AGH University of Science and Technology
Faculty of Physics and Applied Computer Science

Master's thesis

Patryk Lesiak
major: applied computer science

Development of the data quality assurance and visualization system for the Time Projection Chamber in ALICE experiment at the LHC

Supervisor: dr Jacek Otwinowski
The Henryk Niewodniczański Institute of Nuclear Physics,
Polish Academy of Sciences, Cracow

Cracow, October 2016

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

.....

(legible signature)

The subject of the master's thesis and the internship by Patryk Lesiak

student of 5th year major in applied computer science

The subject of the master's thesis: **Development of the data quality assurance and visualization system for the Time Projection Chamber in ALICE experiment at the LHC**

Supervisor: dr Jacek Otwinowski
Reviewer: dr inż. Grzegorz Gach
A place of the internship: The Henryk Niewodniczański Institute of Nuclear Physics, Polish Academy of Sciences, Cracow

Program of the master's thesis and the internship

1. Discussion with the supervisor on realization of the thesis.
2. Collecting and studying of the references connected with the thesis topic.
3. The internship:
   - Preparation of the environment for the software.
   - Incremented implementation of the developed system.
   - Collecting metrics of the developed system.
4. Drawing conclusions based on gathered data.
5. Analysis of the project, discussion and approval by the supervisor.
6. Editorial work on the thesis.

Dean's office delivery date: October 2016

…….. ……..

(Department Chair's signature) (Supervisor's signature)

Supervisor's review

Reviewer's review

Abstract

The following thesis describes architecture of a prototype of the data quality assurance and visualization system for the Time Projection Chamber in ALICE experiment at the LHC. System was created as a module of the new O^2 system. Developed prototype was used to perform tests of computing resources consumption and merging operation efficiency for tests with different parameters such as communication socket buffer capacity, size and type of Quality Control objects and number of producing and merging programs in a topology.

Performed tests shown, that buffer for connection between producing and merging programs has to have capacity of at least the same number of objects as producing programs in topology. The size of messages had no influence on online processing with smaller producing efficiency for larger messages. Data type tests shown, that class *TH1F, TH2F* and *TH3F* provide stabilization of consumed resources in contrast to classes *THnF* and *TTree.* Scalability tests proved that merger online processing is possible for up to 375 producing nodes for one merger. By increasing the number of merging nodes it is possible to improve processing capabilities for inefficient topology of one merger and 500 producers by dividing data stream for many merging nodes.

Streszczenie

Praca magisterska opisuje architekturę prototypu systemu do kontroli jakości i wizualizacji danych Komory Projekcji Czasowej eksperymentu ALICE na LHC. Oprogramowanie zostało stworzone w ramach aktualizacji systemu O^2. Rozwinięty prototyp posłużył do analizy wykorzystywanych zasobów obliczeniowych oraz wydajności procesu scalania danych dla testów o różnych parametrach takich jak rozmiar bufora kanału komunikacyjnego, rozmiar i typ obiektów przechowujących dane oraz ilość programów produkujących i scalających dane.

Przeprowadzone testy wykazały, że bufor dla połączenia pomiędzy programami produkującymi a węzłem scalającym musi być co najmniej równy ilości programów produkujących dane. Rozmiar wiadomości nie wpływał na obniżenie wydajności procesu scalania przy różnej sprawności programów produkujących dane. Testy typów obiektów pokazały, że klasy *TH1F, TH2F* i *TH3F* zapewniają stabilizację używanych zasobów obliczeniowych w przeciwieństwie do klas *THnF* oraz *TTree.* Skalowanie ilości programów produkujących dane odbywało się na korzyść wydajności procesu scalania aż do 375 programów produkujących dane. Testy ilości programów scalających dane wykazały, że nieefektywną topologię taką jak 500 programów produkujących oraz jeden program scalający można poprawić przez podzielenie strumienia danych na więcej programów scalających.

Contents

1 Introduction

1.1 ALICE experiment

ALICE (A Large Ion Collider Experiment) [1] is one of the detectors located at the Large Hadron Collider (LHC) which belongs to European Organization for Nuclear Research (CERN). It is focused on exploring properties of strongly interacting matter at extremely high temperature and density where the formation of a new phase of matter, the quark-gluon plasma, is expected. The central part of the ALICE detector consists of a few devices necessary for particles tracking and identification. One of them is the Time-Projection Chamber (TPC) [2] which is the main tracking detector of the whole system. It was constructed to provide measurements of charged particle momentum, vertex determination and identification of particles.

Operation of the TPC is based on the phenomenon of gas ionization. The chamber is filled with 90 m^3 gas mixture of neon, carbon dioxide and nitrogen in 90/10/5 proportions. Electrons from gas ionization made during collisions are carried in end plates direction where signal amplification process takes place. After this process obtained analog signals can be digitize by front-end electronics of TPC end send to online and offline systems for further processing.

Engineers and scientists from the ALICE Collaboration are constantly trying to enhance capabilities of experiment. During the second long shutdown of LHC between 2018 and 2019 the ALICE detector upgrade [3] is planned in order to allow future physics program to be fulfill.

During Run3 (the period of data collecting) between 2019 and 2023 physicists will focus on studying of heavy-flavour production, low-mass dileptons and production of quarkonia.

To achieve goals mentioned above the ALICE detector upgrade is needed in the following areas: tracking capability, enhancing low-momentum vertexing and enabling data collection with larger rate (up to 50 thousands events per second). One of the consequences will be increased amount of data that will require online and offline processing.

1.2 ALICE O^2 system

According to the Technical Design Report [4] future system should be able to handle processing of continuously incoming data from the TPC with rate reaching 1.1TB/s at 50 kHz interaction rate of Pb-Pb collisions. Current approach does not assume continuous read-out mode of detectors. Thus, different approach is needed to face the new challenges with increased rate of data flow. The ALICE Online-Offline computing system (O^2) [4] will be used in future for processing data collected from the detectors of the experiment.

The concept of upgrading current system consists of moving all data coming from the detector into the computing system. In order to save reconstructed events into a local storage four steps of data processing is needed.

Firstly input of the system organized as constant raw data streams will be transformed into manageable object using global reference clock enclosed to the stream. After calibration and local pattern recognition data will be stored in compressed objects. The second step consists of data aggregation and global reconstruction from all channels of detector measurements. After those steps data will be transported to efficient storage center. Up to this point every activity is performed synchronously with collecting of the data. The final step of processing is asynchronous to the data taking. It consists of further calibration, reconstruction and event extraction. Those events will be available on the ALICE Computer Grid for analysis as a final product of the O^2 system.

1.3 ALICE Quality Control System

The purpose of the Quality Control (QC) system is to provide online, fast feedback about quality of collected data as well as performance of the data calibration and reconstruction algorithms.

A work flow of the QC system begins with receiving data from different steps of data flow described in section 1.2. The next step is to perform quality assessment and storage of QC information. The results should be computed automatically and consist of objects like histograms or values together with meta data describing data quality assessments. Another component of the QC system, Event Display, will allow to manipulate, visualize and analyze reconstructed events.

The key components of QC system are shown in figure 1.

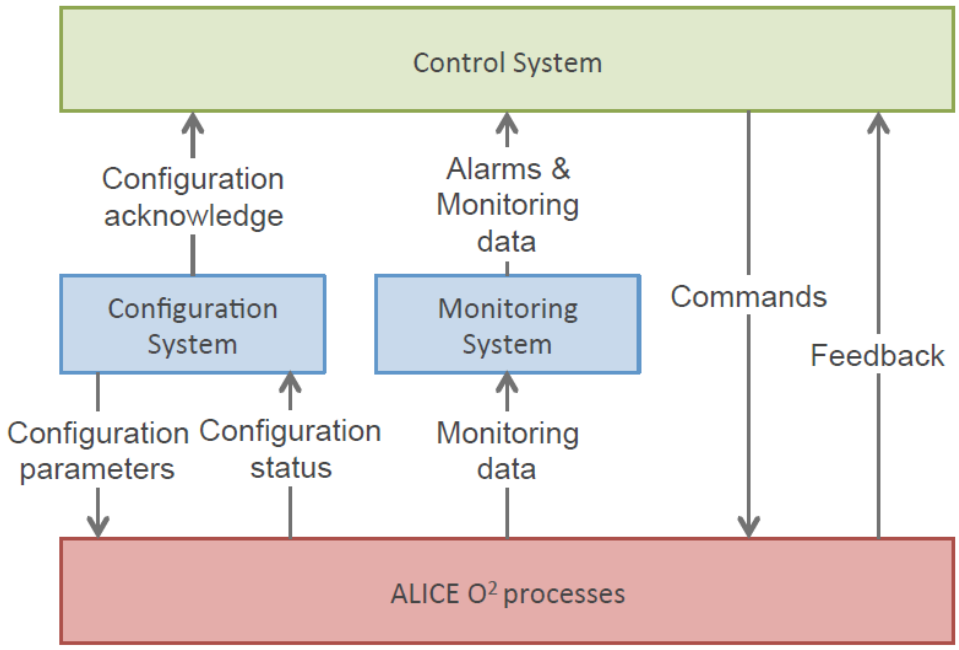

Figure 1: Relationship between the Control, Configuration and Monitoring (CCM) components of the O^2 system, taken from [4].

1.4 The scope of the thesis

The aim of the thesis was to develop a prototype of QC system which consists of the QC data extraction, merging performance examination and results visualization. Implemented system is included in the ALICE O^2 software and will be used in the further system development.

The prototype allows to carry out research on the consumption of resources and performance of the system for various configuration parameters. Those parameters are amount of incoming data, the size and type of data, the size of nodes messaging buffers and number of processing nodes in deployed system.

After results evaluation the most suitable configuration can be chosen for further development of O2 computing system.

The following aspects were covered in this thesis:

- Implementation of the merging prototype
- Implementation of the control component
- Execution of the benchmark tests
- Results evaluation

2 Used tools

2.1 Programming language

C++ programming language was chosen because of its fast execution time and compatibility with frameworks that have been developed by the ALICE Collaboration and GSI/IT group.
To follow the modern C++ programming approach some features of the 11 and 14 versions were applied like *chrono* library, smart pointers or lambda functions.
Some of the modules of popular set of libraries Boost [5] were utilized. The following libraries were used: *Boost.PropertyTree*, *Boost.Date_Time*, *Boost.Algorithm*, *Boost.Program_options* and *Boost.Test*.

2.2 FairRoot framework

FairRoot framework [6] is required for installation of Alice O^2 software because some modules depends on FairRoot implementation of reconstruction, simulation and analysis. The framework provides two necessary modules for developing merging prototype named ROOT [7] and FairMQ.

2.2.1 ROOT

FairRoot is based on the ROOT framework [7] which provides functionality to handle big data processing, visualization, storage and statistical analysis. Already implemented data structures can be utilized for representing and processing of the data in O^2 system. *TH1* class and its subclasses like *TH2*, *TH3* and *THn* are specialized to store data in histogram form. There are also classes implemented for data aggregation such as *TTree* which can hold all kinds of data, such as objects or arrays in addition to the simple types. In order to send objects over transport layer there is a need to serialize them into byte array and deserialize at the receiver side. ROOT framework provides such functionality with *TMessage* class.

Every class within described framework extends base class with name *TObject*. Thus, every class can be visualized using *Draw* function which declaration is in root class. Such functionality helped to create simple *EventDisplay* component described in section 1.3.

2.2.2 FairMQ

Its purpose is to provide possibility of running tasks in different system processes and arrange communication between them. Each independent process can be implemented as a state machine called *device*. Those devices can be connected to each other with one of framework transport layer implementations which is based on efficient messaging library ZeroMQ [8]. This type of communication provides asynchronous messages queues. Together with communication patterns like push-pull, request-response or publish-subscribe it gives possibility to form processing tasks in topologies specific to concrete problem.

2.3 Dynamic Deployment System

More than 10000 programs are needed to run the target environment of the O^2 system. For such purpose Dynamic Deployment System (DDS) [9] was implemented. The system facilities deployment of systems consisting of many independent programs by providing a concepts of topology file. Inside this configuration file is located description of the implemented system in the form of XML. Each of the executable described in the topology deployed by DDS agents is started by DDS commander server, a central part of the system.
Common global variables can be introduced inside the file in order to share access to defined value between all of the programs within the same topology. A target environment can be a local system or grid solution with batch systems as PBS or SLURM.

Example of DDS topology file with system composed of two different programs:

```
<topology id="QA">
    <decltask id="ProducerTaskId">
        <exe reachable="false">runProducer parameter1 parameter2</exe>
    </decltask>

    <decltask id="MergerTaskId">
        <exe reachable="false">runMergerDevice</exe>
    </decltask>

    <main id="main">
        <task>ProducerTaskId</task>
        <task>MergerTaskId</task>
    </main>
</topology>
```

Another important feature is DDS intercom API footsore as C++ library. Its purpose is to provide messages exchange between the separate processes running in one topology. Each program can broadcast a message in the character string form. Program which is designed to receive such messages implements subscription as a message handler:

```
CCustomCmd ddsCustomCmd;

ddsCustomCmd.subscribe([&](const string& command, const string& condition,
      uint64_t senderId)
      {
            // message handling
      });

ddsCustomCmd.subscribeOnError([&](const EErrorCode _errorCode,
      const string& _errorMsg)
      {
            // error message handling
      });
```

DDS intercom API is considered by the ALICE collaborators as a good approach for implementing separate communication channel for exchanging control messages within the QC system.

2.4 Elasticsearch

The QC system produces a large amount of control metrics. There is a need to aggregate measurements and visualize them in a human convenient form.

Nowadays there are many search and analytic engines available on the market. Elasticsearch [10] is one of the solutions which provides a scalable, distributed and real-time processing. It is based on full-text search engine library *Apache Lucene*.
Elasticsearch provides possibility to store text documents with Javascript Object Notation (json) format. Every field of the json can be indexed by the engine to allow efficient search of selected data. In order to create an index which will be used to distinguish collection of documents it is needed to use RESTful API provided by Elasticsearch server.

An example of creating new index for set of metrics can be found below:

```
curl -XPUT http://localhost:9200/newMetric -d '
{
 "mappings" : {
  "_default_" : {
   "properties" : {
    "command" : {"type": "string", "index" : "not_analyzed" },
    "node_id" : {"type": "string", "index" : "not_analyzed" },
    "average_merge_time" : { "type" : "double" },
    "VmRSS" : { "type" : "integer" },
    "request_timestamp" : { "type": "date" },
    "response_timestamp" : { "type": "date" },
    "cpu_clock" : { "type" : "integer" },
    "merged_objects_per_second" : {"type" : "double"}
   }
  }
 }}';
```

Once data is imported it is possible to visualize results as graphs by using Kibana platform which was designed to work with Elasticsearch. Another modules allows to calculate and visualize standard deviation of examined set of data.

3 Description of developed system

3.1 System architecture

Prototype of the QC system consists of four types of executable programs. Each of them represents the different stages of processing data. Running programs perform a single type of task like producing, merging, visualizing or monitoring. Thereby there is a need to provide communication to combine them into one coherent system.

For asynchronous communication purposes FairMQ framework described in section 2.2.2 was used. *SendAsync and ReceiveAsync* functions of *FairMQChannel* class provides possibility to exchange messages between the separated programs with push-pull approach. Developer can decide what to do in the case when buffer is overloaded. In developed prototype in such case send or receive operations are repeated until success. This approach prevents loss of the data.

QC objects which are exchanged between nodes of the system are serialized and saved in objects of *TMessage* class. In order to send messages with various types every object is first cast to *TObject* class which is a root class of all types implemented in the ROOT framework. With this approach number supported types may increase in the future.

Sequence diagram of processing data by the prototype can be found in figure 2.

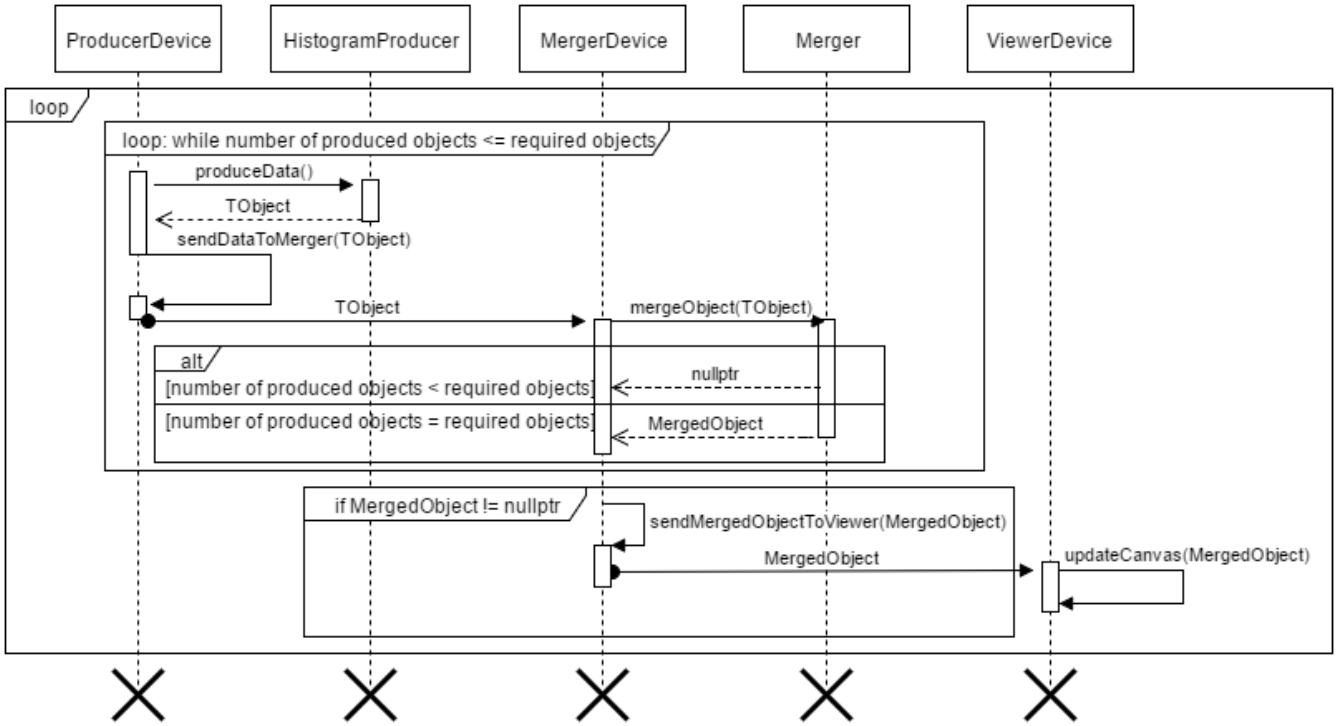

Figure 2: Sequence diagram of processing data by developed QC prototype.

3.1.1 ProducerDevice

This component is responsible for producing QC data. Those objects represent sampled data from all levels of processing data in the ALICE O^2 system.
Each object is filled with the random values from the Gaussian distribution. ROOT framework has been used to exploit already implemented classes for data storage. The size of the objects depends on amount of data and internal implementation of data storage format.
Producers nodes are supposed to constantly provide QC objects to the system. To allow that, buffers of these nodes have extremely high capacity to prevent from buffer overload occurrence.
Each producer node creates one part of complete object. To restore complete information there is a need to merge data from all producers from one cycle of data production.
Every object of *ProducerDevice* class was implemented rules for handling *check-state* DDS custom command in order to provide information about operation status of the node.
Production of the following types of QC objects was implemented:

1) *TH1* – one dimensional histogram
2) *TH2* – two dimensional histogram
3) *TH3* – three dimensional histogram
4) *THN* – n-dimensional histogram
5) *TTree* – tree data structure where data can be saved in branches

3.1.2 MergerDevice

Every producer in the DDS topology sends data objects to *MergerDevice* component where the process of data reduction by merging is held.
At the beginning of the processing received objects are stored in *TList* collection. This container stores pointers to the QC objects. Objects with the same name are stored in one instance of *TList* which is an entry in the map container.
As it was described in 3.1.1 section objects from one production cycle should be merged. In order to simplify implementation of merger component and to avoid problem with handling non complete information from producers the decision was made to perform merge operation when the same number of messages as producers is stored by merger, not taking into account from which cycle it comes. This may cause situation, when objects from different production cycles are merged together. However, this approach allows the monitoring of the system in which none of producers stops sending data.
Merging operation itself is implemented in the classes of the ROOT framework. Every class which allows this operation like *TH1F* or *TTree* has a *Merge* function. The only one argument of the function is a list of objects which will be merged together with object invoking *Merge* function. After merging operation is done collected objects are released from program memory.

3.1.3 ViewerDevice

Like previously described components *ViewerDevice* is implemented as a FairMQ state machine. Merged portions of data are constantly send to the *ViewerDevice* which visualize them using ROOT functionality. Objects with the same name of data are shown in one canvas.
The main benefit of visualizing objects is ensuring human observer that merging process was successful and that new objects are constantly received by *ViewerDevice*.

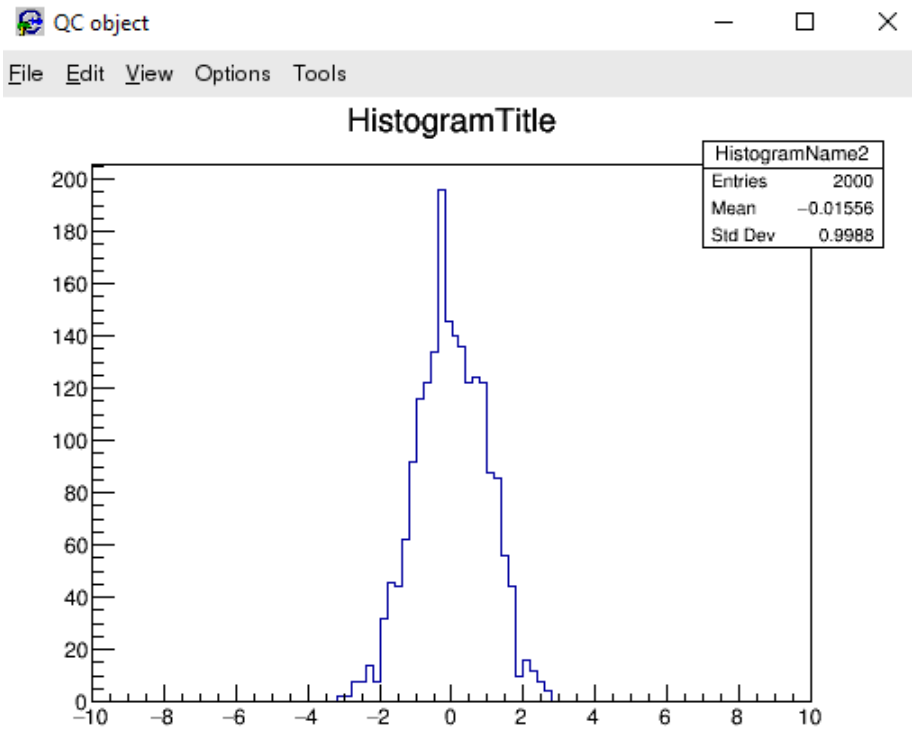

Figure 3: Visualized Quality Control object using the ROOT framework.

3.1.4 MetricsExtractor

The functionality of this program is to gather information about current topology of the running system and performance of the merging process. DDS intercom API is used to provide communication channels separated from Quality Control objects transport. That solution prevents from ending the connection during regular transport layer failure provided by FairMQ.
Every information is transported as characters string with json format.
There are two types of DDS custom command implemented in the prototype:

1) check-state: used to retrieve information about state of every node running in topology
2) get-metrics: used to get metric from *MergerDevice* component

Each command is described in details in section 3.2.2.

3.2 Data flow

Devices are connected to each other with TCP transport layer, creating expected data flow of the system. There are two types of messages exchanged within running programs: QC objects and control messages.

Figure 4 shows an example of the QC system topology and its data flow. Connections between FairMQ devices (*ProducerDevice, MergerDevice* and ViewerDevice) are made expressing exact address and port of desired endpoint e.g. tcp://localhost:5005. Such channel can carry serialized messages within processes. This type of communication ensures the flow of QC messages within the QC prototype with push-pull approach. This strategy allows to arrange communication as a pipeline in which messages are carried only in downstream direction. Blue components in figure 4 represents connections and messages flow between FairMQ devices.

Control messages are carried between MetricsExtractor component and FairMQ devices. Communication is provided by DDS intercom API. Every message consists of serialized char string formatted as json data. An example of carried metrics can be found in section 3.1.4.

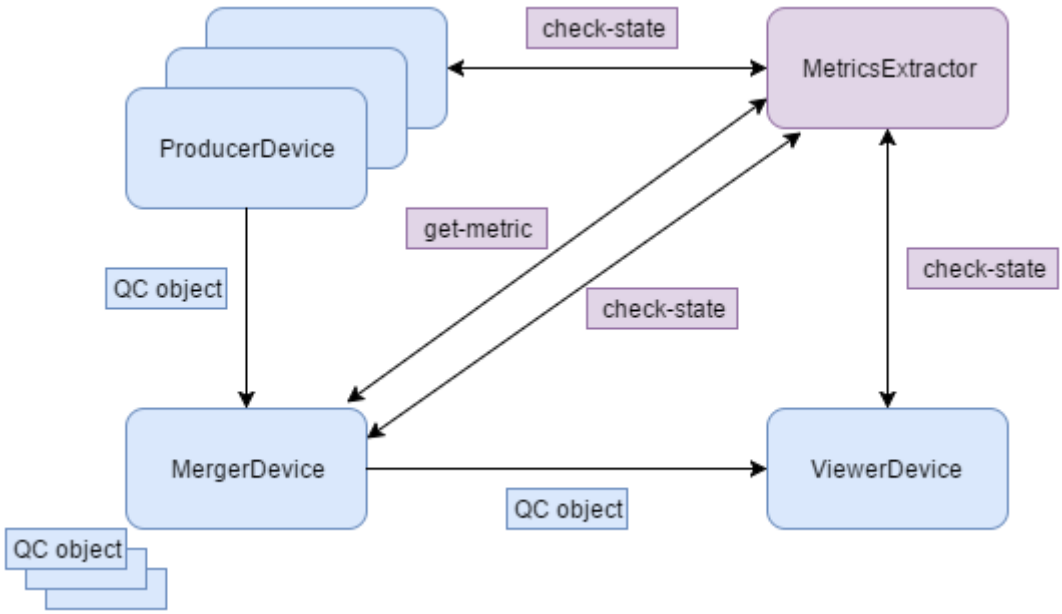

Figure 4: Data flow within developed QC system.

3.2.1 QC Objects

These are objects representing the QC data in the O^2 system. Serialized to byte array objects are sent using *TMessage* class of the ROOT framework.
At the beginning of the transport process messages are placed in the buffer of FairMQ channel. During *AsyncSend* operation messages are immediately pushed into further transport layers or are suspended in case of buffer overload. In such case operation is repeated until success.
Push-pull strategy protects from delays caused by time needed to wait until confirmation message reaches sending component.

3.2.2 Control messages

Control messages provides information about state of the components of the QC system and performance of the merging operation. Different types of DDS custom commands are distinct with *command* field of json message.
First of the command with name *check-state* is used to determine if every component of the topology works correctly. Especially to check if buffers are not overloaded and if every component has *RUNNING* state during test performance.
The second command id used to retrieve hardware consumption metric from *MergerDevice* component.
The collected data were used for result elaboration in the form of graphs and calculation of mean value with standard deviation of selected characteristics of hardware resource consumption.

Example of check-state result:

```
{
	"command":"check-state",
	"node_id":"CentralMerger",
	"node_state":"RUNNING",
	"request_timestamp":"2016-08-28T00:17:08",
	"response_timestamp":"2016-08-28T00:17:08",
	"receive_buffer_size":"1000000",
	"receive_buffer_overloaded":"false",
	"send_buffer_size":"1000000",
	"send_buffer_overloaded":"false"
}
```

Example of get-metrics result:


```
{
	"command":"get-metrics",
	"node_id":"CentralMerger",
	"PID":"14756",
	"request_timestamp":"2016-08-28T00:17:08",
	"response_timestamp":"2016-08-28T00:17:08",
	"average_merge_time":"0",
	"VmRSS":"33032",
	"cpu_clock":"28871",
	"merged_objects_per_second":"0"
}
```


In order to gather metadata about metrics the following fields were introduced in metrics presented above:

- command: the name of requested metric, used by DDS intercom API
- node_id: internal id of FairMQ device
- node_state: state of FairMQ device
- timestamps: time of receiving and sending metric

Timestamps fields facilities moment in time and gives knowledge if there are some delays on DDS intercom API channels. Further details on metrics can be found in chapter 4.

3.3 Software tests

3.3.1 Unit tests

To monitor software development the unit tests of source code should be implemented. The unit tests of the QC system prototype were applied using Boost Test Library.

Basic unit tests were implemented to validate FairMQ devices objects creation. The second part of unit tests consist of validation of system functionality like merging process which is crucial part of the system functionality. *MergerTestSuite* collects tests of merging given number of objects.

Test case *mergeTenHistograms* checks, if objects are correctly merged by *mergeObject* function. Histograms are passed by using a collection of already created histogram objects. Assertion *BOOST_TEST* checks, if function returned valid merged object when there were enough objects in merger component to initialize merging operation or if function returned null pointer otherwise.

3.3.2 Valgrind

Programming languages like C++ which do not have garbage collector mechanism are exposed to memory leaks. In C++ this situation occurs when allocated dynamic memory is not released before program end. Memory leak can lead to worse performance because of reduced amount of available RAM and eventually to crash of the system.
Developed prototype was checked by valgrind tool to assure that there are no memory leaks caused by the QC prototype system.

Despite the fact that there is no memory leak caused by the QC prototype there are two minor memory leak in ROOT framework used for tests executions. Valgrind output can be found below.

```
41 bytes in 1 blocks are definitely lost in loss record 2,897 of 7,845
   at 0x402871D: operator new[](unsigned long)
   by 0xD5BF42B: TCling::TypeName(char const*) (TCling.cxx:4368)
   by 0x5205484: TDataMember::Init(bool) (TDataMember.cxx:199)
   by 0x520524A: TDataMember::TDataMember(DataMemberInfo_t*, TClass*) (TDataMember.cxx:182)
   by 0x5230ABB: TListOfDataMembers::Get(DataMemberInfo_t*, bool)
       (TListOfDataMembers.cxx:314)
   by 0x523118B: TListOfDataMembers::Load() (TListOfDataMembers.cxx:481)
   by 0x5245580: TClass::GetListOfDataMembers(bool) (TClass.cxx:3552)
   by 0x5243FA4: TClass::GetDataMember(char const*) const (TClass.cxx:3225)
   by 0x523985E: TBuildRealData::Inspect(TClass*, char const*, char const*, void const*,
       bool) (TClass.cxx:734)
   by 0xD5B761B: TCling::InspectMembers(TMemberInspector&, void const*, TClass const*, bool)
       (TCling.cxx:2275)
   by 0x5240A95: TClass::CallShowMembers(void const*, TMemberInspector&, bool) const
       (TClass.cxx:2131)
   by 0x5240005: TClass::BuildRealData(void*, bool) (TClass.cxx:1986)
```

```
5,400 bytes in 25 blocks are definitely lost in loss record 7,347 of 7,845
   at 0x4028078: operator new(unsigned long)
   by 0x513B8B9: TStorage::ObjectAlloc(unsigned long) (TStorage.cxx:324)
   by 0x508ABD2: TObject::operator new(unsigned long) (TObject.h:158)
   by 0xD5B303A: TCling::LoadPCM(TString, char const**, void (*)()) const (TCling.cxx:1282)
   by 0xD5B4F64: TCling::RegisterModule(char const*, char const**, char const**,
        char const*, char const*, void (*)(),
        std::vector<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>,
        std::allocator<char> >, int>,
        std::allocator<std::pair<std::__cxx11::basic_string<char, std::char_traits<char>,
        std::allocator<char> >,
        int> > > const&, char const**) (TCling.cxx:1706)
   by 0x5087FEA: TROOT::InitInterpreter() (TROOT.cxx:1785)
   by 0x508199F: ROOT::Internal::GetROOT2() (TROOT.cxx:362)
   by 0x50819CA: ROOT::GetROOT() (TROOT.cxx:376)
   by 0x5242E5A: TClass::GetClass(char const*, bool, bool) (TClass.cxx:2878)
   by 0x524B0E0: TClass::Load(TBuffer&) (TClass.cxx:5339)
   by 0x621B84D: TBufferFile::ReadClass(TClass const*, unsigned int*) (TBufferFile.cxx:2682)
   by 0x662CC2B: TMessage::TMessage(void*, int) (Tmessage.cxx:95)
```

The memory leaks cause increasing memory consumption of approximately 1 MB per 250 seconds of test performance. For five minutes tests it does not have an impact on results presented in this thesis. Nevertheless it will cause allocated memory overload during a long system performance. The erorrs were raported to ROOT developers and are expected to be gone in the future releases of new versions.

3.4 Execution environment

PL-Grid infrastructure was used to provide target deployment environment for the merging prototype. System was executed on the newest cluster "Prometheus". It provides the following hardware for single computing node:
Processors: 2 x Intel Xeon E5-2680v3
Number of cores: 24
CPU clock: 2,5 GHz
RAM: 128 GB

For such environment there is possibility to deploy up to 48 DDS agetns so as not to cause an overhead on performance of individual processes.

4 Benchmark

To fulfill the goal of this thesis there is a need to test and compare merging prototype performance with different configurations. Based on the experience from previous periods of data collecting by ALICE some realistic scenarios was selected to be taken under consideration.
It is expected that benchmark results will help to make decisions about the type and size of the QC objects, merging algorithms, topology of the system and hardware characteristics of the processing nodes. Another important aspect is to investigate the scalability of the whole system.

4.1 Parameters

Five parameters were defined to investigate the influence on resource consumption and performance of the system. Between each execution of the test scenario only one parameter was changed to enable examination of its impact. The following parameters were taken under consideration:

1) Data type – the QC values can be stored in different representation types. ROOT framework provides implementation of the most common representations of data collected by a particle detector. To examine consequences of choosing specific type of data five types of classes from ROOT framework were examined in terms of resources consumption and merging performance. Classes taken under consideration were described in section 3.1.1.

2) Data size – considering one portion of data there is a choice to transport it encapsulated in one message or to send more messages with divided data. By sending different amount of data in one message there is possibility to investigate buffer overloading and impact on the time of merging process. It was decided to examine the following amounts of data for each message from producer component: 1kB, 500kB, 5M and 50MB.

3) Number of input nodes – every producer node is sending about one hundred messages per second. By increasing number of input nodes for one processing unit there is possibility to investigate its performance under bigger pressure of data. To check scalability of the system research covers 2, 5, 10, 50, 100, 250, 375, 425 and 500 input nodes scenarios.

4) Size of the buffers – FairMQ framework introduced buffers for messaging channels. Buffer capacity is expressed by a number of messages that can be stored. Buffer can be set for communication socket which performs connect operation of ZeroMQ functionality. After buffer reaches the limit of messages data transport over channel is suspended. Availability of the buffer between producers and merger were studied. In developed QC prototype objects stored in the buffer are of type *TMessage.* The size of these objects are almost the same as for examined types described in 3.1.1 section. The following values of buffer capacity were taken into consideration: 1, 4, 20, 50, 100, 1000.

5) Number of merging nodes and their topology – the traffic from all producers of the running system can be organized in different topology. Depending on the number of merging nodes there is a possibility to balance the load of each processing unit. This is especially important when some node of the system is overwhelmed by the input of too many producing components. In order to reduce the traffic there is a need to deploy additional merging unit into the system. Because DDS system does not provide yet a possibility to dynamic topology reconfiguration only static connections were executed during tests.

4.2 Metrics

To have the opportunity to compare the system performance in different test scenarios described in chapter 4 there is a need to collect control data. Those metrics will focus on the basic resources which are consumed during the time of system execution. Examination of those metrics will allow to find hardware and process limitations.
The following metrics were gathered during test execution.

4.2.1 Central Processing Unit (CPU) usage

After doing research the conclusion was made that there is no straightforward way to measure current CPU usage on Linux operating system in Hz. Most of the monitoring and snapshotting Linux tools like *top* or *ps* use *proc* pseudo-file system described in section 4.2.2. However, measurements provided with those tools are based on system time which is different than processor time, so there is no 1 to 1 mapping between those two values.

The decision was made to use an approximation of CPU clock time provided by C language library *time.* The function *clock* returns approximation of a processor tick count related to the current process. This is the clock time elapsed since implementation-dependent reference time point.

Despite the fact that this number will not reflect the actual CPU usage it is still a good measure of the CPU usage trend in time.
The maximum value for utilization of one CPU core given by *clock* function is 10^6. This will always be true, regardless of the actual CPU clock speed. Higher values indicates that more than one core was involved in a program execution.
To measure approximated number of processor ticks which were used for the current process performance over time there is a need to compare two values obtained during different time of measurement. An instance of *MergerDevice* class stores the measurement of CPU time when receiving the request of metric from *ControlDevice* object. Doing the next measurement the new value has to be compared with the previously stored result.

In order to present the result in human friendly way the result was converted to tics per one second (Hz) using this formula:

$$Hz = \left(CPU_{previous} - CPU_{current}\right) * \frac{1}{T_{elapsed}} \quad (1)$$

where:

Hz – approximated CPU usage in Hz
$CPU_{previous}$ – previous measurement of CPU time
$CPU_{current}$ – current measurement of CPU time
$T_{elapsed}$ – elapsed time between measurements in seconds

4.2.2 Random Access Memory usage

Random Access Memory (RAM) usage was determined by using a Linux pseudo-file system which provides information about resources consumed by a specific process and the whole system. The file system is located in root directory of the system in *proc* folder. Each process has a separate sub folder with files which represents kernel data structures with always updated values. The most comprehensive memory measurements are present in *status* file. Row with virtual memory resident set size (VmRSS) string represents Resident Set Size which is the current amount of RAM used by a process.

4.2.3 Average merging time

On-line systems are exposed to incorrect operation caused by, among others, delays during data processing. The prototype developed for this thesis performs the merging operation on input objects. It is expected that this operation should be done in real time, without a need of queuing received objects to wait for its turn to be processed.
Therefore it is important to estimate the time needed by the merging process of the QC data. This is a decisive factor that decides whether specific hardware can handle expected influx of data or topology of running components needs to be adapted to the data flow.
To measure elapsed time *high_resolution_clock* class from C++ 11 *chrono* library was used.

It was decided to examine average merging time of the last 10 merge operations for time of metric request.

4.2.4 Merged objects per second

This is the number of processed objects between two metric measurements. This process includes receiving and sending the data from the TCP socket, QC objects deserialization and serialization. When MetricsExtractor sends request *get-metrics* to *MergerDevice* it returns value of sent objects counter multiplied by time factor. This factor is used to express the value in processed objects per one second. After that operation the counter value is reset.
Formula for time factor is as follows:

$$T_f = \frac{1\,s}{T_{elapsed}} \quad (2)$$

where:
$T_{f\_}$ time factor
$T_{elapsed\_}$ elapsed time since last metric request

5 Results

The obtained results on the hardware resources consumption and the performance of the merging operation are reported in this chapter. The metrics are described in detail in the section 4.2.

Each test lasted 300 seconds. Metrics were gathered with 1 second time interval.

Gathered results of hardware consumption resources as CPU and VmRSS usage are biased due to different initial conditions of the computing nodes and external system processes which were executed during tests performance. Therefore, the results of CPU and memory consumption should be considered with precision of 10 percent. This upper limit was established on the basis of observations of tests executions with the same parameters at different times.

Resource consumption has a direct impact on the system performance. For this reason, the results on the average merging time and the number of merged objects should be treated with the same precision of 10 percent.

To avoid such behavior it is necessary to carry out the tests on well isolated grid solution having a full control of processes running at the same time. Unfortunately it was impossible to create such conditions on public access grid solutions as PL-Grid infrastructure.

5.1 Buffer size tests

The following tests covers the hardware resources consumption and the system performance with different buffer capacities between producers and one merger. Four producers where producing 1 kB *TH1F* objects with rate of 100 QC objects per second for each producer.

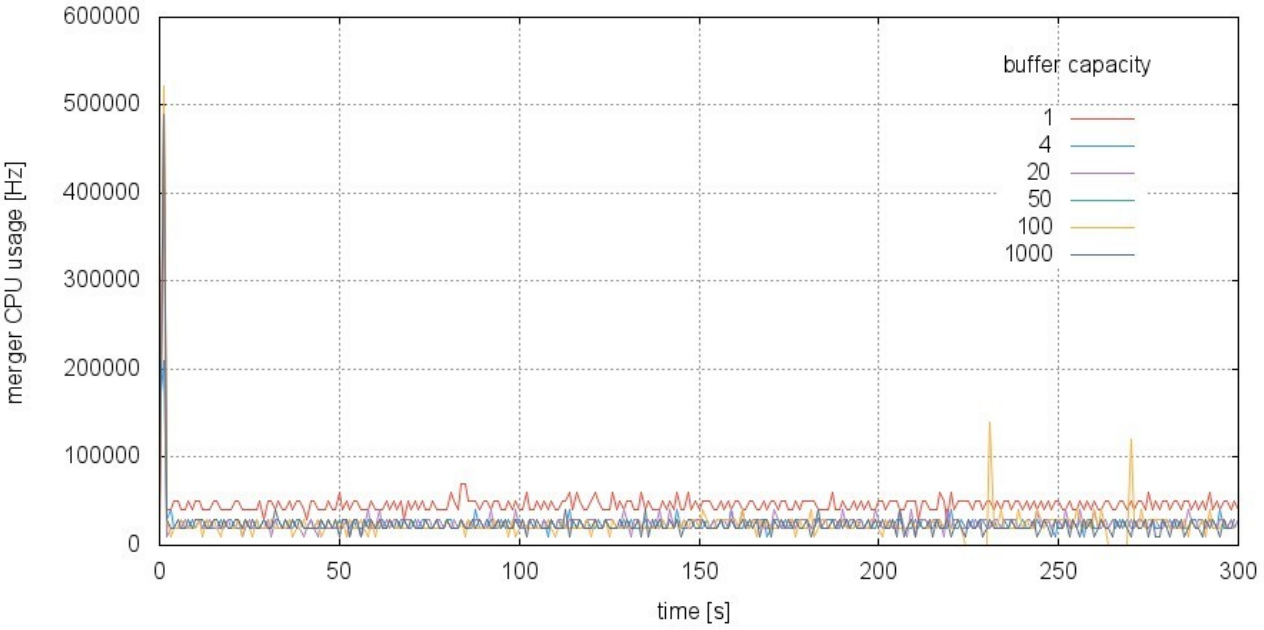

Figure 5: Approximated merger CPU usage.

Table 1: Average value and standard deviation of approximated merger CPU usage.

Buffer size	Average value[Hz]	Standard deviation [Hz]
1	46868	27028
4	25351	14200
20	26656	24894
50	25295	27614
100	24596	31119
1000	24643	27735

Figure 5 shows the merger CPU usage over time. After FairMQ devices initialization during the first second the merger CPU usage for all considered buffer sizes was stabilized. Minor fluctuations in the CPU usage occurred for buffer of 100 objects starting for times around 220 second. For 2 periods of 7 seconds none of the objects were merged. Because the system was not saturated it could handle processing of more objects during next cycles of the program execution. Such behavior may be caused by operating system task scheduling. For buffer size of 1 QC object, CPU usage was nearly two times higher compering to the other results (table 1). Internal reports of buffer overload indicates that processor had to do more operations to handle buffer unloading.

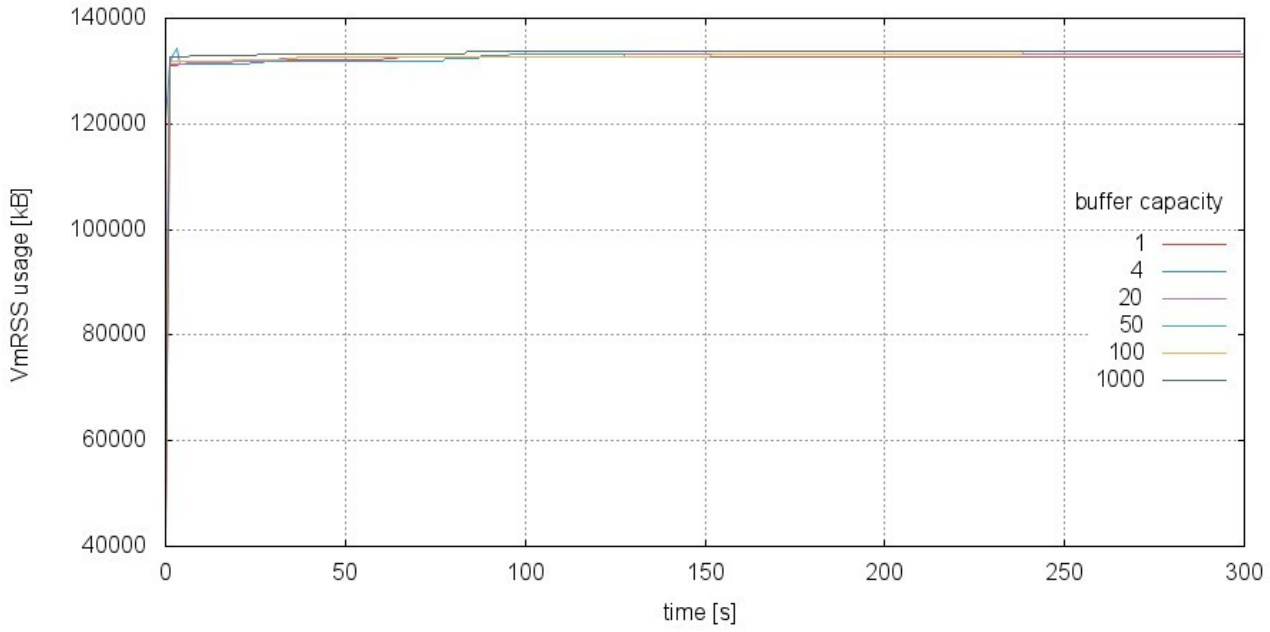

Figure 6: Merger VmRSS usage.

Table 2: Average value and standard deviation of merger VmRSS usage.

Buffer size	Average value [kB]	Standard deviation [kB]
1	132221	5235
4	132730	2956
20	132675	3151
50	134403	5063
100	132691	5114
1000	133221	5034

Differences between the average values of the memory consumption for different buffer sizes shown in figure 6 and table 2 indicates lack of influence of examined capacities on the *MergerDevice* memory consumption. Because it could process the whole producers output online there was no need to use buffer for storage of the QC data. Due to ROOT framework memory leaks described in section 3.3.2 consumption was stabilized only for periods of about 10 seconds. During the entire tests execution it had a growing trend of about 3 kB/s.

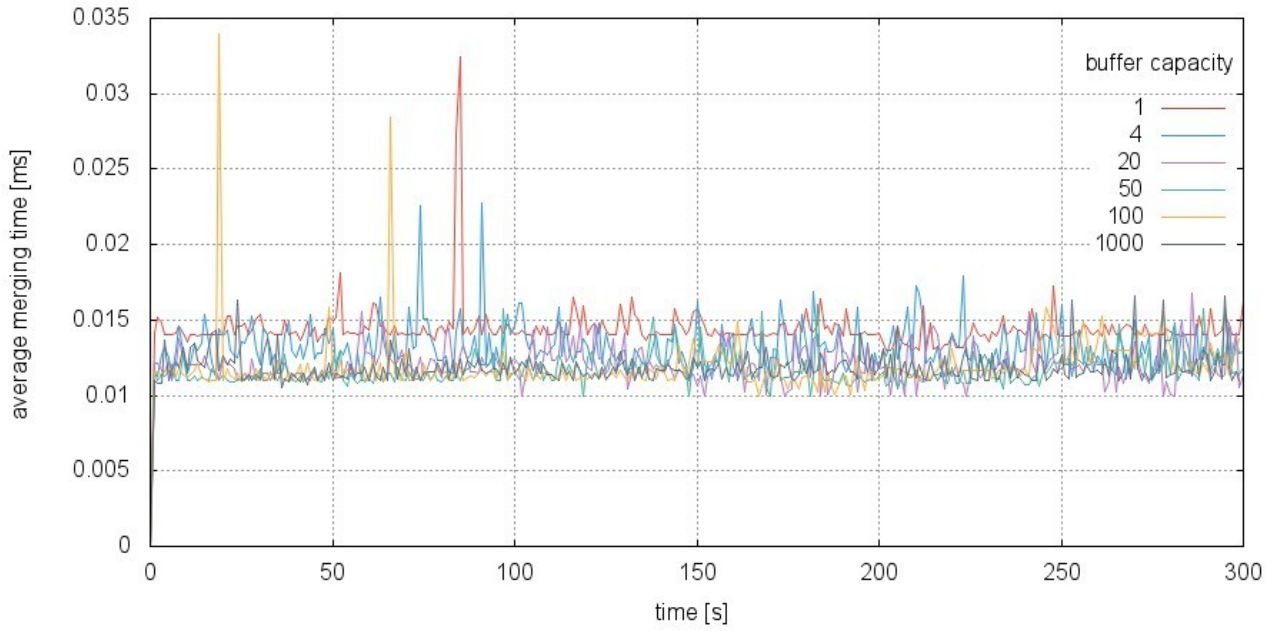

Figure 7: Average merging time.

Table 3: Average value and standard deviation of average merging time.

Buffer size	Average value [ms]	Standard deviation [ms]
1	0.014	0.0017
4	0.013	0.0016
20	0.012	0.0013
50	0.012	0.0012
100	0.012	0.0027
1000	0.012	0.0011

The highest value of average merging time shown in figure 7 and table 3 occurred for buffer of 1 QC object size. It confirms once again, that additional operations were performed to deal with the overloaded socket buffer.

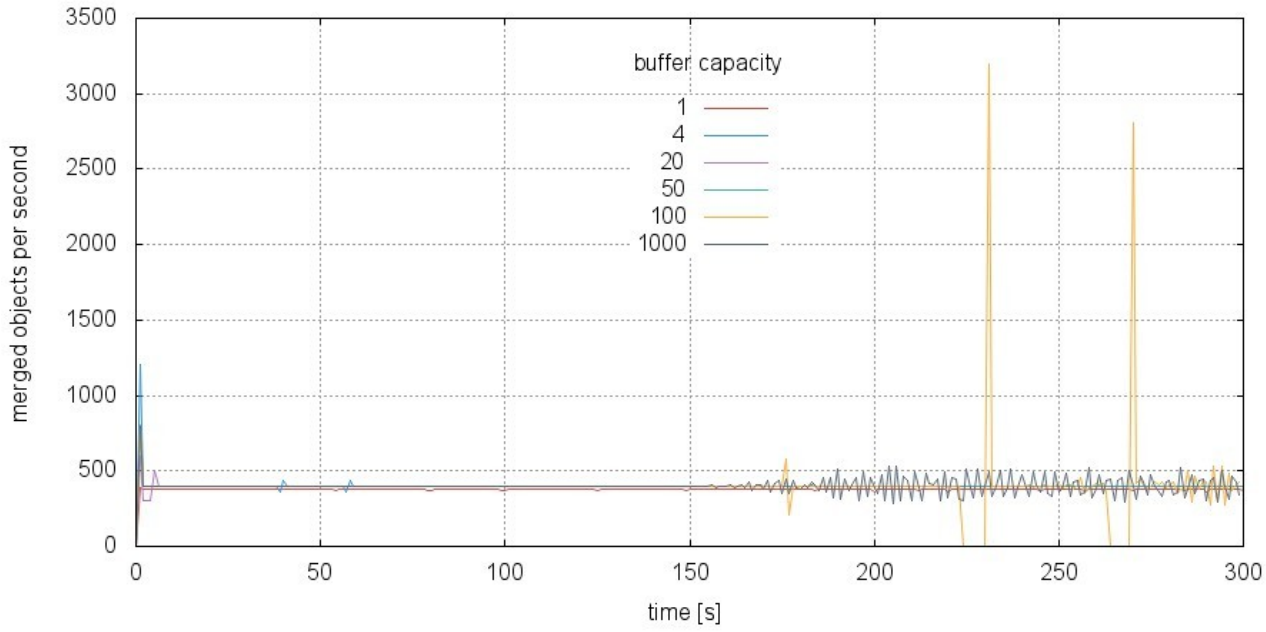

Figure 8: Number of merged objects per second.

Table 4: Average value and standard deviation of number of merged objects per second.

Buffer size	Average value	Standard deviation
1	377	22
4	398	51
20	399	28
50	400	33
100	400	233
1000	400	55

The results from figure 8 and table 4 confirms that for buffer of size 1 less objects were merged due to the socket buffer overloading. By increasing capacity of the buffer to four messages (the same number as producers), processing of the QC data was close to online proceeding.
Beside the fact, that for buffers of capacity more than 100 QC objects some fluctuation appears, the average value was nearly the same for buffer size larger than 20 messages.

5.2 Data size

The second type of tests were focused on a resource consumption influence of the QC objects size. Topology of the system consisted of two producers sending objects to one merger. Capacity of the socket buffer was one thousand messages. Because the time of creating histograms grows with the number of bins it was impossible to keep the same number of produced objects per second for every examined object size. Production of 1 kB and 500 kB objects was limited to twenty new QC objects per second. Objects of size 5 MB and 50 MB were produced at maximal level of producers efficiency.

To create objects with a various sizes the following numbers of the histogram bins were used:

- 1 kB *TH1F*: 100 bins, 1000 entries, Gaussian distribution
- 500 kB *TH1F*: 130800 bins, 1000 entries, Gaussian distribution
- 5 MB *TH1F*: 1310000 bins, 1000 entries, Gaussian distribution
- 50 MB *TH1F*: 13110000 bins, 1000 entries, Gaussian distribution

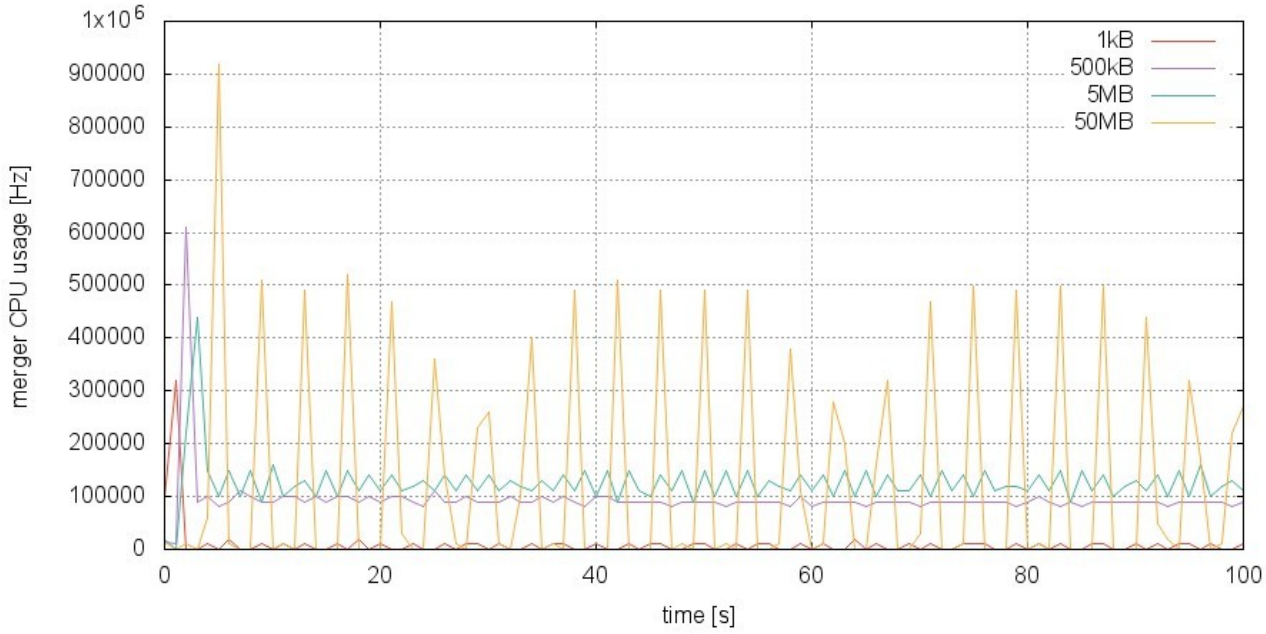

Figure 9: Approximated merger CPU usage.

Table 5: Average value and standard deviation of approximated merger CPU usage.

Object size	Average value [Hz]	Standard deviation [Hz]	Producer efficiency
1 kB	6265	19205	20 objects / s
500 kB	91209	30395	20 objects / s
5 MB	117047	30010	2.5 objects / s
50 MB	122281	188541	1 object / 4 s

The CPU usage increases with the size of the QC object as shown in figure 9. The results in table 5 show that merging operation for 500 kB (the most expected objects in the final QC system) objects was utilizing almost fifteen times more clock tics than for 1 kB objects. For 50 MB objects fluctuations which can be observed in figure 9 were present during the entire test. The reason was waiting for the QC objects by merger while none of the objects was processed.

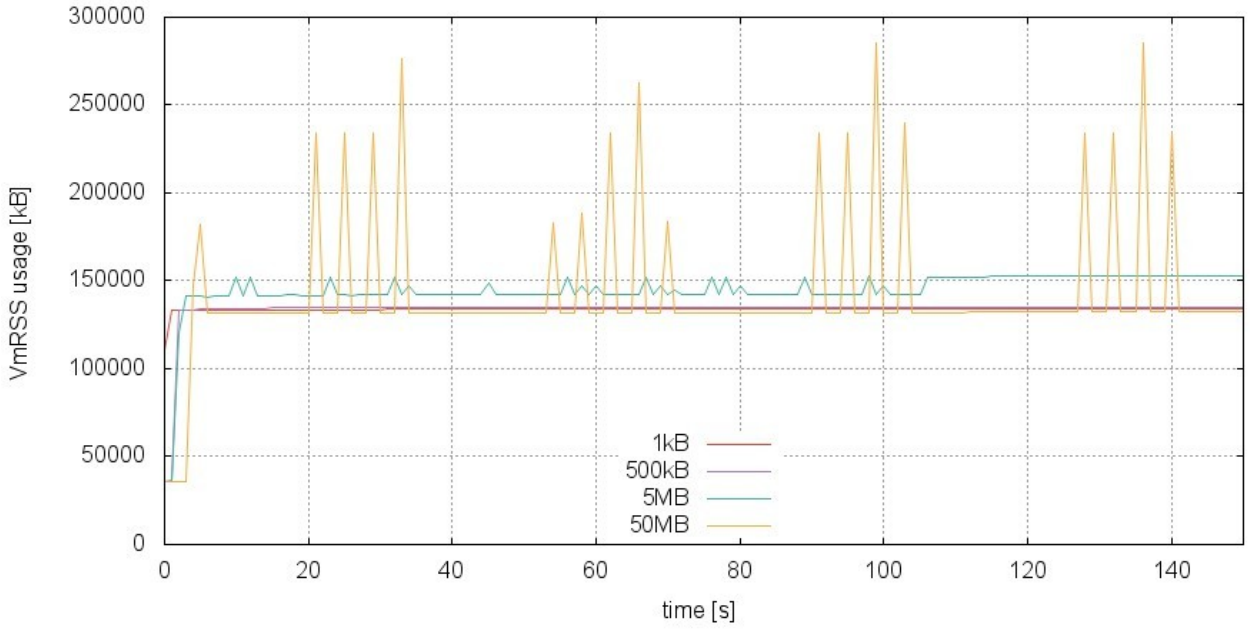

Figure 10: Merger VmRSS usage.

Table 6: Average value and standard deviation of merger VmRSS usage.

Object size	Average value [kB]	Standard deviation [kB]	Producer efficiency
1 kB	134314	1420	20 objects / s
500 kB	134777	7872	20 objects / s
5 MB	151405	11528	2.5 objects / s
50 MB	144080	36913	1 object / 4 s

The Virtual memory Resident Set Size usage for 1 kB and 500 kB QC objects was holding at that same level of 134 MB as shown in figure 10 and table 6. For 5 MB and 50 MB QC objects memory has been freed several times during the test (figure 10). It indicates that the heap of the merger program has been revamped after merging process.

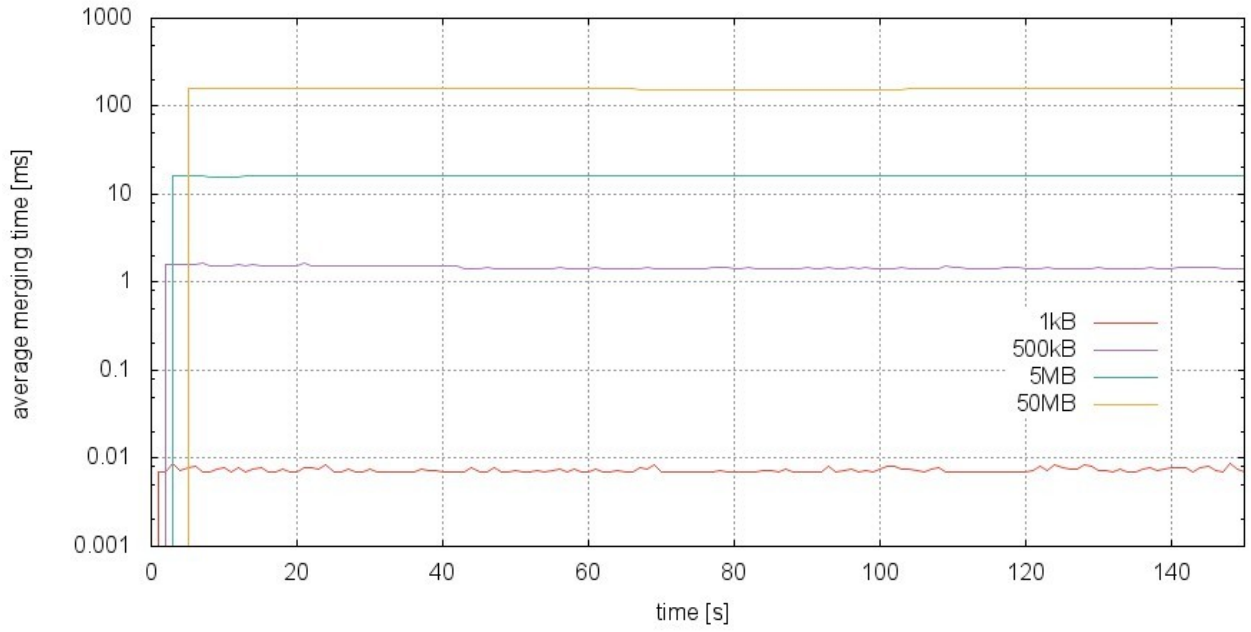

Figure 11: Average merging time.

Table 7: Average value and standard deviation of average merging time.

Object size	Average value [ms]	Standard deviation [ms]	Producer efficiency
1 kB	0.007	0.0006	20 objects / s
500 kB	1.478	0.1349	20 objects / s
5 MB	16.062	1.5767	2.5 objects / s
50 MB	156.059	19.7621	1 object / 4 s

The results presented in figure 11 and table 7 indicates that the average merging time of two QC objects grows proportionally to the size of the QC objects for 500 kB, 5 MB and 50 MB object sizes. Exception to this rule occurs for the 1kB data. Compering to the 500 kB objects the time of merge is over 200 times smaller.

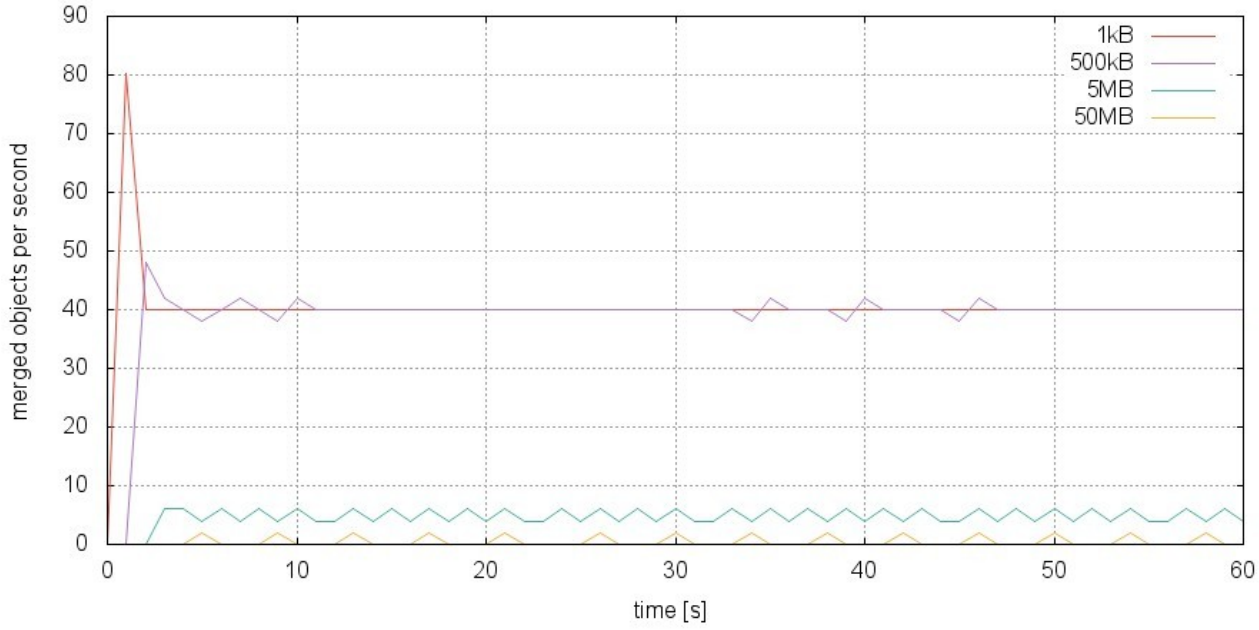

Figure 12: Number of merged objects per second.

Table 8: Average value and standard deviation of number of merged objects per second.

Object size	Average value	Standard deviation	Producer efficiency
1 kB	40	3	20 objects / s
500 kB	40	3	20 objects / s
5 MB	5	1	2.5 objects / s
50 MB	0.5	0.86	1 object / 4 s

The QC objects of all examined sizes were processed online as shown in figure 12 and table 8. The average values shows that fluctuations for these sizes of objects had no impact on the final results which was expected.

5.2.1 Various producers number with 50MB objects

Because it was impossible to send the new QC objects with the same rate for each size the decision was made to examine the influence of increasing number of producers of 50 MB *TH1F* objects. Considered system had the same parameters for the previous tests (section 5.2). The only difference is in number of producing components. For complete information it was required to merge only two objects every time, regardless of the number of producers in the system topology.

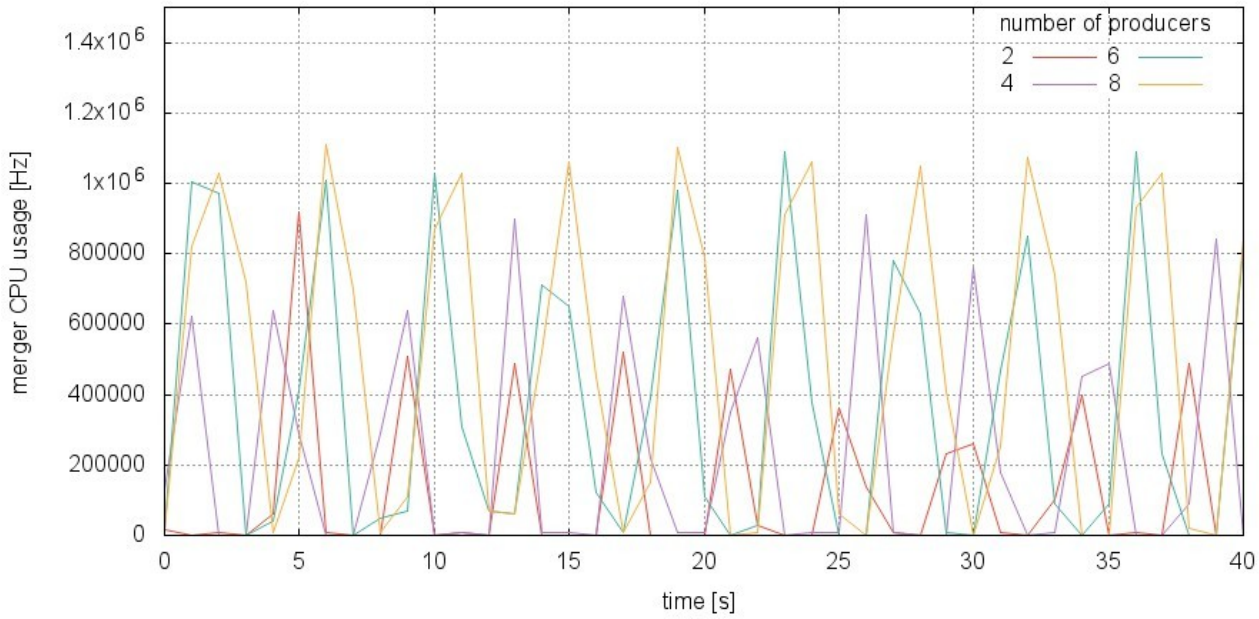

Figure 13: Approximated merger CPU usage.

Table 9: Average value and standard deviation of approximated merger CPU usage.

Producers number	Average value [Hz]	Standard deviation [Hz]	Merger input
2	121391	190227	1 object / 4 s
4	228624	301885	1 object / 2 s
6	335939	383198	1 object / 0,75 s
8	470588	418600	2 object / s

Results in figure 13 and table 9 indicates that the merger CPU average usage increases with the number of producers in the running topology. Characteristic peaks are present for each number of producers, moreover for 6 and 8 producers it reaches the maximal utilization of the computing core. During the times when CPU usage was almost equal to zero merger component was waiting for objects from the producers. Internal FairMQ logs confirmed that objects were not received for every second of execution of the merging instance. It indicates that transport layer is not efficient for such big objects.

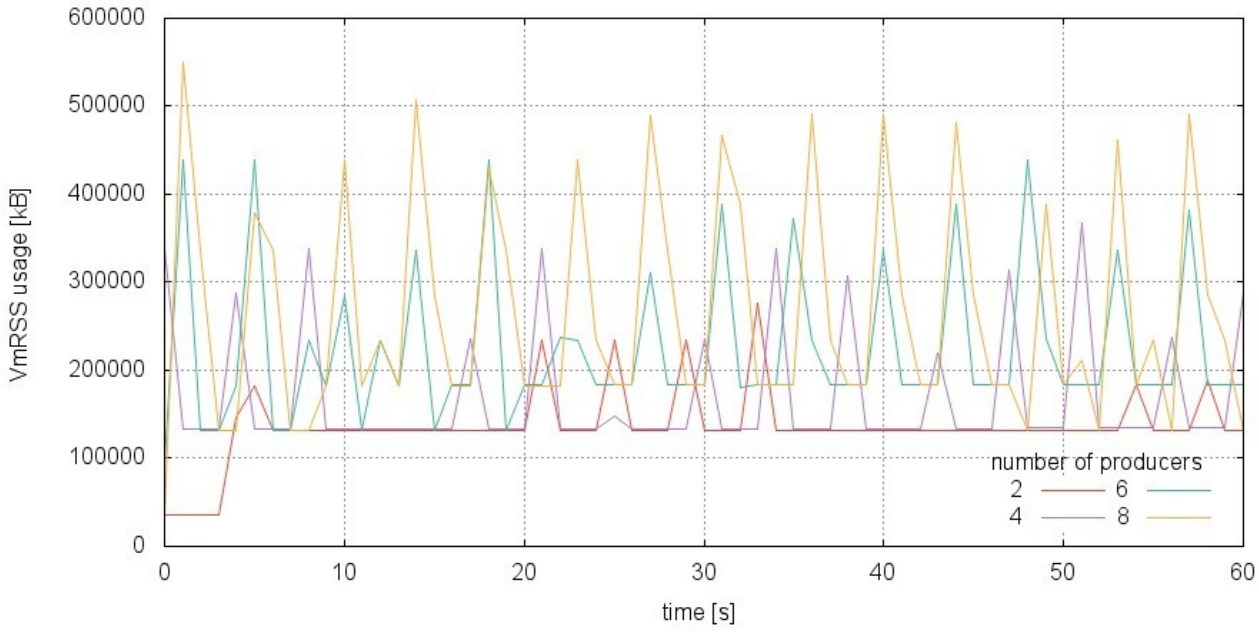

Figure 14: Merger VmRSS usage.

Table 10: Average value and standard deviation of merger VmRSS usage.

Producers number	Average value [kB]	Standard deviation [kB]	Merger input
2	143241	36382	1 object / 4 s
4	165092	62345	1 object / 2 s
6	218132	69115	1 object / 0,75 s
8	239883	112629	2 object / s

Figure 14 and table 10 show that the more objects were received by the merger the more memory was used for the data processing. The standard deviation (table 10) value also grows with the number of producers. It indicates that it is necessary to allocate more memory for processing larger amount of data.

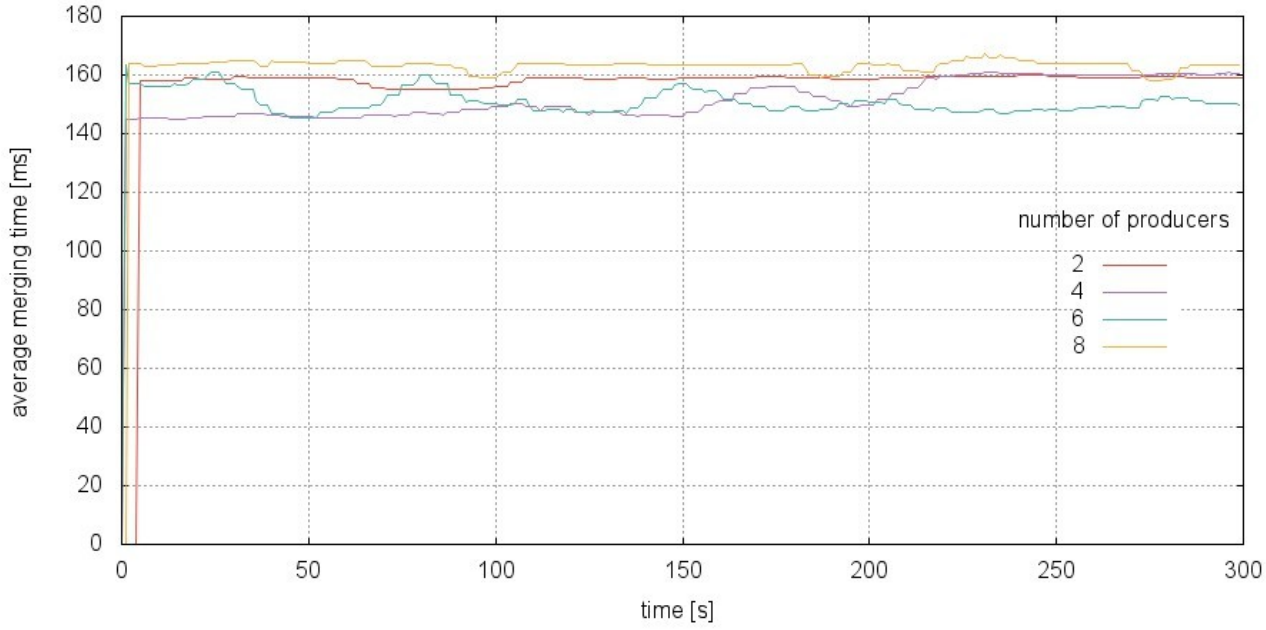

Figure 15: Average merging time.

Table 11: Average value and standard deviation of average merging time.

Producers number	Average value [ms]	Standard deviation [ms]	Merger input
2	155.88	20.33	1 object / 4 s
4	151.25	10.60	1 object / 2 s
6	150.36	9.45	1 object / 0,75 s
8	162.19	13.41	2 object / s

Assuming a 10% precision of the results interpretation the average merging time did not change for various numbers of producers as shown in figure 15 and table 11. It is expected result because exactly two objects were merged for each case.

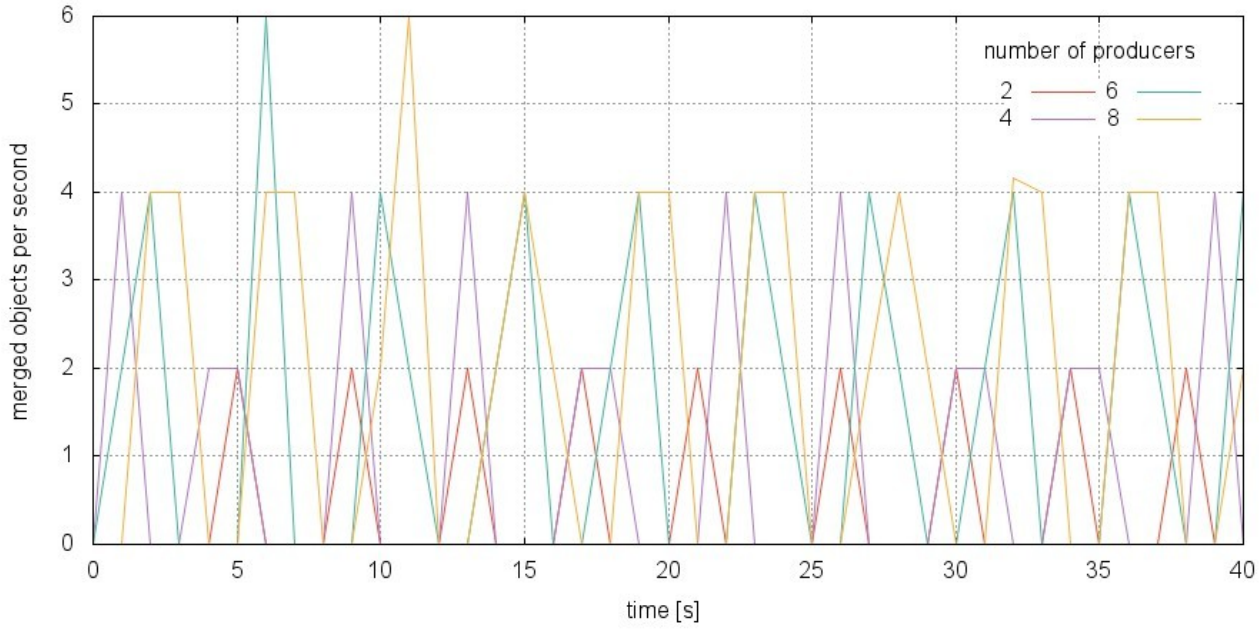

Figure 16: Number of merged objects per second.

Table 12: Average value and standard deviation of number of merged objects per second.

Producers number	Average value	Standard deviation	Merger input
2	0.48	0.85	1 object / 4 s
4	0.95	1.42	1 object / 2 s
6	1.39	1.71	1 object / 0,75 s
8	1.87	1.88	2 object / s

Increasing the number of producers resulted in the larger number of objects which could be merged by *MergerDevice* component. Result values in figure 16 and table 12 show that for 6 and 8 producers merger component could process the data online. The peaks visible in figure 16 indicates that merging operation was not done in each measurement cycle. Instead, this operation was performed later for the larger number of objects (even 6). The cycles with 0 merged object could be utilized for deserialization and serialization of a QC object or for transport purposes.

5.3 Data type

The fourth type of tests was executed with 20 producers which were sending 20 QC objects per second to one merger. Expected input of the *MergerDevice* component was 400 objects per second. The size of an object was always 500 kB. The socket buffer capacity was set to one hundred thousands messages.

Logs of FairMQ *MergerDevice* component shows that for *THnF* and *TTree* objects merger input was smaller than expected. Despite the fact that all of producers were sending 20 new objects per second for each type of the QC data, logs of the merger FairMQ device indicated that its input was at lower level.

To obtain 500 kB objects of every object the following parameters were applied:

- TH1F: 130800 bins, 1000 entries, Gaussian distribution
- TH2F: 360×360 bins, 1000 entries, Gaussian distribution
- TH3F: 49×49×49 bins, 1000 entries, Gaussian distribution
- THnF: 17×17×17×17 bins, 1000 entries, Gaussian distribution
- TTree: 11 branches each with 1000 entries, Gaussian distribution

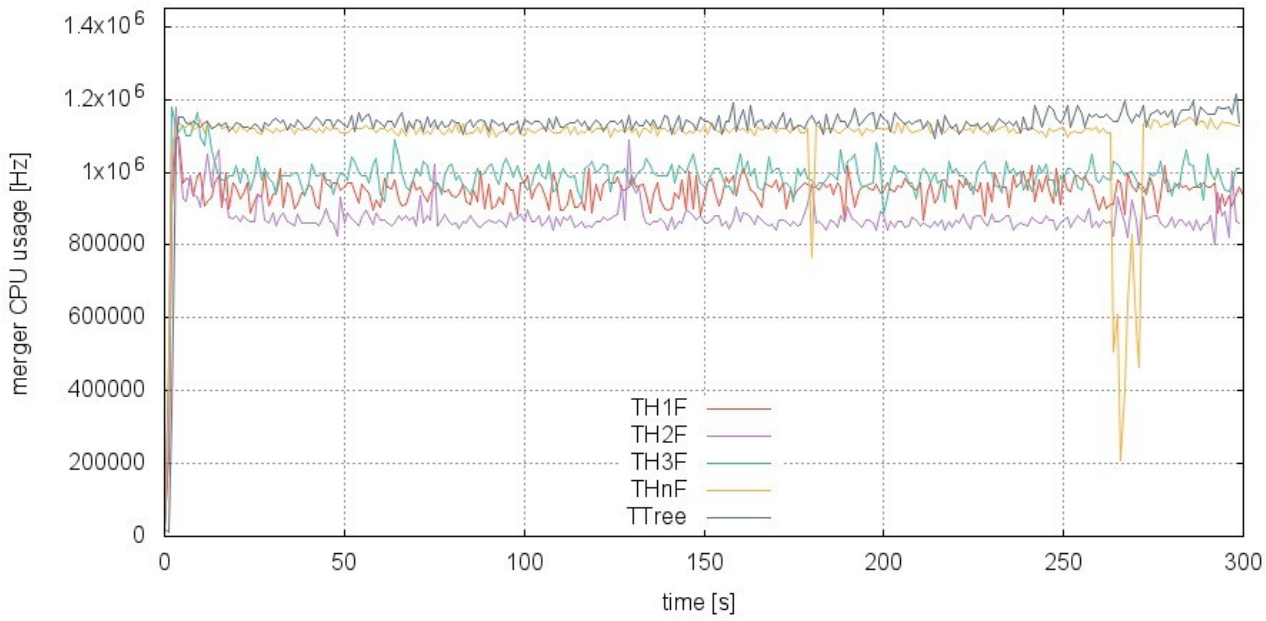

Figure 17: Approximated merger CPU usage.

Table 13: Average value and standard deviation of approximated merger CPU usage.

Producers number	Average value [Hz]	Standard deviation [Hz]	Merger input
TH1F	944330	75643	400 objects / s
TH2F	871817	73544	400 objects / s
TH3F	989825	74378	400 objects / s
THnF	1095021	125530	170 objects / s
TTree	1128613	104134	96 objects / s

Assuming a 10 % precision on the CPU results interpretation the average values for *TH1F*, *TH2F* and *TH3F* are at that same level as shown in figure 17 and table 13. Number of utilized processors tics are visibly higher for *THnF* and *TTree* objects. Values that extends beyond 10^6 Hz indicates utilization of more than one computing core.

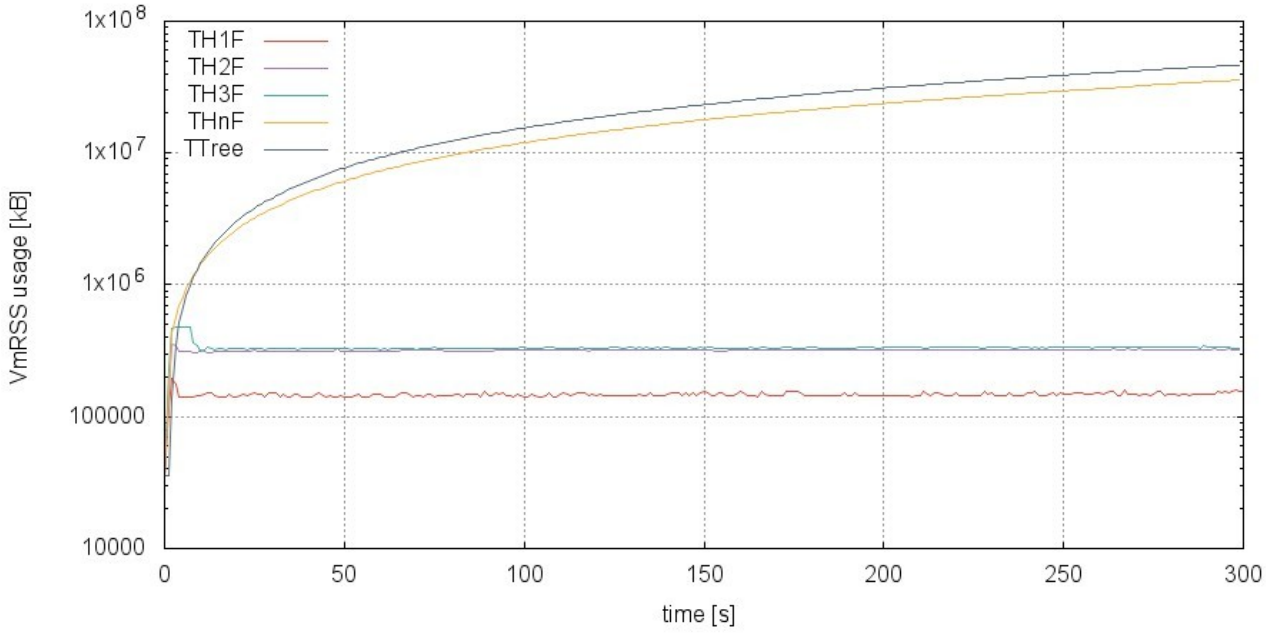

Figure 18: Merger VmRSS usage.

Table 14: Average value and standard deviation of merger VmRSS usage.

Producers number	Average value [kB]	Standard deviation [kB]	Merger input
TH1F	146544	8538	400 objects / s
TH2F	316677	18973	400 objects / s
TH3F	333981	27682	400 objects / s
THnF	17859282	10249448	170 objects / s
TTree	23266926	13564220	96 objects / s

Figure 18 shows that the memory consumption for processing *THnF* and *TTree* objects was

constantly increasing. It is due to the ROOT framework internal implementation of *Merge* function of these objects. The QC data objects are constantly being copied for each call of this function. The results of *TH1F*, *TH2F* and *TH3F* types show good performance with stabilization of consumed resources on the level shown in table 14.

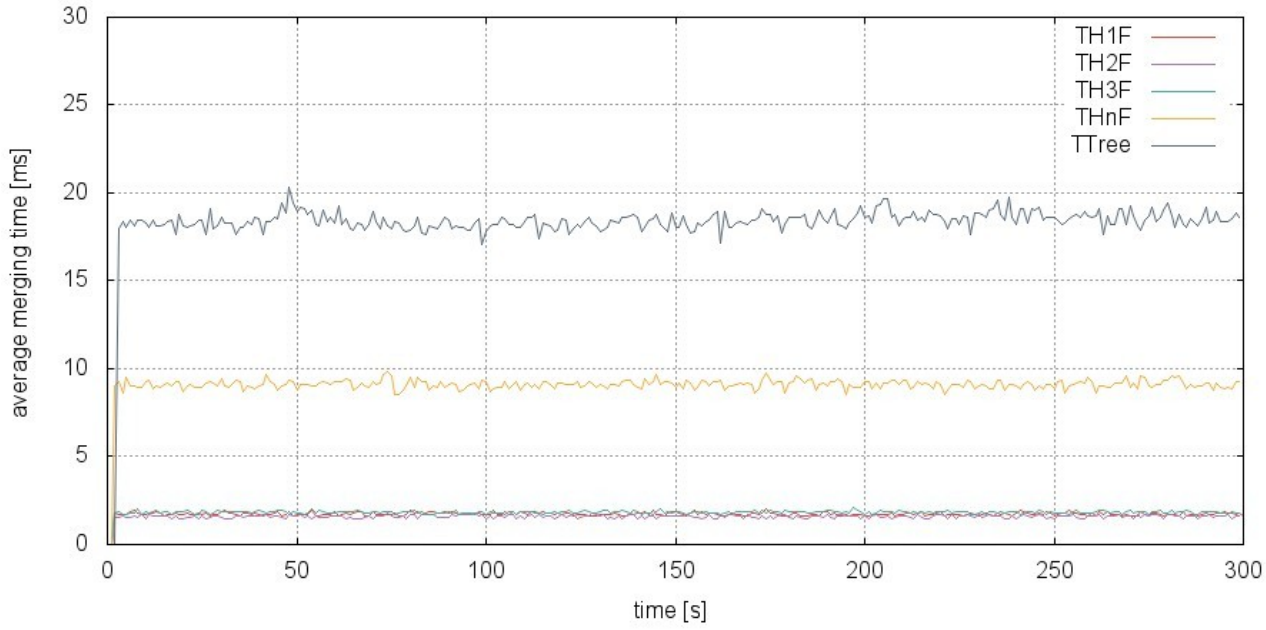

Figure 19: Average merging time.

Table 15: Average value and standard deviation of average merging time.

Producers number	Average value [ms]	Standard deviation [ms]	Merger input
TH1F	1.731	0.161	400 objects / s
TH2F	1.588	0.151	400 objects / s
TH3F	1.827	0.165	400 objects / s
THnF	9.049	0.775	170 objects / s
TTree	18.256	1.891	96 objects / s

The results shown in figure 19 and table 15 indicate that the time needed to merge four objects is the same for *TH1F*, *TH2F* and *TH3F* object types. The time of merging *THnF* and *TTree* object is higher but remains at the same level during the whole test.

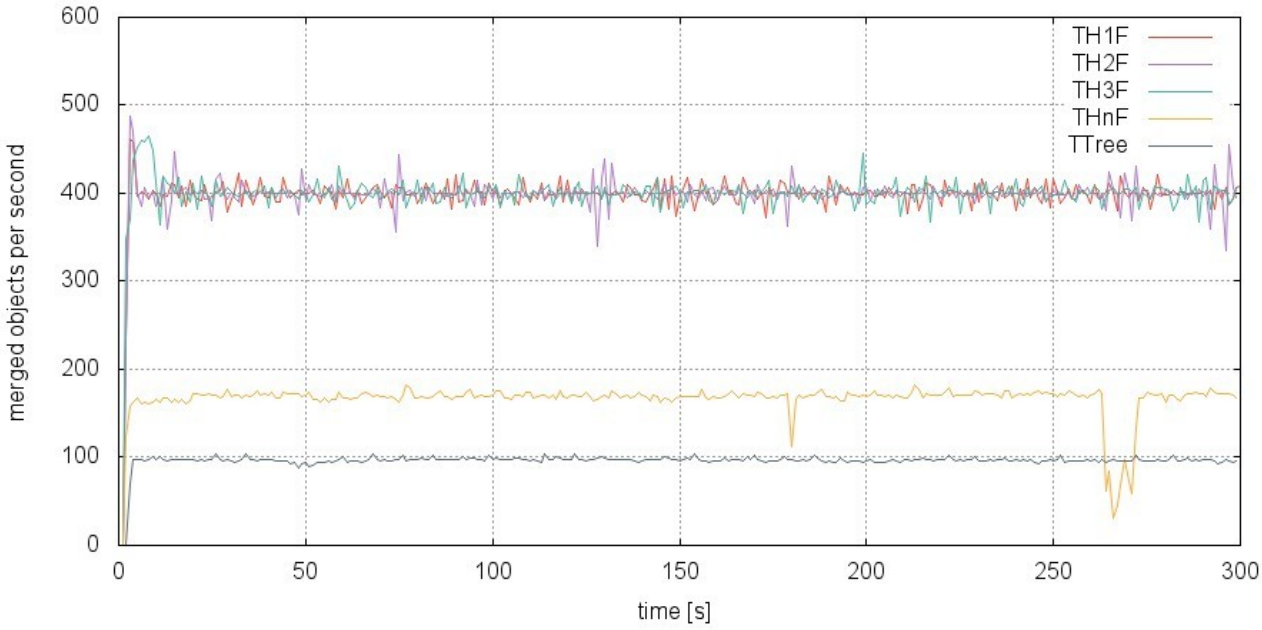

Figure 20: Number of merged objects per second.

Table 16: Average value and standard deviation of number of merged objects per second.

Data type	Average value	Standard deviation	Merger input
TH1F	397	35	400 objects / s
TH2F	397	37	400 objects / s
TH3F	398	35	400 objects / s
THnF	166	23	170 objects / s
TTree	96	10	96 objects / s

The results shown in figure 20 and table 16 indicate that objects of *TH1F*, *TH2F* and *TH3F* were processed online with the maximal merger input of 400 objects per second. Objects of type *TTree* were also merged online but with smaller merger input. For *THnF* type there are observable drops around 180 and 220 times. Nevertheless the average value indicates online processing assuming 10% precision of measurements.

5.4 Number of producers

The fourth set of tests was focused on the influence of increasing the number of producing nodes in the running topology. The main purpose of the following tests was to examine scalability of the system and find the maximal number of producers per one merger instance for efficient data processing.

Each producer was sending 100 QC objects every second. Each data object was of the *TH1F* type and 1 kB size. The socket buffer capacity between producers and merger was set to one hundred thousand QA objects. Number of required objects for one merge operation was defined by the number of running producers.

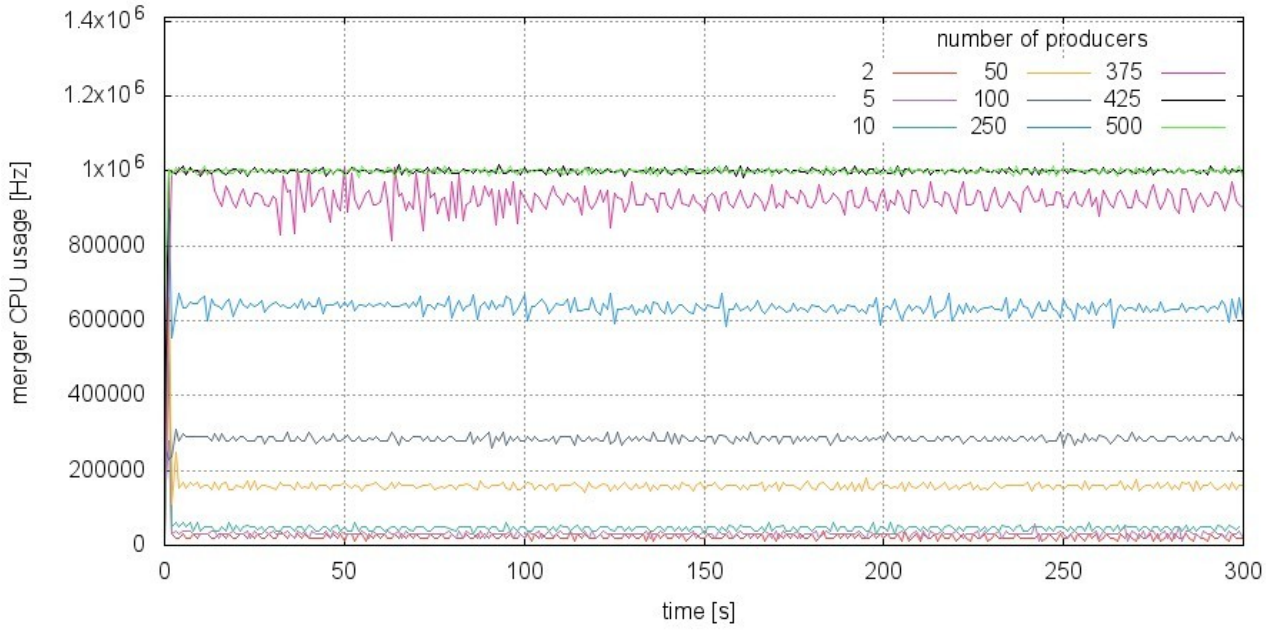

Figure 21: Approximated merger CPU usage.

Table 17: Average value and standard deviation of approximated merger CPU usage.

Producers number	Average value [Hz]	Standard deviation [Hz]
2	24610	27769
5	32341	16632
10	48344	32039
50	159418	31588
100	284361	9067
250	634129	40860
375	922957	56279
425	996926	41565
500	998176	21390

The merger CPU usage as shown in figure 21 and table 17 was increasing by adding more instances of producer program to the running test topology. The maximal value for one CPU core utilization occurred for the topologies with 425 and 500 producers instances.

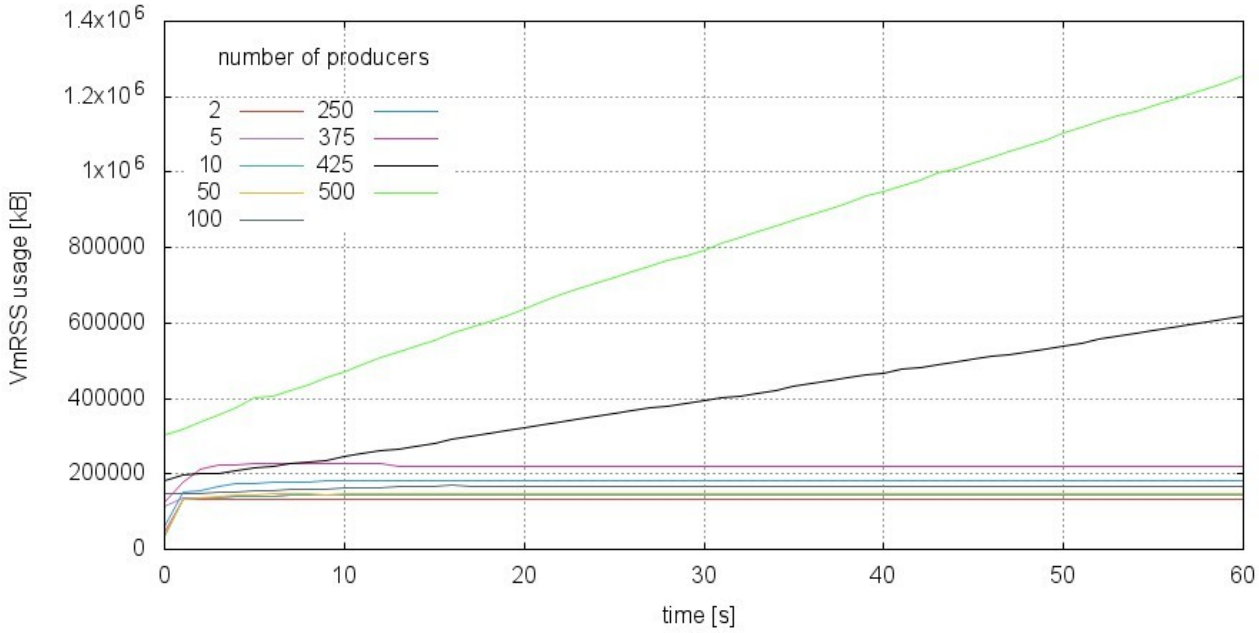

Figure 22: Merger VmRSS usage.

Table 18: Average value and standard deviation of merger VmRSS usage.

Producers number	Average value [kB]	Standard deviation [kB]
2	133222	5043
5	144084	1968
10	144849	6399
50	146473	6424
100	167114	2639
250	182485	7564
375	220472	6093
425	1292891	655169
500	2635385	1337465

The memory was stabilized for every case except for 425 and 500 producers. In figure 22 it can be seen that VmRSS usage was constantly increasing during the entire test for the two highest numbers of producers. The results in table 18 indicates that average memory consumption increases with adding more producers to the test topology.

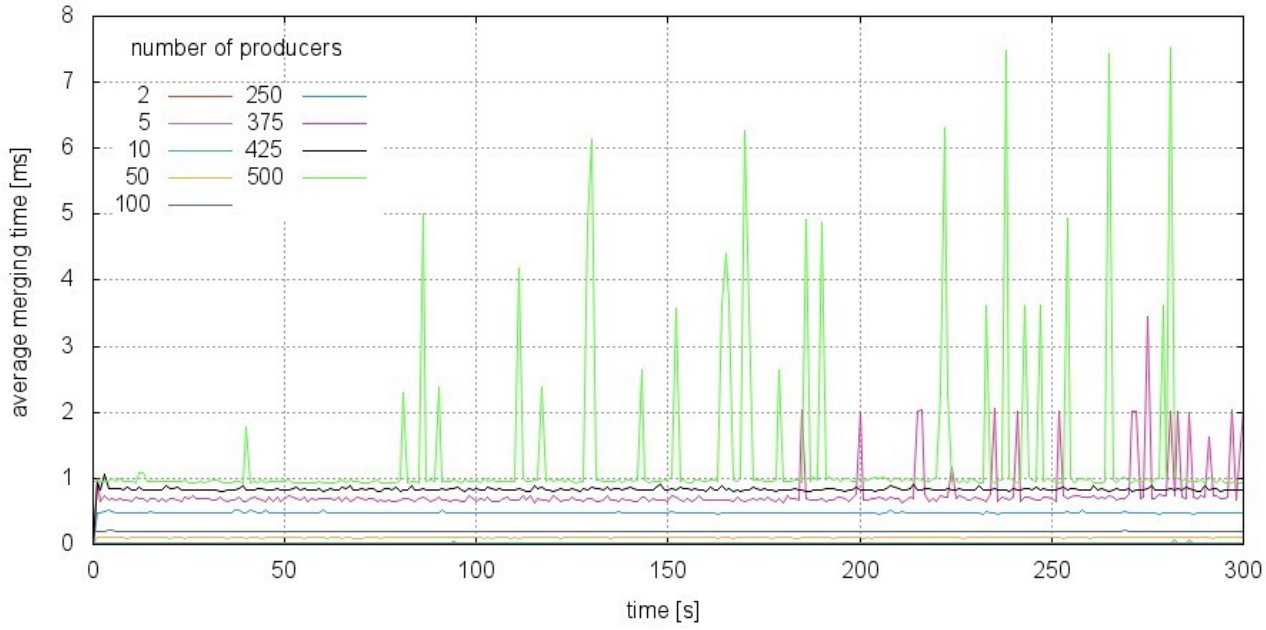

Figure 23: Average merging time.

Table 19: Average value and standard deviation of average merging time.

Producers number	Average value [ms]	Standard deviation [ms]
2	0.012	0.001
5	0.016	0.004
10	0.023	0.002
50	0.098	0.006
100	0.201	0.001
250	0.472	0.029
375	0.765	0.331
425	0.831	0.054
500	1.276	1.086

The results shown in figure 23 and table 19 indicate that the average merge time was increasing with adding more producers to the test topology. It is expected result as more objects needs to be merge for more producers instances.

Fluctuations of the results for 375 and 500 producers cases are visible in figure 23. These fluctuations are higher for 500 producers which is the result of reaching the limit of available computing resources for merger program.

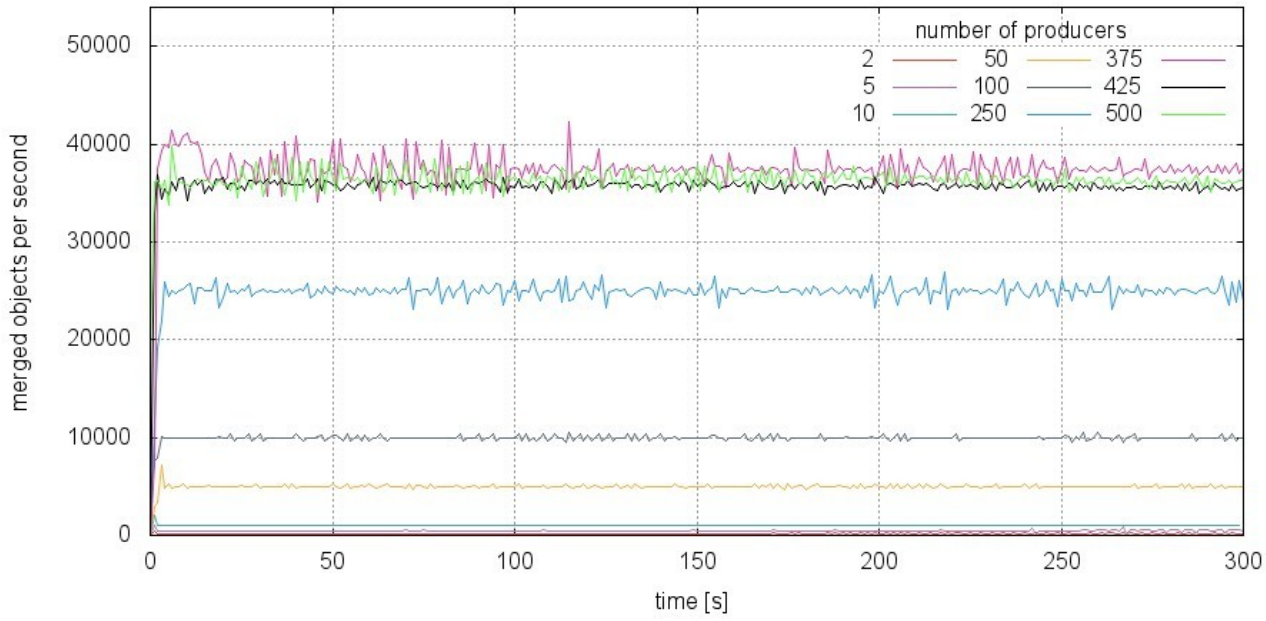

Figure 24: Number of merged objects per second.

Table 20: Average value and standard deviation of number of merged objects per second.

Producers number	Average value	Standard deviation
2	200	28
5	501	64
10	1000	84
50	4977	368
100	9962	428
250	24840	1781
375	37307	3028
425	356578	2105
500	36341	1313

The number of merged objects per second shown in figure 24 and table 20 was increasing by adding more producers to the test topology for up to 375 instances. Further addition of producing programs resulted in a decrease of the number of merged objects per second.

5.5 Number of mergers

The last set of tests was examining the influence of increasing mergers components to the system for predefined number of producers. The tests of number of producers showed that the system is not efficient for 500 producers with one merger. That is why these number of producers was selected for further investigation.

To assure that each merger has the same load, producers were spread into sustainable groups using DDS topology file. Each group was sending objects to predefined merger instance. Number of required objects for merge was equal to the number of producers per merger.

Each producer instance was sending *TH1F* QC objects with size of 1 kB (100 bins with 1000 entries) with rate of 100 objects per second. Each socket for every connection between producers and merger had buffer with capacity of one hundred thousand messages.

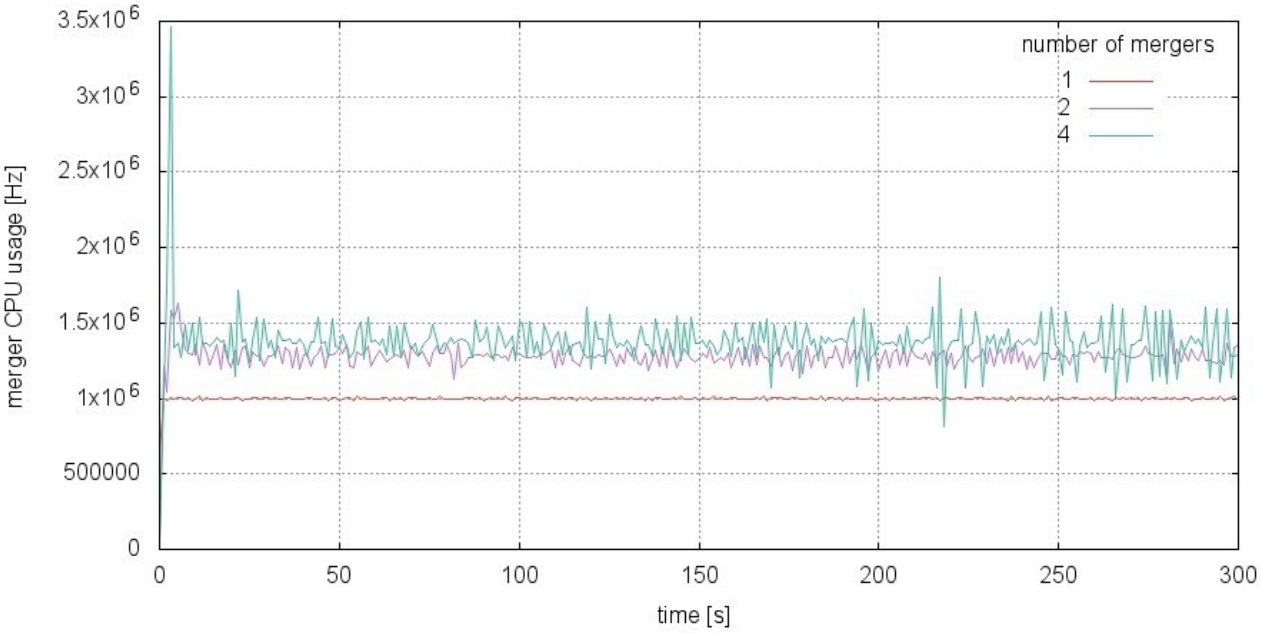

Figure 25: Approximated merger CPU usage.

Table 21: Average value and standard deviation of approximated merger CPU usage.

Mergers number	Average value [Hz]	Standard deviation [Hz]
1	998176	21390
2	1277743	92076
4	1376229	186923

Figure 25 shows that doubling the number of mergers does not result in significant increase of the CPU usage what can be also seen in table 21.

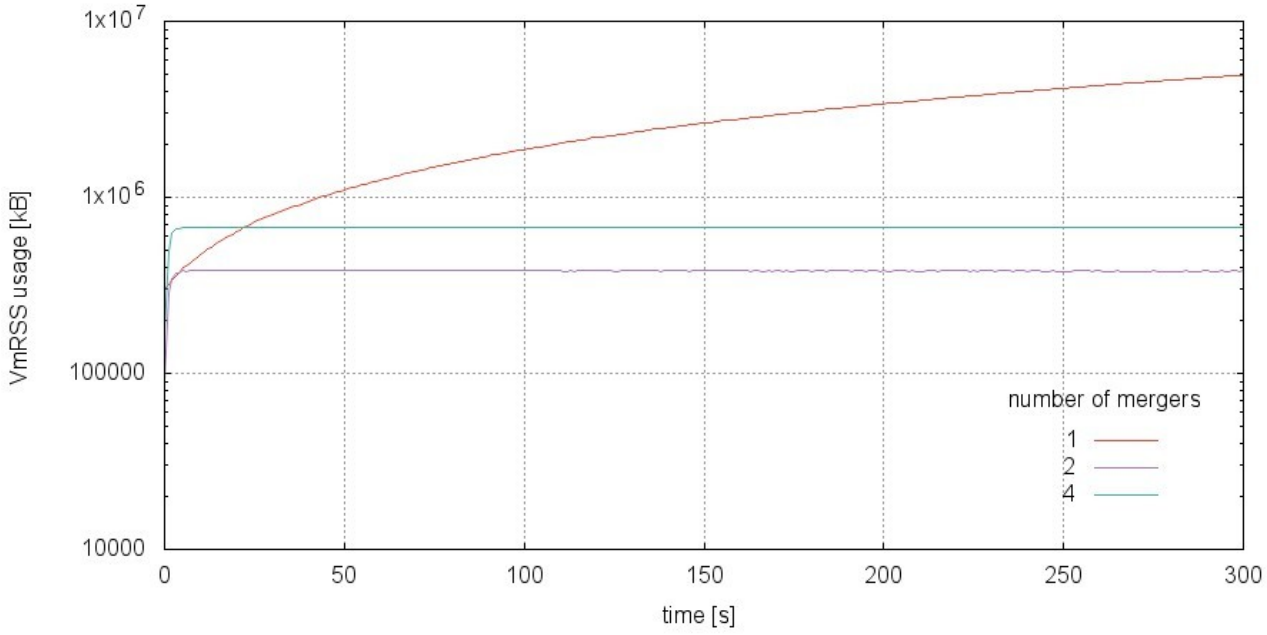

Figure 26: Merger VmRSS usage.

Table 22: Average value and standard deviation of merger VmRSS usage.

Mergers number	Average value [kB]	Standard deviation [kB]
1	2635385	1337465
2	380539	17962
4	677635	31132

The topology of one merger and 500 producers is not efficient because of too big input data to the merger. One of the results is constantly growing memory as shown figure 26. This is because objects in the buffer queue must wait for its processing turn and this queue is growing over the time. Stabilization of the memory usage occurred when added more mergers to the topology. As it was expected based on the previous results from the section 5.4 adding one merger was enough to enable online processing. Stabilization of the consumed resources occurred for both 2 and 4 mergers but at different level. Consumption of the memory is higher when adding more mergers to the system.

The results in table 22 show that the amount of memory allocated for 4 mergers is almost two times bigger than for 2 mergers. It indicates that increasing number of mergers has larger impact on consumed memory than on CPU usage.

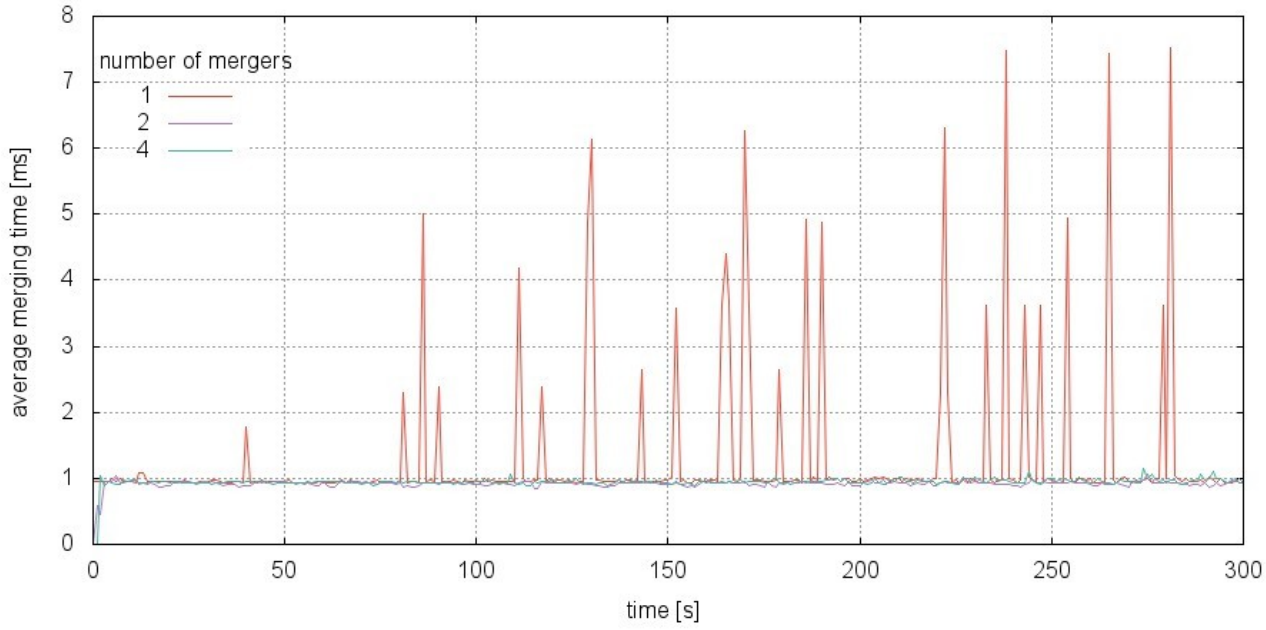

Figure 27: Average merging time.

Table 23: Average value and standard deviation of average merging time.

Mergers number	Average value [ms]	Standard deviation [ms]
1	1.28	1.09
2	0.92	0.07
4	0.94	0.08

As shown in figure 27 major fluctuations occurred only for the topology with only one merger for 500 producers. Moreover average time of merge is larger than for other cases. The average value of the merging time is the same for 2 and 4 merger (table 23) assuming a 10 % precision on the results interpretation.

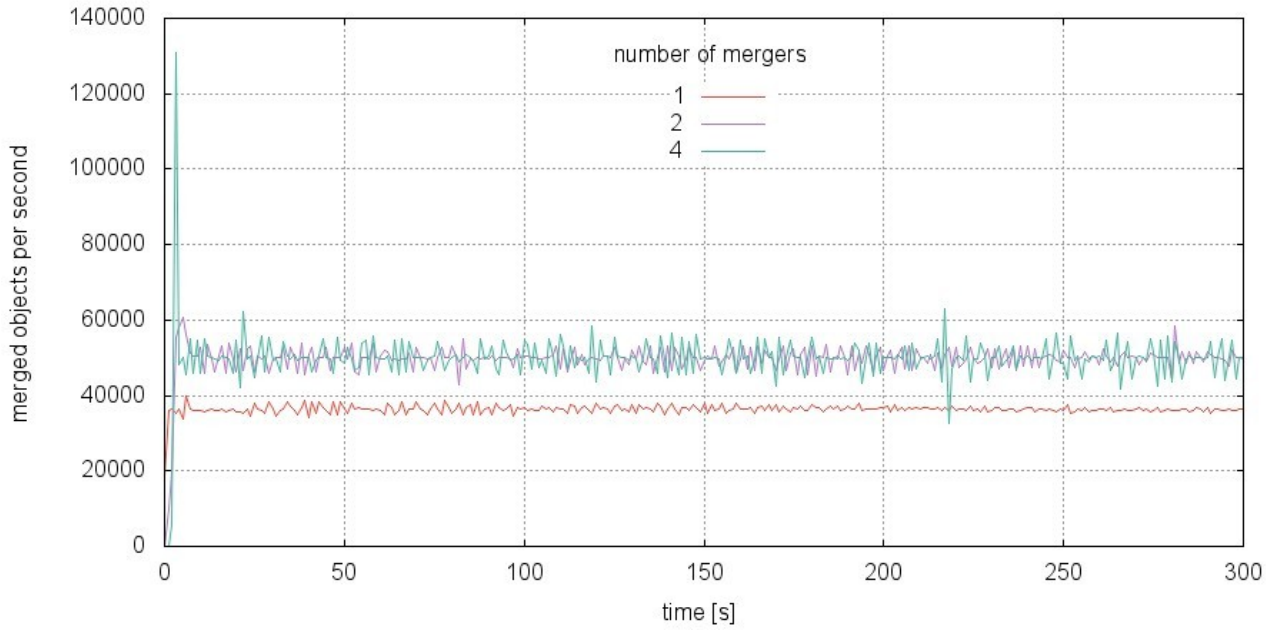

Figure 28: Number of merged objects per second.

Table 24: Average value and standard deviation of number of merged objects per second.

Mergers number	Average value	Standard deviation
1	36341	1313
2	49622	4720
4	49807	7578

The results shown in figure 28 and table 24 indicate that one merger in the topology of QC system is not enough to carry out online data processing. Number of merged objects per second for topology with 2 and 4 mergers demonstrate improved processing capabilities of the QC data. The amount of the processed data for those topologies is close to the expected value of 50 thousand merged objects per second.

6 Summary

Frameworks developed by the ALICE Collaboration and GSI/IT group provided all needed tools to develop QC prototype of the O^2 system. Nevertheless DDS is still in the development phase and it was needed to align the prototype implementation to the newest libraries versions for several times. Some of the important functionality of this framework are still in plans. An example can be a dynamic topology reconfiguration which will be crucial feature of the final QC system implementation.
Another problem is not efficient functionality of the merging operation of the ROOT framework implementation for objects types *THnF* and *TTree*. Furthermore current implementation of the ROOT framework has memory leaks which has to be eliminated in future releases.

The implemented QC prototype system allowed for estimating the performance and resource consumption of different topologies and parameters.

Buffer size of the socket connecting producers with mergers affected system performance when it was more than 20 times smaller than the overall merger input. For capacity of 1 message the number of merged objects per second was equal to 94.21% of the overall input of merger which indicates that some data had to be held in the buffer over time.

Tests carried out have shown that objects of size 1 kB, 500 kB, 5MB and 50 MB could be merged online with the maximal merger input rate which was different for larger objects. Despite the fact that for objects of size 5 MB input of the merger was 8 times smaller the CPU usage was at comparable level as for 500 kB objects.

Data type tests have proved that the most suitable objects types for processing are *TH1F*, *TH2F* and *TH3F*. This is because merging rate was at that same high level and processing was carried online for the whole test. Due to constantly growing memory usage for tests of *THnF* and *TTree* types efficiency of merging operation was not satisfactory because of impact on number of received and processed objects per second.

Scalability of the system topology by increasing the number of producers resulted in increasing performance of the merger component for up to 375 producers in the running topology as shown in figure 29.

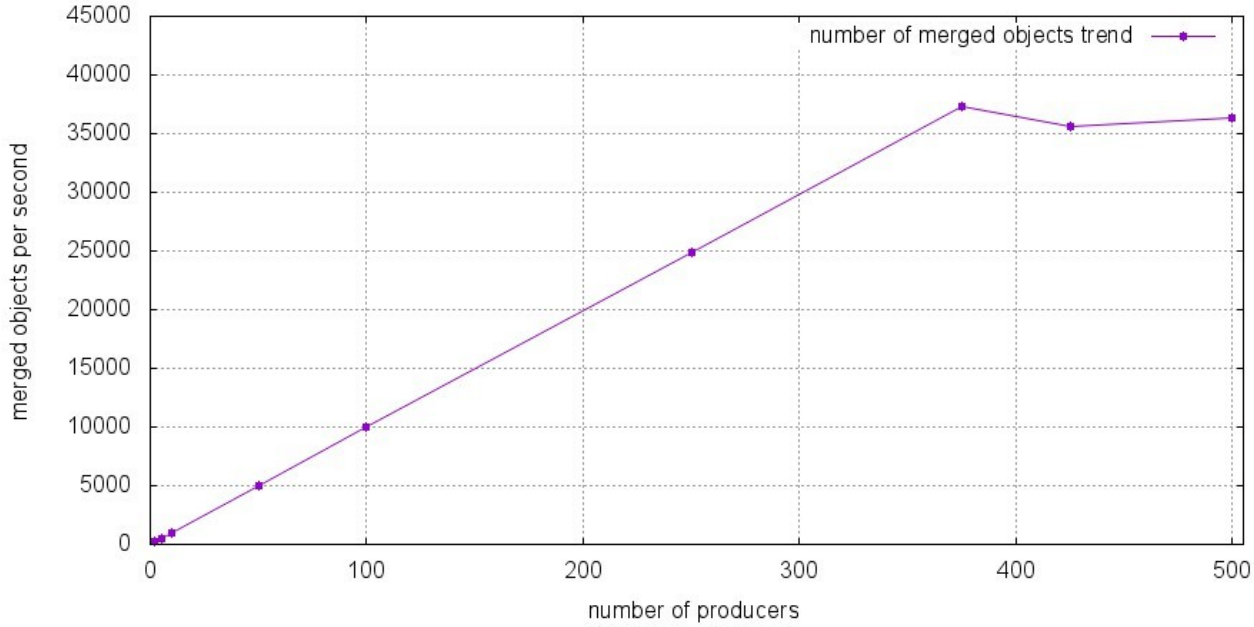

Figure 29: Number of merged objects per second for various number of producers in the test topology.

The number of merged objects per second increases linearly for up to 375 producers. After that point decrease of the performance is observed for 425 and 500 producers. It results in holding more and more objects in the socket buffer resulting in reduce merger capabilities of merging objects.

Tests of various number of merger instances with a fixed number of producers proved that processing of the data can be enhanced by unloading overloaded merger component by a redirection part of input to another instance of merging node. Processing of input from 500 producers was not sufficient for only one merger. By doubling number of mergers online processing was restored for two mergers, each connected to 250 producers as shown in figure 30. Moreover hardware resources consumed during execution does not increase significantly.

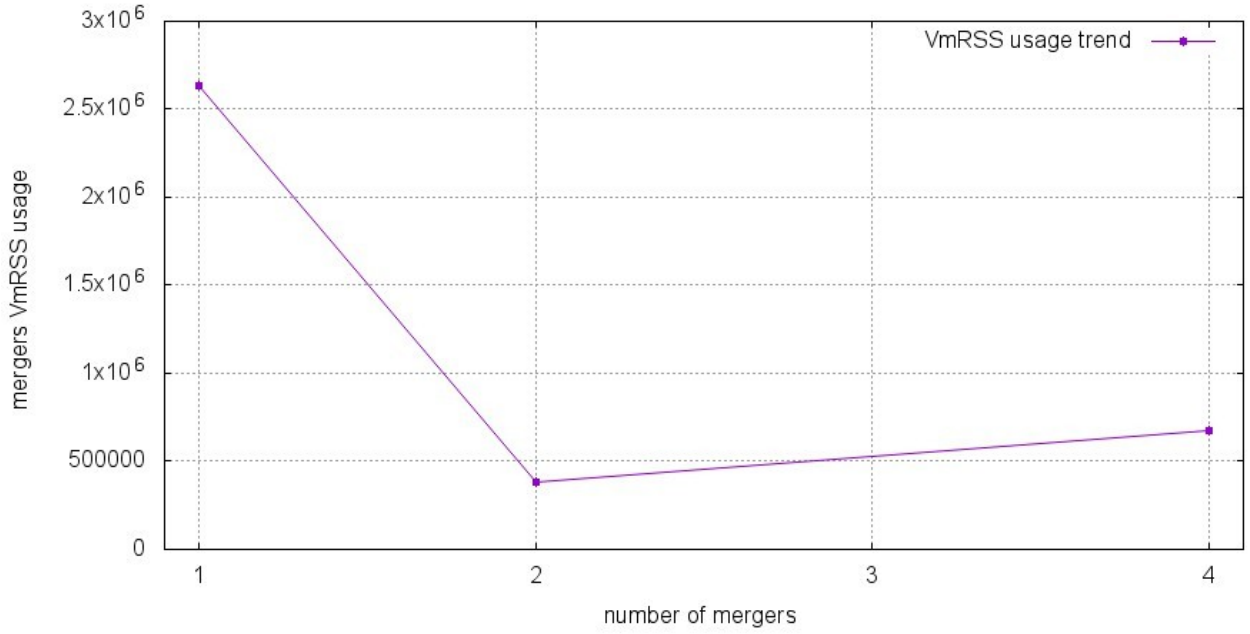

Figure 30: Merger VmRSS usage trend for various number of merger instances.

7 References

[1] K. Aamodt et al. (The ALICE Collaboration) “The ALICE experiment at the CERN LHC”, JINST 3 (2008) S08002, http://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08002/pdf

[2] J. Alme et al. (for The ALICE Collaboration), Nucl. Instrum. Meth. A 622 (2010) 316–367, http://www.sciencedirect.com/science/article/pii/S0168900210008910

[3] B. Abelev et al. (The ALICE Collaboration) “Upgrade of the ALICE Experiment: Letter of Intent”, J. Phys. G: Nucl. Part. Phys. 41 087001 (2014), http://iopscience.iop.org/article/10.1088/0954-3899/41/8/087001/pdf

[4] J. Adam et al. (The ALICE Collaboration) “Technical Design Report for the Upgrade of the Online-Offline computing system”, CERN-LHCC-2015-006 (2015)

[5] “Boost 1.61.0 Library Documentation”, http://www.boost.org/doc/libs/1_61_0/

[6] FairRoot official web site, https://fairroot.gsi.de/

[7] “ROOT Data Analysis Framework, User’s Guide”, (2014), https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html

[8] “0MQ – The Guide”, http://zguide.zeromq.org/page:all

[9] “The DDS User Manual v1.2”, http://dds.gsi.de/doc/nightly/dds.pdf

[10] Clinton Gormley, Zachary Tong “Elasticsearch: The Definitive Guide”, https://www.elastic.co/guide/en/elasticsearch/guide/current/index.html

[11] “The QC prototype project”, https://github.com/AliceO2Group/AliceO2/tree/dev/Utilities/QC

8 Remarks

The source code of developed QC system prototype together with installation and usage instructions can be found at Github repository [11] or on the physical drive included to this thesis.

This research was supported in part by PL-Grid Infrastructure.