Scalable monitoring data processing for the LHCb software trigger

Stefano Petrucci[1,*], *Rosen* Matev[2,**], and *Roel* Aaij[3,***]

[1]University of Edinburgh
[2]CERN
[3]Nikhef

Abstract. The LHCb High Level Trigger (HLT) is split in two stages. HLT1 is synchronous with collisions delivered by the LHC and writes its output to a local disk buffer, which is asynchronously processed by HLT2. Efficient monitoring of the data being processed by the application is crucial to promptly diagnose detector or software problems. HLT2 consists of approximately 50000 processes and 4000 histograms are produced by each process. This results in 200 million histograms that need to be aggregated for each of up to a hundred data taking intervals that are being processed simultaneously. This paper presents the multi-level hierarchical architecture of the monitoring infrastructure put in place to achieve this. Network bandwidth is minimised by sending histogram increments and only exchanging metadata when necessary, using a custom lightweight protocol based on boost::serialize. The transport layer is implemented with ZeroMQ, which supports IPC and TCP communication, queue handling, asynchronous request/response and multipart messages. The persistent storage to ROOT is parallelized in order to cope with data arriving from a hundred of data taking intervals being processed simultaneously by HLT2. The performance and the scalability of the current system are presented. We demonstrate the feasibility of such an approach for the HLT1 use case, where real-time feedback and reliability of the infrastructure are crucial. In addition, a prototype of a high-level transport layer based on the stream-processing platform Apache Kafka is shown, which has several advantages over the lower-level ZeroMQ solution.

1 Introduction

Starting from Run 3 in Q2 2021, the LHCb experiment will run on a trigger-less configuration where the detector will be read by the Event Filter Farm (EFF) at 30MHz. Run 3 HLT will have the same two-level configuration as in Run 2. HLT1 will be synchronous with collisions delivered by LHC writing its output to a local disk buffer which will be asynchronously processed by HLT2. With this trigger-less configuration, monitoring very efficiently the data processed by the application will be crucial to promptly diagnose detector or software problems. For this purpose, scalability and performance of the Run 2 HLT monitoring infrastructure

[*]e-mail: s.petrucci@cern.ch
[**]e-mail: Rosen.Matev@cern.ch
[***]e-mail: Roel.Aaij@cern.ch

were studied and a prototype of a high-level transport layer based on the stream-processing platform Apache Kafka was built and tested.

2 The LHCb HLT monitoring infrastructure

The main task of the HLT is to reduce the background while maintaining a high efficiency for physics signal candidates. This operation happens during the data taking so everything rejected at this stage is permanently lost. In order to detect problems happening in this phase: a small fraction of the HLT computing resources is used to generating monitoring histograms and counters. Since HLT applications are distributed on all the servers of the EFF, the monitoring information are scattered on the entire farm. This requires a system to collect, to merge and to publish them. Due to the different tasks and time requirements of the HLT1 and HLT2, the Run 2 monitoring infrastructure has two separate implementations for each HLT stage.

2.1 HLT1

HLT1 is the selection stage before the buffer therefore it has to perform very quick decisions writing its output on disk as soon as possible in order to avoid dead time. For this reason, the number of monitoring histograms and counters produced is limited. Every HLT1 node generates the partial histograms in the form of "full histogram" containing all the metadata (axis, bin labels, titles, etc.). Those histograms and counters are collected, merged and published using DIM [1] as shown in Figure 2.

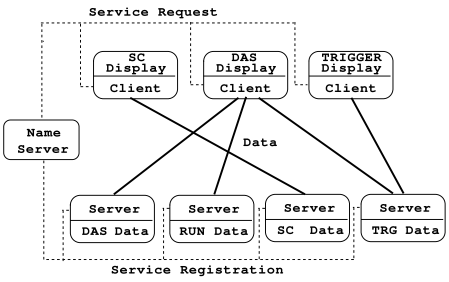

Figure 1. DIM communication scheme [1]. Clients request services to the Name Server and, once they get the coordinates, a one-to-one communication starts.

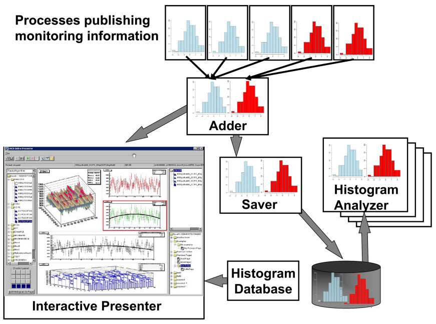

Figure 2. Scheme of DIM collecting, merging and publishing histograms [2].

DIM distributes data as a service: there is a central server, the Name Server, where servers producing information subscribe specifying the information to publish. Clients check with the Name Server if the required service is available and where to find it. Once the client gets the coordinate of the server providing the information, the communication happens between server and client (as shown in Figure 1). In case of missing services, the Name Server takes care of them: if a client requires an unavailable service, the Name Server notifies the client once the service is back online. Due to the one-to-one transfer protocol, the failure of the Name Server doesn't affect communications already started. If the Name Server fails, once it comes back online, all the servers will register their services again and the clients will request again the services they need. This system is very well tested, running smoothly for more than two decades of data taking (LHCb, Delphi and BaBar).

2.2 HLT2

HLT2 has more time to perform decisions and to produce monitoring information since it runs asynchronously reading from the buffer. For this reason, the request of monitoring information in the HLT2 is way higher compared to the HLT1: around 50000, processes each one producing 4000 histograms, are running on the farm. This results in roughly 200 million histograms that need to be aggregated and published. To satisfy the HLT2 requirements, a different custom system based on ZeroMQ as messaging library was developed and put in place. The HLT2 monitoring system was designed to deal with a higher number of histograms and counters keeping the network load as low as possible. For this reason, some new features were introduced in respect to the HLT1 system: the messages were split into increment and metadata and the EFF was grouped in sub-farms and top level adders.

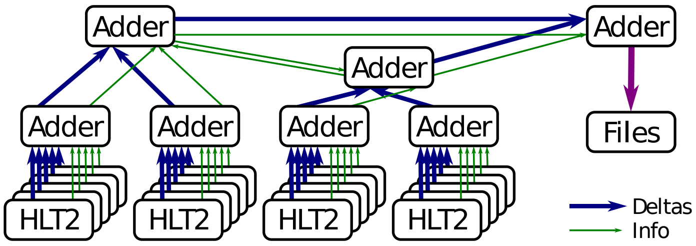

Figure 3. HLT2 monitoring system scheme. The EFF is split in sub-farms pushing deltas and info messages to the sub-farm adders. Sub-farm adders merge and push the information to the top-level adders, responsible to merge and publish the full histograms and counters.

Metadata are all the information related to the style of the final histograms (axis, labels, etc.) and they are sent only during initialization. Increments are the difference between the latest histogram produced and the last one sent. To better distribute the workload, avoiding having a single node aggregating $O\left(10^8\right)$ histograms and counters, the farm is split into sub-farms. Each sub-farm has a specific node responsible to collect, merge and push the fragments to the next level of aggregation. While ZeroMQ takes care of finding the best path to deliver the messages, the queue parameters need to be adjusted to meet the implementation's requirements. Tuning the queue is a crucial part of the HLT2 monitoring implementation requiring a lot of attention: if the queue doesn't work smoothly, there is a high chance of dropping messages. For this reason, a software simulating the behavior of a typical HLT2 monitoring workload was developed and strongly used to optimize the queue parameters.

3 Prototyping a monitoring system using Kafka

The experience gained from developing the HLT2 monitoring system was used to prototype a new infrastructure running on a commercial queue system in order to have a product already optimized and tested. Since Apache Kafka was already used in the LHCb Online for logs and monitor [3], it was decided to implement a prototype resembling the HLT2 monitoring system running on Kafka. In details, Kafka is a distributed streaming platform with the follow capabilities [4]:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system
- Store streams of records in a fault-tolerant durable way

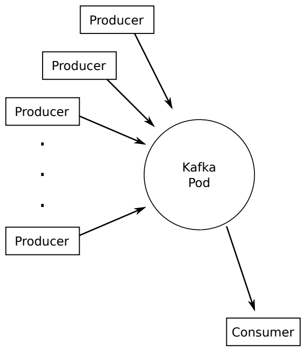

Figure 4. Scheme of the prototype: multiple producers write in the Kafka instance running on a Kubernetes' pod, a single consumer reads from it.

- Process streams of records as they occur.

Kafka allows multiple clients to read from the same queue. This feature is fundamental to share the workload between different machines.

3.1 Infrastructure and configuration

The prototype was developed using the Kafka infrastructure already in place in the Online environment. In details the Kafka instance was hosted on a Kubernetes' pod and Confluent-Kaka [5] was used to interact with it. This API allows very easy access to the queue for both pushing and reading the messages. In Kafka there are four core APIs, two of them are interesting for this project: producers, publishing messages into the queue and consumers, reading the messages from the queue. With this configuration in place, multiple producers were used to emulate the HLT nodes pushing the monitoring information inside the queue and a single consumer was used to collect the monitoring information, to merge them and to publish them. This configuration with multiple producers and a single consumer was chosen in order to test how many producers are needed to saturate a single consumer. For this reason the hierarchical structure of the farm used for the Run 2 HLT2 monitoring system was not implemented.

3.1.1 Requirements

In order to start with a reduced workload, the prototype was tested emulating a HLT1 task with each producer sending:

- 10KB/s to emulate ~ 1000 counters
- 100KB/10s to emulate ~ 100 1D histograms
- 10MB/60s to emulate multiple 1D and 2D histograms

The target for this test is to have around 400 producers, which is expected to be representative of the HLT1 case for Run3. Since this is a prototype, the messages' payload is synthetic. Due to the threshold Kafka puts on the message size, the 10MB message was split into 50 parts which were sent one after the other. Future implementations will deliver real histograms and counters instead of random generated ones.

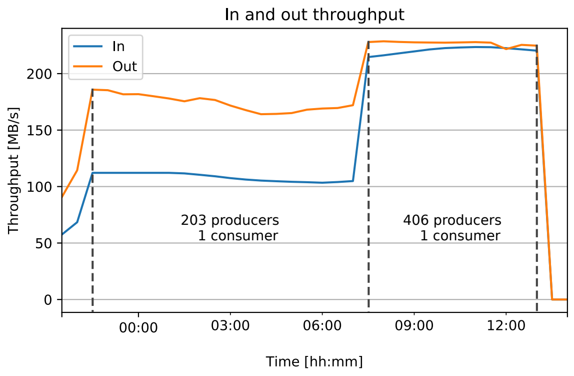

Figure 5. Kafka queue, throughput measurement: in blue the messages coming inside the queue, in orange the ones going out. Two configurations were tested and the difference is clearly visible at ~7:30 where the plot shows a "step". During the first part of the test, the prototype was running with 203 producers and a single consumer. In the second part, the number of producers was increased to 406 keeping only a single consumer.

3.1.2 Tests

The prototype was first tested on a single machine running consumer, producer and the Kafka instance. After testing and debugging the code, the entire test was moved on multiple machines connected to the same Kafka instance like shown in Figure 4. The very preliminary test in this configuration was to check the integration with the instance provided by LHCb Online performing a stress test with multiple producers. In Figure 5 it is possible to see that the prototype reacted very well running for several hours with 203 producers first and 406 after. In particular, a single consumer was able to handle a HLT1-like workload without any problems. This test showed that the prototype was stable during several hours of test. It also showed that, running on a 406 producers configuration, without an intermediate aggregation level, increases the stress on the network, with the output throughput hitting 220MB/s. Future implementations of the prototype will run with multiple consumers sharing the workload and a top-level consumer reading the consumers' output and merging it. This configuration will benefit from Kafka as well: there will be another Kafka instances collecting the first level of aggregation and a top level consumer will perform the same operations as the consumer used in this prototype. The last test performed was checking the resilience to failures. This operation was performed shutting down the consumer for five minutes: in this way it is possible to check how the entire system reacts in case the queue starts filling up. There are two major points to verify: if the Kafka instance can handle non-consumed messages for five minutes and, once the consumer is back, if the consumer can catch up the producers, increasing the output throughput. In Figure 6 is shown the failure test. The producer (blue line) constantly injects data during the entire test without being slowed down by the increasing queue. The consumer is shut down for five minutes after running on a stable configuration; once it comes back online it is able to catch the producer increasing its throughput. This is a clear sign of the system showing great response over a failure. Two additional features are visible from the plot: the periodical spikes are due to the 10MB/60s messages while the fact that the output throughput doesn't show a sharp changes when the consumer is turned on and off is because the measurements were averaged.

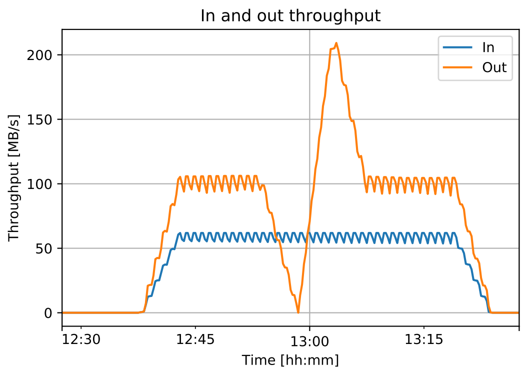

Figure 6. Kafka queue, throughput measurement: in blue the messages coming inside the queue, in orange the ones going out. From this plot it is visible that producers input messages constantly for the entire test while the consumer stops for 5 minutes dropping the output throughput. Once the consumer is back its throughput increases way above the normal level in order to catch-up the producers. Once the queue is emptied, the output throughput goes down to the same level as the one at the beginning of the test. Also in this case, the difference between the input and output throughput is due to protocol overhead and it is under investigation. During this test, 105 producers and a single consumer were used.

4 Conclusion

In this proceedings, the HLT1 and HLT2 monitoring systems running in Run 2 were presented as well as a prototype using a commercial software to handle the distribution of the monitoring information. HLT1 has a very well tested system generating "full histograms" and running on DIM. In HLT2 a different approach was chosen using ZeroMQ to allow communication between the HLT nodes. The HLT2 implementation is focused on lowering the network load splitting the messages into increments and metadata. This system performed very well during Run 2 but it strongly relied on a simulation software to tune the queue parameters. Following the same approach used for the HLT2 monitoring system, a prototype running on Apache Kafka was implemented. Kafka was the most reasonable choice because it was already used by LHCb Online and, since it is a commercial software, it is already very well optimized. Tests performed on the prototype showed that it is well integrated in the Kafka instance provided by the LHCb Online infrastructure and it can handle a HLT1-like workload with a single node reading all the monitoring information delivered by the HLT1 machines. It also showed effective recovery in response to a failure. This feature is fundamental because, in case of problems, it gives time to either restart the consumer or to create a new one. Future implementations will resemble the workload required by HLT2. In order to lower the input throughput on the consumer, a configuration with multi level aggregation will be implemented and tested.

References

[1] C. Gaspar and M. Dönszelmann, DIM a distributed information management system for the Delphi experiment at CERN
[2] O Callot et all, IOP Publishing **119**, 022015 (2008)
[3] H. Mohamed, Deploying a ”push” model Prometheus, CHEP 2018
[4] https://kafka.apache.org/intro
[5] https://github.com/confluentinc/confluent-kafka-python