto/emt.html`, 1998.

[20] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, **7**(1):3–30, 1998.

[21] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, 1992.

[22] S. Park and K. Miller. Random number generators: good ones are hard to find. *Comm. ACM*, **31**:1192–1201, 1988.

[23] K. Pawlikowski, H.-D. Joshua Jeong, and J.-S. Ruth Lee. On Credibility of Simulation Studies of Telecommunication Networks. *IEEE Communications Magazine*, **40** (1):132–139, 2002.

[24] A. Varga. OMNeT++ Discrete Event Simulation System. `http://www.hit.bme.hu/phd/vargaa/omnetpp.htm`.

[25] A. Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*. June 6-9, Prague, Czech Republic, 2001.