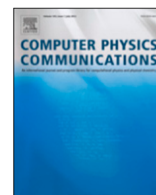
Computer Physics Communications 258 (2021) 107581

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc



Data Sampling methods in the ALICE O^2 distributed processing system[☆]

Piotr Konopka[a,b,*], Barthélémy von Haller[b]

[a] Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland
[b] European Organization for Nuclear Research (CERN), 1211 Geneva 23, Switzerland

ARTICLE INFO

Article history:
Received 6 May 2020
Received in revised form 17 July 2020
Accepted 26 August 2020
Available online 12 September 2020

Keywords:
CERN
ALICE
O^2
Data Sampling
Message-based system
Distributed processing system
Data quality control
Pseudo-random number generators

ABSTRACT

The ALICE experiment at the CERN LHC focuses on studying the quark-gluon plasma produced by heavy-ion collisions. Starting from 2021, it will see its input data throughput increase a hundredfold, up to 3.5 TB/s. To cope with such a large amount of data, a new online-offline computing system, called O^2, will be deployed. It will synchronously compress the data stream by a factor of 35 down to 100 GB/s before storing it permanently.

One of the key software components of the system will be the data Quality Control (QC). This framework and infrastructure is responsible for all aspects related to the analysis software aimed at identifying possible issues with the data itself, and indirectly with the underlying processing done both synchronously and asynchronously. Since analyzing the full stream of data online would exceed the available computational resources, a reliable and efficient sampling will be needed. It should provide a few percent of data selected randomly in a statistically sound manner with a minimal impact on the main dataflow. Extra requirements include e.g. the option to choose data corresponding to the same collisions over a group of computing nodes.

In this paper the design of the O^2 Data Sampling software is presented. In particular, the requirements for pseudo-random number generators to be used for sampling decisions are highlighted, as well as the results of the benchmarks performed to evaluate different possibilities. Finally, a large scale test of the O^2 Data Sampling is reported.

© 2020 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. The ALICE experiment

ALICE (A Large Ion Collider Experiment) [1] is one of the four major particle detectors at the CERN Large Hadron Collider (LHC). It is designed to study the quark-gluon plasma by observing fundamental and composite particles appearing in the debris produced by heavy-ion and proton collisions. Starting from November 2009, the ALICE experiment has successfully recorded data, which allowed several measurements of the properties of that primordial state of matter.

2. The ALICE upgrade

During the years 2019–2020 CERN is taking an operation break, called the Long Shutdown 2 (LS2) [2], during which the accelerator complex and associated experiments are refurbished and upgraded. The ALICE experiment will, among many activities, replace entirely the key tracking sub-detectors in order to achieve higher resolution and increase the amount of data they produce [3].

The necessity to move away from hardware triggering will make a crucial difference from the data acquisition point of view. The physics topics that will be covered by ALICE after the LS2 are characterized by a very small signal-to-noise ratio, which will result in triggering techniques being inefficient or even inapplicable. To the same consequences leads a need to cope with an increased heavy-ion collision rate of 50 kHz, which will require the ALICE's Time Projection Chamber to send data in a continuous mode to avoid dead time generated by triggers and event pile-up. It is estimated that the full upgraded detector will generate a data stream of up to 3.5 TB/s, which is a number 100 times higher than the highest values achieved in the last data taking period.

To cope with the given requirements a new online-offline computing system, called O^2, is being deployed [4]. The software relies on message-based architecture, consisting of multiple processes exchanging data via message queues with the zero-copy approach. The computing farm is estimated to consist of around 1000–2000 servers arranged into two major groups. The First Level Processors (FLPs) will perform detector-specific tasks, while

[☆] The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author at: Faculty of Electrical Engineering, Automatics, Computer Science and Biomedical Engineering, AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland.

E-mail addresses: piotr.jan.konopka@cern.ch (P. Konopka), barthelemy.von.haller@cern.ch (B. von Haller).

https://doi.org/10.1016/j.cpc.2020.107581
0010-4655/© 2020 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

the Event Processing Nodes (EPNs) will take care of data aggregation from all FLPs for a given time-period. The architecture will be highly heterogeneous. The Common Readout Units (CRUs) with FPGAs (Field-programmable gate array) will receive data from the sub-detectors on optic links, optionally compress it and directly transfer to the FLPs' memory [5]. Each FLP is expected to host between one and three of these FPGA cards. Whenever suitable, the processing tasks on the EPNs will be delegated to graphics cards.

During the online processing, the raw data will be filtered, compressed and reconstructed, while not keeping the original bit-stream. Given the risk associated with losing precious data, an immediate data Quality Control (QC) is crucial to assess the correctness of the detectors' operations, their calibration as well as the data processing itself [6].

3. Quality Control in the O^2 system

The online data Quality Control will be performed by more than 100 QC Tasks running each in parallel on many nodes, which will spy on various data types generated in consecutive processing stages. These will be user algorithms unified under a common interface and executed by the framework. The algorithms will produce Monitor Objects out of data, which are expected to be mostly histograms of varied dimensions. QC Tasks will run either on the main processing machines or on dedicated servers, depending on their computational cost.

In the O^2 system partial data will be distributed among multiple nodes. When QC Tasks will be run on the main processing machines in parallel, they will produce incomplete results, which will be collected and combined by Mergers. In case that a parallel QC Task shall be distributed among 750 EPNs (the maximum expected number), one Merger will have to sustain a flow of 750 objects updated each minute.

Complete Monitor Objects will constitute a base for Qualities – Null, Bad, Medium or Good marks with optional metadata. These will be produced by user-written Checks, executed by the framework.

Post-processing software will further transform a portion of Monitor Objects in order to obtain correlations between various observables and trend them in time. The results will also be evaluated and assigned a Quality, in this case relying partially on machine learning methods.

A database will permanently store Monitor Objects and Qualities, discarding only some intermediate versions generated during data-taking runs. Shifters and experts will examine these in a general Quality Control GUI or dedicated applications based on the same web-based framework.

4. Data sampling requirements

Moving the full data stream to QC Tasks, as well as analyzing it, will require excessive CPU, memory and bandwidth resources. Therefore, whenever possible, the QC Tasks will perform the quality assessment based on a fraction of the events which might be selected randomly or by matching certain criteria. Data of different structure and generated on several processing stages will need to be evaluated. In rare cases, the main data stream should be blocked, to make sure that there are no messages dropped on the way to a task.

The task of sampling and providing the data to QC tasks as well as other potential clients will be performed by the Data Sampling software, which is the topic of this paper.

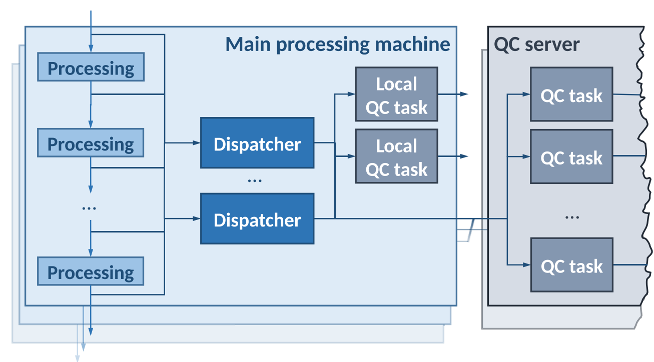


Fig. 1. The message-passing process topology.

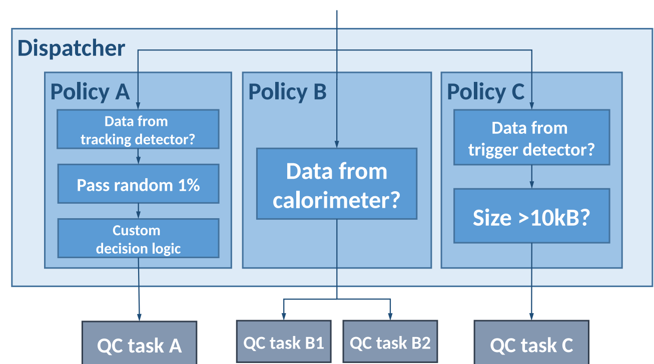


Fig. 2. A Dispatcher's inner logic example.

5. Data sampling design

Fig. 1 presents an example of the message-passing process topology which incorporates a main processing chain, Quality Control infrastructure and Dispatchers. The latter are the entities responsible for sampling and forwarding data. The design foresees one or more parallel Dispatchers, which subscribe to messages produced by the main processing workflow, receiving them in a round-robin schedule. Selected messages are passed accordingly to local and remote clients.

The characteristics of desired data are defined in the Data Sampling Policies (Fig. 2). Messages are matched to Policies by comparing their headers. The rules of passing data forward can be specified as a set of Data Sampling Conditions, which can perform for example:

- Random sampling
- Filtering messages matching certain payload size range
- Filtering a number of consecutive messages in a given interval
- Any custom filtering, defined by inheriting the Data Sampling Condition class

A Dispatcher can be reconfigured at run-time — Data Sampling Policies can be switched on and off, and the list of Data Sampling Conditions can be modified. Together with a passed message payload, the Dispatcher pushes an additional header at the top of the header stack. It encapsulates potentially useful information i.e. sampling decision time, total numbers of matching messages seen and passed.

The features provided by the design allow for an advanced data selection, which should help to avoid transporting unnecessary messages, therefore minimizing the amount of required CPU, memory resources and bandwidth. Especially, one can consider

moving from the traditional approach of evaluating frequency of certain events in a histogram to a more lightweight method, which involves filtering and counting only unwanted ones.

6. Statistical soundness of random sampling

6.1. Rationale and approaches for sampling data

In many cases analyzing only a portion of the full data stream is completely sufficient, provided that the samples are chosen in a statistically sound manner and a desired percent of data samples is successfully passed to clients. Selecting every 100th message might be indeed simple and efficient performance-wise, but at the same time it might introduce unwanted biases if sampled data stream contains any patterns, for example every 25th payload being much larger.

As the first level of processing in the new ALICE data acquisition system will be sub-detector-dependent, parts of data corresponding to the same events will be spread over a large group of nodes, though identified with a common identifier, later referred to as *TimesliceID*. It introduces an additional level of difficulty to provide data to QC tasks, which need samples from multiple machines, especially given that the correct order of arriving messages is not guaranteed. Choosing pseudo-randomly a fraction of related messages would require either having a centrally-driven decision-making mechanism or having each Dispatcher operate independently, while taking the same decisions.

The first alternative might be problematic to implement. Distributing and storing decisions beforehand for all possible *TimesliceIDs* would require extensive amounts of memory (around 18 EB for 64-bit *TimesliceID*). On the other hand, requesting and receiving sampling decisions in real-time from a central node would introduce a significant overhead. A `ping` measurement showed an average round-trip time of 145 μs in a setup of two servers with Intel Ethernet CNA X520 cards connected with each other via a Dell Force10 S4810 ultra-low-latency switch. Because of that, this approach would not fit the expected message rates of tens of thousands messages per second on each computing node. Waiting for sampling decisions synchronously would significantly slow down the performance, while receiving the answers asynchronously would require additional buffering of messages, increasing RAM usage. Also, a central decision-making node would constitute a single point of failure and its load would rise proportionally to the number of requests, making this solution more difficult to scale.

To follow the decentralized approach, one would need to use a pseudo-random number generator (PRNG), which can return a value given a seed and an input number (respectively: a data acquisition run number and a *TimesliceID* in this case). Its result should be deterministic, independent from computer architecture and, while not being completely random, it should satisfy the statistical soundness requirements. As most PRNGs are intended to produce either ones or zeros with equal probability or equally distributed numbers in a range of [0, 1], additional operation has to be performed in order to receive positive decisions with a desired probability (Eq. (1)). A random number *rand* is compared with a threshold, which is a product of the *fraction* parameter and the highest possible random number $rand_{max}$. Having a greater or equal comparison for negative decisions prevents any messages to be dispatched when $fraction = 0$ is set.

$$Decision = \begin{cases} 1 & rand < fraction \cdot rand_{max} \\ 0 & rand \geq fraction \cdot rand_{max} \end{cases} \quad (1)$$

6.2. Evaluated sampling methods

Several pseudo-random number generation methods, which could fulfill the presented needs, have been considered. The research was narrowed to those which have a ready-to-use, open-source, implementation in C++. The following methods have been evaluated:

- PCG (Permuted Congruential Generator) [7] which has a jump ahead or backwards feature. This allows the user to retrieve a pseudo-random number corresponding to any *TimesliceID* by iterating the state forward or backward. The cost of jumping is in order of *O(log k)*, where *k* is the length of the jump. In particular, the `pcg32_fast` variation was tested.
- PRNGs available in the ROOT framework [8] unified under a common interface, i.e. `TRandom`, `TRandom1`, `TRandom2`, `TRandom3`, `TRandomMT64`, `TRandomRanlux48`, `TRandomMixMax17`, `TRandomMixMax`. Since none of them provided a possibility to move through the state freely, a workaround has been applied. Before requesting a new random value, a PRNG was reinitialized with a product of seed and *TimesliceID*. However, this approach was expected to decrease performance and potentially impair the randomness' quality — the scale of these effects had to be evaluated.
- Two simple hash functions, one of them based on the `splitmix64` generator [9], found on the website [10]. Similarly to the workaround for ROOT PRNGs, the input value was being multiplied by a seed. The second algorithm took use of the `hash_combine` function from the boost library [11], which can produce a hashed value out of two argument numbers.

6.3. Sampling methods tests

The presented PRNGs had to be examined in an environment representing the use-case of the Data Sampling software. The `dieharder` test suite [12] has been chosen to evaluate the generated numbers which were used for comparison with a threshold value (Eq. (1)). All tests in the package are executed in a similar fashion (described in detail in [13]). The benchmarks are run multiple times, each one resulting in a single statistic, which is transformed into a *p*-value. By default, 100 p-values are produced, however, this number is configurable. In case that tested binary sequences are truly random, the resulting p-values should be uniformly distributed in the range of [0, 1], which is checked using the Kolmogorov–Smirnov test. This way, obtaining a large portion of seemingly acceptable, but closely packed p-values is treated as not likely to be random (e.g. three tests resulting in p-values 0.271, 0.272, 0.275). The philosophy of `dieharder` tests assumes that one can never confirm a PRNG randomness, but only disprove it. Moreover, a good generator might occasionally fail the tests by accidentally producing an unlikely bit sequence, therefore, one should not take one result as decisive. The default number 100 of p-values in one test run makes a compromise between the test duration and accuracy, but it should be increased in case of receiving ambiguous results.

In comparison with just the `dieharder` suite, the PRNGs included in the ROOT framework were at disadvantage, since they produced floating-point numbers between 1 and 0, which cannot be completely random bitwise due to notation method, even after scaling it to maximum integer values. At the same time, the `dieharder` tests are designed to examine the equal probability of 0s and 1s. For these reasons, aside from using the test suite, additional evaluation methods have been proposed, which are explicitly designed to analyze bitstreams with non-equal probabilities of seeing 1 or 0.

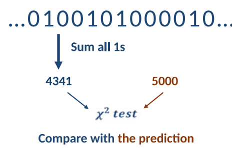


Fig. 3. The χ^2 test (A).

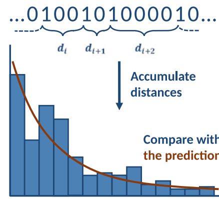


Fig. 4. The Runs test (B).

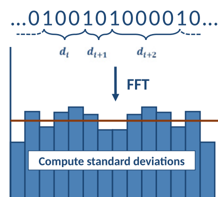


Fig. 5. The FFT test (C).

The first test (A) is supposed to be the easiest to pass. It involves counting the number of 1's in a sequence of N decisions, computing a χ^2 value (Eq. (2))(Fig. 3) and a p-value with the `TMath::Prob` function in the ROOT package. Following the philosophy of the `dieharder` suite, the test is ran 100 times and the distribution of p-values is checked to be uniform using the Kolmogorov Test. The result should indicate whether a given method returns the specified amount of positive decisions while being statistically sound. However, it cannot possibly detect repeating patterns of 1's and 0's.

$$\chi^2 = 2\frac{(N \cdot fraction - trues)^2}{N \cdot fraction} \tag{2}$$

The second benchmark (B) is supposed to address situations when 1's are distributed among the bitstream with 'suspiciously' repetitious arrangement (Fig. 4). The distances between consecutive 1's in a sequence of N decisions are stored in a histogram. They are compared with the predicted negative binomial distribution (Eq. (3)), which for this case takes the form of Eq. (4). The χ^2 is obtained with the `TH1::ChiSquare` method against the expected distribution Eq. (4). To obtain the p-value, `TMath::Prob` is used. Again, the test is performed 100 times and the distribution of resulting p-values is checked to be uniform.

$$f(k; r, p) \equiv Pr(X = k) = \binom{k + r - 1}{k} p^k (1 - p)^r \tag{3}$$

$$f(k; 1, 1 - f) = (1 - f)^k f \tag{4}$$

This test might still be passed by a properly designed sequences of bits. Thus, to complement it, the sequence of distances between following 1's was additionally analyzed in frequency domain, using Fast Fourier Transform (C)(Fig. 5). It is expected to uncover some repetitive patterns in the form of uneven frequencies distribution. The main benchmark result is a sum of standard deviations of frequencies' powers (excluding the constant), calculated for 100 test runs.

The statistics toolbox provided in the ROOT framework has been used to implement the tests. For validation, the benchmarks have been run against the Linux `/dev/urandom` source.

Aside from the randomness benchmarks, also the performance of each method has been evaluated using a server with Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz.

6.4. Tests results

Table 1 presents the tests results.

The `dieharder` suite tests consumed raw numbers generated by each method. For the reasons mentioned earlier (floating-point numbers generators being in disadvantage), its results were treated only as a suggestion rather than definitive confirmation or disqualification of a method. In the case of PRNGs provided by ROOT framework, it is crucial to state that the negative results do not indicate a bad quality of these generators as they were abused by setting their seed each time before drawing a new number. This fact hindered their computational performance as well.

For the first and second custom randomness tests, achieving a p-value higher than 0.005 and smaller than 0.995 was put as a requirement. It corresponds to the probability of 1% that tested bit sequences would be incorrectly perceived as too bad or suspiciously good while actually being random, which was an acceptable risk in this context. In the `dieharder` suite exceeding these thresholds indicates a weak test result, while achieving values out of the range of $[0.0005, 0.9995]$ is treated as a test failure. The sum of standard deviations returned by the third test was supposed to be close to the result obtained for the `/dev/urandom` generator. The three benchmarks were ran assuming $fraction = 0.01$.

The results show that the family of TRandom generators cannot be successfully used in the described application. Setting the seed hinders the performance and possibly the randomness as well. The methods which have very large state (MixMax) suffered the most, being unable to provide enough data to the benchmarks in a sufficient time. The two methods which took advantage of hash functions produced surprisingly satisfying results. Their simplicity ensured an excellent performance, while providing a good quality of randomness. The PCG generator has passed each benchmark and it has proved to be lightweight in terms of computational resources needs. The result of 2.95 ns per call was achieved for evaluating incremental *timesliceIDs*. For more realistically distributed values it requires additional 5–20 ns, which is still expected to be negligible in comparison to more computationally demanding parts of the code.

PCG has been chosen for the application, as its good quality is also confirmed by other studies [14,15]. At the same time, in case there is a need for performance improvement, the possibility to switch to the first hash function is still available.

7. Comprehensive data sampling benchmarks

Having chosen the random sampling method, a global performance benchmark was carried out. Its aim was to measure the maximum amount of messages which can be passed forward by Dispatcher in various conditions. Fig. 6 presents the topology of processes taking part in the test. A configurable number of data producers publishes messages at the highest possible rate. They are received and forwarded by one or more parallel Dispatchers. All of the messages are sent to the receiver, which is only used as an endpoint of the topology. Each configuration was ran 5 times for 5 min to observe variations of the results, which could occur due to e.g. changing affiliation between processes and CPU cores. Moreover, as the performance of Dispatcher is largely influenced

Table 1
The test results of random sampling methods for $N = 10^7$ and *fraction* $= 0.01$.

PRNG	Test				
	`dieharder`	ns/call	A (KS test p-value)	B (KS test p-value)	C (σ) (lower is better)
Hash function 1	PASSED	2.66	0.0101	0.2809	1552
Hash function 2	A dozen FAILED	2.07	0.5806	0.3667	1551
PCG	PASSED	2.95*	0.2106	0.8127	1552
TRandom	All FAILED	7.97	0.0000	0.0000	6067
TRandom1	All FAILED	300.2	0.0000	0.0000	2665
TRandom2	A half FAILED	30.60	0.1545	0.0000	1554
TRandom3	PASSED	1378	0.0243	0.6994	1552
TRandomMT64	PASSED	2360	0.0541	0.9671	1552
TRandomRanlux48	Almost All FAILED	118.1	0.0000	0.0000	2418
TRandomMixMax17	Too slow	22 224	0.0101	0.5806	1551
TRandomMixMax	Too slow	4 487 360	Too slow	Too slow	Too slow
`/dev/urandom`	PASSED	923.0	0.0366	0.2105	1552

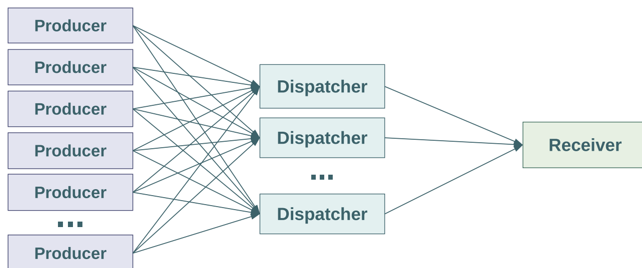


Fig. 6. The performance benchmark's process topology.

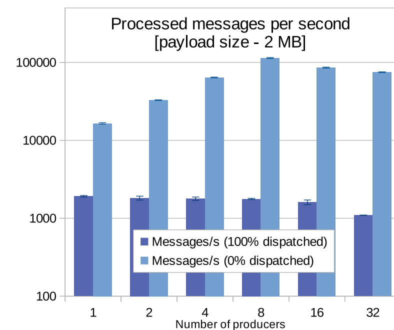


Fig. 8. Dispatcher's performance depending on the number of data producers.

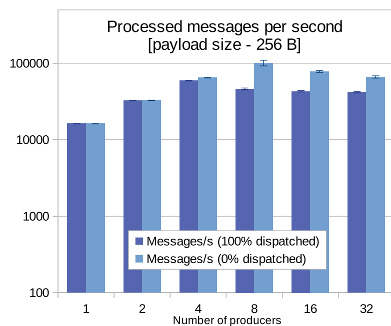


Fig. 7. Dispatcher's performance depending on the number of data producers.

by the desired fraction of messages passed forward, the benchmark required repeating it for two extreme values (100% and 0%) to develop a better understanding of the performance range. All of the presented results were achieved on a machine with Dual Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz and 128 GB RAM.

The goal of the first test was to understand how the number of message producers influences the performance of Dispatchers. When very small (payload of 256 B) messages are distributed, 4 producers are needed to saturate Dispatcher and reach the highest message passing rate of ∼59000 per second, while for 8 producers the peak performance of ∼100000 messages rejected per second was reached (Fig. 7). For larger payload sizes, the overhead of copying memory takes a bigger toll (Fig. 8). As a room for improvement, the decrease of received messages for larger numbers of producers might be mitigated by designating multiple Dispatchers to separate channel groups.

The relationship between the amount of processed messages and the payload size is presented in Fig. 9. The plot confirms that transferring data with too fine granularity hinders the overall throughput — the performance over 2 GB/s was achieved for payload sizes between 256 kB and 1 GB, with the highest value of 3480 MB/s with 256 kB payloads. The performance of raw message passing (1 B sized payloads) reaches ∼58 000 messages/s when dispatched and over ∼114 000 messages/s when ignored. This translates to around 8.7 μs required to receive and reject each message.

A possible way to increase the number of processed messages is to spawn parallel Dispatchers which can receive data in the round-robin order. This method, of course, is not anticipated to reduce the total CPU and memory resources required, but it constitute an additional approach of improving the Data Sampling performance. Fig. 10 does not show any improvements for cases of dispatching very small messages. However, using parallel Dispatchers can increase the total number of sampled messages for larger payload sizes, as indicated in Fig. 11.

8. Summary

The Data Sampling software plays a significant role in the data Quality Control of the O^2 system. Its efficiency will have an impact on how much data can be analyzed without disturbing the main processing flow. It is intended to help reducing the bandwidth requirements, especially by providing several methods of data selection and keeping sampling statistics.

The random sampling methods were thoroughly evaluated to provide statistically sound representation of data to the QC Tasks while keeping the performance overhead minimal. A set of randomness benchmarks, which can be applied to similar use-cases, have been presented.

Global performance tests of the Data Sampling software have also been carried out in order to achieve a thorough understanding of its performance and how it is influenced by different configurations. The authors hope that this research will prove useful to fellow designers and developers of message-based systems, as the results are expected to map to analogous topologies of processes.

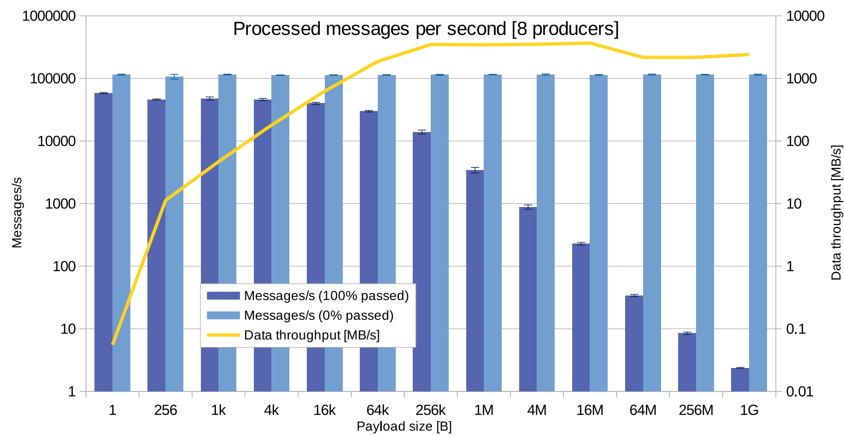


Fig. 9. Dispatcher's performance depending on message payload size.

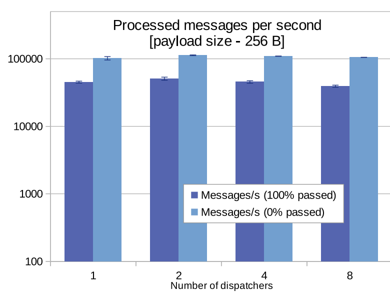


Fig. 10. Dispatcher's performance depending on the number of parallel dispatchers (payload size of 256 B).

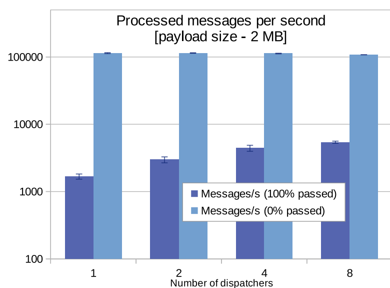


Fig. 11. Dispatcher's performance depending on the number of parallel dispatchers (payload size of 2 MB).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

[1] ALICE Collaboration, J. Instrum. 3 (2008) S08002, http://dx.doi.org/10.1088/1748-0221/3/08/S08002.
[2] M. Bernardini, K. Foraz, CERN Yellow Rep. 2 (00) (2016) 290, URL https://e-publishing.cern.ch/index.php/CYR/article/view/159.
[3] The ALICE Collaboration, J. Phys. G: Nucl. Part. Phys. 41 (8) (2014) 087001, URL http://stacks.iop.org/0954-3899/41/i=8/a=087001.
[4] The ALICE Collaboration, Technical Design Report for the Upgrade of the Online–Offline Computing System, Tech. Rep. CERN-LHCC-2015-006, CERN, 2015.
[5] J. Mitra, S. Khan, S. Mukherjee, R. Paul, J. Instrum. 11 (03) (2016) C03021, http://dx.doi.org/10.1088/1748-0221/11/03/c03021.
[6] B. von Haller, P. Lesiak, J. Otwinowski, The ALICE Collaboration, J. Phys. Conf. Ser. 898 (3) (2017) 032001, URL http://stacks.iop.org/1742-6596/898/i=3/a=032001.
[7] M.E. O'Neill, PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation, Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 2014.
[8] R. Brun, F. Rademakers, ROOT - an object oriented data analysis framework, 1996, pp. 81–86, URL http://root.cern.ch/.
[9] S. Vigna, A fixed-increment version of Java 8's Splittable Random generator, URL http://xorshift.di.unimi.it/splitmix64.c.
[10] Lear, T. Mueller, What integer hash function are good that accepts an integer hash key? Stack Overflow, URL https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-accepts-an-integer-hash-key/12996028#12996028.
[11] D. James, Combining hash values, URL https://www.boost.org/doc/libs/1_70_0/doc/html/hash/combine.html.
[12] R.G. Brown, D. Eddelbuettel, D. Bauer, Dieharder: A random number test suite, URL http://webhome.phy.duke.edu/~rgb/General/dieharder.php.
[13] R.G. Brown, dieharder(1) - Linux man page, URL https://linux.die.net/man/1/dieharder.
[14] J. Lockhart, K. Rawashdeh, C. Purdy, Verification of random number generators for embedded machine learning, 2018, pp. 411–416, http://dx.doi.org/10.1109/NAECON.2018.8556780.
[15] D. Lemire, Testing non-cryptographic random number generators: my results, URL https://lemire.me/blog/2017/08/22/testing-non-cryptographic-random-number-generators-my-results/.