Gathering Entropy at Large Scale with HAVEGE and BlobSeer

Alin Suciu, Bogdan Nicolae, Gabriel Antoniu, Zsolt Istvan, Istvan Szakats

Gathering Entropy at Large Scale with HAVEGE and BlobSeer

Alin Suciu[1], Bogdan Nicolae[2], Gabriel Antoniu[2], Zsolt Istvan[1], and Istvan Szakats[1]

[1] Technical University of Cluj-Napoca, Romania
[2] INRIA, France

Abstract. Large sequences of random information are the foundation for a large class of applications: security, online gambling games, large scale Monte-Carlo simulations, etc. Many such applications are distributed and run on large-scale infrastructures such as clouds and grids. In this context, the random generator plays a crucial role: it needs to achieve a *high entropy*, a *high throughput* and last but not least a *high degree of security*. Several ways to generate high-entropy random information securely exist. For example, HAVEGE generates random information by gathering entropy from internal processor states of the machine where it is running alongside the user applications. These internal states are inheritably volatile and impossible to tamper with in a controlled fashion by the applications running on it. A centralized approach however does not scale to the high throughput requirement in a large scale setting. In order to do so, the output of several such instances needs to be combined into a single output stream. While this certainly has a good potential to solve the high throughput requirement, the way the outputs of the instances are combined in a single stream becomes a new weak link that can negatively impact all three requirements and therefore has to be addressed properly. In this paper we propose a distributed random number generator that efficiently addresses the aforementioned issue. We introduce a series of mechanisms to preserve a high entropy and degree of security for the combined output result and implement them on top of BlobSeer, a data storage service specifically designed to offer a high throughput in large-scale deployments even under heavy access concurrency. Large-scale experiments were performed on the G5K testbed and demonstrate substantial benefits for our approach.

1 Introduction

Random numbers play a very important role for a wide range of applications in domains like cryptography, online gambling, large scale Monte-Carlo simulations or physics modeling. With increasing demand at large-scale, distributed computing [5, 4] applications tend to require an ever increasing amount of high quality random numbers in order to provide such services at large scales. Therefore, there is a need to design a scalable random number generator that is able to sustain a high throughput, generating massive amounts of random numbers

in a timely fashion, yet satisfy the quality requirements of the target application at the same time. While some domains may accept or even require reproducible random sequences, in a large number of applications *security* is paramount: random number sequences generated for such applications need to satisfy the requirements of *unpredictability*, *irreproducibility* and *uniform distribution* of the employed randomness.

However, generating high quality randomness has always been a challenge. Aiming to meet the high throughput requirement, applications frequently rely on pseudo-random generators, whose security guarantees are limited yet they can be easily optimized for speed. More secure approaches typically leverage unpredictable entropy sources to meet stronger security requirements: devices designed to be deterministic, but due to the complexity of the underlying phenomenon or the unpredictable nature of human-computer interaction can yield good quality randomness. The down side of this approach however is the fact that such devices are usually slow entropy sources and have limited potential to enable high throughput random number generation.

Ideally, both high throughput and high security are desired. A centralized approach certainly does not scale to the high throughput requirement in large scale settings. In order to do so, the output of several distributed random number generators needs to be combined. While this certainly has a good potential to solve the high throughput requirement, the way the outputs of the instances are combined in a single stream becomes a new weak link that can negatively impact the security requirement.

This paper proposes a distributed random number generator that efficiently addresses the aforementioned issue. We introduce a series of mechanisms to preserve a high entropy and degree of security for the combined output result and implement them on top of BlobSeer (presented in more detail in Section 2), a distributed storage service specifically designed to offer a high throughput in large-scale deployments even under heavily concurrent access. Experimental results on the Grid'5000 [6] testbed shows large benefits for using our proposal both in terms of throughput and security guarantees.

2 BlobSeer

This section introduces BlobSeer, a distributed data storage service designed to deal with the needs of data-intensive applications: *scalable aggregation of storage space* from the participating nodes with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency*.

Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a blob, reading/writing a range of *size* bytes from/to the BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. This access interface is designed to support versioning explicitly: each time a write or append is performed by the client, a new snapshot

of the blob is generated rather than overwriting any existing data (but physically stored is only the difference). This snapshot is labeled with an incremental version and the client is allowed to read from any past snapshot of the BLOB by specifying its version.

Architecture. BlobSeer consists of a series of distributed communicating processes. Each BLOB is split into chunks that are distributed among *data providers*. *Clients* read, write and append data to/from BLOBs. Metadata is associated to each BLOB and stores information about the chunk composition of the BLOB and where each chunk is stored, facilitating access to any range of any existing snapshot of the BLOB. As data volumes are huge, metadata grows to significant sizes and as such is stored and managed by the *metadata providers* in a decentralized fashion. A *version manager* is responsible to assign versions to snapshots and ensure high-performance concurrency control. Finally, a *provider manager* is responsible to employ a chunk allocation strategy, which decides what chunks are stored on which data providers, when writes and appends are issued by the clients. A *load-balancing* strategy is favored by the provider manager in such way as to ensure an even distribution of chunks among providers.

Key features. BlobSeer relies on *data striping*, *distributed metadata management* and *versioning-based concurrency control* to avoid data-access synchronization and to distribute the I/O workload at large-scale both for data and metadata. This is crucial for achieving a high aggregated throughput for data-intensive applications, as demonstrated by our previous work [7–9].

3 The Generation Process

We designed and implemented a distributed random number generator that runs the HAVEGE unpredictable random number generator [2, 3] in a distributed fashion. HAVEGE produces unpredictable random numbers by gathering entropy from internal processor states using the memory hierarchy and the branch prediction mechanism. These processor states are inheritably volatile and impossible to tamper with in a controlled fashion by any application running on the target system.

Running the HAVEGE generator in a distributed setting with a large number of nodes in parallel, has several significant advantages like the fast generation of random numbers, availability of a large entropy pool (hundreds of CPUs) and the possibility of efficiently mixing output results from several entropy sources.

In order to obtain a single large sequence of random numbers that is shared at global scale, we leveraged BlobSeer, presented in Section 2. BlobSeer has two advantages that are highly relevant in our context. First, it enables sustaining a high throughput despite massively concurrent accesses, both for reading, writing and appending. Second, thanks to its versioning model, no distributed locking is necessary, yet strong consistency guarantees are provided: all operations are atomic. It is for these two reasons that we chose it as the underlying storage service for our random numbers.

3.1 Simple appending

The simplest method of obtaining large amounts of random sequences is to initiate the generation process on several nodes simultaneously and append the resulting sequences to a single BLOB. Besides the easily achievable solution of appending the whole output of a generator to the output of another, Blobseer provides the facility to interleave the results by appending a chunk (fixed amount of data) to the BLOB as soon as it is generated. In the context of this latter method it is important to note that the append operation is not deterministic and works on a first-come, first-served basis. Thus, due to due to the jitter experienced during the generation process, it will be impossible to determine in the final BLOB the provenance of each chunk. Fig. 1 depicts the process of simple generation and appending.

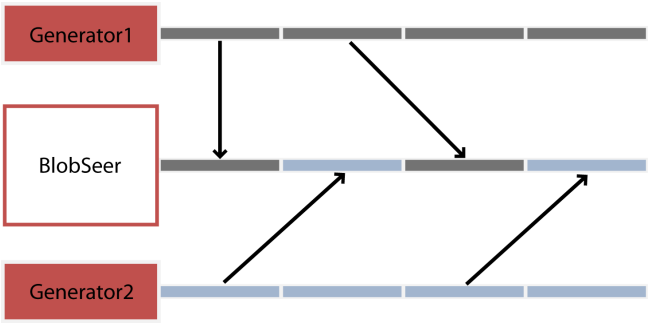

Fig. 1. Simple generation scheme based on appending

3.2 Combining multiple chunks

Considering the hypothetical scenario in which there is no jitter and all HAVEGE generators would run at the same speed, given the times when each generator started w, it could be theoretically possible to a certain extent for an adversary to determine the exact node which generated a particular chunk in the final output. One method to solve this problem is to create a new BLOB by combining several chunks of the original BLOB either in a deterministic or in a random manner. The combining function has to be chosen carefully to preserve the randomness of the inputs, for example the XOR logical function which is very easy to implement and also exhibits the property of preserving randomness. The process of combining chunks can be carried out in parallel with the generation of the initial BLOB, as depicted in Figure 2.

3.3 Next address logic based on previous data

Even if the results from a set of nodes are combined, as shown in the previous method, the results from the combination get appended one chunk at a time,

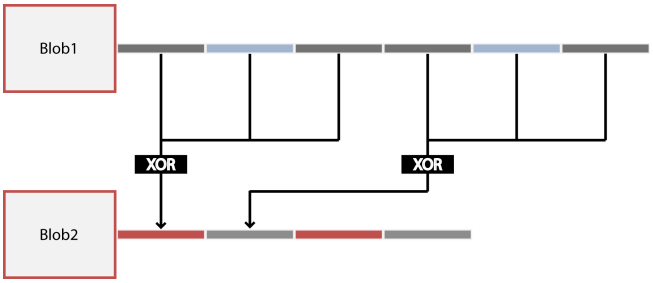

Fig. 2. Chunks in a blob are consecutively overwritten using the XOR operation

giving rise to the same problem as in the simple append approach. Hence it would be better not to place the results in the sequential order of their generation but rather at random locations in the BLOB. The method applied for computing the address where the next chunk will be written to is based on the value of previously generated data.

The writing of chunks is driven by the XOR operation such that new data is XOR combined with the existing data at the given address and hence instead of simply overwriting and losing certain results, this operation will preserve the randomness of the produced data. Consequently, when the the data to be XOR-ed is read from the target location, its value is used to determine the next address for the writing process. Furthermore, ignoring the read-write race conditions renders the results less predictable.

The writing of chunks starts at address 0, but since all generators initially read a 0 value to be XOR-ed, they will immediately overwrite the same location. Hence the first chunk is overwritten at least N+1 times in case N generators are used and due to this process the starting chunk source is very well hidden. The size of the output can be chosen at the beginning of the generation process hence the more chunks are generated the quality of randomness increases even more, but care should be taken to allow the execution until (almost) every bit of the output buffer has been written at least once. Fig. 3 depicts the process of determining the address for the next writing operation.

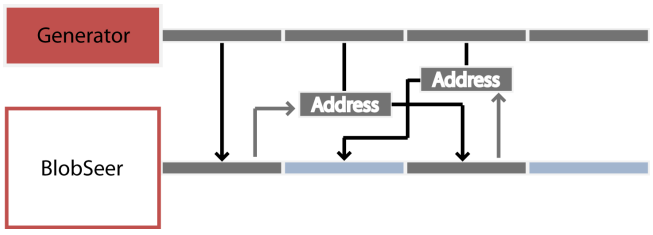

Fig. 3. The address of the next write is dependent on the just written page

4 Experimental results

4.1 Throughput and scalability

With the aim of testing the performance of the generator, together with the throughput of our Grid-based solution and in order to inspect how speed varies depending on the number of used parallel nodes, the first generation approach (i.e. interleaved generation) was chosen for measurements. For performing the tests, the chunk-size in BlobSeer was set to 16 MB and the measurements revealed that, since the HAVEGE generator's speed is inversely proportional to the applied *stephiding* number, when *Hvg1* is used, BlobSeer quickly becomes the bottleneck of the system. However, in case *Hvg32* (stephiding factor = 32) or an even higher version is used, the speed scales linearly with the number of employed computational nodes (as generation is slower than data management and persistence).

An important detail that needs to be considered when employing BlobSeer regards the direct influence of the BLOB's chunk-size on the maximum attainable write speed in the system. Consequently, when applied in conjunction with the HAVEGE generator, the faster the generator version is, the larger the chunk-size has to be, otherwise the system can not work in a continuous flow of generation, networking and persistence, but rather in bursts. These bursts appear when the generator fills up the buffers too fast, and the overhead of the network and the data management system (BlobSeer) is still significant when compared to the actual time needed for writing the information to the hard-disk or other storage devices. As a result, the above mentioned 16 MB chunk-size is chosen because it balances fairly well the load on the CPUs, the network interfaces and the storage units.

All measurements were performed on the *Grid5000* platform, clusters *paradent* and *paramount* situated in Rennes (Bretagne). The nodes connected by a Gigabit Ethernet interface have the following specification:

```
Carri System nodes (paradent)
   CPU:          Intel Xeon L5420
   2.5 Ghz / 6MB
      * 64 nodes x 2 cpus per node = 128 cpus
      * 128 cpus x 4 core(s) per cpu = 512 cores
   Memory:      32 GB

Dell nodes (paramount)
   CPU:         Intel Xeon 5148 LV
   2.33 Ghz
      * 33 nodes x 2 cpus per node = 66 cpus
      * 66 cpus x 2 core(s) per cpu = 132 cores
   Memory:      8 GB
```

Fig. 4 depicts the graphic for the generation and storage performance of the interleaved generation scheme, while Fig. **??** shows the generation and storage performance as a function of the BLOB's chunk-size.

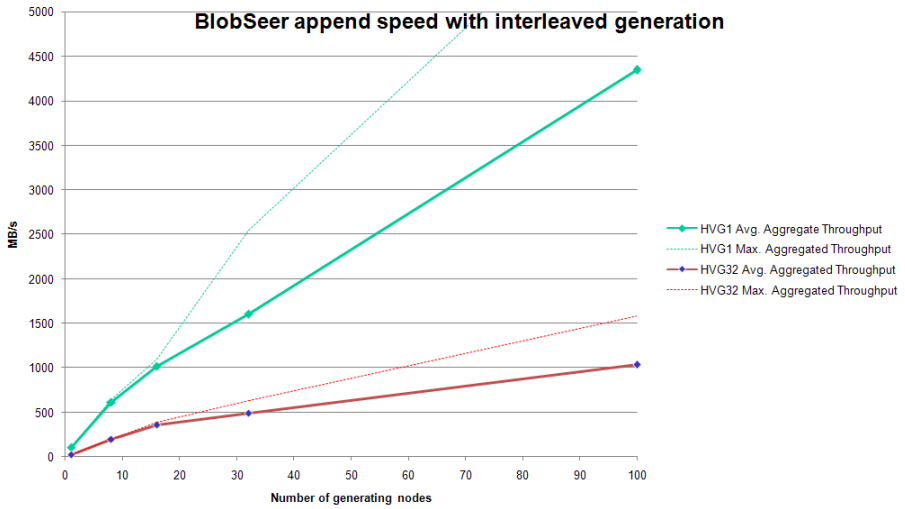

Fig. 4. Speed of generation and storage for the interleaved generation scheme

4.2 Randomness quality

In order to qualitatively test the developed distributed random number generator and determine the degree to which the applied generation schemes are having a positive effect on the produced output, we turned to statistically test the resulting volume of random data with the help of the well-known NIST statistical test suite [1].

The HAVEGE generator produces cryptographically strong randomness and the improvement process focuses on increasing the quality of these sequences even more by using a very simple but very effective operation, the XOR. Thus, in order to highlight the significant positive effect of the XOR operation on the quality of generated randomness, a series of tests were performed and the experimental results are presented in the following.

One category of test data is formed by 1,000 random number sequences of 1 MB each generated with *Hvg32*. The second category consists of four times the amount of previously generated randomness, or in other words 4,000 sequences that are combined together in groups of four using the XOR operation. This method was repeated until one new sequence was formed by a number of 128 original sequences combined together.

The results show that the XOR operator has a powerful feature of maintaining and even of improving the overall quality of the combined sequences. As usual with the HAVEGE generator, a percentage of around 98% of the NIST tests were passed, in all the generation scenarios.

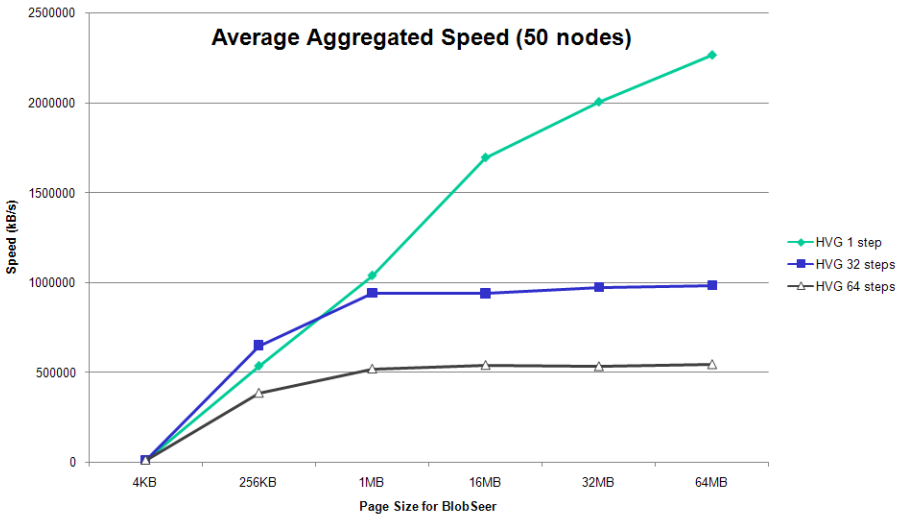

Fig. 5. Speed of generation and storage as a function of chunk-size of the BLOB

5 Conclusions

This work proposed a distributed random number generator that aims to achieves high throughput and scalability at large scale without sacrificing the quality of the generated sequences compared to serial generators. To this end, it leverages HAVEGE to generate local fine-grained random subsequences on multiple nodes of a cluster in parallel, which are then combined into a global random sequence through BlobSeer, a high-throughput distributed storage system. How to combine the subsequences is a key contribution of this work: we propose a XOR-based scheme that in itself introduces randomness in the combination process and thus increases the resilience to attacks that are launched from compromised nodes.

Experimental results performed on the Grid'5000 testbed demonstrate both the scalability and randomness quality of our proposal: we show an aggregated throughput that grows from 500 MB/s for 10 nodes to 4500 MB/s for 100 nodes, all this achievable with a 98% pass rate for the NIST randomness quality tests. Furthermore, our experiments emphasize the significant positive effects of the XOR operation on improving the quality of random number sequences compared to simple append, which is an added beneficial side-effect of the original intent to increase resilience.

Given our encouraging preliminary results, we plan to develop this work further by performing a more in-depth analysis of resilience and refine our combination method to adapt and tolerate a given a target precent of compromised nodes.

References

1. A. Rukhin et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST Special Publication 800-22 (with revisions dated April, 2010).
2. A. Seznec, N. Sendrier. HAVEGE: a user-level software heuristic for generating empirically strong random numbers. ACM Transaction on Modeling and Computer Simulations, Vol. 13, Issue 4, 2003.
3. A. Seznec, N. Sendrier. Hardware Volatile Entropy Gathering and Expansion: generating unpredictable random numbers at user level. INRIA Research Report, RR-4592, 2002.
4. A. Suciu, R. Potolea. A Taxonomy for Grid Applications, Proc. of International Conference on Automation, Quality and Testing, Robotics AQTR 2008, Vol. 3, pp. 365-368, 2008.
5. Buyya, Rajkumar and Yeo, Chee Shin and Venugopal, Srikumar and Broberg, James and Brandic, Ivona Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst., Vol. 25, Issue 4, pp. 599-616, 2009.
6. Jégou, Yvon and Lantéri, Stephane and Leduc, Julien and Melab Noredine and Mornet, Guillaume and Namyst, Raymond and Primet, Pascale and Quetier, Benjamin and Richard, Olivier and Talbi, El-Ghazali and Iréa, Touche Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. International Journal of High Performance Computing Applications, Vol. 20, Issue 4, pp. 481-494, 2006.
7. Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise and Alexandra Carpen-Amarie. BlobSeer: Next Generation Data Management for Large Scale Infrastructures. Journal of Parallel and Distributed Computing, Vol. 71, Issue 2, pp. 169-184, 2011.
8. Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé and Matthieu Dorier. BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications. IPDPS '10: Proc. 24th IEEE International Parallel and Distributed Processing Symposium, pp. 1-12, Atlanta, USA, 2010.
9. Jesús Montes, Bogdan Nicolae, Gabriel Antoniu, Alberto Sánchez and María Pérez. Using Global Behavior Modeling to Improve QoS in Data Storage Services on the Cloud. CloudCom '10: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science, pp. 304-311, Indianapolis, USA, 2010.