Pseudo Random Numbers Generators available as Web Services

Sébastien Rumley and Markus Becker, *Members, IEEE*

Abstract— **Series of Pseudo Random Numbers are often used as simulation input, and they strongly influence the results. Thus, their usage and the usage of their generator need to be taken care of very well. Qualified generators are available on the web as source code or libraries. However, they require an additional middleware to adapt them to the running environments, and this can lead to misuses.**

This paper proposes Pseudo Random Number Generators embedded inside a Web Service for the use in simulations. This service, while offering a unique interface accessible from any platform, eases the important task of retrieving correct pseudo random numbers. The general architecture of the service is presented, as well as a reference implementation. The performance of the Web Service generator is compared to the performance of local generators.

Index Terms— **Random Number Generation, Web Services, Load-testing.**

I. Introduction

PERFORMANCE evaluation of a computer system or network infrastructure (later designated as "system") is and has always been a key task. It permits to verify if an existing infrastructure can fulfill precise requirements, or to foresee the qualities and flaws of novel ones. This evaluation can be based on a mathematical analysis. It can also be conducted over the system itself. However, when the situation becomes to complex to be formulated in equations, or if usage of the real system is to costly, dangerous or time consuming [1], performance evaluation can be conducted by means of simulation.

The description of a system can be partitioned into two parts. One part is devoted to the system itself (the internal part). The second part is related to the external events that "stimulate" the system (the external part). Simulation of the first part consists in reproducing the behaviors of each component of the system inside a computer program. As most of these devices present a deterministic behavior (notable exception is the IEEE 802.3/Ethernet random waiting time after collision, or the 802.11/Wifi backoff period), the system itself can be considered as deterministic. Oppositely, the external part (i.e. the real world) is non-deterministic and triggers random events (e.g. transmission errors, packet arrival, or telephone pick-up) which directly affect the internal part. This non-deterministic behavior of the external environment is modeled using statistical distributions. Within simulation, random numbers are needed to generate samples of these distributions, and in turn triggering events.

Truly random sequences of numbers cannot be generated by finite state machines, thus by software means. They can be obtained from specific physical processes [2], although these processes require to be implemented in external interfaces, inducing costs. More often, sequences generated by algorithms called Pseudo Random Numbers Generators (PRNGs) are substituted for truly random sequences. PRNGs do not provide real randomness, but presents other advantages, such as reproducibility of the sequences, which is useful for debugging, and very high generation rate. For these reasons, PRNGs are particularly suitable for simulation [1]. They should however comply with several criteria (for instance, high sequence length, unpredictability, or high entropy) to be eligible as good ones [3],[4].

PRNGs are based on complex algorithms themselves based on number theory and algebra. Performing a correct implementation requires specific knowledge, while testing and validating it can be very time consuming. It is therefore profitable and rewarding to reuse implementations of PRNGs made by experienced persons, and having been tested and validated by a wide user community.

Unfortunately, this reusability principle is not straight-forward to apply: the source code of PRNGs can only be integrated into projects sharing the same development environment, while PRNGs available as libraries have to be compatible with the execution environment. Some middleware can be added to bypass these road-blocks, but the reproducibility of the PRNGs might be then difficult to guarantee. It might also be affected by differences at the hardware, operating system, or runtime environment level.

This study proposes an approach where both the PRNG and a part of the required middleware are embedded into a

Manuscript received February 25, 2008; revised May 15, 2008. The research presented here has been undertaken by members of the European Cooperation in the field of Scientific and Technical Research (COST) Actions 285 ("Modeling and Simulation Tools for Research in Emerging Multi-service Telecommunications") and 291 ("Toward Digital Optical Networks").

This research was also supported by the German Research Foundation (DFG) as part of the Collaborative Research Centre 637 "Autonomous Cooperating Logistic Processes", and by the Swiss Secretariat for Education and Research.

S. Rumley is with the Laboratoire de Télécommunications, Ecole Polytechnique Fédérale de Lausanne, Station 11, 1015 Lausanne, Switzerland (e-mail: sebastien.rumley@epfl.ch).

M. Becker is with the TZI, University of Bremen, Otto-Hahn-Allee NW1, 28359 Bremen, Germany. (e-mail: mab@comnets.uni-bremen.de).

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

component deployed as Web Service. This approach solves many of the problems formerly enounced, as the variability of the service is almost null: a precise, tested and assessed implementation of PRNG is executed over a single hardware platform, within a unique runtime environment and library set. Furthermore, it can be called by any user working on any platform, using a single unified interface provided by an identical middleware. This service is therefore a reliable and independent source of pseudo random numbers.

This contribution is organized in the following manner. In the next Section, a short presentation of random number generation is given. Section III situates this contribution in the context of generic component oriented and service oriented architectures. In Section IV, an abstract description of the approach is given, while reference implementations and performance measurements are given in Section V. A conclusion is given in Section VI.

II. PSEUDO RANDOM NUMBER GENERATION

A general description of the pseudo random number generation is available in several reference books [5]. The first pseudo generator function (called middle-square method) has been imagined by John von Neumann in 1946. Even if von Neumann was aware of its poor quality, he preferred it to true random sequences stored on punch card for performance reasons. After 1946, many generators have been presented [6], [7]. Their quality, in particular their cycle length, increased together with the computing power, which allowed simulations requiring more and more numbers. More recently, Matsumoto *et al.* [3] presented a generator called Mersenne Twister providing a mathematically proven period of 219937-1. Other recent advances, generators and additional references have been summarized by Panneton *et al.* [8].

Depending on their properties, PRNGs can be more or less suited for different usages [9]. There exist also several algorithmic methods to test PRN sequences, and assess their corresponding generators [10], [11]. Utilization and impact of PRNGs in communication network simulations has been addressed in various publications, for instance in [12], while the effects induced by bad generators, or incorrect usage of good ones are described in [13]-[15], [32].

Software implementations of PRNG algorithms can be found in many locations of the Internet, and for various platforms. In particular, several packages offer implementations of the Mersenne Twister [16]-[18], as well as other PRNGs. Among them, the GNU Scientific Library [16] is also offering known bad PRNGs, which permits comparisons.

III. COMPONENT/SERVICE ORIENTED ARCHITECTURE

The Web Service (WS) in which the PRNGs are packed is the basic building block for a Service Oriented Architecture (SOA) for simulations [19]-[21]. WS provide increased interoperability in comparison to older distributed computing schemes like Remote Procedure Call (RPC): while RPC allows only communications between remote machines sharing at least a programming environment, WSs can be used between any pair of computers, whatever being the operating system, the programming environment or the programming language used. SOA applies the code reusability principle. A service can be seen as a reusable component, following in this way a Component Oriented design [22], [23]. The utilization of the SOA in the context of academic research or education has already been proposed: network planning tool deployed as WS [20], or scientific devices controllable through WS [33].

Request to WSs are sent using the Simple Object Access Protocol (SOAP), also employed to format the response. SOAP is a synchronous and stateless protocol [24] and defines a simple XML structure in which transmitted data must be incorporated. As SOAP messages are generally transported using the well known HTTP protocol, Web Services are often implemented beside conventional web servers like Apache [25].

Advantages of the SOA are multiple. The yield of the code constituting the service increases, as this service can be accessed by more users. It avoids the multiplication of the implementations, and thus diminishes the risks of mutation among the implementations, which can lead to incompatibilities. Additionally, less code means less development time and less bugs. Besides these quantitative considerations, qualitative gains can also be disclosed. In a SOA, implementation should be performed by experienced persons. People missing the appropriate knowledge should only write the connecting interfaces. This guarantees a high quality of the offered service. If in addition, the maintenance is operated by skilled people, the service will present high availability and robustness.

The SOA however presents several drawbacks, too. In particular, the stateless property of the SOAP protocol imposes a one-time transmission of all information, which makes reference passing calls impossible. In the case of PRNGs, this obliges to attach the state of the PRNGs in each call, which is resource consuming.

Performances of WSs are also subject to the potentially high latency or low bandwidth induced by remoteness. When payload data is of limited size and when the offered bandwidth towards the destination is large, this limitation has no real impact. In other cases, this can be an important performance limitation factor. Finally, WSs, due to the high-level interfaces they provide, require XML over HTTP, written in ASCII. Additional processing time is thus required for each message reception/emission [26], [27]. Several tracks have been envisaged to reduce the required processing time, by means of compression, different encoding styles or SOAP attachments [27].

Nevertheless, one expects in the future to see these limitations being mitigated by the appearance of high-capacity and low latency computing grids. Synergies between WSs and Grid Computing have already been exposed so far [28].

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

IV. PRNG Web Service Architecture

The architecture of the realized PRNG service is sketched in Fig. 1. As already mentioned, a classical web server engine is used to handle incoming TCP connections and parse HTTP messages. Once the HTTP payload is extracted, the WS is accessed.

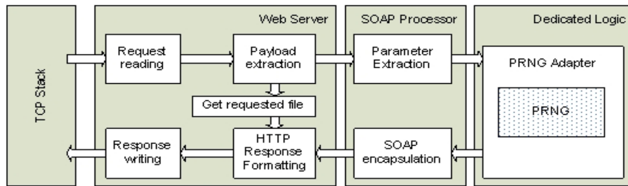

Fig. 1. Server side Implementation of the PRNG Web Service.

Inside the WSs, the SOAP message is parsed to recover the call parameters, which are in the present case:

- The name of Pseudo Random Number Generator (The WS may embed various ones)
- The number of desired samples
- A single seed value OR the generator state

An example of SOAP request message is shown in Fig. 2. The state is transmitted as a collection of columns. In the example, it consists of one array of integers (first column, included in element `<m:c0>`) plus one pointer over this array (unique element of the second column `<m:c1>`). This format tolerates more columns, and has been selected to allow PRNGs using states structured differently. The state size element contains a list of comma separated integers. The first integer indicates the number of columns, while the following integers denote the length of each column.

Because certain PRNGs provide a mechanism to populate their state register from a single value (the seed) at the initialization, the full state element listed in the Fig. 2 can be replaced by a simple seed element in the first request. Successive requests must however include the state to keep continuity in the sequence.

The WS also includes the logic transforming ASCII numbers into integers. Once the parameters are extracted, the WS initializes the embedded generator, either using the given seed, either replacing the state values by the ones contained inside the request. The WS then retrieves *n* requested samples from the PRNG, retrieves again the state of the PRNG, and packs all results in the response message, whose example is listed in Fig. 3. This message is eventually forwarded to the web server for HTTP formatting and serialization. On the client side, once the message is received, parsing operation is conducted and the response values are extracted. Depending on whether more samples will be requested or not, the state can be stored or discarded.

To prevent any interpretation difference on the client side of ASCII double or long values, SOAP messages include only integer values, ranging from -2^{31} to 2^{31}-1. Whether the way of storing integer arrays inside the SOAP messages is the right option or not stays an open question. The format used for state in Fig. 2 and 3 has a high overhead and may require more efforts from the XML parser, but is less error prone. Another possibility consists in using the format used for the state size: comma separated integers. This would however require an additional parsing pass, as XML parser will only extract the string contained inside the element. For the moment, no definitive solution has been selected.

V. Reference Implementation

To validate the proposed approach, several implementations have been realized, both on the client and server side. A main Java implementation has been achieved, and various designs have been tested and measured within it. Additionally, C and Python based implementation have been setup, to verify

```
< SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-
ENV:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance>"
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
   <m:get_prng xmlns:m="http://tcom.epfl.ch/pnrgweb/v1">
      <m:prngtype xsi:type="xsd:string">mt19937</m:prngtype>
      <m:samples xsi:type="xsd:type">9</m:samples>
      <m:state_size>2,624,1</m:state_size>
      <m:state>
         <m:c0>
            <m:s>-200633805</m:s>
            <m:s>-111674796</m:s>
              :              :
            <m:s>1516013983</m:s>
         </m:c0>
         <m:c1>
            <m:s>30</m:s>
         </m:c1>
      </m:state>
   </m:get_prng>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 2. SOAP request including state, requesting 9 samples

```
< SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-
ENV:encodingstyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
   <m:get_prngResponse
      xmlns:m="http://tcom.epfl.ch/pnrgweb/v1">
      <m:nb_samples>9</m:nb_samples>
      <m:samples>
         <m:v>3434214777</m:v>
            :           :
         <m:v>2248176379</m:v>
      </m:samples>
      <m:state_size>2,624,1</m:state_size>
      <m:state>
         <m:c0>
            <m:s>1221865289</m:s>
            :              :
            <m:s>-2030900162</m:s>
         </m:c0>
         <m:c1>
            <m:s>39</m:s>
         </m:c1>
      </m:state>
   </m:get_prngResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 3. SOAP response, including state and returning 9 samples.

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"

    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Header/>
<SOAP-ENV:Body>
    <m:get_prng xmlns:m="http://tcom.epfl.ch/prngweb/v1">
        <m:seed xsi:type="xsd:int">498374</m:seed>
        <m:samples xsi:type="xsd:int">21</m:samples>
        <m:prngtype xsi:type="xsd:string">mt19937</m:prngtype>
    </m:get_prng>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<SOAP-ENV:Body>
    <get_prng SOAP-ENC:root="1">
        <seed xsi:type="xsd:int">498374</seed>
        <samples xsi:type="xsd:int">21</samples>
        <prngtype xsi:type="xsd:string">mt19937</prngtype>
    </get_prng>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 4. Comparison between SOAP messages. Left message has been created by cSOAP, right message by SOAPpy. Differences have been highlighted

interoperability, compatibility and portability of the approach over various platforms.

WebServer architecture:

On the server side, C and Python implementations rely on embedded HTTP servers to handle incoming TCP connection and parse HTTP messages. The Java implementation discarded Apache/Tomcat/Axis for performance and lack of flexibility reasons and uses a lightweight HTTP Server library called Simple [29].

SOAP/WS architecture:

Various packages provide support for SOAP message handling, for instance SOAPpy library in Python, cSOAP in C, or Axis in Java. However, we experienced difficulties with several libraries, either because they offer too high level functions and miss of flexibility and configurability (Axis, in particular), either because they use different definitions and/or implementations of the WS/SOAP mechanisms. Fig. 4 lists SOAP messages generated by SOAPpy and cSOAP clients and highlights the differences. cSOAP, additionally, requires the SOAP header element, although it is mentioned as optional in the W3C's specification [30]. For these reasons, dedicated logic for SOAP message creation and parsing has been developed within the Java implementation. This provided a good compatibility with all clients, as well as more flexibility to test various design choices.

XML Parsing:

Parsing of the SOAP message is clearly the most time consuming task of the approach, both on the server and on the client side. The selection of an appropriate parser is thus a crucial step. In this study, both DOM and SAX parsers have been tested. DOM parsers are slower and more memory consuming, as they first store the XML document as an object structure, and later let the user accessing it. They however allow more flexibility regarding the way the elements are placed inside the XML structure. SAX parsers permit on the contrary to parse message in one pass, reading directly on the input stream. They are very efficient but less robust, and impose a strict organization inside the message. Fig. 5 illustrates performance of the implementation and compares the two parsers. The WS server is executed locally on the same multi CPU machine than the client and thus performance does not suffer from any network latency. It appeared that SAX parsers are about 30% faster than DOM. By replacing DOM parsers by SAX engines on both side, performances nearly double. The SAX Parser also tolerates larger messages due to its reduced memory consumption.

Selection of the PRNGs:

The three packages mentioned in the introduction have been included in the reference implementations. GSL [16] is used by the C and Python servers, while SSJ [17] and Mantissa [18] are implemented in the Java server. All three include the Mersenne Twister algorithm, which has been selected as reference generator. The way the function is implemented varies however between the packages. GSL and Mantissa populate the initial state vector using a unique seed, while SSJ requires a full sequence. SSJ generates double values but Mantissa outputs integers. GSL is configurable on that point. After many trials, adaptations and reverse engineering operations, we eventually generated identical sequences using the different methods. Additionally, as previously mentioned, the state of the PRNG must be accessible. However, to guarantee the consistency of the sequences, implementations declare the state variables private. Thus, we also modified the implementation of both Mantissa and SSJ packages, in order

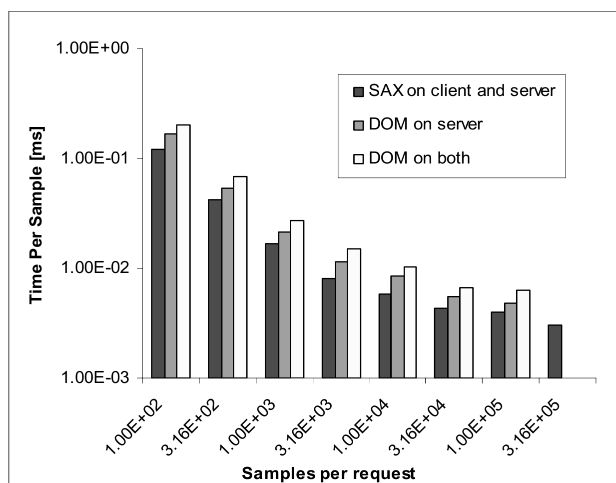

Fig. 5. Comparison between SAX and DOM parsers. A total of 4 millions of samples have been retrieved in all cases

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

to retrieve and set the state. GSL did not require any modification: the state of the different pseudo random number generators is accessible.

Besides the Mersenne Twister, another generating function has been tested: the FourTap shift register [31], included in the Mantissa package. Performances of two WSs implementing each of these two generators have been measured and are compared (Fig. 6). Again, WSs are executed on the same machine. The FourTap has a much bigger state than the Mersenne Twister, thus imposes larger messages which lead to a performance penalty. The penalty is however reduced when large amount of samples are retrieved within the same request. In Fig. 6, a local minimum appears at 10k samples for the two series corresponding to 10k of samples. This is due to the fact that the request size matches the number of reclaimed samples perfectly.

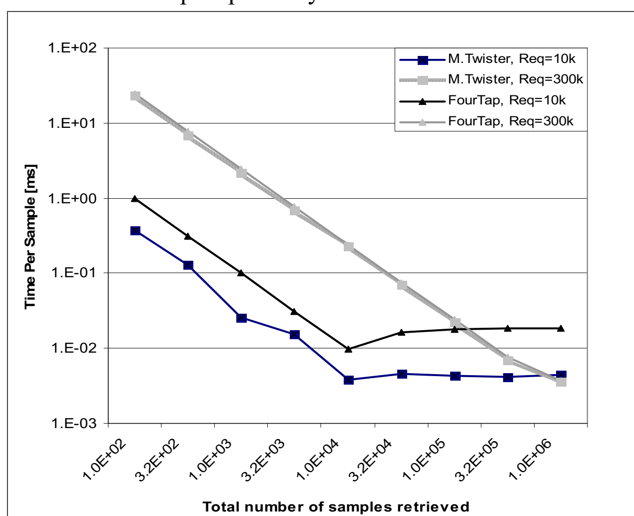

Fig. 6. Comparison between two PRNG functions (Mersenne Twister and FourTap), and between two request size.

XML Serialization:

Different options also exist regarding XML serialization. SOAP messages can be constructed using a DOM object, which are eventually serialized over the stream. This is however memory and time consuming. A progressive serialization of the XML has been tested but provided no benefits, probably due to packet fragmentation at the TCP/IP stack level. In the retained solution, XML text is progressively written in a byte buffer, which is then linearly serialized to the stream.

Client caching mechanism:

To mitigate the influence of the multiple overheads (i.e. SOAP envelopes, XML tags and transmission of state) over the performance, the Java client has been equipped with a caching mechanism. In this way, the client first fills its cache by sending one or more request to the server. Once filled, random numbers stored in the cache can be consumed by the user application, i.e. the simulation. When the last number is consumed, the client fills the cache again.

The effect of cache size over performances has been measured, and is represented in Fig. 7. The request cannot be larger than the cache, to avoid loosing numbers. By increasing the cache size, larger requests can be emitted, which improves performance. A cache much larger than the request size is not valuable, and can even bring penalties. In order to see the effect of an increased delay, the same experiment has also been performed in alternative conditions: the server has been moved on a laptop connected to the server through a wireless and Virtual Private Network (VPN) connection.

Several scenarios have also been set up to measure the influence of the ratio request size/cache size. When the WS is running on the same machine, the overhead is mostly due to XML parsing. By maximizing the size of the requests, less state messages have to be parsed. A ratio of 100% is therefore giving the best performance. If the WS is more distant, smaller request permit a better anticipation. Some results are listed in Fig. 8.

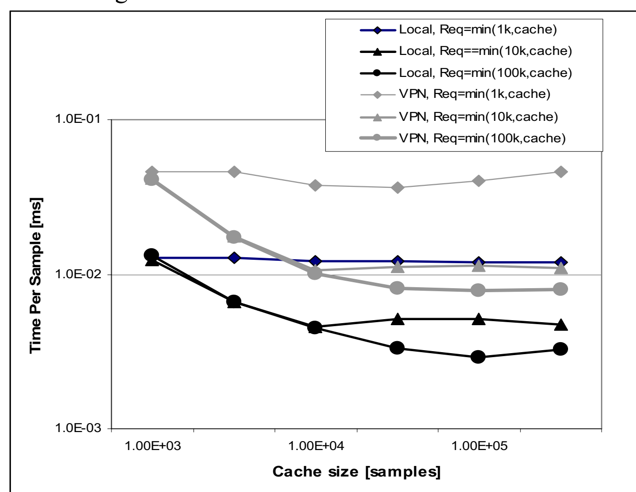

Fig. 7. Effect of cache size on performance.

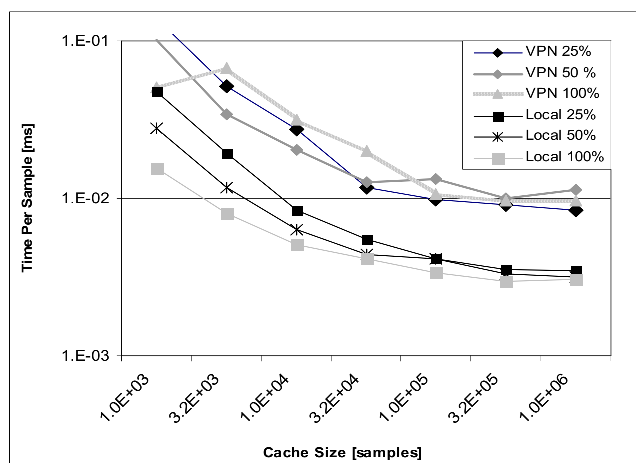

Fig. 8. Effect of different ratios request size/cache size (25%, 50%, 100%) with a local and distant WS.

Parallel cache reloading mechanism:

One can take advantage of multi-threading to concurrently consume random numbers and refill the cache. Such a possibility has been included into the client. The reading

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

thread (i.e. the one which consumes the samples) periodically verifies the level of the cache. When this level falls under a certain threshold, it triggers a new thread, which requests more numbers from the server, while the reading thread continues its normal execution. If all numbers are exhausted, the reading thread waits until the requesting thread terminates.

In order to provide a global overview of the performances, four situations are compared:

- server and client are running on the same machine (Intel Quad CPUs, Windows environment)
- server is moved on a Pentium 4 based machine working on the same LAN (Linux environment).
- server is moved on a Pentium M laptop connected to the LAN via a wireless connection and through a VPN (Windows environment).
- random numbers are retrieved directly without using the WS.

Additionally, a block acting as simulator is introduced. This block simply performs a configurable number of floating point divisions. Each time a sample is read, this block is executed.

Results corresponding to these four situations are illustrated in Fig. 9. The x axis corresponds to the number of divisions made in the block. Multithreading (MT) gives shorter times when samples are requested locally or on the LAN, but not when requested through the VPN: due to the reduced bandwidth of the VPN, the reading thread consumes the results too quickly, and waits for the other thread to finish. Synchronization mechanisms between threads are thus required, and occupy additional resources.

Fig. 9 shows that direct generation is without comparison better for pure sample generation. WS based generation becomes competitive if several operations have to be conducted with each random number. In terms of rates, a PRNG directly accessed furnishes between 10 to 100 millions of samples per second, whereas the WS based approach is limited to 0.1 to 1 million when used locally, and to 10 to 100 thousands when used in a campus network (wireless and VPN). The impact of the remoteness becomes acceptable, if the required rate is smaller than these values.

VI. CONCLUSION

A WS providing Pseudo Random Number series has been presented. Such a service could be made available over a campus wide network or over an academic grid, and accessed by any user of the network. The underlying hardware and software platform over which the service is running does not vary over time. Neither does the implementation of the PRNG algorithms, and the additional middleware. Therefore, an almost perfect reproducibility of the generated sequences is guaranteed. As the service should be setup and maintained by people having experience in pseudo random number generation, only valid sequences are generated. Those sequences can be later used in many fields, for instance in simulation.

A reference implementation has also been described. By using a cache mechanism and requesting large amount of numbers at the same time, the performance of this approach is acceptable, although much slower than a direct implementation. Utilization in simulations requiring a limited amount of pseudo random numbers is thus possible.

The implementation also revealed that the same PRNGs packaged in different libraries could generate different sequences if not initialized in a very similar way, which is not straightforward. Finally, incoherencies between SOAP implementations have been highlighted.

The described service intends to cure in parts the credibility crisis of simulation studies denounced in [12]. On one hand regarding the limited attention paid to pseudo random number generation, on the other hand regarding the non-reproducibility of the results. Indeed, if the simulator itself is proposed as a service, and if the PRNG function and seed are published along the results, anybody could check their validity.

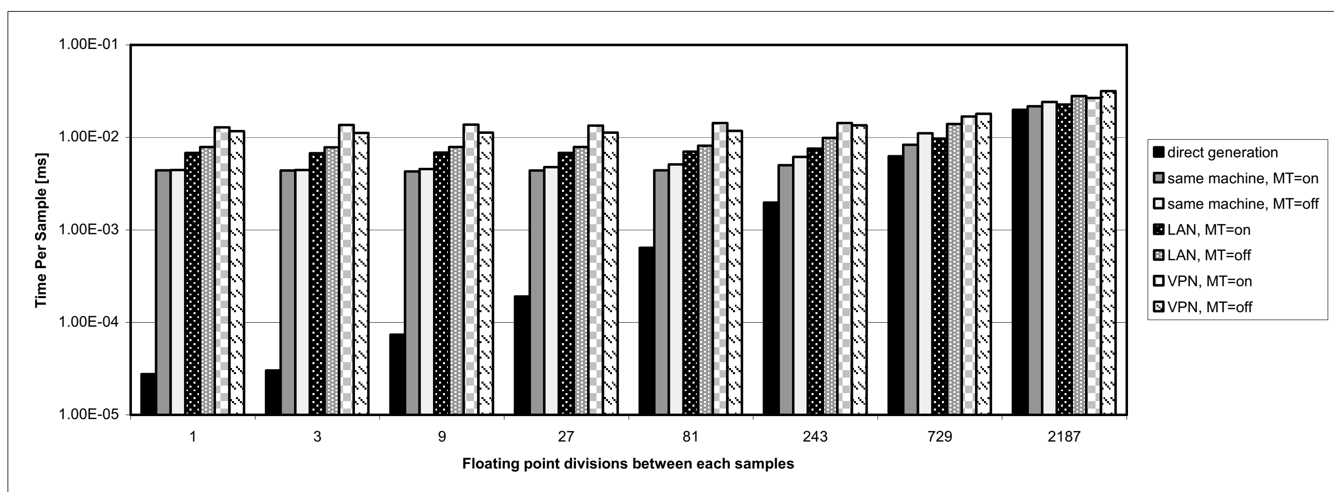

Fig. 9. Comparison between direct generation, using WS running on the same machine, on the same LAN, or in the same campus network, through VPN. MT means multithreading.

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] C. Phillips. “A Review of High Performance Simulation Tools and Modeling Concepts”, *Recent Advances in Modeling and Simulation Tools for Communication Networks and Services*. Springer, 2008.

[2] True Random Numbers, 2007. http://www.random.org

[3] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator”. *ACM Trans. On Modeling and Computer Simulation*, 8(1):3-30, Jan. 1998.

[4] Stevanovic, R., Topic, G., Skala, K., Stipcevic, M., Rogina, B.M.: “Quantum Random Bit Generator Service for Monte Carlo and Other Stochastic Simulations“. *Lecture Notes in Computer Science*, Springer. 2007.

[5] D. E. Knuth. *Art of Computer Programming*, Volume 2: Seminumerical Algorithms. Addison-Wesley Professional, 3 edition, Nov. 1997.

[6] D. Lehmer. “Mathematical methods in large-scale computing units”. *Proc. 2nd Symposium on Large-Scale Digital CoCalculating Machinery*, pages 141-146. Harcard University Press, 1951.

[7] R. Tausworthe. *Random numbers generated by linear recurrence module two*. In Math. Comp., volume 19, pages 201-209, 1965.

[8] F. Panneton, P. L'Ecuyer, and M. Matsumoto 2006. “Improved long-period generators based on linear recurrences modulo 2”. *ACM Trans. on Mathematical Software*, 32(1):1-16, March 2006.

[9] P. L'Ecuyer, “Software for uniform random number generation: distinguishing the good and the bad”. *Proceedings of the 33nd conference on Winter simulation (WSC '01)*, 2001.

[10] J. Soto. “Statistical Testing of Random Number Generators”. *Proceedings of the 22nd National Information Systems Security Conference*, Oct. 1999.

[11] P. L'Ecuyer and R. Simard. “TestU01: A C Library for Empirical Testing of Random Number Generators”. *ACM Transactions on Mathematical Software*, 33(4),Aug. 2007.

[12] K. Pawlikowski, H.-D. J. Jeong, and J.-S. R. Lee. “On credibility of simulation studies of telecommunication networks”. *IEEE Communications Magazine*, 40(1):132{139, Jan. 2002.

[13] M. Becker, T. L. Weerawardane, X. Li, and C. Görg. “Extending OPNET Modeler with External Pseudo Random Number Generators and Statistical Evaluation by the Limited Relative Error Algorithm”. *Recent Advances in Modeling and Simulation Tools for Communication Networks and Services*. Springer, 2008.

[14] B. Hechenleitner. “Defects in Random Number Routines of Well-Known Network Simulators and Appropriate Improvements”. PhD thesis, School of Scientic Computing of the University of Salzburg, 2004.

[15] P. Hellekalek. “Good random number generators are (not so) easy to find”. *Proceedings Second IMACS Symposium on Mathematical Modelling*, 1998.

[16] GNU Scientific Library Reference Manual - Revised Second Edition, 2007. http://www.gnu.org/.

[17] SSJ: Stochastic Simulation in Java, http://www.iro.umontreal.ca/~simardr/ssj/indexe.html

[18] Mantissa (Mathematical Algorithms for Numerical Tasks In Space System Applications), http://www.spaceroots.org/software/mantissa/index.html

[19] J. Bih. *Service oriented architecture (SOA): A new paradigm to implement dynamic e-business solutions*, 2006.

[20] S. Rumley, C. Gaumier. “Distributed RWA Tools via Web Services”. *Proceedings Optical Network Design and Modelling (ONDM 2008) Conference*, March 2008.

[21] S. Chandrasekaran, G. Silver, J. Miller, J. Cardoso, and A. Sheth. “Web service technologies and their synergy with simulation”. *Proceedings of the Winter Simulation Conference (WSC '02)*, volume 1, pages 606 615, Dec. 2002.

[22] O. Nierstrasz and D. Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[23] M. Lackovic and C. Bungarzeanu. A Component Approach to Optical Transmission Network Design. *Modelling and Simulation Tools for Emerging Telecommunications Networks*. Springer, 2006.

[24] Web Services Architecture, 2007. http://www.w3.org/TR/ws-arch/.

[25] Apache Web Server, 2007. http://httpd.apache.org.

[26] K. Chiu, M. Govindaraju, and R. Bramley. “Investigating the Limits of SOAP Performance for Scientific Computing”. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[27] R.A. van Engelen, “Pushing the SOAP Envelope with web services for scientific computing”, *1st International Conference on Web-services*, June 22-26, 2003.

[28] M. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. Lewis. “A Benchmark Suite for SOAP-based Communication in Grid Web Services”. *Proceedings of the ACM/IEEE SC 2005 Supercomputing Conference*, 2005.

[29] http://simpleweb.sourceforge.net/

[30] http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

[31] R.M. Ziff, “Four-tap shift-register-sequence random-number generators”, *Computers in Physics*, vol. 12, number 4, 1998.

[32] M. Matsumoto, I. Wada, A.k Kuramoto and H. Ashihara. “Common Defects in Initialization of Pseudorandom Number Generators”. *ACM Transactions on Modeling and Computer Simulation*. Article 15, Volume 17, Number 4, 2007.

[33] Y. Yan, Y. Liang, X. Du, H.S. Hassane, A. Ghorbani, “Putting labs online with Web services”,*IT Professional* Volume 8, Issue 2, March-April 2006.

Authorized licensed use limited to: EPFL LAUSANNE. Downloaded on June 04,2010 at 13:08:32 UTC from IEEE Xplore. Restrictions apply.