Silesian University of Technology

Faculty of Automatic Control, Electronics and Computer Science

Fault Tolerant Data Acquisition through Dynamic Load Scheduling

PhD Thesis

MSc Eng. Michał Simon

Supervisor: Prof Stanisław Kozielski

Geneva-Gliwice 2013

To my parents

and my loving wife Marta

who liked the idea of me being well educated

Fairy tales are more than true — not because they tell us dragons exist, but because they tell us dragons can be beaten.

G. K. Chesterton

Acknowledgements

I would like to thank my supervisors Prof Stanisław Kozielski and Dr Hannes Sakulin for all their help and support during my PhD studies, for all the stimulating discussion and most importantly for showing me what science is.

Special thanks go to whole CMS Data Acquisition group for both teaching me the secrets of their DAQ System and for sharing the infrastructure with me.

I am indebted to my doctor-wife Marta for her valuable feedback from a different (area of science) perspective.

I would like to thank my parents and my sister Joasia for being such a great family and (what has been essential for my PhD studies) for laying the foundations of my knowledge when I was young.

And finally last but not least, I would like to thank my brother in law and his mother (Dr Piotr Stec and Dr Krystyna Stec) for showing me that old-school scientists are really cool.

Table of Contents

Abbreviations

ARMOR	Adaptive, Reconfigurable, and Mobile Objects for Reliability	**42**	
aTTS	asynchronous Trigger Throttle System	44	
BCN	Builder Control Network	44	
BDN	Builder Data Network	44	
BM	Builder Manager	44	
BU	Builder Units	19	**56**
CMS	Compact Muon Solenoid	15	**15**
CORBA	Common Object Request Broker Architecture	39	
CSC	Cathode Strip Chambers	15	
CSN	Computing Service Network	44	
D2S	Data to Surface	44	
DAQ	Data Acquisition	**16**	
DCN	Detector Control Network	44	
DCS	Detector Control System	44	
DFM	Data Flow Manager	37	
DQM	Data Quality Monitoring	61	
DSN	DAQ Service Network	44	
DT	Drift Tubes	15	
ECAL	Electromagnetic Calorimeter	16	
EDF	Earliest Deadline First	**32**	
EPR	Equal Partitioning Rule	**32**	
ETL	Extract, Transform, Load	31	
EVM	Event Manager	19	**57**
FTM	Fault Tolerance Manager	**42**	
FB	FED Builder	44	
FBO	FED-Builder Output	56	
FEC	Front-End Controller	44	
FED	Front-End Driver	45	

FES	Front-End System	44	
FFN	Filter Farm Network	44	
FIFO	First In First Out	32	
FRL	Front-End Readout Link	45	
FS	Filter Subfarm	44	
FSM	Finite State Machines	59	
FU	Filter Units	19	61
FU-EP	Filter Unit Event Processor	61	
FU-RB	Filter Unit Resource Broker	61	
GTP	Global Trigger Processor	44	
HCAL	Hadron Calorimeter	16	
HEP	High Energy Physics	13	
LHC	Large Hadron Collider (LHC)	13	14
LV1	Level-1 Trigger Processor	44	
MEP	Multi-Event Packets	36	
MWDF	Maximum Workload Derivative First	32	
NIC	Network Interface Controller	46	
OPR	Optimal Partitioning Rule	32	
QCD	Quantum Chromodynamics	14	
RCMS	Run Control and Monitoring System	45	
RCN	Readout Control Network	44	
RM	Readout Manager	44	
RPC	Resistive Plate Chambers	15	
RTP	Regional Trigger Processor	44	
RU	Readout Units	18	58
SM	Storage Manager	19	
SPMD	Single Process, Multiple Data	18	
sTTS	synchronous Trigger Throttle System	44	
TDAQ	Trigger and Data Acquisition System	37	

TPG	Trigger Primitive Generator	44
TTC	Timing, Trigger and Control	44
VLA	Very Lightweight Agent	42

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 High Energy Physics

High Energy Physics (HEP) is a subfield of physics that investigates the elementary particles of matter and the forces between them. The name, of the field is related to the fact that many subatomic particles do not occur under ordinary conditions on Earth and can only be obtained during high energy collisions of regular particles that can be found in nature. These collisions are carried out in specialized instruments called particle accelerators. Currently, the world largest and highest-energy particle accelerator is the Large Hadron Collider (LHC) at CERN in Geneva, Switzerland (shown in Figure 1-1).

Figure 1-1 LHC Ring

Basically, two attributes characterize a particle collision: loss of energy by the particle and a deflection of the particle from its original motion direction [1]. These attributes are often registered for further analysis in order to extend the knowledge about the phenomena occurring during collisions. Cross section is the basic concept used in particle physics for evaluating the probability of interaction or collision between two particles. The second important property describing the collisions is the luminosity which is determined by storage ring operating conditions. By definition, the luminosity is the number of particles in the beam per unit time and can be increased by increasing the beam intensity. The luminosity is proportional to the number of particles of each of the two colliding beams and to the beam revolution frequency. It is inversely proportional to the beam size in the collision point. The luminosity multiplied by the cross-section gives the process collision rate. The two discussed quantities essentially allow estimating how many particle collisions will occur per unit of time and as a result, define the needed design data-taking rate.

1.1.1 Large Hadron Collider

The LHC is the largest circular accelerator with circumference length of 27 km, located about 100 to 150 m underground. The LHC is designed to accelerate two beams of hadrons (protons or lead ions) in opposite directions and then to collide them with energy of 7 TeV per each beam (14 TeV in total) and with luminosity of 10^{34} $cm^{-2}s^{-1}$ (which corresponds to a bunch crossing frequency of 40 MHz) [2]. The main goal of LHC is to study the electroweak symmetry breaking for which the Higgs mechanism theory is supposed to be an explanation. These experimental studies can also contribute to the mathematical consistency of the Standard Model at energy scales above about 1 TeV. Moreover, the LHC opens up opportunities for discoveries that could lead towards a unified theory such as extra dimensions or supersymmetry theory. Furthermore, the heavy-ion collisions carried out at LHC allow for conducting research on Quantum Chromodynamics (QCD) matter under extreme conditions of temperature, density, and parton momentum fraction (low-x).

1.1.2 Compact Muon Solenoid

The Compact Muon Solenoid (CMS) is one of two large general-purpose experiments at LHC for studying proton-proton and heavy ion collisions at TeV scale. In order to avoid mixing a single event of particles collision with other collisions from the same bunch crossing CMS has been designed using high-granularity detectors with good time resolution (which results in low occupancy) and equipped with millions of readout channels.

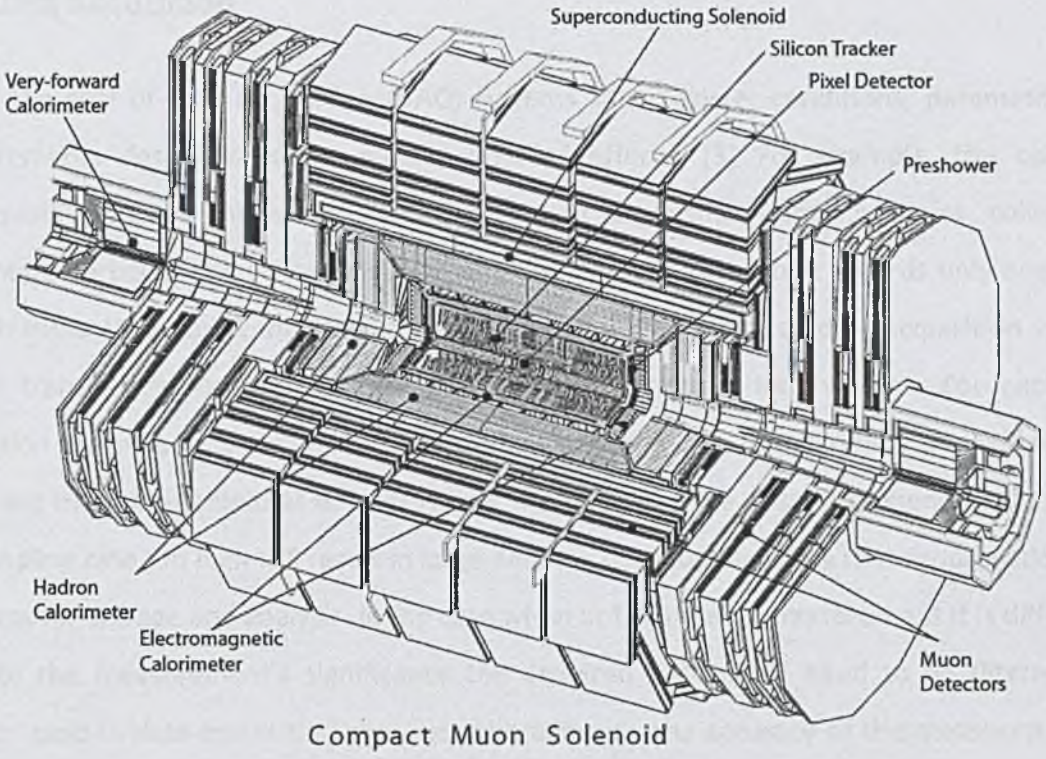


Figure 1-2 The Overall layout of CMS detector

As shown in Figure 1-2, the CMS detector has an onion-like structure. It is constructed of multiple layers that are responsible for concurrent measurements of numerous parameters and phenomena caused by hadron collisions. The magnetic field is provided by a 4-T superconducting solenoid. In order to guarantee reliability and full geometric coverage, four muon stations have been installed, each of them composed of a number of layers of aluminum drift tubes (DT), and cathode strip chambers (CSC), completed by resistive plate chambers (RPC). The tracking volume consists of 10 layers of silicon microstrip detectors that ensure

desired granularity and accuracy. Moreover, 3 additional layers of silicon pixel detectors are used to further enhance the observation quality of the impact parameter of charged-particle tracks, along with the position of secondary vertices. The electromagnetic calorimeter (ECAL) is made from lead tungstate (PbWO4) crystals and placed in the central barrel part, with preshower detector in front. ECAL is encapsulated inside a hadron calorimeter (HCAL) that is a brass scintillator sampling calorimeter.

1.2 Data Acquisition

The goal of data acquisition (DAQ) systems is to register conditions, parameters and measurements describing some specific physical effects. [3] For example, the observed phenomena could be the effects of a drug on an organism, seismic activities, collision of elementary particles, and so on. Basically, any data acquisition system records only one thing, which is the voltage (or eventually the electric current). Therefore, all data acquisition systems require transducers that convert the phenomena of interest into voltage. For each data acquisition system a proper sampling rate has to be chosen. Too low sampling rate will result in inaccurate measurements that will not reflect the nature of the observed phenomena. Setting the sampling rate too high will result in large amount of redundant data that require additional resources for storage and analysis. In the case when at the time of measurement it is difficult to estimate the measurement's significance the acquired data often need to be filtered. Any detector used in data acquisition requires calibration, so the accuracy of the measurements is known.

1.2.1 Data Acquisition in High Energy Physics

The modern detectors used in high energy physics (HEP) experiments are complex instruments designed to register collisions of elementary particles at extremely high energies. Only a small fraction of such collisions results in interesting, new phenomena. Therefore, in order to maximize the probability of a discovery in particle physics, a collision rate in the MHz range is needed. Data that correspond to a single collision of particles referred to as an event are acquired from millions of readout channels. Those readout channels are merged into

several hundreds of detector front-ends. Each detector front-end serves as a source of event fragments. The event fragments corresponding to a single event contain a common, unique ID, so they can be combined into a whole event later. One of the major challenges that today's HEP Data Acquisition Systems are facing is to process the huge amounts of data produced by the detectors. Since it is not possible to send all collected events to persistent storage due to the required space, a drastic event filtration has to be achieved. The goal is to select only those events that describe the phenomena of interest (e.g. confirm the existence of Higgs Boson). The filtration process has to begin as soon as possible and therefore the first selection decisions are made based only on some partial event information (e.g. using an event fragment). Afterwards, when an event has been fully reconstructed more sophisticated algorithms are used in order to take the final selection decision. Depending on the filtering stage the selection algorithms are implemented either in hardware (early stages) or in software.

1.2.2 Drawbacks of static workload distribution

After the first event selection step that is completely realized in hardware, the data acquisition systems used in HEP experiments, acquire event fragments from numerous sources. [4] Further filtering steps are implemented in software running on a set of computing farms. In the very first software-selection step due to the still high rate (the order of 100 kHz), the data are usually distributed in a static way between filtering nodes. In case of systems, with only one stage of software filtration, this is also the final stage, where the event reconstruction has to be done. In this case the static distribution determines strongly the system. The processing power of the participating computing nodes and farms has to be easily measurable, so that the distribution schema could be prepared precisely. Subsequently, it is difficult to introduce heterogeneity to the discussed group of data acquisition systems. Moreover, static data distribution decreases fault tolerance and introduces additional single points of failure. The main goal of our research is to replace the static workload distribution policy with an algorithm that allows for dynamic adjustment to the changes in the available processing power and thereby increases the system's overall fault tolerance. The algorithm should also facilitate measurements of load on the system, and evaluation of the available computing power so it could be easily applied to heterogeneous systems.

1.2.3 CMS Data Acquisition System

The CMS is a multi-purpose detector for studying proton-proton and heavy ion collisions at TeV scale [2]. CMS is designed to collect data at the LHC bunch crossing frequency of 40 MHz (as described in subsection 1.1.1). The first level trigger pre-selects events with interesting signatures reducing the incoming data rate to a maximum of 100 kHz. The DAQ System (shown in Figure 1-3) acquires event fragments from about 500 sources and combines them into full events. Each data source delivers event fragments of an average size of 2 kB at a rate of 100 kHz. Event fragments are transported by a non-blocking network [5] (based on Myrinet [6] technology) to the surface and statically distributed (usually in round-robin fashion) amongst several autonomous processing units called DAQ Slices.

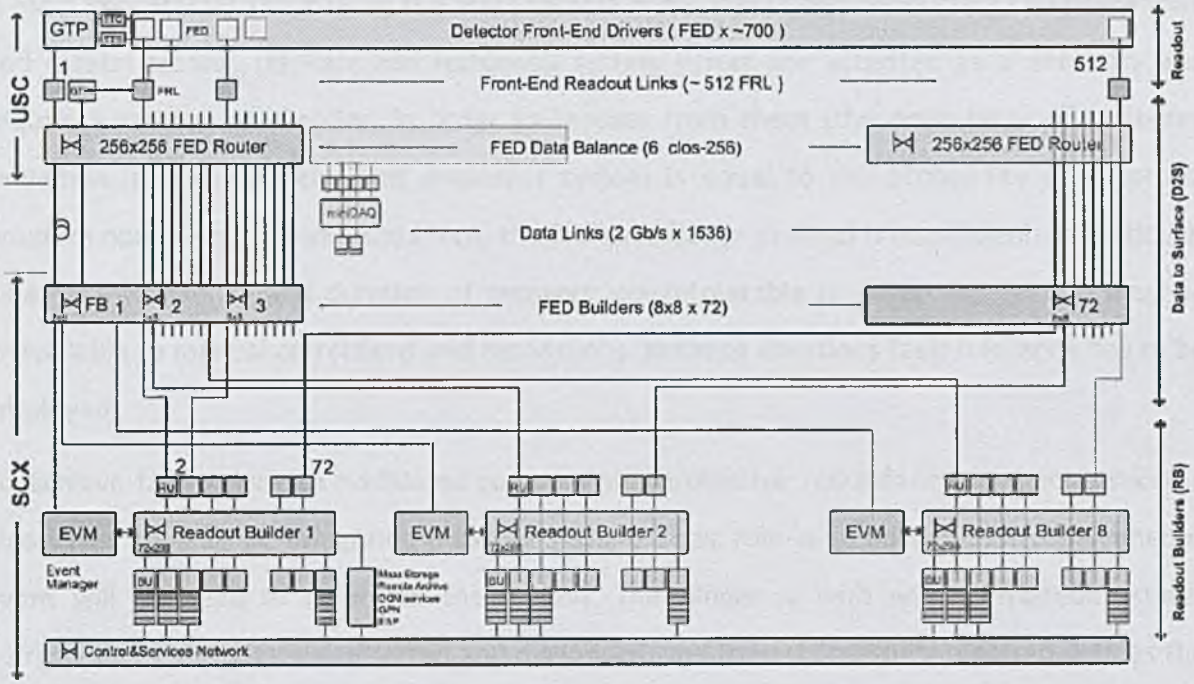


Figure 1-3 Schematic view of the CMS DAQ System [2], described in more details in section 3.1

A DAQ Slice is a computing farm organized around a Terascale Force10 switch, where parallelization is achieved through SPMD (Single Process, Multiple Data [7]) technique. In the first event building stage event fragments are received by a DAQ Slice through distributed readout consisting of computing nodes called Readout Units (RU), and then assembled into super-fragments inside these RUs. Subsequently, in the second stage, in each of the DAQ Slices,

an Event Manager (EVM) node assigns super-fragments to Builder Units (BU) that construct the whole event. The complete events are then delivered to Filter Units (FU) that run the High Level Trigger selection algorithm (BU and FU are hosted on the same node). Events accepted for storage are transmitted to Storage Manager (SM) nodes connected to a Storage Area Network. Currently when one DAQ Slice becomes less efficient, e.g. because of some fault like a failing computing node it slows down other DAQ Slices. Moreover, there are several potential single points of failure like the EVM, SM and RU nodes.

1.3 Fault Tolerance

The most popular way of achieving reliable computing is to employ fault avoidance that is most importantly about using the most reliable components and conducting comprehensive and careful testing. [8] Rare and incidental system errors are accepted as a necessity and require a manual intervention in order to recover from them (the probability of fault-free execution in a completely fault intolerant system is equal to the probability of a correct program operation). In some situations, the fault avoidance method is insufficient, in particular when the frequency and duration of recovery are intolerable or when the system may be unavailable to manual corrections and reparations. In these situations fault tolerance has to be employed.

To achieve fault tolerance additional components (protective redundancy) and sophisticated algorithms have to be integrated into the system. Their role is to ensure that an erroneous event will not lead to failure of the system. The efficiency with which erroneous states corresponding to faults are detected and diagnosed, and then successfully repaired defines the degree of fault tolerance. After a fault occurrence, depending on the extent of fault tolerance, and also on the complexity of the problem, the system may perform with its full efficiency, or may provide only reduced performance or limited functionality (fail-soft capability). In order to ensure reliability often many run time mechanisms have to be adopted and the system needs to be kept as close as possible to the correct state. [9] Most common fault tolerance mechanisms are:

- Error confinement: Each procedure has only least possible rights granted and a minimum domain of access. Also no operations on incoherent data are allowed. This policy limits error damage before detection.
- Detection and categorization: Each fault has to be detected and categorized in order to trigger appropriate reaction.
- Reconfiguration: For example excluding a failed unit from the system (whether it is hardware or software) or moving the system to a backup state.
- Restart: If the fault caused the system to stop a restart is needed.

1.4 Load Scheduling

The goal of load scheduling is to assign the incoming workload (data, calculations, etc.) to available resources of a distributed computing system. [10] The workload is allocated only once, at the point of load emergence in the system, before the actual processing of the workload starts. The load should be distributed in such a way that the execution time would be as short as possible and the available resources would be optimally utilized. Load scheduling algorithms (as well as load balancing algorithms) aim to balance the load to prevent coexistence of overloaded and idle resources, as well as to prevent from slowing down the more efficient parts of the system by the slower parts. [11] Such scheduling is also important because it helps to increase the fault tolerance of the whole system, amongst others, by removing single points of failure. Load scheduling techniques can be divided into two categories. [12] Static load scheduling allocates workload to computing nodes probabilistically or deterministically, without carrying out any analysis of runtime events, and as a result reduces communication delays. This method is lightweight, as well as efficient, as long as the workload is precisely identified, the available computing power is constant and the load scheduler is pervasive, which means that it is in charge of the whole incoming workload (or it is aware to a certain extent of the background load of the system). Difficulties arise when it is hard to predict the workload of incoming tasks, there are fluctuations in background load, or the computing power varies (e.g. due to some fault occurrence). Dynamic load scheduling is a method for distributing the load between available resources, based on the resources efficiency over the time. It is designed to

handle the problems of unknown or unidentified workloads and runtime variations. As a rule, load indices are collected during workload processing, and then used in the load assignment procedure in order to enhance resource utilization and minimize the overall processing time of newly emerged workload.

1.5 Load Balancing

Load balancing also aims to enhance the performance of a distributed system, especially in terms of resource availability. [10] [12] However, in this case, the workload (processes, data) is being reassigned among a group of cooperating computing nodes during execution time. Typically, workload indices are monitored and measured so appropriate action may be taken in order to achieve exact load distribution. The concepts of dynamic load scheduling and load balancing are closely intertwined by definition. Both may be classified based on the method used for triggering load distributing and redistributing activities, load assigner location and workload data exchange pattern. The advantage of load scheduling over load balancing is that the overhead due to workload transfer from one computing node to another can be avoided. On the other hand, load balancing can be far more effective in case of several, not equally time consuming, interdependent tasks. Although, the main subject of this dissertation is dynamic load scheduling, we will also discuss some aspects of load balancing because of the analogies.

1.6 Problem Formulation

In the classical load scheduling problem (Figure 1-4) a load scheduling algorithm aims to distribute the incoming workload that is produced by one load source between available computing nodes. In our case (Figure 1-5), the workload takes the form of a distributed stream by which we mean that the stream consists of numerous sub-streams, each of them provided by a separate load source. Such a single sub-stream by itself carries no information and therefore cannot serve as the basis for any computations. However, all the sub-streams together constitute a logical wholeness that outlines the hadron interactions that take place in the LHC accelerator and were selected by the first level trigger. Furthermore, the workload stream has another interesting property, namely it is divisible in the sense that it consists of a

sequence of independent data that may be processed individually. This independent data correspond to single collisions of two bunches of particles and are referred to as events. Each load source produces event fragments in the same sequence and with the same frequency. The goal of the proposed load scheduling algorithm is to balance the incoming workload between several computing farms that are carrying out the task of selecting events with interesting signatures for persistent storage. The filtering farms are receiving the incoming events through a distributed readout that consists of a set of readout nodes. Each readout node receives the workload only from several load sources, and the workload transfer from load sources to readout nodes is not synchronized. The workload allocation has to be done with a particular emphasis on fluctuations in computing power of filtering farms that may be caused by faults.

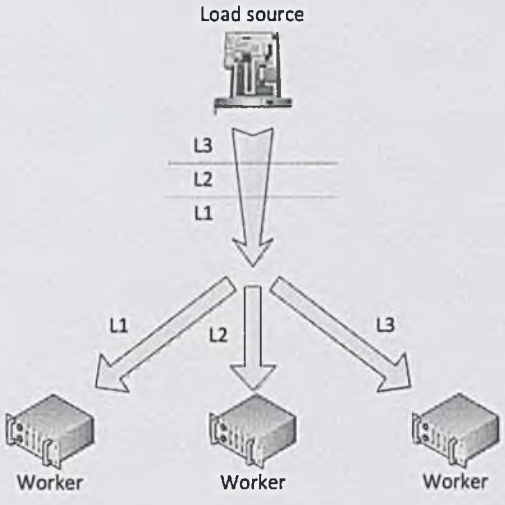


Figure 1-4 Load Scheduling, classical case

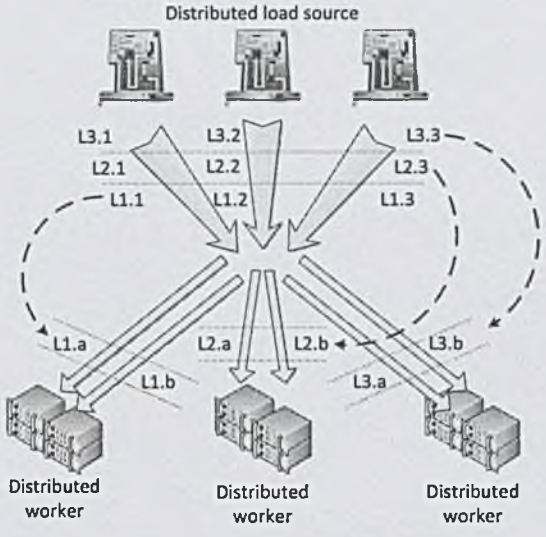


Figure 1-5 Load scheduling, investigated case

1.6.1 Objectives

i. Investigating of the extent to which the load scheduling can increase the reliability of a distributed data acquisition system.
ii. Providing dynamic load scheduling for heterogeneous computing farms, as well as, homogeneous computing farms, where the imbalance could be caused by faults.
iii. Increasing the efficient utilization of available resources.

iv. Proposing a scalable load scheduling protocol along with a distributed asynchronous load assignment policy and a robust load index.

1.6.2 Hypotheses

i. Dynamic load scheduling increases the overall fault tolerance of a distributed data acquisition system.

ii. Asynchronous, distributed load scheduling can be performed on workload fragments (constituting coherent wholeness) produced by numerous load sources, provided that each load source is producing the workload fragments in the same sequence.

2 State of the Art – Dynamic Load Distribution and Fault Tolerance

Although, the available literature discussing workload distribution (load scheduling and load balancing), data acquisition and fault tolerance separately is very rich, the number of papers dealing with the problem of workload scheduling and fault tolerance and at the same time addressing requirements comparable to those of the Compact Muon Solenoid DAQ system is very modest. Even in High Energy Physics there are only several experiments that have comparable data-taking and event size conditions to the CMS.

In this chapter our aim is to give a complex and most up to date overview of the state of the art in the fields of workload distribution and fault tolerance. However, we will only focus on research and methods that could be applied in our system. First we will discuss the taxonomy and classification of load distribution algorithms and the possible workload indices. Subsequently, we will present several load scheduling and balancing methods, as well as fault tolerance strategies and designs that have some features essential for our research project. Additionally, we will give an overview of dedicated load scheduling/balancing algorithms that have been applied in other data acquisition systems of HEP experiments, which have similar requirements to the CMS in terms of data taking.

2.1 Load Distribution Strategies

The following load distribution taxonomies will be the basis for classification of algorithms mentioned latter in this dissertation. Figure 2-1 shows the taxonomy of dynamic load distribution algorithms proposed by Osman and Amar in [13]. In the discussed classification method, in order to specify a load distribution algorithm, four major sub-strategies have been characterized: Initiation, Load Assigner Location, Information Exchange and Load Selection. [13] The initiation method defines the procedure for invoking activities that lead to exact load distribution. In the proposed classification two types of initiation may be distinguish: periodic and event driven. The event driven initiation is usually based on local

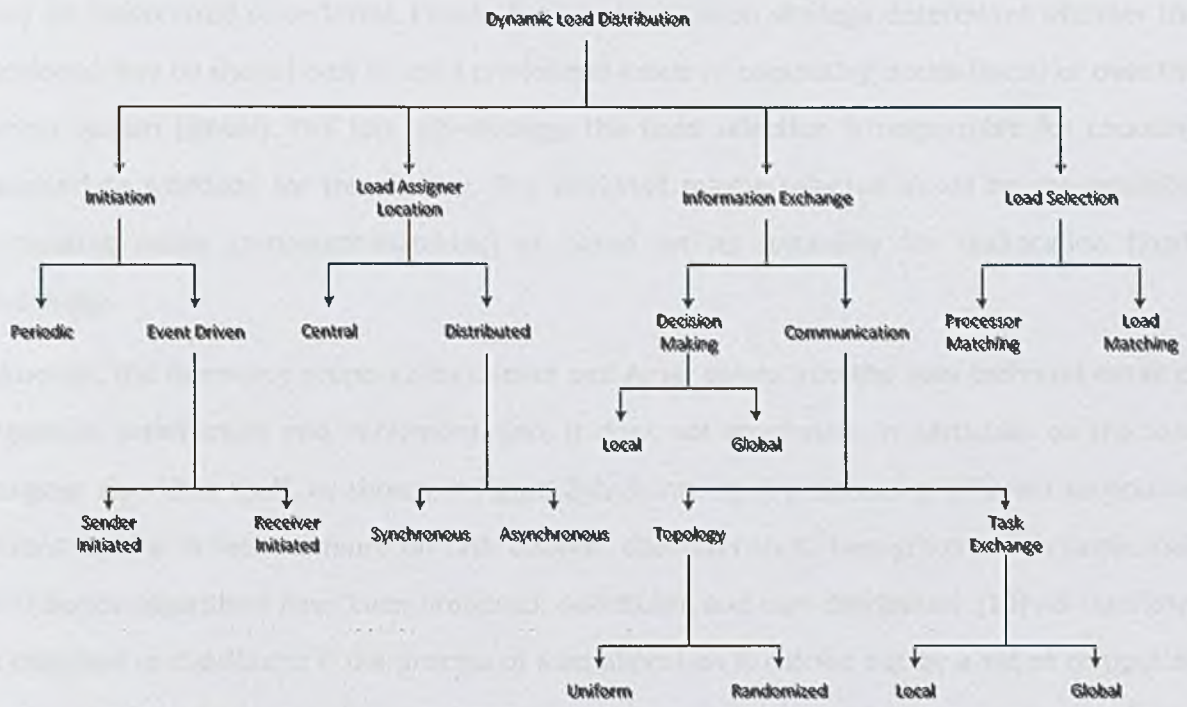


Figure 2-1 Taxonomy of dynamic load distribution algorithms [13]

workload observation and can be triggered either by sender or receiver. In general, event driven policies are more sensitive to workload fluctuations, while periodic policies are easier to implement. The Load Assigner Location policy defines the placement of the workload assigning algorithm itself. If there is a single agent supervising and controlling the whole process of load distribution the load distribution algorithm is classified as central. On the other hand, if each workload source (whether it is the original point of load emergence in the system, or an overloaded node) may decide about the load assignment independently the algorithm is said to be distributed. The distributed strategy may be further characterized as synchronous or asynchronous. The central policy introduces an additional single point of failure to the distributed computing system, and may lead to a bottleneck. The distributed policy, in turn, depends heavily on load indices propagation over the system, which may result in higher communication cost. The workload and load indices flow through the system is defined by the Information Exchange strategy. This policy specifies whether the information used by Load Assigner is local or global. Furthermore, it characterizes the connection topology that is used for information exchange and that in practise specifies the neighbourhood of each node (this

may be randomized or uniform). Finally, the communication strategy determines whether the workload may be shared only inside a predefined group of computing nodes (local) or over the whole system (global). The last sub-strategy, the Load selection is responsible for choosing appropriate workload for transferring. The workload maybe selected based on the available computing nodes (processor-matching) or based on its suitability for reallocation (load-matching).

Although, the taxonomy proposed by Osman and Amar delves into the very technical detail of algorithm architecture and implementation, it does not emphasize in particular on the load assigner algorithm itself. As shown in Figure 2-2, Shirazi et al proposed a different scheduling taxonomy that is focused more on task assigner characteristics. Two groups of dynamic load distribution algorithms have been proposed: distributed and non-distributed. [10] An algorithm is classified as distributed if the process of load allocation is carried out by a set of computing nodes. On the other hand, if the assignment decision is taken by a single computing node the algorithm is classified as non-distributed. The distributed algorithms are further divided into cooperative and non-cooperative. In case of non-cooperative strategies, the scheduling decision is carried out by each of the load scheduling nodes autonomously of the actions of other nodes. In the cooperative case, each of the load scheduling nodes is responsible for a fraction of the workload, but yet they are pursuing a common, global goal. As a result, the assignment decision is taken in the context of the whole system. The cooperative method may be further characterized as optimal or suboptimal. It can be easily noticed that workload scheduling is optimal if and only if all available computing nodes finish workload processing exactly at the same time, and there were no interruptions and pauses during workload processing. If this conditions are fulfilled it is guaranteed that the execution time is minimal. Unfortunately, finding an optimal solution is an NP-complete problem [14] [15] [16]. Therefore the suboptimal strategies are more common. In the discussed taxonomy two types of suboptimal policies are distinguished. The approximate algorithms use identical computational schema as the optimal algorithms, but rather than generating optimal schedules, they terminate when a solution is obtained that is considered as 'good' enough. The evaluation

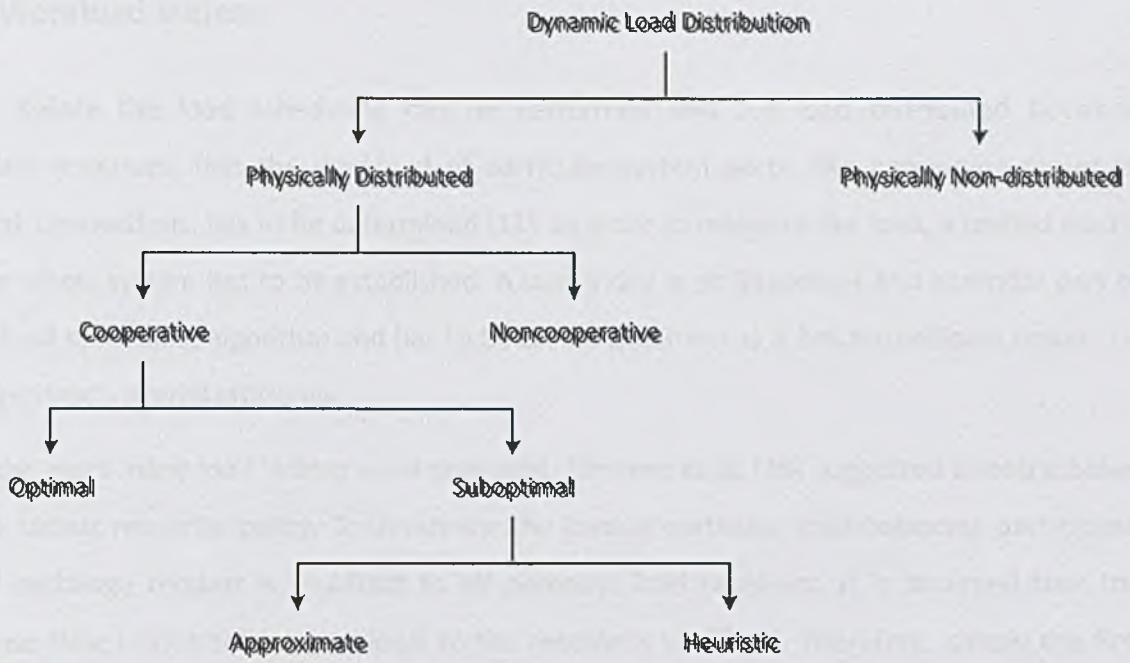


Figure 2-2 Taxonomy of dynamic load distribution algorithms [10]

method of the solution is a significant component of such algorithms, and its computational complexity, along with its accuracy determines whether this approach is appropriate or not. However, if such an easily computable method exists, the approximate policy can cut down the time needed for finding sufficient schedule significantly. Heuristic methods, in turn, often make use of some specific properties of the system that are rather easy to monitor and calculate and influence the overall performance in an indirect way. Usually the heuristic strategy is the most robust one in terms of time performance and resource utilization. However, it is not always possible to prove a first-order correlation between the adopted heuristic solution and desired outcomes.

All the workload distribution strategies discussed above employ a single stage decision procedure. There are also load allocation methods that adopt multi-stage decision-making procedures that are constructed like an OR-decision tree [17]. Such solutions are used to process job consisting of a chain of tasks, which are dynamically created accordingly to tree-like precedence constraints.

2.2 Workload Indices

Before the load scheduling can be performed and the load distributed between available resources, first the workload of particular system parts, like computing nodes or network connections, has to be determined [11]. In order to measure the load, a unified metric for the whole system has to be established. A load index is an important and essential part of every load scheduling algorithm and has to be carefully chosen as it has a significant impact on the algorithm's overall efficiency.

Over the years many load indices were proposed. Theimer et al. [18] suggested a metric based on the fastest response policy. To determine the load of particular load-balancing participant, a load exchange request is multicast to all potential load receivers. It is assumed that the response time is inversely proportional to the receiver's workload. Therefore, simply the first receiver who responds is regarded as the least loaded. Borzemski et al. also investigated the possibility of adopting the request response time as a load index in [19]. The studied algorithm is based on a fuzzy-neural decision-making scheme, which allocates the workload (HTTP requests) accordingly to the expected request response time. The request response time is being estimated by a broker node, which is taking into account historical latency measurements, the class of the request and the current workload on a given worker node (collected from local switches). More common load metrics usually explore the availability of resources more directly. They take into account e.g. CPU queue length, I/O queue length, memory utilization etc. Werstien et al. [20] presented a load metric reflecting CPU utilization, memory utilization and network traffic. He proposed four-level-hierarchy for evaluating the load of computing nodes: idle, low, normal and high. To assign a load-level to a particular computing node, first the average workload in the cluster has to be estimated. Afterwards, the load of the computing node is compared to the average value and on this basis a load-level is assigned to the node. A different strategy has been introduced by Fonlupt et al. [21], in this approach the load of a processor is measured by the data it owns. The total load of the system is estimated as the sum of load of all processors participating in load-balancing (which in this case are all data owned by participating processors). As previously reported by Regina et al. [22]

two groups of load metric can be distinguished: generic and specific ones. Discussed above metrics were examples of generic load indices. Applying such a metric to the CMS data acquisition system would not result in achieving desired objectives. Generic load metrics are designed to rate a single computing node's performance, and not, as it is needed in case of CMS experiment, to rate whole computing farm's performance. Nevertheless, there are some analogies between above mentioned metric's requirements and those in CMS DAQ system. Those analogies may be used while creating a specific load-metric dedicated for the discussed system (as described in section 5.1).

2.3 Dynamic Load Scheduling

There are numerous, both generic and dedicated strategies for scheduling the incoming workload in distributed computer systems. A large fraction of the currently conducted research concentrates on cluster based web-systems [23] [24] [25]. In this section we aim to focus on generic algorithms that are most interesting from standpoint of our studies. Therefore, we concentrate on scheduling multisource loads, divisible loads and distributed stream processing.

There are two main concepts for generating schedules for multisource workloads: a strategy based on superposition and an approach referred to as network partitioning. [26] In the superposition strategy all computing nodes are assigned with multiple fragments of workloads from numerous load sources accordingly to their computing power. The main drawback of this method is due to the need of additional communication between nodes, which as a result may lead to large overheads and difficulties in exercising control. Network partitioning, in turn, involves partitioning the whole network into disjoint areas, of equivalent computational power, corresponding to each load source. The idea is that each source will only send its workload through its own network area. This way a source node may dispatch the workload independently of other sources. Unfortunately, solving the problem of partitioning the network involves finding an optimal spanning tree of an arbitrary graph, which is proven to be NP-hard [27].

Jia et al proposed in [26] a dynamic load scheduling algorithm for multisource loads that follows the network partitioning concept. The algorithm is designed for a system of m load sources and

n processing nodes. Each of the load sources has an independent workload inflow, and also participates in load processing. The partitioning mechanism takes into account the communication time between source nodes and processing nodes. Each processing node is assigned to the source for which the communication time is minimal. This way, m regions are created, each of them being the shortest path spanning tree. Subsequently, the source with the smallest workload processing time t_{min} is determined. The source nodes distribute the workload fraction inside their regions, in such a way that the expected processing time in each region is t_{min}. From that point on, all incoming workloads are queued in source's buffers. After the distributed portion of workload has been processed new partitioning is carried out, and again the source with lowest load processing time is being determined. This process is repeated until the entire workload is consumed. Note that after a load processing cycle is finished the source that has been previously recognized as the one with lowest processing time t_{min}, not necessarily needs to own any further workload. In this case, the resources assigned to this source, along with this source node itself, will be reallocated to other regions. On the other hand, if an idle source found itself in possession of new workload a new region will be created. As a result the number of sources and regions may fluctuate. The discussed algorithm is difficult to apply for a HEP data acquisition system because it assumes that the data inflowing in different sources are not related. The algorithm also implies that the source nodes have unlimited buffers, which could turn as a drawback due to the huge amount of data acquired at high rate by the HEP experiments. Nonetheless, the discussed scheduling strategy has some properties and mechanisms that are considered as desirable for our system. Moreover, the algorithm analysis method presented in [26] is very interesting.

Yu and Robertazzi, in turn, proposed in [28] a dynamic load scheduling strategy for multisource loads that follows the idea of superposition. Similarly as in electric circuit theory, they conduct the workload distribution analysis for each load source separately, as if other sources would not exist. Subsequently, the single-source workload has been assigned to available computing nodes proportionally to their computing power. Since the Divisible Load Theory is linear [29], and there is a pre-assumption made that the load is indeed arbitrary divisible, and that the inflowing workload in different load sources is not related, the solutions obtained for each

source can be superimposed algebraically. As a result, the multi-source workloads have been allocated to computing nodes proportionally to their computing power. What is more, the amount of workload that has to be transferred between nodes is minimized. As already mentioned, the discussed solution is unfortunately meant for load sources that deliver unrelated workloads, and as a result cannot be directly applied to a HEP data acquisition system.

Another branch of load scheduling, which is important for our research, is distributed stream processing. A stream is a potentially unlimited set of continuously incoming data produced by a data source [30]. In contrast to the other workload types, the stream data are produced in real time. What is more, the stream data are feasible only at a given point, only for a short moment, and as a result the stream itself as wholeness is not available. Broberg et al proposed a set of three algorithms for addressing the problems of: feasibility, maximally proportional throughput of output streams and best allocation of resources in [31]. The load scheduling strategy for a continuous flow of loads (feasibility) will be regarded as stable if the whole number of loads staying in the system remains limited. In order to fulfil this condition, an algorithm has been presented that depending on a potential function's outcome balances the workload in a distributed way using backpressure. A potential function corresponding to each queue is defined. The arguments for such a function are the heights of the queue. Each computing node assigns the workload in such a way that the potential of its outgoing queues is kept as low as possible. In order to find the maximally proportional throughput of output streams, the maximum concurrent number of flows has to be found (the input load at respective rate has to be feasible). The optimal value can be found by adding bisection search to the original algorithm. To address the problem of best allocation of resources the algorithm needs to be further extended by adding buffer control (maximum allowed queue height). Flows that exceed this newly added restriction will be deleted and the optimal weighted throughput will be obtained. Although the discussed multi-commodity flow algorithm has been extended by an available computing power constraint, the main concern of multi-commodity flow modelling is still to achieve maximal flow between sources and sinks. The issue of fault tolerant stream processing has been addressed in [32] and [33] by Gorawski and Marks. The aim of their

research is to provide continuity and reliability of an ETL [34] process conducted on data streams from multiple sources. The studied system remotely and automatically reads out media consumption meters (electricity, water, gas). The data are first acquired by so called collecting nodes, and then are passed further to telemetric servers. Subsequently, in order to facilitate the data analysis (e.g. predict media consumption), those data are transferred into a stream data warehouse. To ensure that a stream has not been corrupted while being processed, it is being replicated first (redundantly transmitted) and then processed in parallel on several nodes. The obtained outcomes are then compared: the timestamp analysis allows to identify missing tuples and the attribute analysis makes it possible to detect processing errors. A stream is being considered as reliable if more than half of the analysed tuples were confirmed as errorless by other replica streams. The parallel telemetric data warehouse described in [35] is a system that handles the data incoming from the telemetric servers. The warehouse is composed of a set of computing nodes that do not share any resources and are not homogeneous. Every query served by the system is passed further to each of the computing nodes in an unchanged form. Subsequently, the nodes are carrying out the requested operation on their subset of data. In order to achieve the optimal performance the data that are being loaded from telemetric servers are allocated in such a way that each worker node should finish its part of a query-task at the same time.

Lin et al gave us in [36] an in-depth analysis of a few generic, real time load scheduling strategies. The scope of their work concerns algorithms that are a combination of studied scheduling, node assignment and task partitioning policies. When it comes to the scheduling strategy, three possibilities are considered: First In First Out (FIFO), Earliest Deadline First (EDF) and Maximum Workload Derivative First (MWDF). The FIFO method assumes that the loads will be processed in the order of their arrival. The EDF algorithm, in turn, schedules loads by their deadlines. The MWDF policy is meant for divisible loads and adopts the following rule: the costliest workload is ordered first. As for node assignment, two strategies are analysed. The first one allocates all available computing nodes to the scheduled workload in order to process it as soon as possible. The second one allocates the minimum number of computing nodes needed by the scheduled workload to meet its deadline. This way, other resources are saved

for new, incoming workloads. Likewise, two partitioning methods are studied. Optimal Partitioning Rule (OPR) results from divisible load theory, and aims to ensure that all computation finish at the same time. On the other hand, Equal Partitioning Rule (EPR) addresses the partitioning problem by dividing the scheduled workload into *n* possibly equal sub-loads, where *n* is the number of allocated computing nodes. The main conclusion of the discussed research is that the algorithms adopting OPR strategy achieve lower load reject ratio and therefore outperform the corresponding algorithms. This confirms the working hypothesis that it is beneficial to adopt divisible load theory for real time scheduling in cluster environment.

2.4 Load Balancing

Although load balancing is not the main topic of this thesis, it is closely related to load scheduling, and therefore it is also crucial to present here a brief load balancing overview. In the following section we focus on algorithms adopted for SPMD and distributed stream processing, as well as on solution for achieving asynchronous, distributed load balancing. These topics seem to be most analogous to the subject of this dissertation.

Thome et al [37] gave us a compact overview and comparison of load balancing strategies used in a SPMD systems adopted for computing macroscopic thermal dispersion in porous media. The discussed research focuses in particular on the mechanisms used for triggering load balancing activities, as well as on workload indices communication. For us, the most important outcome from the presented analysis concerns the approach for gathering internal load indices and for workload redistribution. It has been shown that the global, collective load balancing led to the best results. The global strategy implies that the load data should be gathered from the whole system at once. The collective strategy in turn, implies that load balancing should lead to exact workload redistribution in the whole system. Using these strategies provides the fastest reaction to imbalance in the system. Another interesting fact is that algorithms using these strategies obtained almost identical results for distributed and centralized load balancers.

Osman and Amar [38] also proposed a load balancing algorithm for the pipelined SPMD computational model. Although, the discussed data processing model has many advantages,

especially when it comes to scientific applications, it also has a major drawback. Namely, presence of a slower computing node in the system results in a slowdown of other computing nodes. The load balancing strategy introduced in [38] is distributed by which we mean that there is no central workload scheduler. Moreover, the algorithm is asynchronous in the sense that the load balancing activities carried out in a computing node are autonomous and there is no need for synchronization with other computing nodes. The initiation of load balancing activities, in turn, is triggered by the overloaded node. The algorithm is design with a particular emphasis on scalability, and therefore the decision process is performed based on exchange of local workload indices. This way the communication overhead that usually increases with the number of computing nodes in the system, is reduced. The load index is expressed in a unit called *data point*. A *data point* is the smallest, distinguishable amount of data that can be defined by requiring a given number of computational operations and storage space. Based on the measurement of *data points* allocated to a computing node, the node is assigned to one of the three categories: overloaded, normal and underloaded. When a computing node reaches the overloaded state the load balancer is triggered. First, the load balancer queries its neighbours to see how many *data points* can be accommodated before the underloaded neighbours reach normal state. If the neighbour nodes can accept the extra workload without reaching overloaded state the data are transferred. Otherwise, the overload is averaged between all underloaded and normal computing nodes. In this case, calculation of the workload to be exchange is crucial, because the overload transfer may result in bouncing-load effect: the workload receiver becomes overloaded itself, and as a result it decides to transfer the load back to the load originator, which in turn becomes overloaded and performs again the same load balancing activities, and so on. The described bouncing-load effect generates a substantial overhead and also prevents the system from reaching the balanced state. The discussed distributed, asynchronous strategy is especially interesting from the standpoint of our research. In case of a multi load source system applying such a strategy may result in substantial reduction of the overhead of the workload distribution algorithm. Moreover, the proposed load index (*data point*) seems to be a natural choice when it comes to SPMD computing model.

Cherniack et al [39] discuss two stream processing systems: Aurora* and Medusa. Both architectures are designed to support large scale, distributed stream-based computations. Aurora* provides infrastructure for a system where all computing nodes are subordinated to a single administrative domain. On the other hand, Medusa assumes that there are several administrative boundaries, and therefore supports federated operations. Nevertheless, both architectures have to address the problem of workload management. The case of Medusa is slightly more complicated since it implies that the computing nodes not necessarily need to be under common control. In both cases, an overlay network is defined that is independent from the physical network topology. Each event produced by a data source is labelled with a stream name, and then transferred to one of the computing nodes in the overlay network. A load balancing daemon is invoked periodically on each node. The daemon is responsible for offloading its machine or accepting additional workload depending on its node state. Since in case of stream processing, the workload is related to a constant and continues data flow, rather than to some computational tasks running on single nodes, load sharing is achieved through overlay network repartitioning. There are two main workload sharing policies: *box sliding* and *box splitting*. *Box splitting* is a heavier and more complicated operation since it has to be ensured that the result before and after splitting will be the same. It is crucial to choose a suitable frequency of load balancing activities so that the daemon will be able to handle not only the incoming workload changes, but also the changes in the overlay network topology. It is also critical to take into account the bandwidth availability before repartitioning of the overlay network takes place. Finally, it is important that a sub-network split has a long-lasting effect. Besides load management the discussed architectures have also interesting fault tolerance properties. There is a heartbeat mechanism implemented between neighbour nodes. If a computing node timeouts on the heartbeat of its neighbour it triggers a recovery procedure. The backup node starts to emulate the processing of the failed computing node. Subsequently, the load balancing mechanism is used to offload the backup node.

2.5 Data Acquisition in High Energy Physics

It is not always possible to adopt the general workload balancing/scheduling strategies discussed so far in the data acquisition systems of high energy physics experiments. In this section dedicated algorithms are presented, together with a short introduction to the data acquisition systems and experiments they are employed for.

2.5.1 LHCb experiment at CERN

LHCb is one of four experiments at CERN's new LHC. [40] It seeks to discover new phenomena, in particular CP-violation in B-decay and other new rare decays. LHCb expects 10 MHz rate of visible interaction and 100 kHz of b-anti-b-pair production. LHCb employs two level trigger selecting only the interesting events. The first level is a low-latency high-rate trigger implemented in FPGAs and carrying out the task of reducing the incoming data taking rate from 40 MHz to 1 MHz. The second one, the software trigger (HLT) is implemented on a CPU filtering farm and performs further reduction to 2 kHz. The LHCb readout consists of detector-specific front-end electronic boards connected to a group of 320 Readout Boards (total bandwidth of 4 Tb/s). A router with the strongest density of Gigabit Ethernet switching within reach guarantees full connectivity with the filtering farm and a data throughput of 35 GB/s. A single Readout Board holds only a fragment of data describing an event. Therefore, event building and HLT selection is carried out by transferring all event fragments to the same filtering node in the filtering farm. The expected event fragment size is 120 Bytes. In order to improve network utilization a packing of event fragments into Multi-Event Packets (MEP) is performed (typical packing factor is ten). The communication between Readout Boards and filtering farm is provided by UDP protocol and the packet loss detection (very rare) is implemented on event building level. The whole process of event building and filtering is supervised by a single entity called Readout Supervisor (implemented in FPGAs). Among others, the Readout Supervisor assigns beam-synchronous clocks and synchronous resets, and receives back-pressure from Readout Boards. Although the Readout Supervisor is designed with a particular emphasis on reliability, it appears to be a potential single point of failure. There are several spare Readout Supervisors normally used for concurrent standalone runs with sub-detectors. In order to

activate adding event fragments to the current MEP the Readout Supervisor broadcasts a Trigger Type command. The Destination command in turn, closes the MEP that is under construction and also contains the IP address of the destination node in the filtering farm. The destination node is assigned depending on a credit scheme. Credit corresponding to each destination node acts as a counting semaphore. Initially, at the beginning of the run, as well as after processing of each MEP, each filtering node asks for new events throughout sending a MEP Request to the Readout Supervisor. The credits corresponding to a filtering node are incremented based on the MEP Requests and decremented whenever the corresponding node is used as the destination for the next MEP (push mode with passive pull mechanism). If a credit is zero or negative the destination is skipped but the credit is still decremented. This way dynamic load scheduling of the filtering farm has been obtained. This strategy is also very convenient for detecting failing nodes because credit corresponding to such a node becomes increasingly negative.

It can be noticed that the central agent policy that has been adopted for Readout Supervisor increases the number of single points of failures rather than decreases it, which is our objective. Moreover, LHCb data acquisitions system operates on much smaller event fragments than CMS (more than ten times smaller) which allows for packaging during data transfer and results in less time consuming filtering. There is also a possibility that the policy of requesting new events only after the whole MEP is being processed will result in idleness.

2.5.2 ATLAS experiment at CERN

ATLAS is one of two general purpose experiments at CERN's new LHC. It is designed to discover the same phenomena as CMS, and therefore it has to fulfil similar design conditions. ATLAS is supported by a large, distributed trigger and data acquisition system (TDAQ) that employs three stages of event filtering to reduce the initial data acquisition rate of 40 MHz to a rate of stored events of 200 Hz. [41] The first level trigger that is implemented in dedicated custom hardware, is responsible for the first selection step whose output will be in the order of 100 kHz. Subsequently, the trigger information is built by the Region of Interest Builder, and then assigned in round robin fashion and transmitted to one of the second-level trigger

supervisors. This entity acts as a central agent that performs the load scheduling of the incoming events between processing applications running in the second level trigger sub-farm. A processing unit requests partial event data from the Readout System, runs a filtering algorithm, and passes the result back to the second-level trigger supervisor, which in turn sends the information to the DataFlow Manager (DFM). The pre-filtering step done on the basis of partial event information causes further data rate reduction down to 3 kHz. The DFM is also a central load scheduling agent, who supervises the event reconstruction process. For each accepted event the DFM allocates an event-building node according to the pull-requests obtained from those nodes. This way, a demand driven load scheduling has been obtained. Afterwards, the event-building node requests all event fragments, assembles the whole event (expected event size is 1.5 MB) and transfers it to the Event Filter that performs the final selection step and reduces the output rate to final 200 Hz. It can be easily noticed that it was possible to adopt central agent policy only because an additional filtering step has been introduced. However, applying this policy to the CMS data acquisition system would rather increase the number of single points of failures than decrease it, which is our goal. Moreover, in the very first step data distribution is done in a static way using a round robin algorithm.

2.5.3 DZERO experiment at Fermilab

The DZERO experiment at Fermilab's Tevatron is one of two experiments designed to investigate high energy proton-antiproton collisions. The data acquisition system of the DZERO experiment [42] (similar solutions are also used in Zeus [43] and CDF [44] experiments) handles an incoming data rate of 2.5 MHz. The filtration of acquired data is carried out in three stages. The first level trigger, implemented in custom high speed electronics, performs the filtration based on simple criteria. As a result the first level trigger has an output rate of about 5 kHz. The second level trigger, composed from custom electronics, as well as from generic processors, assembles partial information from detector subsystems and executes more precise selection algorithms to further reduce the rate to 1 kHz. The Level three data acquisition and trigger system is built around a single CISCO 6509 switch. The data sources are connected to the central switch through five CISCO 2948G switches (up to 20 data sources per switch). Each data source provides event fragments containing up to 20 kB of data. The event building and filtering

is supervised by a single process called Routing Master running on a single board computer. The Routing Master chooses the destination using a table containing the information about the number of free buffers on each farm node. First, a set of the least loaded nodes is identified, and then the destination node is chosen in round-robin manner. After assigning an event to a farm node the corresponding entry in the table is decremented. Farm nodes update the table entries periodically, through messages with the number of available buffers. The farm node assembles the whole event (average event size is 250 kB) and then runs sophisticated filtering algorithms in order to finally reduce the rate to 50 Hz. Potentially, the load scheduling solution applied in DZERO level 3 data acquisition could be adopted for higher data taking rates provided that the assignment decision would be made and distributed for bunches of events, which is possible since the load scheduler is aware of the space available in farm node's buffers. However, the Routing Master is an additional single point of failure.

2.6 Fault Tolerance

At this point we aim to focus on fault tolerance issues. We will discuss cluster environments with self-stabilization properties and frameworks that combine their fault tolerance features with workload scheduling/balancing utilities. Moreover, we will also consider fault tolerance mechanisms employed in the data acquisition systems of high energy physics experiments.

Sevilla et al analysed in [45] the impact of applying aspect oriented programing paradigm (particular emphasis is laid on software modularity) to High Performance Computing. They propose a framework for automated code generation called CORBA-LC that is an extension of standard Common Object Request Broker Architecture (CORBA). The framework, amongst others, facilitates implementing load balancing and fault tolerance. The fault tolerance, in particular, is achieved by replicating a given component on a set of computing nodes. Every time an action is called on the fault tolerant component, a set of threads (each of them corresponding to one instance of the component) is used to communicate all component replicas distributed over several nodes. Subsequently, failed nodes are detected and a voting is performed in order to determine the computing node that will carry out the submitted task.

Load balancing is provided in a similar way, the module whose operations should be balanced between several nodes in the cluster is replicated. In this case, however, the worker node is chosen according to the least loaded policy. It can be easily noticed that the two discussed attributes are implemented in a similar way, and therefore it is easy to provide components that are both fault tolerant and load balanced, and that they performance will not differ significantly from components that have only one of those attributes.

Another example of combining load balancing with fault tolerance was proposed in [46]. The discussed strategy is designed for Peer-to-Peer (P2P) systems, which are mainly used for sharing files, streaming multimedia and hosting social networks. The idea behind the P2P model is to allow users for registering voluntarily their hosts as a P2P node. Once the machine becomes a P2P node, it may use the facilities provided by the P2P system. The O-Ring architecture introduced in [46] adopts the ring topology in order to enhance and facilitate load balancing and reliability of the overall P2P system. The participating P2P nodes are organized in an overlay ring network, each of the nodes in the network shares part of its load (e.g. data) with its predecessor and its successor. A P2P node manages its own workload and also due to load sharing it is assigned with replicas of partial workload of its neighbours. These replicas are used in case of fault occurrence, but also they are used during load balancing. This way, the overhead of load balancing due to load transferring has been reduced. Short term fluctuations, in the workload of given a P2P node, are balanced by passing some of the node responsibilities to its neighbours. This involves only some changes in the request routing and does not require sending of data. A major advantage of this strategy is that it is easily reversible. Long-term, big fluctuations, in turn, can be addressed by propagating the replicated data over the system. This operation can be performed in the background without involving the overloaded P2P node itself that is responsible for handling requests concerning the overload. On the other hand, in the case of fault occurrence, the neighbour nodes can take over the responsibilities of the failed P2P node. In order to do so, the neighbour nodes extend their own workload by the replicas of the load of the faulty node. Subsequently, the newly obtained load needs to be replicated in the successor/predecessor node of the neighbour nodes. If there is an idle, spare peer in the ring it can be used as a replacement for the failed node in the ring. Otherwise, the ring has to

be closed by cross-replication of the data between the neighbour nodes themselves. In this case, it is very likely that the neighbour nodes will be overloaded, and as result load balancing actions will be necessary. This, in turn, confirms that combining load balancing and fault tolerance mechanisms is advantageous.

As it has been shown in the above discussed examples fault tolerance and load scheduling/balancing complement and enrich each other. Moreover, both fault tolerance and load scheduling/balancing require similar resources, and therefore the overhead caused by adding load scheduling/balancing features to a fault tolerant system (or vice versa) is relatively small.

Flatebo et al observed in [47] that dynamic load scheduling algorithms have self-stabilizing features. Self-stabilization, first introduced by Dijkstra in [48], is a system property such that regardless of the system's initial state (in particular illegal state), the system will stabilize itself to a legal state in finite number of steps. In the proposed algorithms the system is considered as being in an illegal state if a new workload is received, processing of an old workload has been completed or in case of error occurrence, i.e. node or link failure. [47] The algorithm 1 is triggered only by newly received workload, the receiver node first checks its neighbours, and then evaluates their workload. Subsequently the load is transferred to the least loaded neighbour node. The scenario is repeated until the workload reaches the least loaded node in the system. In case if the receiver node is one of the least loaded nodes, the workload is processed on the receiver node. This is a potential drawback when it comes to systems where one particular node is the only source of load. The second, more sophisticated, algorithm assumes that each computing node has three variables used for monitoring the system and for deciding about state changes and workload transfers. These variables are the lowest known workload in the system *L*, id *info* of the node that has the most up to date information about *L*, and the id *P* of the least loaded computing node. A node's state is being updated either because of receiving new workload or because of processing of some part of current workload is completed. In both cases this can lead to updates in state variables of neighbour nodes, which is also considered as a state update. When a computing node receives new load, it is first tested whether the receiver node itself is the least loaded node. In order to do so, the values of the

above mentioned variables are taken into account, as well as the states of the neighbour nodes. If it is the case, the receiver node starts to process the newly assigned workload. Otherwise, the workload is transferred to the computing node indicated by *info* variable, which subsequently executes the algorithm again. This continues until the workload is transferred to the least loaded computing node. When, in turn, a computing node finishes processing a workload and it becomes the least loaded node the information is propagated over the system using the *L* variable. The system is considered as being stable if the state variables of all the computing nodes are set to the same value. Both algorithms 1 and 2 have a distributed workload scheduler. It Is also worth mentioning that the algorithm 2 is an optimal self-stabilization algorithm.

We will discuss a dedicated fault tolerance solution for a data acquisition system based on the research presented in [49] and [50]. The proposed method is studied for the BTeV experiment, which is a high energy physics experiment, and is meant for a Real Time Embedded System in general. The data acquisition system and high-speed trigger of the BTeV experiment are designed to collect data at 500 GB/s, and then, after the selection of interesting events is done, to send data to mass storage at 200 MB/s. The idea is to create a subsystem that will be responsible for handling faults and errors. This subsystem should carry out local actions that correspond to a single computing node or application, like changing thresholds, and global actions related to the overall system such as load shifting. The two mainly employed technologies in this project are ARMOR (Adaptive, Reconfigurable, and Mobile Objects for Reliability [51]) and VLA (Very Lightweight Agent [50]). The ARMOR is a multithreaded, event driven solution organized around objects that are being supervised. The ARMOUR approach implies one Fault Tolerance Manager (FTM) per system that initializes the reliability features and triggers the recovery procedure from faulty states (e.g. node failure). Robustness of the FTM itself is provided by a heartbeat mechanism (heartbeat AMOR). Moreover, there is one daemon per computing node responsible for inter ARMOR communication, and several objects that provide monitoring services. The VLA method, on the other hand, follows the idea of submission architecture [52] and highly supports reactive behaviour. The agents responsible for fault tolerant features are organized in layers and their actions are determined by finite state

machines. The main advantage of this approach is that several rules can be triggered in parallel. Potential conflicts between such concurrent activities are solved using a set of priorities. In general the layer of the agent determines the priority of its actions (higher layer means higher priority). Moreover, the lower the layer of the agent, the more lightweight the agent should be. In order to provide reliability, a VLA agent may perform corrective and preventive actions, request more data from agents within its layer, or pass the problem to higher layer. Conclusions on the state of a computing node are made, amongst other, on the basis of the CPU workload, e.g. a low load may indicate that there is a CPU or network problem, on the other hand, a high load may indicate that the workload limits are badly evaluated. At this point it is important to notice that workload measurements are a key even for a heavily integrated fault tolerance solution.

Our goal and priority is to provide fault tolerance in such a way that the number of additional components and their impact on the system is minimized as much as possible. Therefore, only some specific attributes that will allow for concluding about the entire system, have to be monitored. Also, the additional components have to be as lightweight as possible. As a result, in our case, it is not possible to apply fault tolerance mechanism that requires detailed monitoring of the system at several levels.

3 Case Study: The Compact Muon Solenoid Data Acquisition System

As a case study we consider the Data Acquisition (DAQ) System of the Compact Muon Solenoid (CMS) experiment at CERN [53] [54]. The discussed DAQ system is designed to sustain a maximum data taking rate of 100 kHz of 1 MB zero-supressed events coming from approximately 512 sources that corresponds to an average input of about 100 GB/s. Each data source is expected to deliver event fragments of average size of 2 kB (in some case two sources are merged so that the nominal size is obtained). The system has to carry out software filtration of the incoming events to reduce the rate of stored events by a factor of 1000. In order to reach the desired rejection factor substantial computing power is needed that corresponds to thousands of computing nodes. The configuration of the CMS Data Acquisition cluster is carried out dynamically at run-time so the DAQ System may adapt to required performance and also can be partitioned in order host concurrent test- and data-taking runs [55].

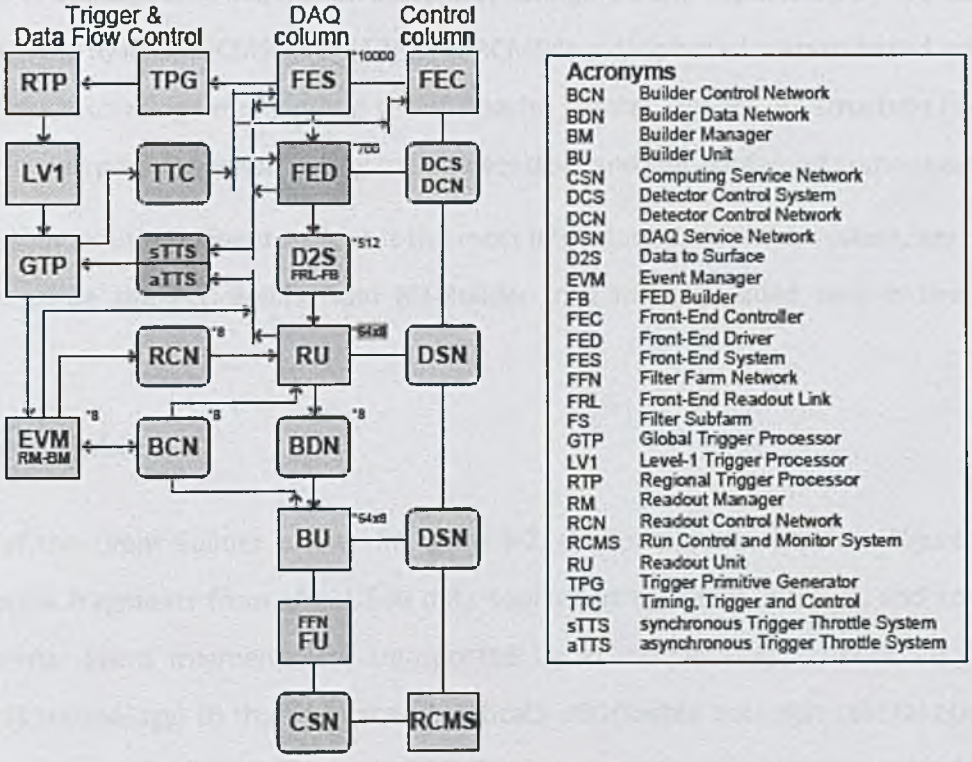


Figure 3-1 Functional decomposition of the CMS DAQ System (for clarity, multiplicity of each entity is not shown). [2]

The component architecture of the CMS DAQ system, along with the data flow is shown in Figure 3-1. The sub-detector front-end systems store data constantly in 40 MHz pipelined buffers. For accepted events, a L1 trigger signal is delivered through the Timing, Trigger and Control (TTC) system. The data of selected events are transferred from buffers to the Front-End Drivers (FEDs), and then are read into the Front-End Readout Links (FRLs) that, as previously mentioned, are able to combine data from two FEDs and act as the data sources for the CMS DAQ system. Afterwards, the data fragments provided by FRLs are assembled to whole events by the Event Builder, and then are passed to the Event Filter for further processing. The Event Builder is composed of a so-called FED-Builder and RU-Builder, which, in turn, is divided into several autonomous computing farms called DAQ Slices (the RU-Builder can be deployed in up to 8 DAQ Slices). [11] In case of congestion, back-pressure is propagated from the RUs to the FRLs and then to the FEDs. FEDs in turn, in order to avoid buffer overflows, may throttle the trigger rate through the Trigger Throttling System (TTS).

The whole process of data acquisition is started, configured and supervised by the Run Control and Monitoring System (RCMS) [56] [57]. The RCMS is a distributed system based on Java and Web Services technology running in a set of Apache Tomcat servers. Its structure is organized into several subsystems corresponding to sub-detectors and self-contained components.

From our standpoint the Event Builder is the most important part of the system, and therefore we will describe the FED-Builder and RU-Builder in a more detailed way in the following sections.

3.1 Event Builder

The goal of the Event Builder (shown in Figure 3-2, also previously shown in Figure 1-3) is to acquire event fragments from about 500 data sources at a rate of 100 kHz, and to construct whole events. Event fragments are transported by a non-blocking network [5] (based on Myrinet [6] technology) to the surface and statically distributed amongst several autonomous computing farms called DAQ Slices. A DAQ Slice is a computing farm organized around a Terascale Force10 switch, where parallelization is achieved through the SPMD (Single Process, Multiple Data [7]) technique. In the first event building stage (FED-Builder level) event

fragments are received by a DAQ Slice through distributed readout consisting of computing nodes called Readout Units (RU), and then assembled into super-fragments inside RUs. Subsequently, in the second stage (RU-Builder level), in each of the DAQ Slices, an Event Manager (EVM) node assigns super-fragments to Builder Units (BU) that construct the whole event. It has to be ensured that all data fragments corresponding to one event are sent to one and only one DAQ Slice, and then after assembling to super fragments go into one BU. Since it is assumed that each event requires similar computing power in order to be processed, the events are usually distributed in round robin fashion between DAQ Slices, which are identical in the sense that they consist of the same number of identical computing nodes. For this reason a non-blocking network has been employed, so the transfer of event fragments to one DAQ Slice is non-blocking in respect to other DAQ Slices. This way, each DAQ Slice has not only the same computing power, but also access to the same bandwidth. Furthermore, the round robin assignment policy can be carried out in constant time, which is undoubtedly an important asset, since DAQ Slice allocation has to be done at least once per 10 μs on a RISC processor with clock speed in order of 10 MHz.

3.1.1 FED-Builder

The FED-Builder is composed of multiple $N \times M$ networks that carry out the task of building super fragments from event fragments obtained from N sources and distributing them between M DAQ Slices. [58] For the standard system $N \leq 8$, and $M = 8$. Physically the FED-Builder network takes form of a non-blocking network based on Myrinet [6] technology. Myrinet is a network solution composed of Network Interface Cards, containing user-programmable RISC processors (NICs) and cross-bar switches interconnected with bidirectional fiber-optic links. In order to guarantee lossless packet transfer two mechanisms are adopted: wormhole routing and flow control. The wormhole routing [59] [60], also called wormhole switching is a technique that allows for dividing larger network packets into smaller parts, which are then sent using

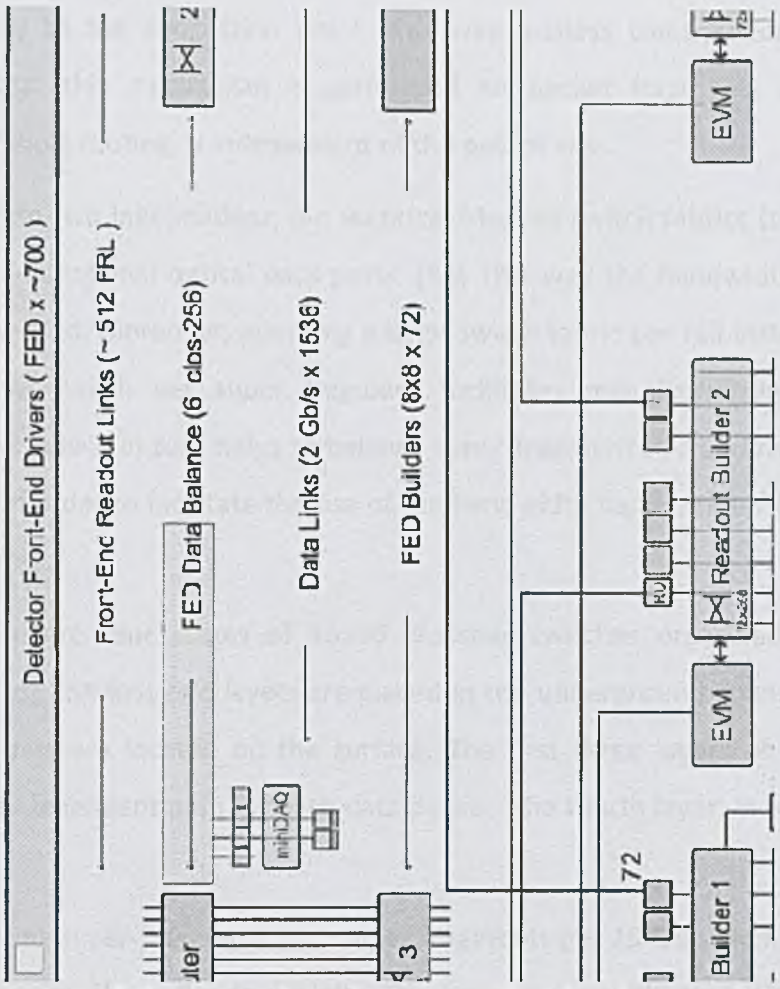


Figure 3-2 Schematic view of the CMS DAQ System [2]

common routing. The first fragment of the network packet, a so-called header contains the information about the route (namely the destination address). The routing decision is taken only once while the header is passing through the network. This way a temporary circuit is created via which the subsequent packet parts flow (therefore it may be considered as dynamic circuit switching [61]). It is worth noticing that the transmission of a given packet may be pipelined across series of devices, and as a result the destination node may receive the header before the source node finishes sending the whole packet. The last packet part, called tail is responsible for closing the connection between the sender and receiver nodes. This method successfully minimizes the latency in the message transmission and is independent of the package length. The flow control [60], in turn, aims to ensure that no buffers are overwritten at any of the stages of package forwarding and receiving. This means that in the case when it is not possible for a packet to proceed, the data source may continue to transfer data only until

all buffers are full all the way to the congestion point (this way lossless transmission is guaranteed). It is assumed that this mechanism is performed on packet fragments and therefore, similarly as the wormhole routing, is independent of the packet size.

Each Myrinet NIC is connected to two independent, but identical Myrinet switch fabrics (two-rail configuration) using two bi-directional optical data ports. [58] This way the bandwidth is doubled and redundancy is provided. Moreover, adopting a large switch fabric per rail instead of using an individual $N \times M$ switch per super fragment facilitates reconfiguration of composition of super fragments, which in turn helps to balance super-fragment size and route avoiding erroneous hardware. In order to facilitate the use of the bandwidth barrel-shifter [62] traffic shaping has been employed.

As shown in Figure 3-3, there are four stages of *16×16* crossbar switches organized as a reconfigurable Clos network [5], the first two layers are placed in the underground counting room, while the other two layers are located on the surface. The first three layers of the network provide a completely independent path for each data packet. The fourth layer, in turn,

carries out the task of constructing super-fragments (two super fragments per *16×16* switch) by sending respective event fragments to a Readout Unit. This however, entails throughput reduction due to head-of-line blocking [63] [64].

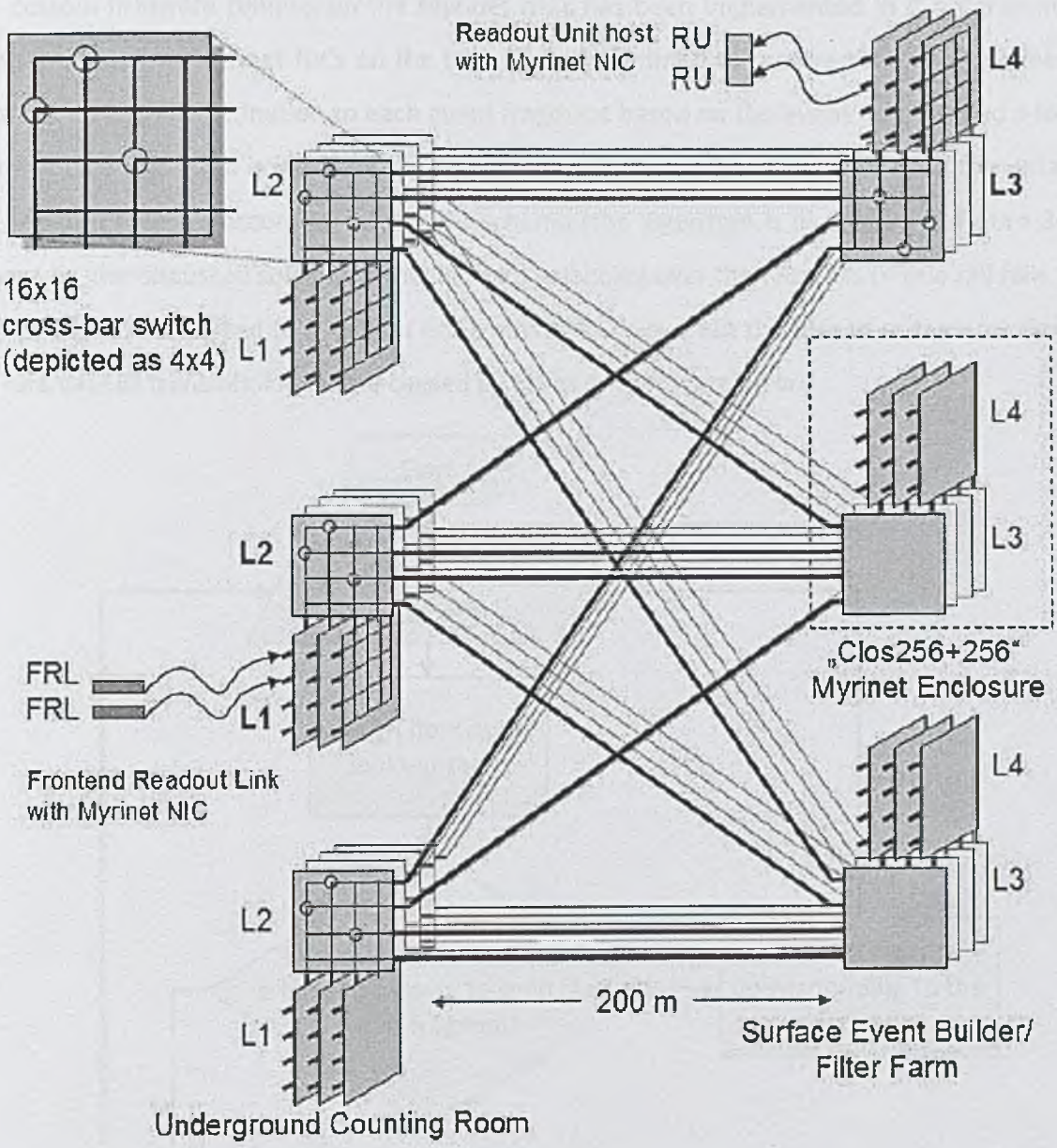


Figure 3-3 The FED-Builder non-blocking network (only one rail shown), the four stages of the Clos network correspond to L1, L2, L3 and L4. The number of crossbar switches in each layer and the number of input/output per crossbar have been scaled down by a factor of 4 for clarity. [58]

The custom firmware running on the Myrinet NICs has been implemented in C programming language. [58] The Myrinet NICs on the FRLs are programmed to receive the event fragments from FRLs, assign a destination to each event fragment based on the event number and a look-up-table (the algorithm is illustrated in Figure 3-4) and then subsequently to send the data to the selected receiver according to a credit schema (the algorithm is illustrated in Figure 3-5). Moreover, the discussed software provides load balancing over the two rails (if one rail fails the whole traffic is redirected to the other one), as well as carries out the task of re-transmission in the rare case of transmission failure caused by fibres or hardware errors.

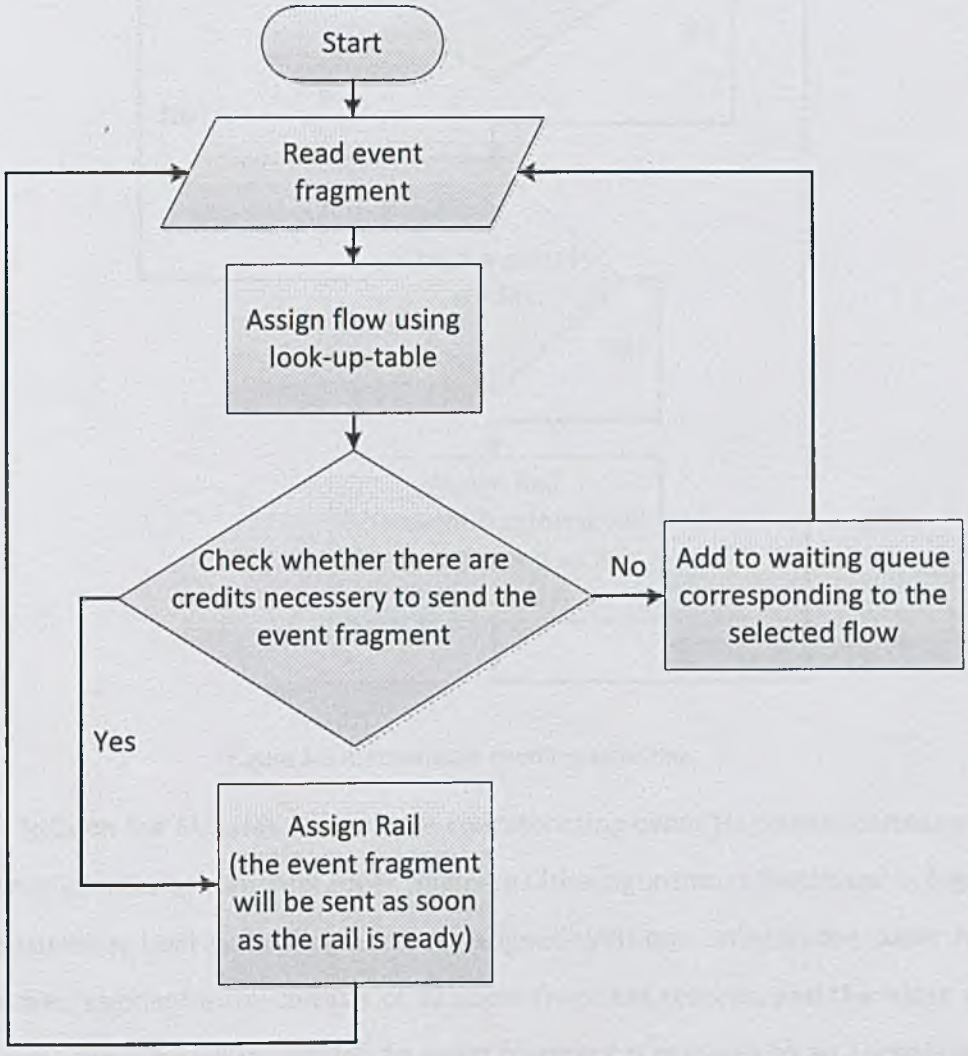


Figure 3-4 Event fragment polling algorithm used to receive fresh blocks pushed by FRL.

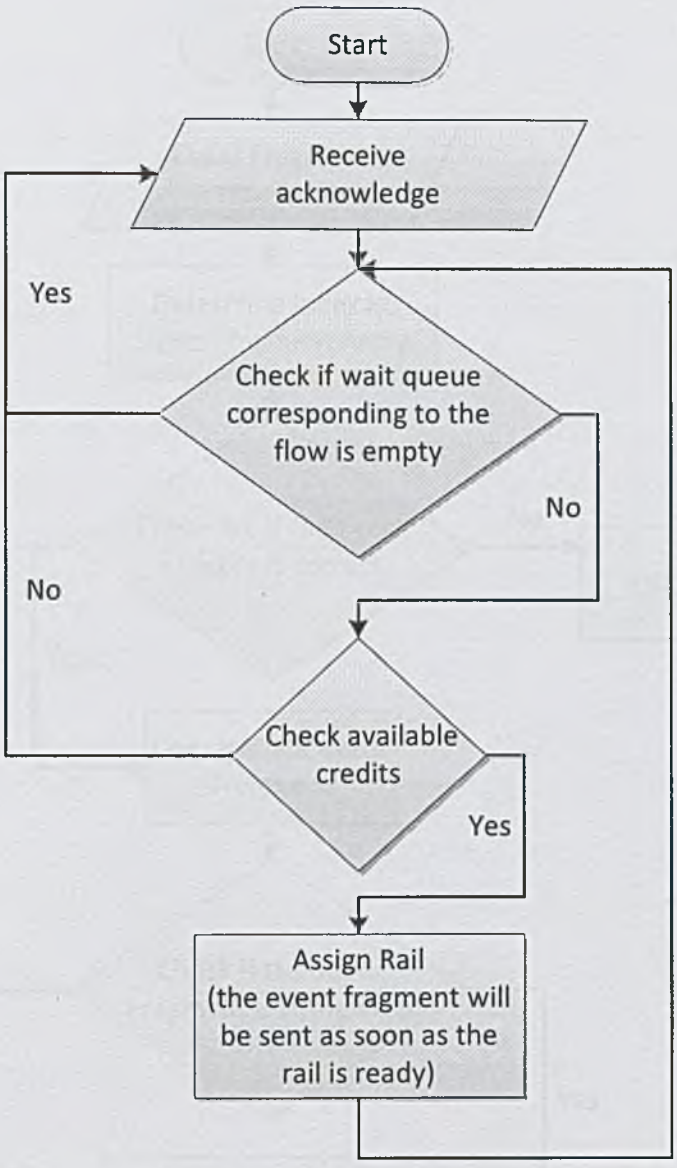


Figure 3-5 Acknowledge handling algorithm

The Myrinet NICs on the RUs side, in turn, are concatenating event fragments corresponding to the same event in order to construct super fragments (the algorithm is illustrated in Figure 3-6). Each of the currently built super fragments is assigned with one entry in the super fragment array. The super fragment array consists of 32 super fragment records, and therefore up to 32 super fragments may be built in parallel. An event fragment is assigned to an appropriate array entry based on a hash function result. The hash function employs the same look-up-table that is used to assign the event fragments to their receivers.

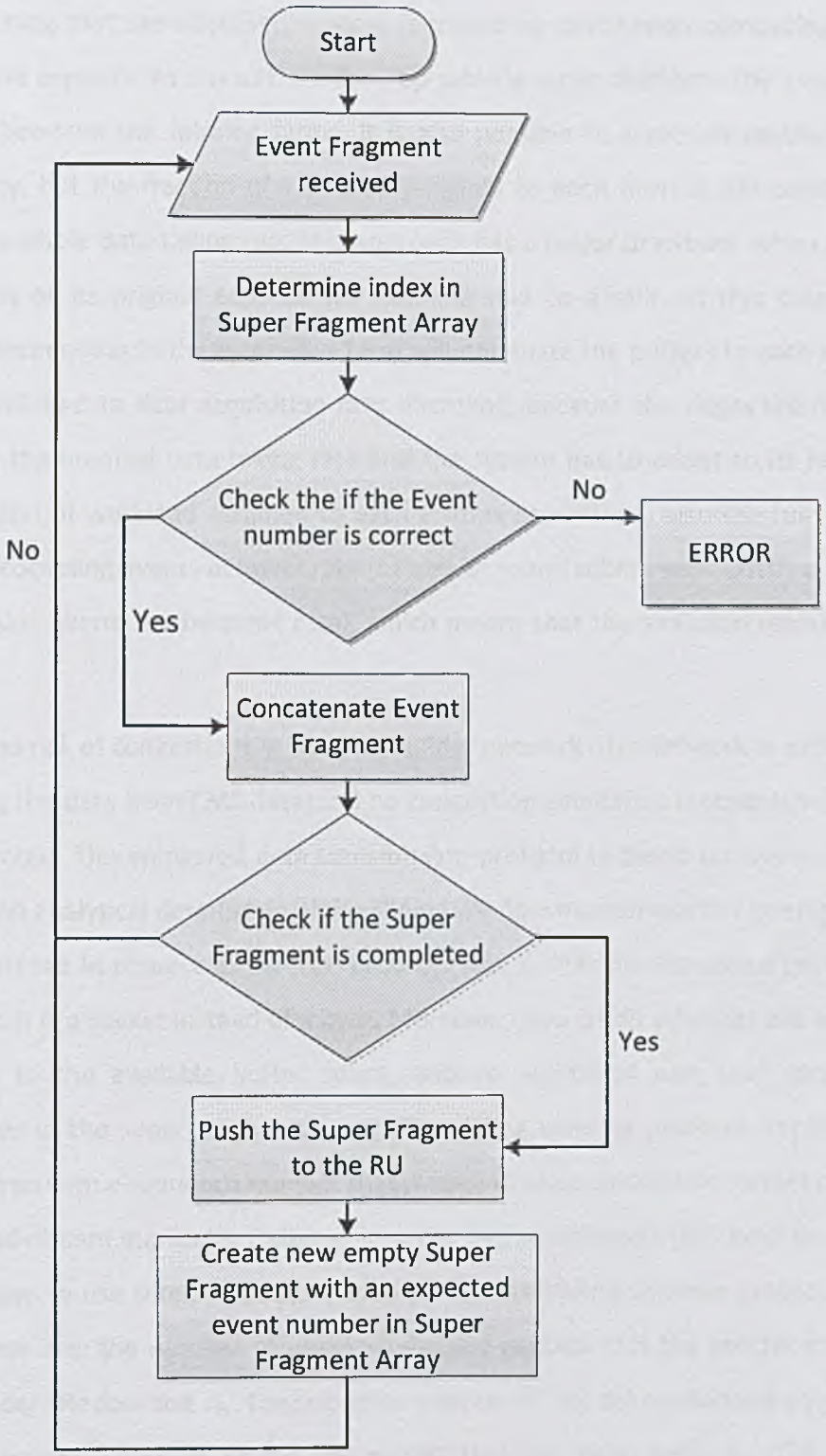


Figure 3-6 Super Fragment concatenation algorithm

It is worth noticing that the discussed strategy is meant for destination computing farms with exactly the same capacity. As a result, the look-up-table is set to distribute the events in round robin fashion between the filtering farms. It is also possible to serve computing farms with diverse capacity, but the fraction of workload assigned to each farm is still constant for the duration of the whole data-taking run. This approach has a major drawback when one filtering farm loses part of its original capacity for example due to a fault. In this case, the event fragments corresponding to the erroneous farm will dominate the buffers in each Myrinet NIC. This, in turn will lead to data acquisition rate throttling, because the degraded filtering farm cannot handle the nominal data-taking rate and the system has to adapt to its new capacity. Since the fraction of workload assigned to each farm is predefined, also the fully operational farms will be processing events at lower rate (in case of round robin event distribution all farms will be processing events at the same rate), which means that the available resources are not fully utilized.

Since there is no risk of congestions in the FED-Builder network (the network is exclusively used for transferring the data from CMS detector) no congestion avoidance mechanism [65] [66] has been implemented. The employed data transmission protocol is based on the sliding window protocol [67]. An analytical description of the sliding window mechanism has been given in [68]. The main difference in respect to the TCP protocol [69] is that the discussed protocol's basic transmission unit is a packet instead of a byte. Moreover, two credit schemas are adopted, one corresponding to the available buffer space, and an additional one that corresponds to available entries in the super fragment array. The sliding window protocol implies that each packet is assigned with a sequence number that is used to place packets in correct order, detect lost packets and discard duplicates (reliable, in-order packet delivery). [67] Both the sender and the receiver have to use three dedicated variables for the sliding window protocol. In case of the sender these are: the number of unacknowledged packets that the sender may transmit, also called sender window size s_W, the sequence number of last acknowledged packet received s_{LA}, and the sequence number of the last packet that has been send s_{LS}. The relationship between these variables is determined by the following invariant:

$$s_{LS} - s_{LA} \leq s_W$$

A packet is retransmitted if it timeout before an acknowledgement is received which means that the sender may have to buffer up to s_W packets. When an in order acknowledgement arrives s_{LA} is incremented, allowing the sender to transmit more packets. The s_{LA} variable is possibly further incremented for each out of order previously received acknowledgement provided that all packets with lower sequence number are now acknowledged. For example, let us assume that $s_{LA} = 5$, and that the acknowledgement is received first for packet *7*, and then for packet *6*. At the point when packet *6* acknowledgement arrives the s_{LA} is incremented, but it is also incremented for packet *7* since its acknowledgement is now also in order, and as a result $s_{LA} = 7$. The receiver, in turn, keeps track of the number of packets that can be received out of order, which is called receiver window size r_W, the highest acceptable sequence number r_{HA}, and the last packet sequence number received r_{LR}. The relationship between these variables is determined by the following invariant:

$$r_{HA} - r_{LR} \leq r_W$$

When a packet arrives the receiver checks its sequence number. If it is within the receiver's window it is accepted, otherwise it is discarded. An acknowledgement is not sent if the accepted packet is received out of order. However if the packet is received in order, the acknowledgement is sent for the currently highest, in order packet (cumulative acknowledgement). For example, suppose that $r_{LR} = 5$, and that the packet *7* is received first, and then the packet *6* (let us assume both packets are within the receiver's window). At the point when packet *6* is received an acknowledgement is sent, but since the highest in order packet is *7*, it is the one that will be acknowledged. After an acknowledgement is sent, r_{LR} is set to the sequence number of the recently acknowledged packet, and $r_{HA} = r_{LR} + r_W$. This way the receiver is allowed to accommodate more packets.

3.1.2 RU-Builder

As previously mentioned, the RU-Builder is composed of up to 8 autonomous computing farms called DAQ Slices. Each DAQ Slice (shown on Figure 3-7) is assigned with equal workload, which means that at the nominal data acquisition rate of 100 kHz it is building events at a rate of 12.5 kHz. The network of a DAQ Slice is implemented by one switch (Terascale Force10 switch). The super fragments are received by RU applications (there are about 72 RU nodes per DAQ Slice), and then assigned by an EVM supervisor to a BU processes (there are about 126 BU nodes per DAQ Slice), which carries out the task of constructing the whole events.

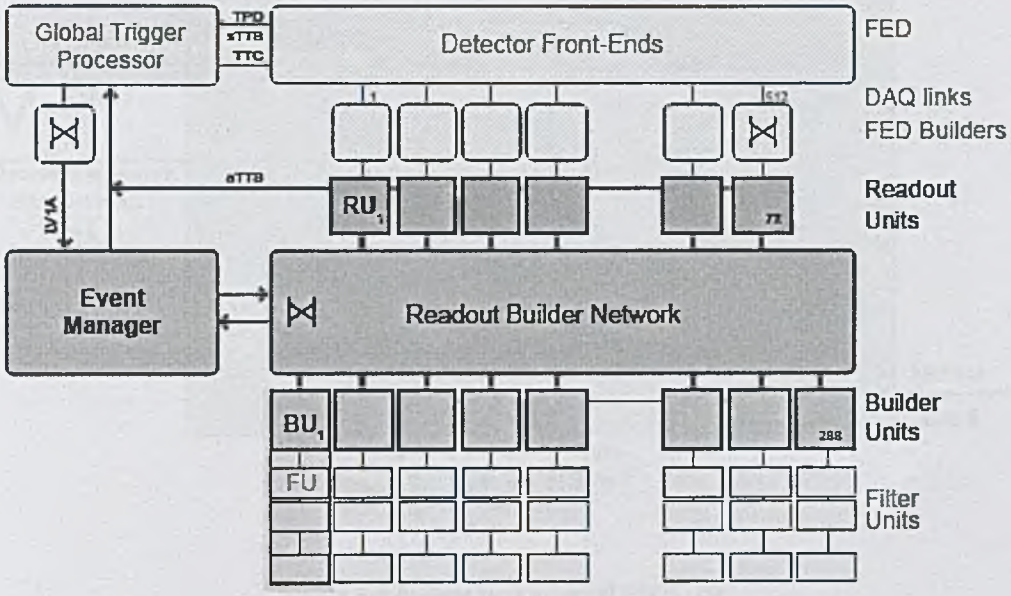


Figure 3-7 DAQ Slice schematic view [2]

At this point we would like to focus on RU-Builder applications (EVM, RU, BU), with the particular emphasis on the employed communication protocol and FIFO queues. First we will discuss the BU application, which carries out the task of building events. An event consists of one trigger super fragment (obtained from an EVM) and *n* RU super fragments, where *n* is the number of RUs. Afterwards we will present the EVM application, which supervises the event flow inside a DAQ Slice of the RU-Builder. In the end we will give a short overview of the RU application, which carries out the task of buffering super fragments until they are allocated

to a BU. The EVM and RUs are communicating with the FED-Builder via FED-Builder Output (FBO) application.

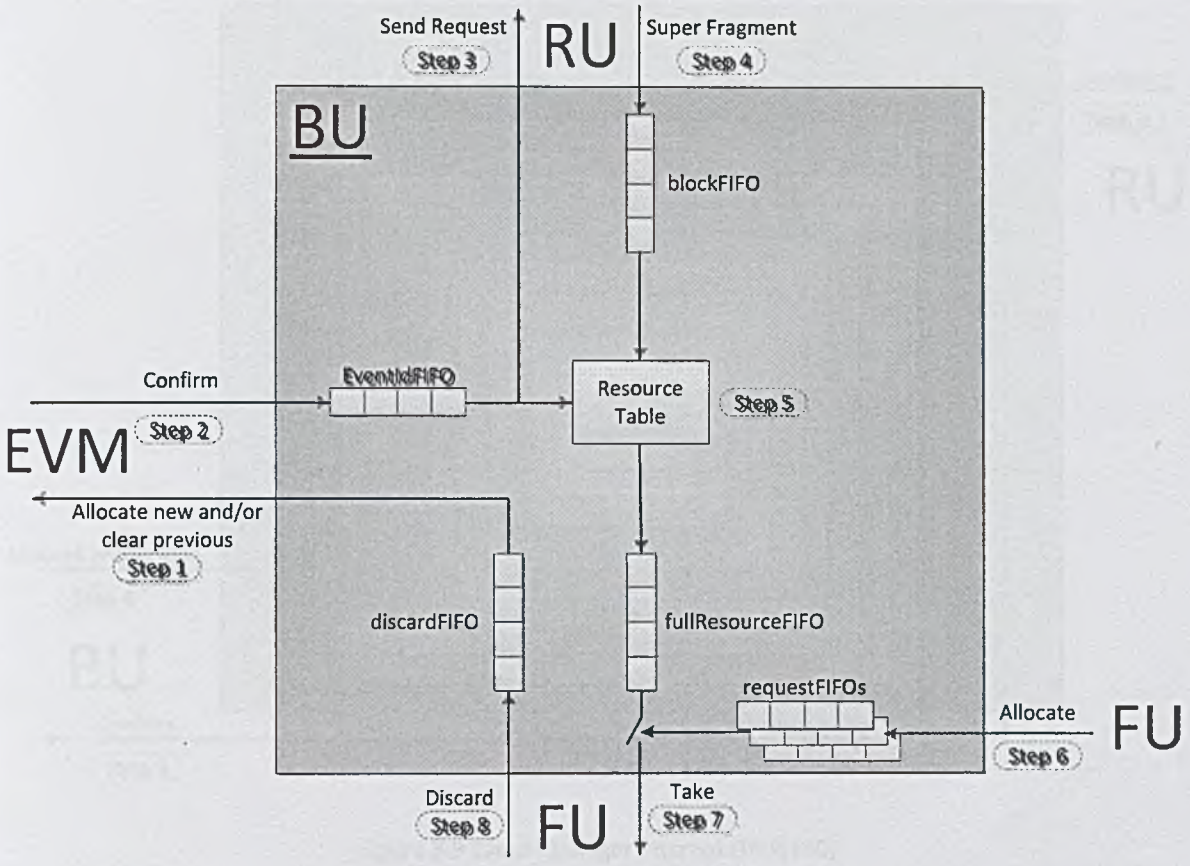


Figure 3-8 Builder Unit internal FIFOs [70]

A schematic view of the BU component is shown in Figure 3-8. If a BU is able to accommodate new events it notifies the EVM (step 1). [70] The EVM assigns the event by transferring a super fragment (trigger data), along with the event ID to the BU (step 2). Subsequently, the BU asks the RUs to transfer it the remaining super fragments of the event (step 3). The BU constructs an entire event from the received super fragments (step 4) using the Resource Table (step 5). The built events, in turn, are reserved by FUs (step 6). The BU handles the allocation request by making the recently built event available to the FU (step 7). As soon as the FU executes the filtering algorithms it sends a discard message to the BU (step 8).

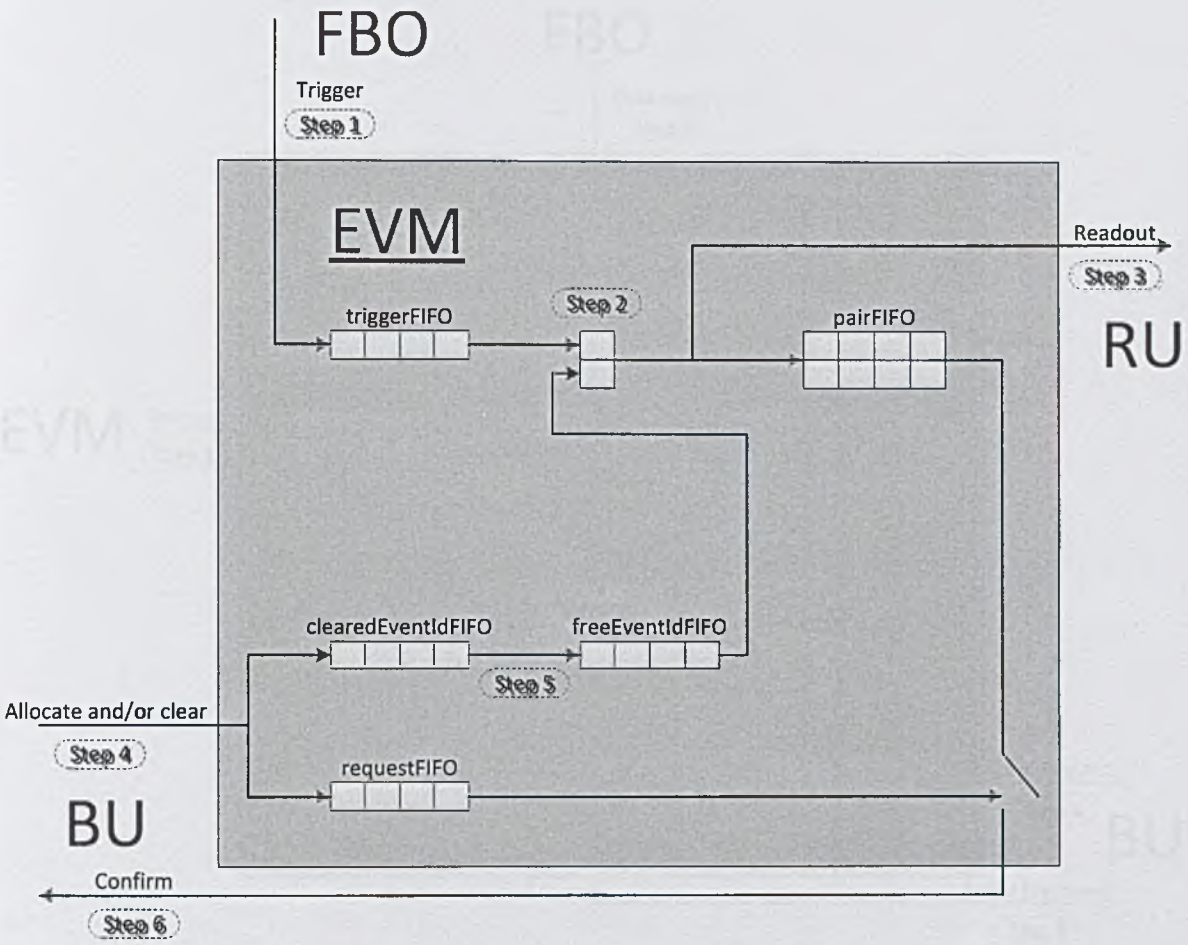


Figure 3-9 Event Manager internal FIFOs [70]

A schematic view of the EVM component is shown in Figure 3-9. The trigger data of an event are transferred to the EVM via the FBO application (step 1). [70] The received trigger data are assigned with free event IDs (step 2). Subsequently, the EVM requests RUs to readout super fragments that correspond to the recently received trigger data (step 3). A BU with free capacity available will send a request to the EVM to allocate an event to it (step 4). Within such a request, an ID of an event will be returned to the EVM in order to be cleared. Each cleared event ID will become a free event ID (step 5). The EVM allocates an event to the BU by transferring the trigger data of the assigned event, along with the event ID (step 6).

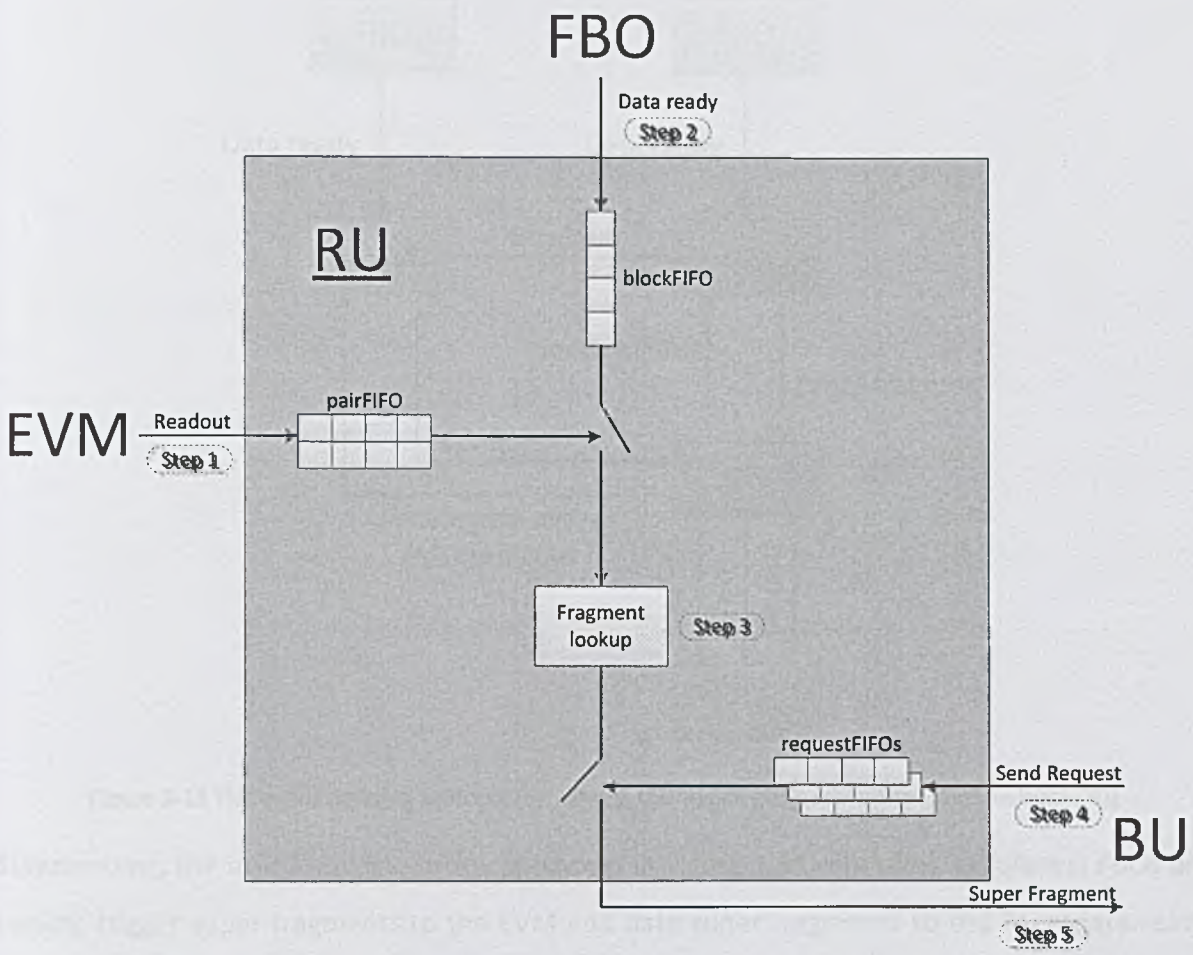


Figure 3-10 Readout Unit internal FIFOs [70]

A schematic view of the RU component is shown in Figure 3-10. The EVM requests the RU to read out an event fragment by sending the trigger event number, along with the assigned event ID (step 1). Simultaneously, the FBO notifies the RU about super fragments that are completed and available for processing (step 2). Each super fragment for which a RU received an event ID – event number pair is positioned in the super fragment lookup table (step 3). BUs are requesting super fragments of the events that they are constructing (step 4). The RU is handling those requests by retrieving appropriate super fragments from the lookup table and then transferring them to respective BUs for further processing (step 5).

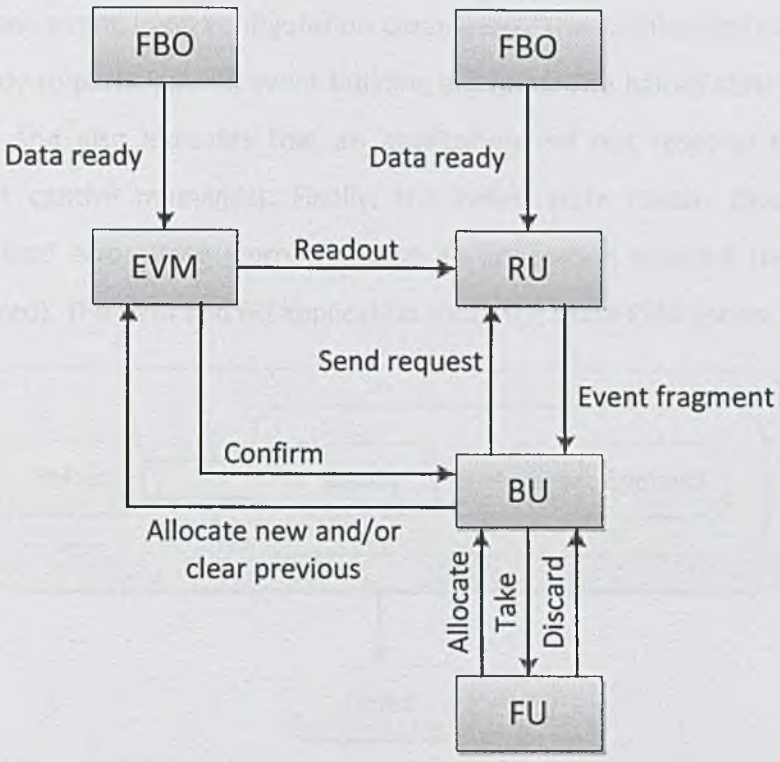


Figure 3-11 The event building protocol (for clarity, multiplicity of each entity is not shown) [70]

Summarizing, the event building protocol (shown in Figure 3-11) operates as follows: FBOs are serving trigger super fragments to the EVM and data super fragments to the RUs (data ready message). Simultaneously, whenever a BU has free capacity to accommodate an event, it requests the EVM to allocate it an event (allocate new and/or clear previous message). Subsequently, the EVM assigns the BU with an event by sending it the trigger super fragment, along with the event ID (confirm message). The trigger event number, along with the event ID, is also sent to all RUs (readout message). After receiving a send request a RU is transferring an appropriate super fragment to the BU. The BU is constructing entire events and then assigning them to FUs (take message), accordingly to allocate requests received from them. The event is stored in shared memory, and is only discarded after a FU finishes executing the selection algorithms (discard message). Therefore, the BU may request a new event only after a previously built event had been processed by a FU.

The behaviour of BU, RU and EVM applications is determined by respective finite state machines (FSM). [70] Those FSM share several common characteristics. The Configured state is

used for reading and acting upon configuration parameters. The enabled state indicates that an application is ready to participate in event building activities. The halted state is used to clean all internal data, and also indicates that an application will not respond to any incoming messages (except control messages). Finally, the Failed state means that an application encountered an fatal error state (currently once an application entered the Failed state it cannot be recovered). The EVM and BU application share the same FSM shown in Figure 3-12.

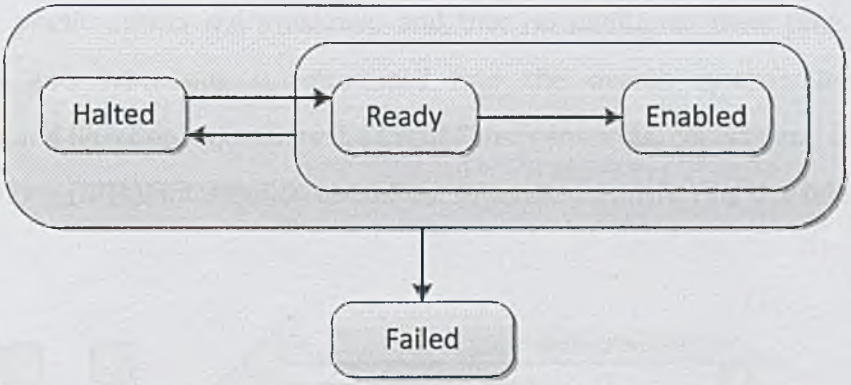


Figure 3-12 Finite State Machine of BU and EVM applications [70]

As shown in Figure 3-13 there are two additional states in the RU application. In both cases the RU is back pressuring the Global Trigger Processor. The Timed Out state indicates that the FBO

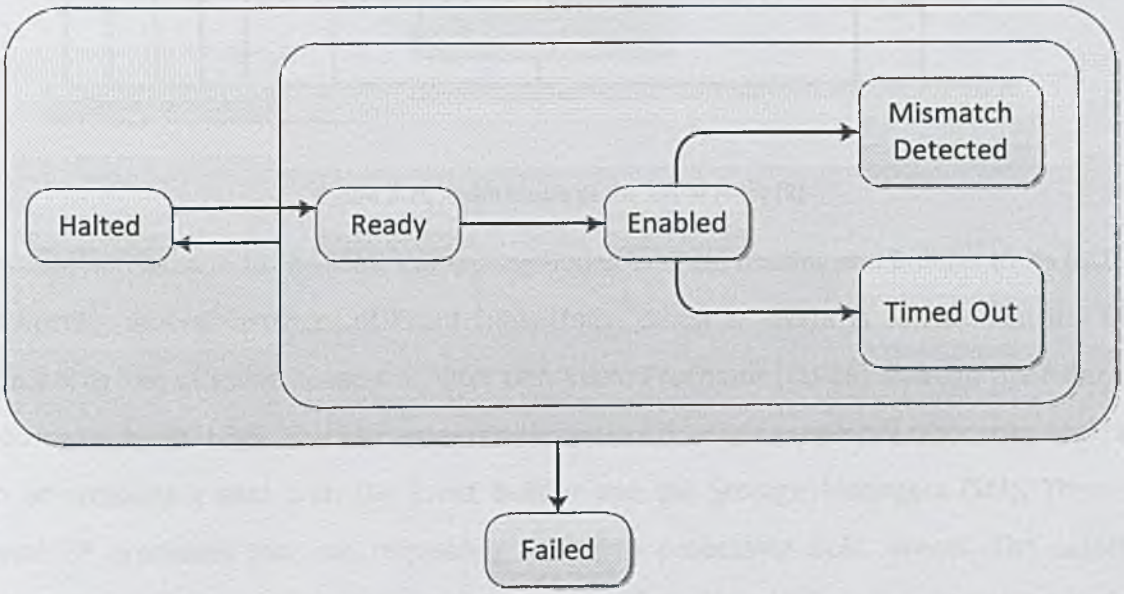


Figure 3-13 Finite State Machine of the RU application [70]

stopped sending super fragments. Mismatch Detected, in turn, indicates that some event fragments were assigned to a wrong DAQ Slice.

3.2 Event Filter

The Event Filter aims to reduce the nominal input rate of accepted events so the output stream is manageable for mass storage and offline processing. Moreover, it has to ensure that all interesting physic events are preserved and that no additional dead-time due to event reconstruction and processing is introduced into the overall system. Besides running reconstruction and filtration algorithms the Event Filter generates, collects and distributes Data Quality Monitoring (DQM) information as well as, supervises transferring the selected events to local storage.

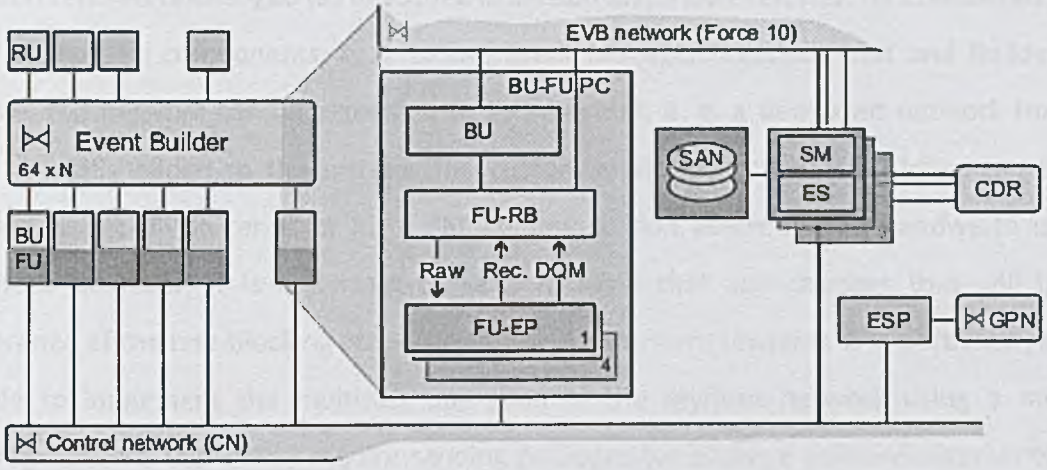


Figure 3-14 Architecture of the Event Filter [2]

As shown on Figure 3-14 the same computing nodes that are hosting the Builder Units (BU) are also hosting several instances of Event Filter Units. When an event is constructed the BU is passing it to one of several copies of Filter Unit Event Processor (FU-EP) through the Filter Unit Resource Broker (FU-RB). The RB carries out the task of managing memory resources, and takes care of exchanging data with the Event Builder and the Storage Managers (SM). There are several EP processes that are requesting and then processing built events. The selection algorithms are chosen and configured at the start of each data-taking run. When an event is select for offline analysis it is passed via RB to a SM node. The SM computing nodes are

connected to a Fibre-Channel SAN that has a throughput of 1 GB/s and a capacity of several hundred TBytes.

3.3 Summary

The research goal of this PhD thesis is to study whether it is possible to enhance the current CMS Data Acquisition System so the static event-fragment distribution can be replaced with a more fault-tolerant dynamic workload scheduling mechanism. The proposed solution has to meet the CMS experiment requirements. Foremost, the system has to be able to sustain an average input of about 100 GB/s, and it has to be guaranteed that all event-fragments will end up in same filtering farm (DAQ Slice). Also, the scheduling mechanism should be lightweight so the current Myrinet network topology, component hierarchy and the two-step event building algorithm remains unchanged (as described in section 3.1). However, new functionalities can be added to existing components, e. g. to the Event Manager, Readout Unit and Builder Unit. Likewise, the network can be expanded to some extent, e. g. a dedicated network for EVMs could be easily added to the system. The custom Myrinet protocol and driver can be also modified, especially in terms of RU – FRL communication where a high bandwidth is being unutilized. However, it is important to keep in mind that any changes that will lead to interference of the non-blocking properties of the network are unwanted. In particular, it is not possible to implement the multicast operation in the Myrinet network using a multicast tree [71], because this would require sending packages (or package acknowledgements) from one source to two or more different destinations.

In parallel other kinds of research is conducted on the CMS DAQ System, including studies on possible replacement of a current electrical extension the S-LINK64 (Simple Link Interface 64 bit) [72] for reading out the detector front-ends. A solution based on 10 Gigabit Ethernet that would allow for larger throughput and for simplification of the architecture is under consideration [73].

4 Requirements analysis

In this chapter we consider the requirements and present use cases of the proposed load scheduling method. We aim to indicate that there is a clear need for a dynamic load scheduling algorithm in the CMS DAQ system, and that such an algorithm enhances the system and facilitates the data acquisition.

4.1 Lost luminosity analysis for CMS experiment

The luminosity loss gives us the information about the amount of potentially useful data that have been lost during the data acquisition process (the concept of luminosity has been described in more details in subsection 1.1.1). Around 10% of the luminosity during stable beam is lost due to various types of malfunctions and general human errors. The luminosity loss, as shown in Figure 4-1, is caused by many factors, amongst others by errors in the sub-detectors and the data acquisition system's hardware and software. In order to reduce the downtime it is necessary to increase the reliability of all components that are subject to faults. The downtime due to both the hardware and software problems in the CMS DAQ system (CDAQ_HW and CDAQ_SW, shown in Figure 4-1) corresponds to 5% of the total downtime. Potentially, the problems causing up to 43% of the CMS DAQ downtime (1 hour, 7 minutes and 43 seconds) could be addressed and solved by the dynamic workload scheduling algorithm, meaning that in the discussed period of 7.5 months additional 387.5 TB of data could be acquired for the online filtering and then approximately 0.4 TB of selected events could be sent to persistent storage for further offline analysis.

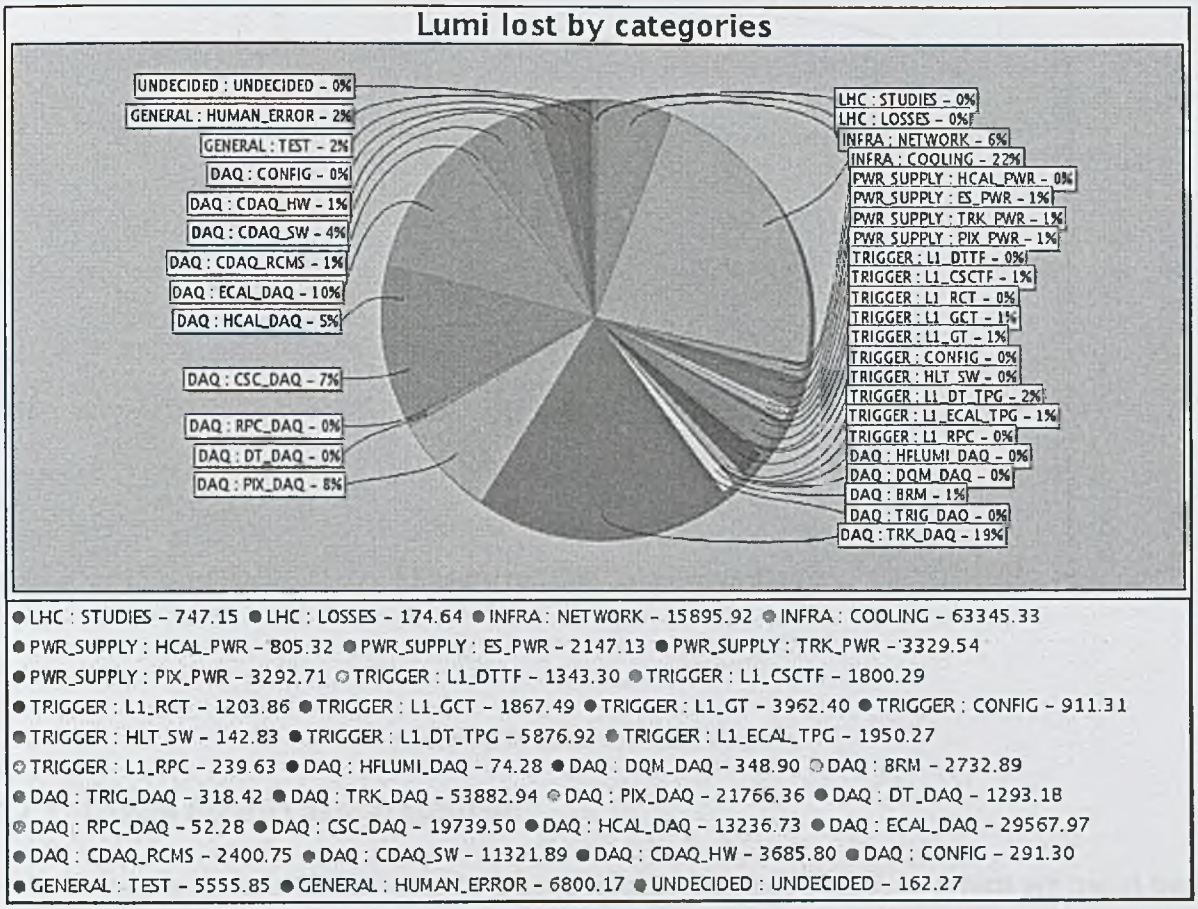


Figure 4-1 Lost luminosity analysis for CMS experiment (measured in the periods form 01/03/2011 to 12/10/2011)

4.2 Business Use Cases

The two most important business use cases for the dynamic workload scheduling algorithm (presented in Figure 4-2) are: ensuring reliable data acquisition and providing more efficient resource utilization. Both business use cases are described and analysed in a more detailed way below.

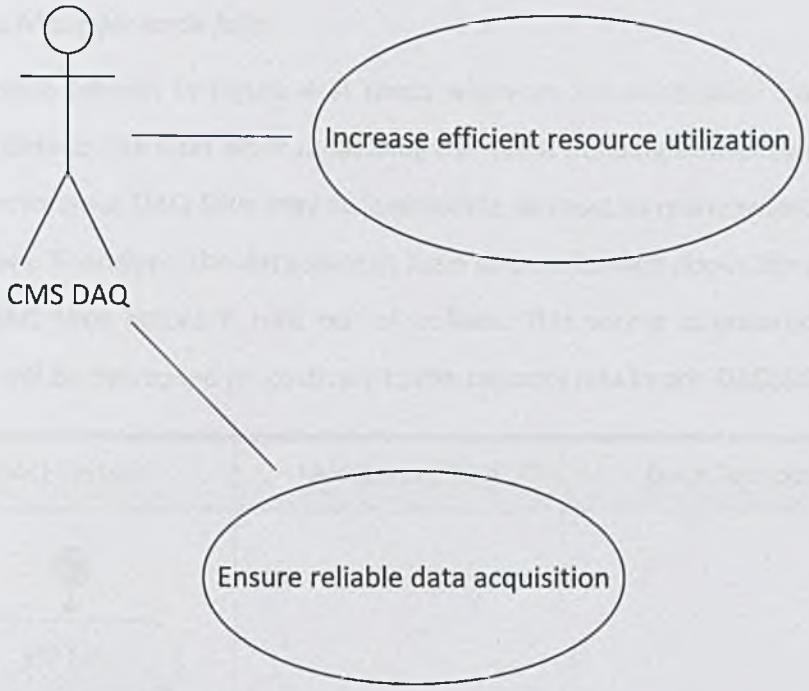


Figure 4-2 Business use cases diagram

4.2.1 Ensure reliable data acquisition

The proposed algorithm should ensure reliable data acquisition by which we mean that in case of some fault that is critical and reduces heavily the capacity of a processing farm the congestion that will appear in this farm should not affect other fully functional filtering farms, and in particular should not cause a data taking run to stop. This is especially important for all single points of failure that are local to a single processing farm (DAQ Slice), but because of the currently used, static event fragment distribution they are becoming global and affect the whole system.

Use case main flow:

1. There is a fault occurrence in the DAQ system
2. The fault is detected by the algorithm
3. The capacity of the damaged filter farm is estimated
4. The workload flow adopts to the new conditions dynamically

Sample use case scenarios corresponding to the discussed use case and illustrating how the proposed algorithm should work have been given below.

4.2.1.1 Storage Manager node fails

The scenario (shown in Figure 4-3) starts when an SM node fails. Subsequently, the monitoring tool detects the fault while measuring the event building efficiency of the damaged DAQ Slice. The erroneous DAQ Slice may accommodate at most as many events as can fit into its readout buffers. Therefore, the data sources have to be informed about the new capacity of the damaged DAQ Slice before it runs out of buffers. This way it is ensured that the data acquisition rate will be decreased respectively to the capacity loss in one DAQ Slice.

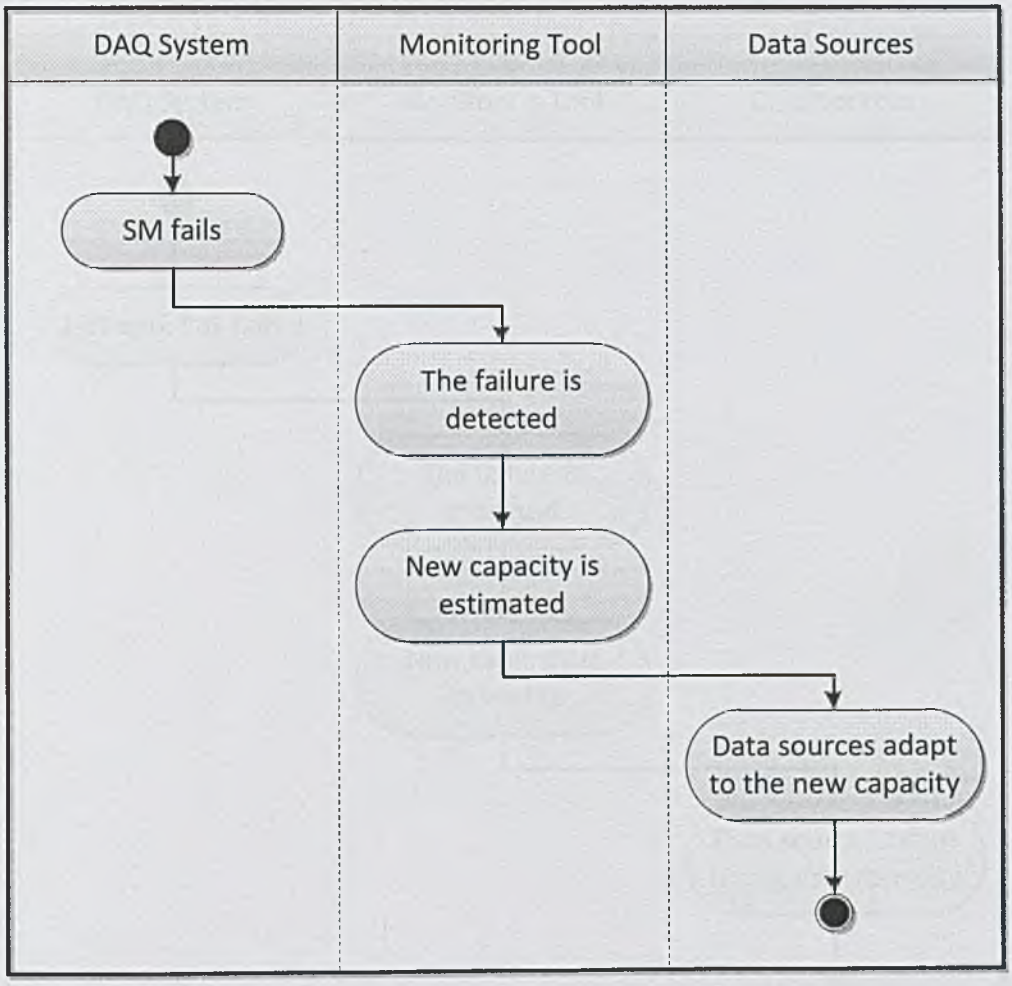


Figure 4-3 SM node fails scenario

4.2.1.2 A network connection of the multiple rail configuration breaks

The scenario (shown in Figure 4-4) starts when one network connection of a multi-rail configuration breaks. Subsequently, the monitoring tool detects the fault while measuring the event building efficiency of the damaged DAQ Slice. The erroneous DAQ Slice may accommodate at most as many events as can fit into its readout buffers. Therefore, the data sources have to be informed about the new capacity of the damaged DAQ Slice before it runs out of buffers. This way it is ensured that the data acquisition rate will be decreased respectively to the capacity loss in one DAQ Slice.

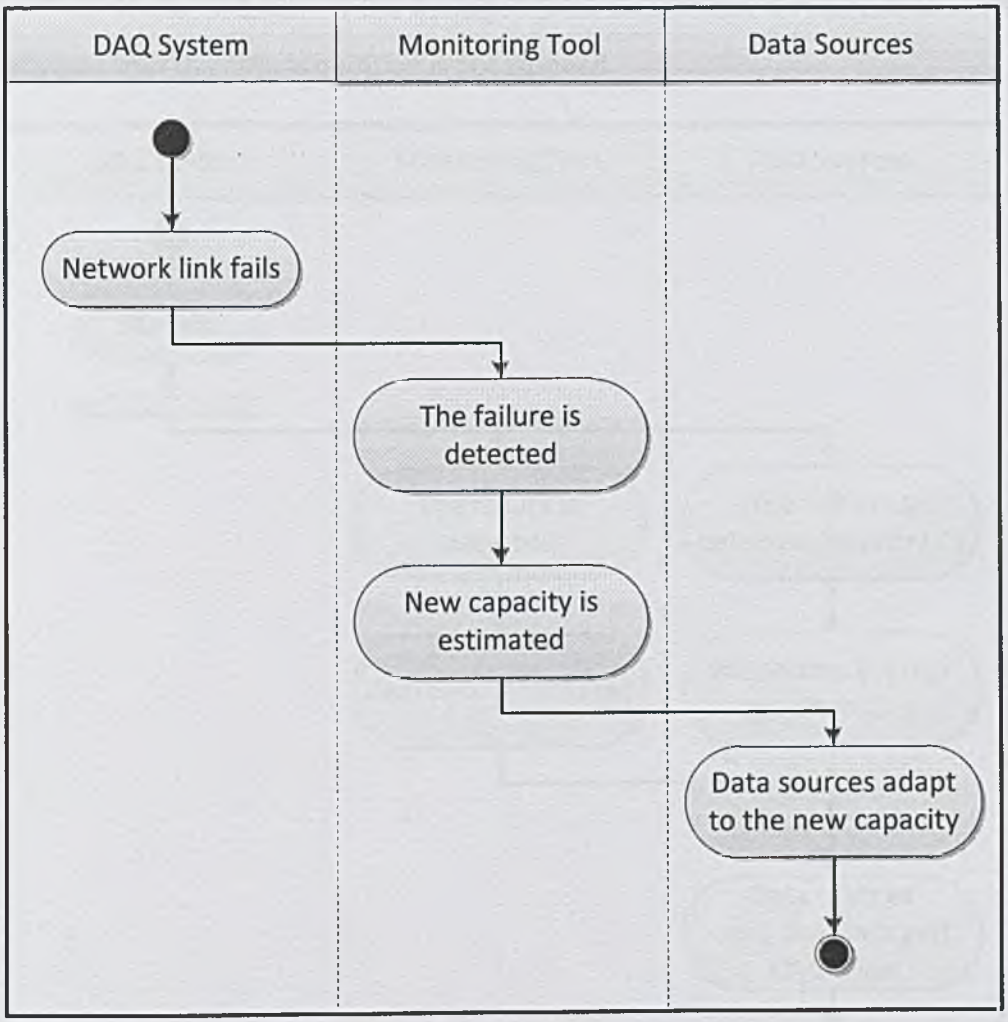


Figure 4-4 Network connection breaks scenario

4.2.1.3 Readout node fails

The scenario (shown in Figure 4-5) starts when a RU node fails. Subsequently, the respective data sources detect the fault using a heartbeat mechanism. Also, the monitoring tool detects the error during measurements of the event building efficiency of the damaged DAQ Slice. Since a Readout node is a local single point of failure, the erroneous DAQ Slice will be excluded from the data taking run. The corrupted DAQ Slice may accommodate at most as many events as can fit into its readout buffers, excluding the event fragments that correspond to the failed RU (those have to be discarded if necessary).Therefore, the data sources have to be informed about the exclusion of the damaged DAQ Slice before it runs out of buffers. This way it is ensured that the data acquisition is not stopped.

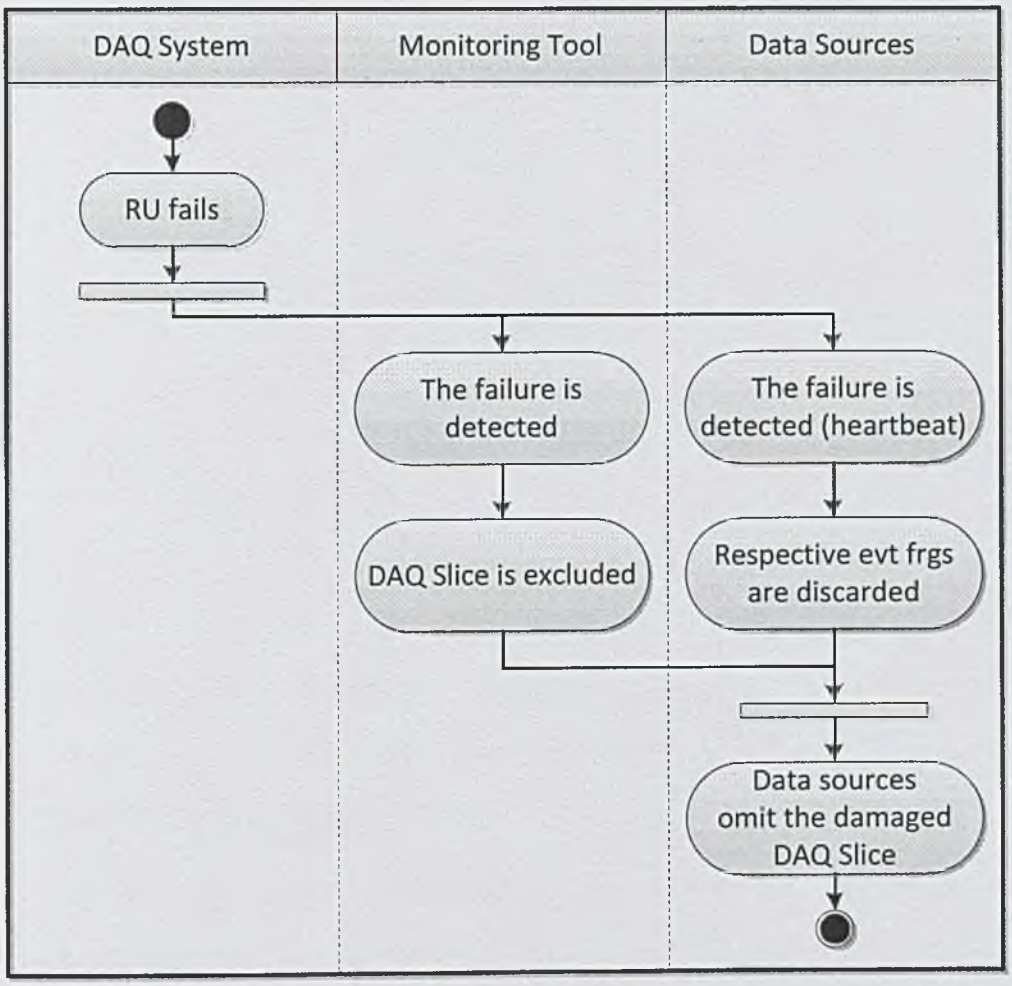


Figure 4-5 RU node fails scenario

4.2.1.4 Event Manager node fails

The scenario (shown in Figure 4-6) starts when an EVM node fails. Subsequently, the respective data source, as well as the monitoring tool detects the fault using a heartbeat mechanism (measurements of the event building efficiency are no longer possible because the EVM is the supervising node). Since an EVM node is a local single point of failure, the erroneous DAQ Slice will be excluded from the data taking run. The corrupted DAQ Slice may accommodate at most as many events as can fit into its readout buffers, excluding trigger super fragments, which correspond to the failed EVM (those have to be discarded if necessary).Therefore, the data sources have to be informed about the exclusion of the damaged DAQ Slice before it runs out of buffers.

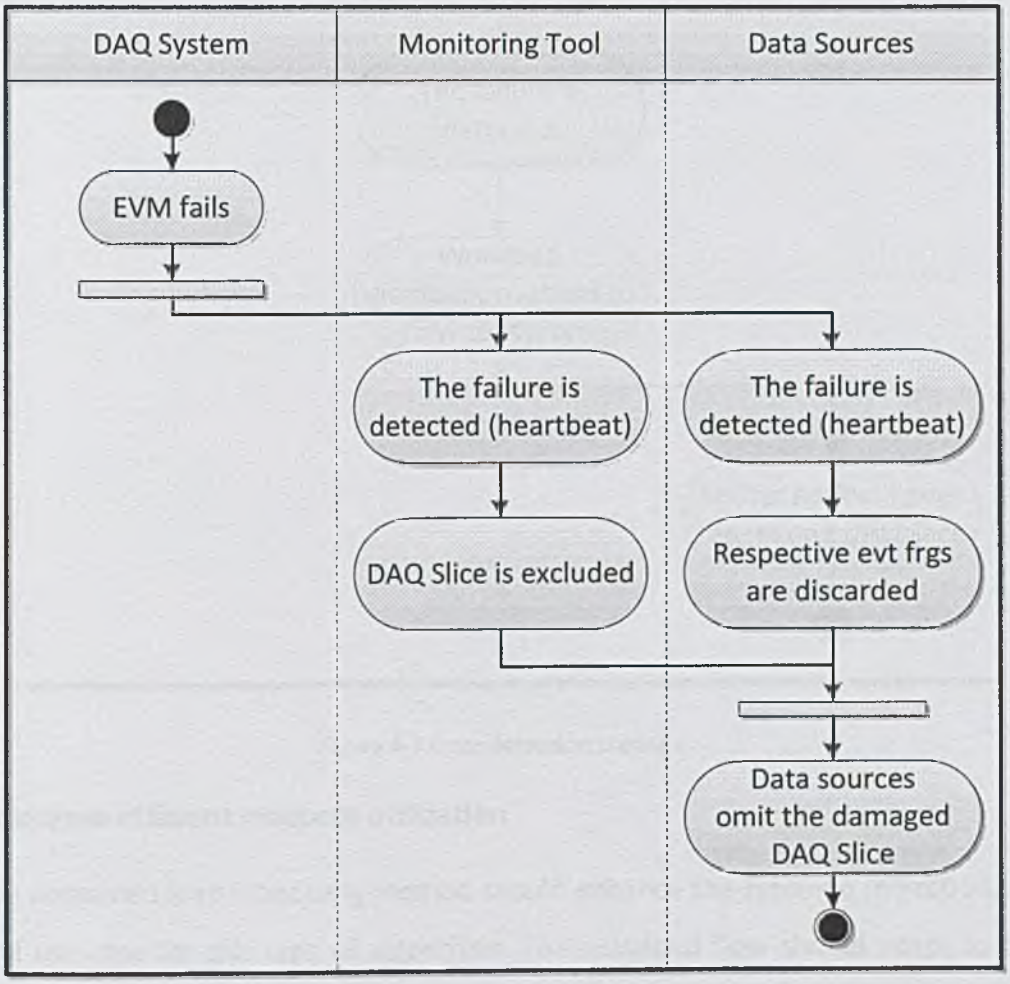


Figure 4-6 EVM node fails scenario

4.2.1.5 Detect problems in an individual DAQ Slice

The scenario (shown in Figure 4-7) starts with a failure in a DAQ Slice that reduces its capacity. The load scheduling algorithm starts to assign fewer events to the damaged DAQ Slice. DAQ shifter notices in the DAQ View monitoring page that the erroneous DAQ Slice collect data at lower rate than expected.

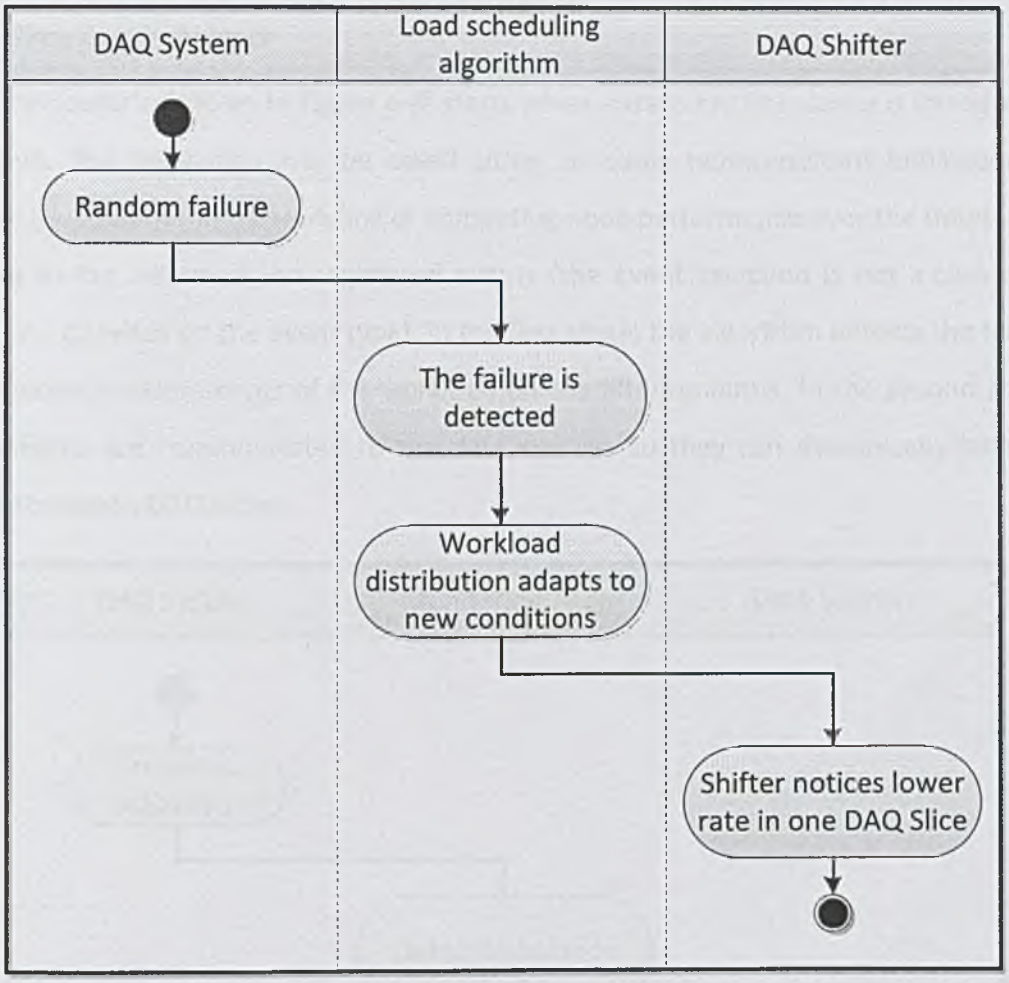


Figure 4-7 Error detection scenario

4.2.2 Increase efficient resource utilization

The presented load scheduling method should enhance the resource utilization, which is a classical use case for this type of algorithms. The workload flow should adapt to the new conditions (throughput, computing power, nature of registered events) dynamically. As a result, higher data event building and filtering rate should be achieved.

Use case main flow:

1. There is an imbalance in the DAQ system
2. The imbalance is detected during routine measurements of the workload on DAQ Slices
3. The workload flow is balanced between DAQ Slices dynamically

Sample use case scenarios corresponding to the proposed use case have been presented below.

4.2.2.1 Transient imbalance

The scenario (shown in Figure 4-8) starts when a transient imbalance is introduced into the system. The imbalance may be cased either by some non-persistent behaviour of the hardware (fluctuations in network link or computing node performance over the time) or by the variations in the nature of the registered events (the event selection is not a constant time process and depends on the event type). In the first stage, the algorithm detects the imbalance during routine measurements of the workload on the filtering farms. In the second stage, the measurements are communicated to the data sources so they can dynamically balance the workload between DAQ Slices.

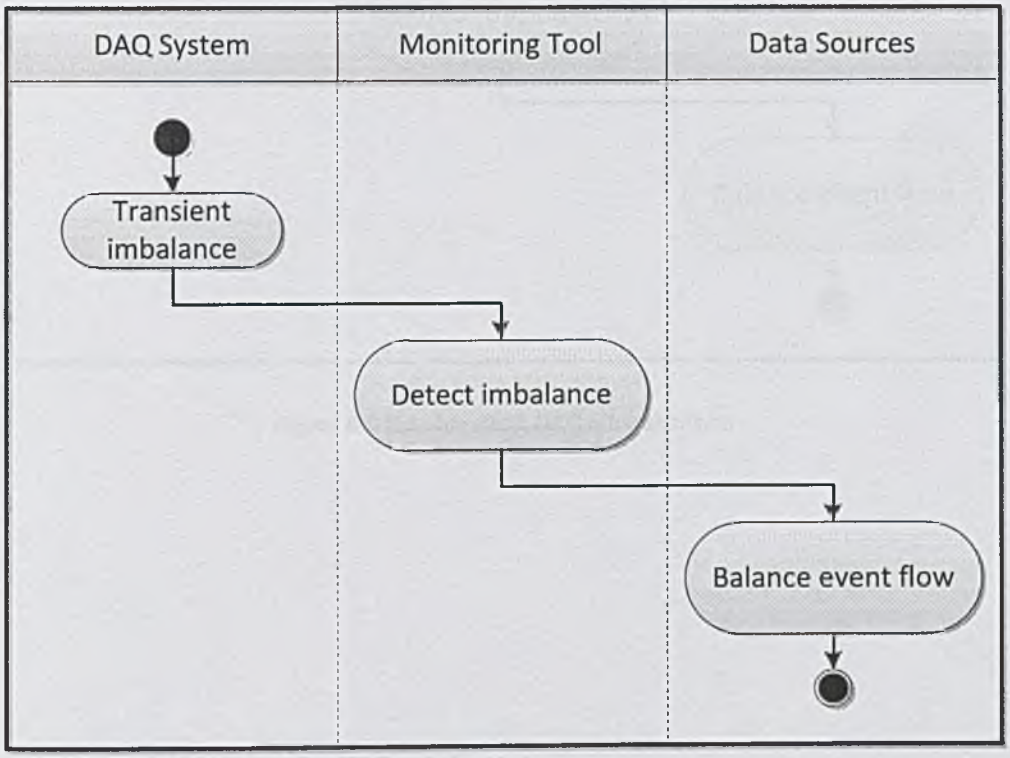


Figure 4-8 Transient imbalance scenario

4.2.2.2 *Non-identical DAQ Slices*

The scenario (shown in Figure 4-9) starts when an additional DAQ Slice is introduced into the DAQ system. The new DAQ Slice is non-identical in respect to other DAQ Slices. It could be either a test farm running experimental algorithms on a small fraction of incoming events, or a powerful farm realized in a completely new technology. The algorithm will have to estimate the capacity of the new DAQ Slice and pass the measurements to the data sources. Subsequently, the data source will have to assign the recently introduced DAQ Slice with a fraction of the workload that corresponds to its capacity in order to fully utilize its resources.

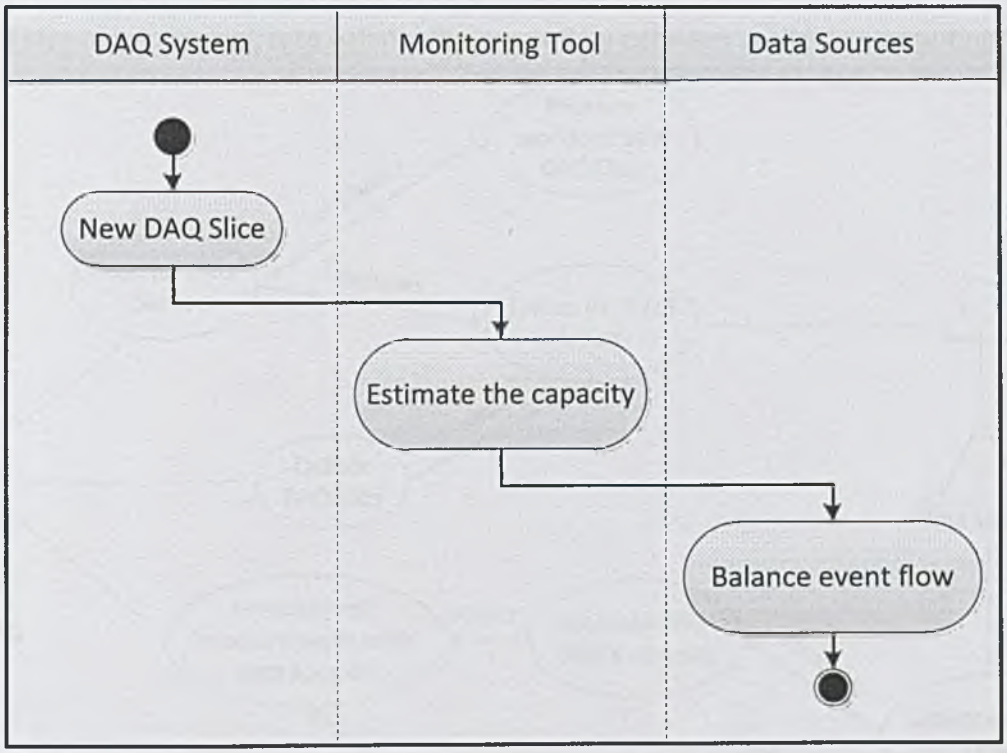


Figure 4-9 Non-identical DAQ Slice scenario

4.3 System Use Cases

The following system use cases (Figure 4-10) were developed based on business use cases and scenarios presented in previous section. The Monitor DAQ Slices use case covers workload measurements, as well as detecting the failure of critical nodes, which in turn is also a use case for the data sources that have to detect the failure independently. The Exclude DAQ Slice use case fulfils the need for masking out a computing farm during data taking run. The workload measurements and exclusion decisions have to be communicated with data sources (Communicate measurements with data sources use case). Finally, the most important use case, and also our main goal, is to balance the event flow between available computing farms.

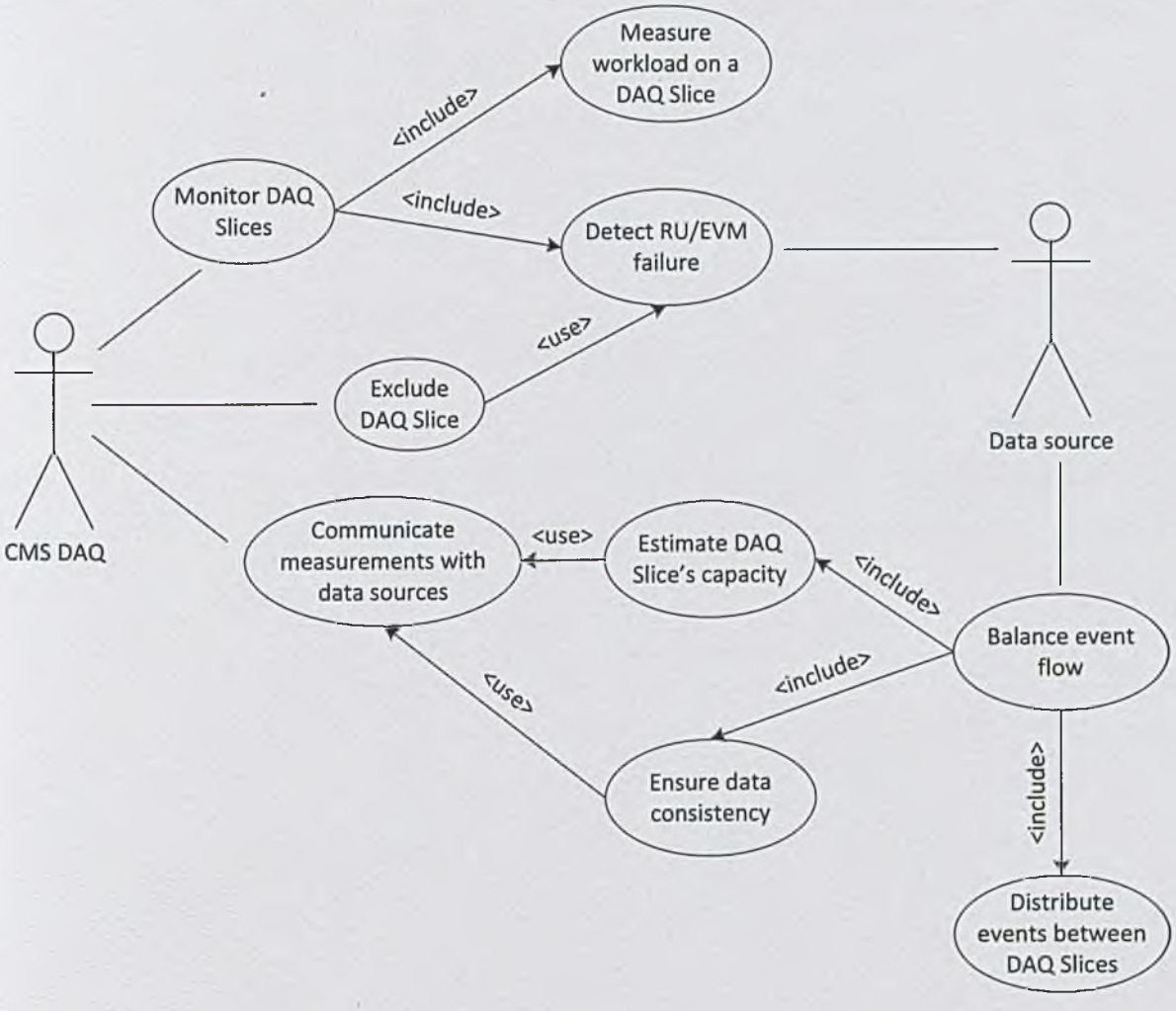


Figure 4-10 System use cases diagram

It is important to notice that our aim is to enhance the system so it is more reliable and the resources are utilized in more efficient way. Several system use cases have been identified that have to be implemented in order to achieve these goals. A monitoring utility is needed so workload measurement can be conducted and computing node failures can be detected. Those measurements need to be communicated to the data sources that in turn will use them to make the event fragment assignment decision. In the worst case scenario, when a critical node fails a DAQ Slice has to be excluded from the on-going data-taking run.

5 Proposed workload scheduling method

In this chapter we will propose a dynamic load scheduling algorithm for a distributed data stream, which allocates the workload between several autonomous computing farms. As previously mentioned the processing farms are accommodating the load through distributed readout. The proposed method employs event driven initiation of load scheduling activities, which is considered as very sensitive to fluctuations and therefore allows for detecting and adapting to faults. Since the algorithm should be as lightweight as possible, and most importantly should not introduce additional bottlenecks and single points of failure into the system, we decided to employ a distributed and asynchronous load assigner. The algorithm is also cooperative in the sense that all load assigner's instances pursue to a common, global goal. Finally, the load assignment method is suboptimal and the monitoring of the workload on the filtering farms is heuristic.

5.1 General idea

In this section we aim to give a brief overview of the proposed workload scheduling method. A schematic view of all important components (computing nodes, networks, etc.), along with the scheduling algorithm's workflow, has been presented in Figure 5-1. Firstly, the capacity of DAQ Slices and their workload have to be estimated. All Builder Units (BUs) in

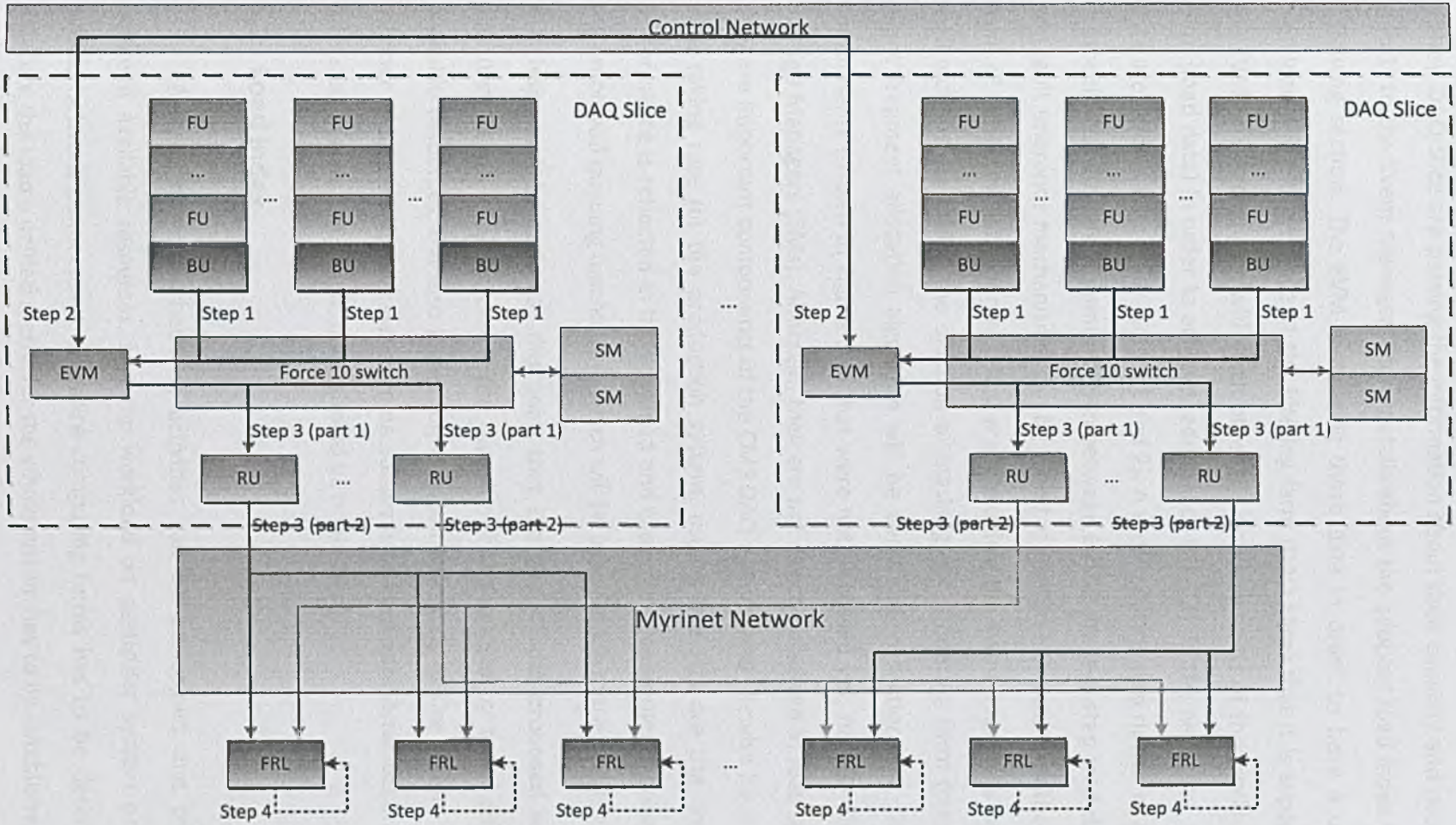


Figure 5-1 Schematic view of CMS DAQ components, along with the scheduling algorithm's workflow [74]

a given DAQ Slice are passing the information about their capacity and occupancy (Figure 5-1, step 1) to the Event Manager. More details about the adopted load index will be given in the following section. The EVM is merging those data in order to have a unified view of the workload and performance of the filtering farm (DAQ Slice) that it is supervising. Afterwards, the EVMs are exchanging with each other the measurements of the workload in each filtering farm (load data) in order to achieve redundancy (step 2). In the next step, the redundant load data are sent to FRLs (step 3, part 1 and 2). A more specific description of the load scheduling protocol (including the communication between EVMs, the two-step load-data transfer to FRLs and fault tolerance mechanisms) can be found in section 5.3. Based on the load data received from all EVMs the data sources (FRLs) are scheduling the event fragments and in case of serious malfunctions are taking the decision of masking out a filtering farm (step 4). The proposed event fragment allocation algorithm will be described in section 5.4. There are two more components (shown in Figure 5-1) that were not discussed yet, namely Filter Units (FUs) and Storage Managers (SMs). Although, they are not directly involved in load scheduling activities, they are important components of the CMS DAQ system, and likewise have a crucial impact on data taking rate (in the production system, usually the FUs are the limiting factor). Their performance is reflected in the workload and capacity measurements done by BUs because of the employed queuing mechanism, which will be explained in more details in section 5.3.

It is important to notice at this point that the goal of the proposed workload scheduling algorithm is not only increasing the capacity of the system by more efficient utilization of available resources, but also increasing the fault tolerance of the system and minimizing the number of lost events in case of some software or hardware error occurrence (event loss is tolerated to a certain extent, but it should be minimized).

5.2 Load index

Before the load-scheduling activities can be performed and the load distributed between available resources, first the workload on particular system parts, like computing nodes, network connections, or entire computing farms has to be determined. In order to measure the load a unified metric for the whole system has to be established. A load index is an

important and essential part of every load scheduling algorithm and has to be carefully chosen as it has a significant impact on the algorithm's overall efficiency.

FRLs deliver data with 100 kHz frequency, which means that approximately every 10 μs there are about 500 new event fragments that need to be assigned to a DAQ Slice. Sending a single message between computing nodes in the discussed system, depending on the network type takes about 10 to 100 μs. The latest networking technologies are an order of magnitude faster, however adopting them for the CMS DAQ system is only in an experimental stage [73]. As a result, currently it is not possible to calculate the workload and exchange the load-data separately for every event [11]. This operation has to be rather made for bigger groups of events in advance. However, it has to be guaranteed that a DAQ Slice can accommodate the assigned group of events. Otherwise, congestion in one DAQ Slice may result with idleness of other DAQ Slices. For this reason, it is desirable that the load index carries the information about the occupancy of the readout buffers in the readout nodes. Furthermore, the available processing power of each DAQ Slice, as well as the workload on the DAQ Slices has to be estimated in order to assign them with an appropriate fraction of the incoming workload. On the other hand, accurate monitoring of all computing nodes in terms of workload, processing power and buffers availability is very resource consuming and thus in case of the discussed system not acceptable. Therefore, a more general way of determining the properties of interest is needed. Since the parallelism inside of a DAQ Slice is achieved by SPMD [7] technique, the workload on a DAQ Slice is directly proportional to the size of data that it has to process. Therefore, the best solution in the discussed case is to measure the workload on a DAQ Slice by the size of the data it owns (as proposed by Fonlupt et al. [21]). The capacity of a DAQ Slice, in turn, can be estimated as the number of events built in a given period of time. Taking into account the structure of the employed FIFO queues and the event building protocol, described previously in chapter 3, it can be noticed that the estimated capacity also reflects the efficiency of event filtration and the transmission rate to persistent storage. The workload and capacity measurements can be easily combined by assigning an initial number of events n to each DAQ Slice, and then, after a given period of time, by checking the numbers of events n_a that still have to be built in each DAQ Slice (an EVM, as it supervises its DAQ Slice, keeps track of how many

events were built, which carries the information about the number of events under construction as long as the original number of events allocated to the DAQ Slice is known). This way, both the workload (n_a events) and the capacity ($n - n_a$ events per measurement time) have been estimated. Determining the occupancy of all readout buffers at once is currently not possible because the addressed system is a distributed real-time system. Nonetheless, the proposed measurement gives us the information about worst case readout buffer occupancy. Since the received super fragments that were not yet assembled into whole events are stored in the buffers of Readout Units, the highest possible occupancy is n_a super fragments. The occupancy might be lower because not necessarily all super fragments that were assigned to a DAQ Slice have been already sent. The information about the available space in the readout buffers is inaccurate because the event size is variable. Nevertheless it is sufficient because the average event size, as well as the event size distribution is known. When it comes to the initial number of events n it cannot be greater than the readout buffer size divided by the average super-fragment size. This way, it is guaranteed that all events assigned to a DAQ Slice will be accommodated regardless of its capacity, which is crucial in order to avoid idleness periods. For example let us consider a DAQ System consisting of two DAQ Slices (Slice 0 and Slice 1). Let us assume, for simplicity, that there are no queues on the data sources' side, and that each data source provides 1 event per time t (*1 evt / t*). Moreover, suppose that an event is transmitted exactly after time t from the previous event transmission, and that the transmission time from the data source to the Readout Unit is negligible. If an event is considered as constructed, all respective super fragments are removed from readout buffers. If a Slice's readout buffers are empty the DAQ Slice is considered as idle since there are no events to build. The data sources are distributing the event fragments in round robin fashion for the initial group of events, and will start to send the next group only after receiving the information about the workload measurements. An event may only be send if there is space for it in the receiver's readout buffers. Slice 0 is building 1 event in time *3 t* (*1 evt / 3 t*) while Slice 1 in time *2 t* (*1 evt / 2 t*), both DAQ Slices have readout buffers for 3 events. Initially each DAQ Slice is assigned with n = *12* events (24 events in total), and the workload measurement will be performed after time *21 t* (from receiving the first event). As shown in Figure 5-2, Slice 1 is building and receiving events

at the same rate and for this reason there is never more than one event in its readout buffers. On the other hand, after time *10 t* (from receiving the first event), the readout buffers from Slice 0 are completely filled in, and as a result receiving of the next event fragment is delayed by *t*. This, in turn, delays sending the next event fragment to Slice 1, and results in idleness periods. Of course if the initial event group would be smaller, the workload measurement would be carried out earlier and communicated fast enough to the data sources, the system could start sending less events to Slice 0 than to Slice 1 and avoid idleness in Slice 1. Nevertheless, this example helps to understand that a Slice has to be able to accommodate all the incoming events in the readout buffers. This property is especially crucial when a fault leads to loss of entire capacity of a DAQ Slice. In practise, in order to minimize the probability that the assigned group of *n* events will not fit into the readout buffers, the buffer space needs to be *n* × *(average super fragment size)* plus some additional reserve due to the variable even size.

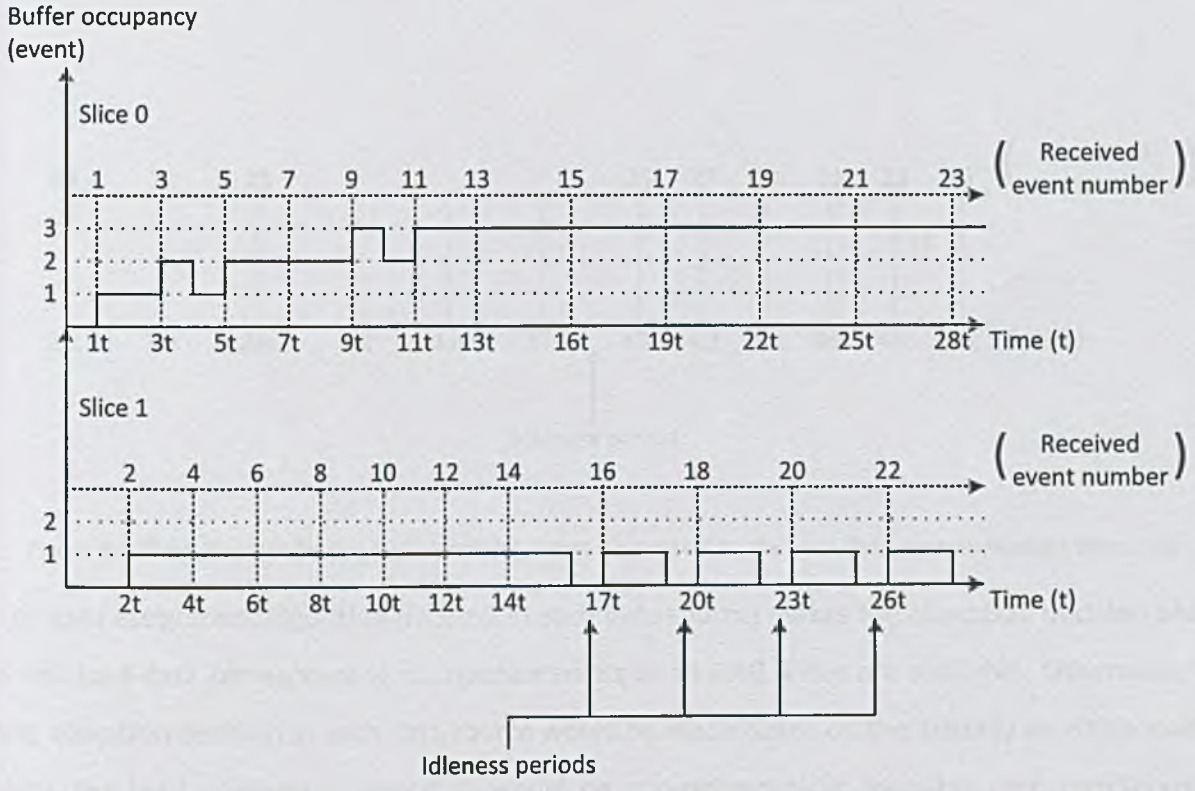


Figure 5-2 Time diagram for a not sufficient readout buffer case

Now let us assume that the load-data communication time is *18 t*. As shown in the Figure 5-3 (continuation from Slice 0 time diagram shown in Figure 5-2), after the workload

measurements were triggered 3 more events were received. Also, there are 2 additional events in the readout buffers. In total the time needed for processing these 5 events is *15 t*. This, in turn, results in *3 t* time idleness until the load-data reach data sources and new events are sent. This leads us to the conclusion that the approximate processing time needed for the remaining events (the events that were assigned to a Slice but were not yet build) has to be higher than the time needed for transferring the load-data to the data sources. Therefore, the workload has to be measured when the number of events assigned to a DAQ Slice for building reaches a minimum (an event driven approach) that allows for sending the obtained load-data to the data sources without an idleness period. When a DAQ Slice reaches this minimal number of events it is considered as underloaded. An advantage, of the event driven strategy is that the frequency of the workload measurements depends on event building efficiency of DAQ Slices, and as a result it depends automatically on the data acquisition rate.

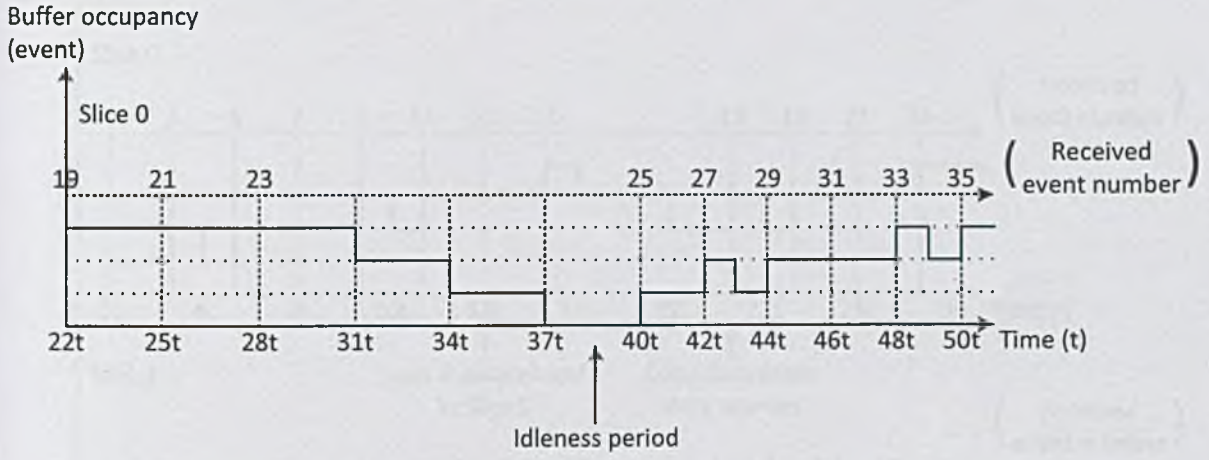


Figure 5-3 Time diagram for too few remaining events in respect to the load-data communication time case

The load assignment algorithm (located in each data source) makes the allocation decision only if the load-data corresponding to measurements in all DAQ Slices are available. Otherwise, if the allocation decision in each data source would be made based on the actually available load-data, the load assignment algorithm would be non-deterministic (possible race conditions). This, in turn, would lead to mixing the event fragments corresponding to one event between several DAQ Slices, which is unacceptable. Since, on the one hand, the measurements are triggered autonomously in each DAQ Slice, but on the other hand the data sources need the

load-data from all DAQ Slices to allocate them with new blocks of events, an idleness period is still possible. Let us consider a DAQ system exactly the same as the one described in the previous examples, except that now both Slices initially are assigned with 8 events (16 events in total) and have enough buffers to accommodate them. Let us also assume that a DAQ Slice is considered as underloaded if less than 4 events are allocated to it for constructing, and that measuring the workload and sending the load-data to data sources takes *8 t*. As shown in Figure 5-4, Slice 1 reaches the underloaded state after time *8 t* (from receiving the first event). The load-data, in turn, reach the data-sources after another *8 t*. At that point the Slice 1 processed all events that where assigned to it. However, since the load-data from Slice 0 did not reach the data sources yet (they were sent later because it is building the events slower), new block of events cannot be allocated to the Slice 1. As a result, there is an idleness period in

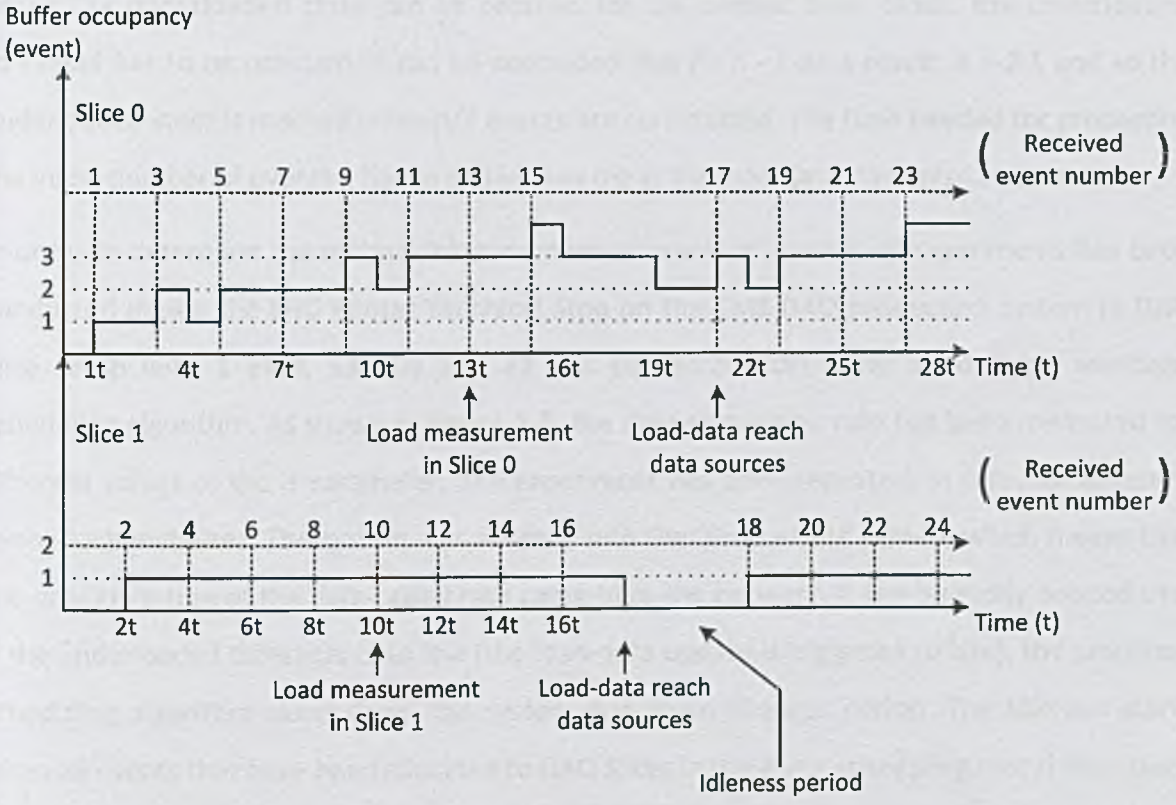


Figure 5-4 Time diagram for an unsynchronized load measurement case

Slice 1. In order to solve this problem, the workload measurement and load-data transfer has to be triggered at once in all participating DAQ Slices. Therefore, the DAQ Slice that becomes

underloaded has to notify other DAQ Slices. This way, the workload estimation and load-data update will be carried out in parallel in all DAQ Slices. Subsequently, the data sources will allocate new blocks of events to all DAQ Slices, which will be distributed after the currently used blocks are all sent. Now, let us consider how to determine the initial number of events that will be assigned to each DAQ Slice. Suppose that the underloaded state is reached after building *l* events. It is desirable to keep *l* as small as possible to achieve more accurate load scheduling. On the other hand, the time needed for processing $n - l$ events has to be higher than or equal to the transmission time of load-data from EVM to FRLs (as shown above, this is a necessary condition that has to be fulfilled in order to avoid lowering the data taking rate). The second set of load-data may be sent to the data sources only after the first one has been received, so $n - l$ is actually the smallest possible number of events after the construction of which the underloaded state can be reached for the second time. Since, the underloaded threshold has to be constant, it can be concluded that $l = n - l$. As a result, $n = 2l$, and so the underloaded state is reached when $n/2$ events are constructed. The time needed for processing the initial number of events *n* has to be twice as big as the load-data transmission time.

In order to determine the optimal initial number of events *n*, a series of experiments has been conducted during the LHC Winter Technical Stop on the CMS DAQ production system (8 DAQ Slice setup with 1 EVM, 63 RUs and 82 Bus per each slice) using a prototype workload scheduling algorithm. As shown in Figure 5-5, the data acquisition rate has been measured for different values of the *n* parameter. The experiment has been repeated, in turn, for different event-fragment sizes. The system was running with the '*Drop at BU*' option, which means that the only limitation of the data-taking rate came from the network. It can be easily noticed that if the underloaded threshold is to low (the load-data update is triggered to late), the workload scheduling algorithm slows down the system due to an idleness period. The idleness starts when all events that have been allocated to DAQ Slices in the given scheduling round have been sent, and continues until new load-data from all DAQ Slices are received. For 2 KB event-fragments (the expected event fragment size for the CMS experiment) the data-taking rate stops being limited by the scheduling algorithm (becomes constant) for $^{n}/_{2}$ greater than 1200 events. The curve corresponding to the 3 KB event-fragment reaches its maximum at the latest

($^{n}/_{2}$ value equals to 2000 events). Hence, the underloaded threshold, including a reserve due to the variable fragment size, should take a value in the range from 1500 to 2000 events.

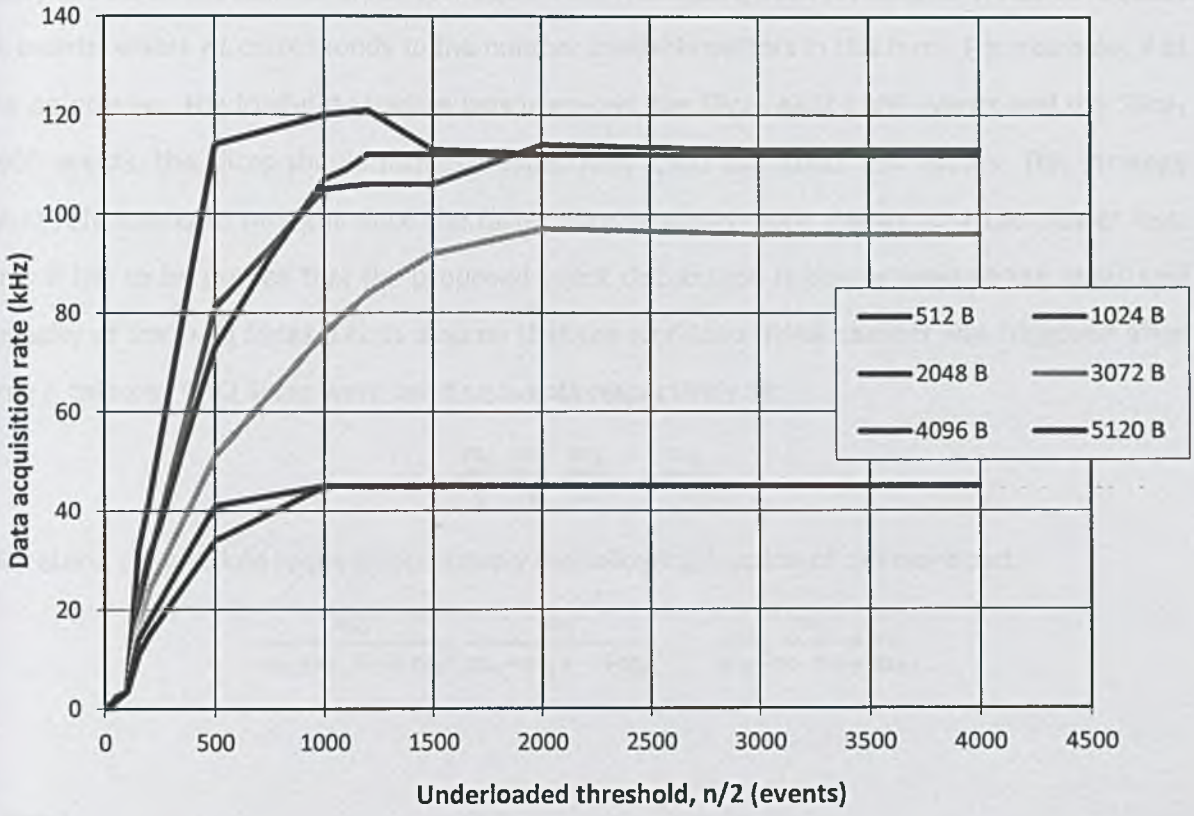


Figure 5-5 The Data acquisition rate depending on the underloaded threshold for various event-fragment sizes

So far, it has been defined how to conduct the workload measurement but the number of events that each DAQ Slice should request by sending the load-data still has be considered. It has been determined that when the load-data update is triggered each DAQ Slice has constructed m_i events, where i is the index of a given DAQ Slice. It is also known that the Slice$_i$ is still assigned with $n - m_i$ events, where n is the number of initially assigned events. Moreover, $n - m_i$ is the maximal possible occupancy of the readout buffers and so m_i is the number of available buffers, in the respective DAQ Slice. The maximal workload that can be requested in total (for k DAQ Slices) is:

$$m_0 + m_1 + m_2 + \cdots + m_k$$

Since, the number of requested events cannot adversely affect the frequency of workload measurements (it depends only on the load-data transmission time) it is desirable to ask for as many events as possible. In order the request the maximum, each filtering farm has to request m_i events, where m_i corresponds to the number available buffers in this farm. For example, if at the point when the load-data update was triggered, the Slice_0 built 1500 events and the Slice_1 1000 events, the slices should request respectively 1500 and 1000 new events. This strategy intuitively seems to be right since the faster Slice requests more events, and the slower less. Now it has to be proven that the proposed event distribution is proportional to the measured capacity of the DAQ Slices. Let us assume that the workload measurement was triggered after time t, this way DAQ Slices were building events respectively at:

$$\frac{m_0}{t}, \frac{m_1}{t}, \frac{m_2}{t}, \cdots, \frac{m_k}{t}$$

Therefore, they should request respectively the following fraction of the workload:

$$\frac{m_0}{m_0+m_1+\cdots+m_k}, \frac{m_1}{m_0+m_1+\cdots+m_k}, \cdots, \frac{m_k}{m_0+m_1+\cdots+m_k}$$

Hence:

$$\text{Slice}_0: \frac{m_0}{m_0+m_1+\cdots+m_k} \times (m_0 + m_1 + \cdots + m_k) = m_0$$

$$\text{Slice}_1: \frac{m_1}{m_0+m_1+\cdots+m_k} \times (m_0 + m_1 + \cdots + m_k) = m_1$$

$$\vdots$$

$$\text{Slice}_k: \frac{m_k}{m_0+m_1+\cdots+m_k} \times (m_0 + m_1 + \cdots + m_k) = m_k$$

This proves that the requests for new events are proportional to the estimated event building efficiency of DAQ Slices. Moreover, after triggering the load-data update each DAQ Slice is again assigned with the initial number of events n (n - m_i + m_i). This is an important property of the proposed load index because it makes the discussed procedure repeatable, by which we mean that the upcoming workload measurements can be done exactly in the same way as the first

one. Whenever a DAQ Slices reaches the underloaded state the workload calculation and the load-data update are triggered.

5.3 Load scheduling protocol

The load scheduling protocol is responsible for transferring the load-data from DAQ Slices to the data sources and gathering the workload indexes. Key features of such a protocol, in case of the discussed DAQ system, are efficiency and scalability. It is also crucial that the protocol is as lightweight as possible so it does not decrease the available processing power and bandwidth. In case of an asynchronous, distributed algorithm, the load scheduling protocol is also responsible for ensuring coherency of the allocation process that is made in numerous data sources concurrently.

5.3.1 Collecting workload indices from Builder Units

As it has been said in previous section, each DAQ Slices is assigned with an initial group of events for building. When the number of these events drops to a certain level the DAQ Slice is considered as underloaded. Therefore, the number of constructed events has to be monitored. Since an EVM is a central point of each DAQ Slice that supervises its activities, it seems to be a natural choice for collecting the workload indices from BUs.

In the standard system the only message that a BU is sending to the EVM is *Allocate new and/or clear previous*. This message is sent each time the BU receives a *discard* message, which, in turn, is sent by a Filter Unit after running the HLT selection algorithms. As a result, monitoring of this message would give us the information about the number of built and filtered events in a given period of time, and the information about the number of not yet filtered events. Since the filtration time is match longer than the event construction time, the estimation about the (worst case) readout buffer occupancy would be very inaccurate (large fraction of the unfiltered events would be actually stored in BUFU's buffers rather than in RUs' buffers). Therefore, as shown in Figure 5-6, the Builder Unit has to be modified so it sends an additional *Event constructed notification* message (step 6) each time a new event had been assembled.

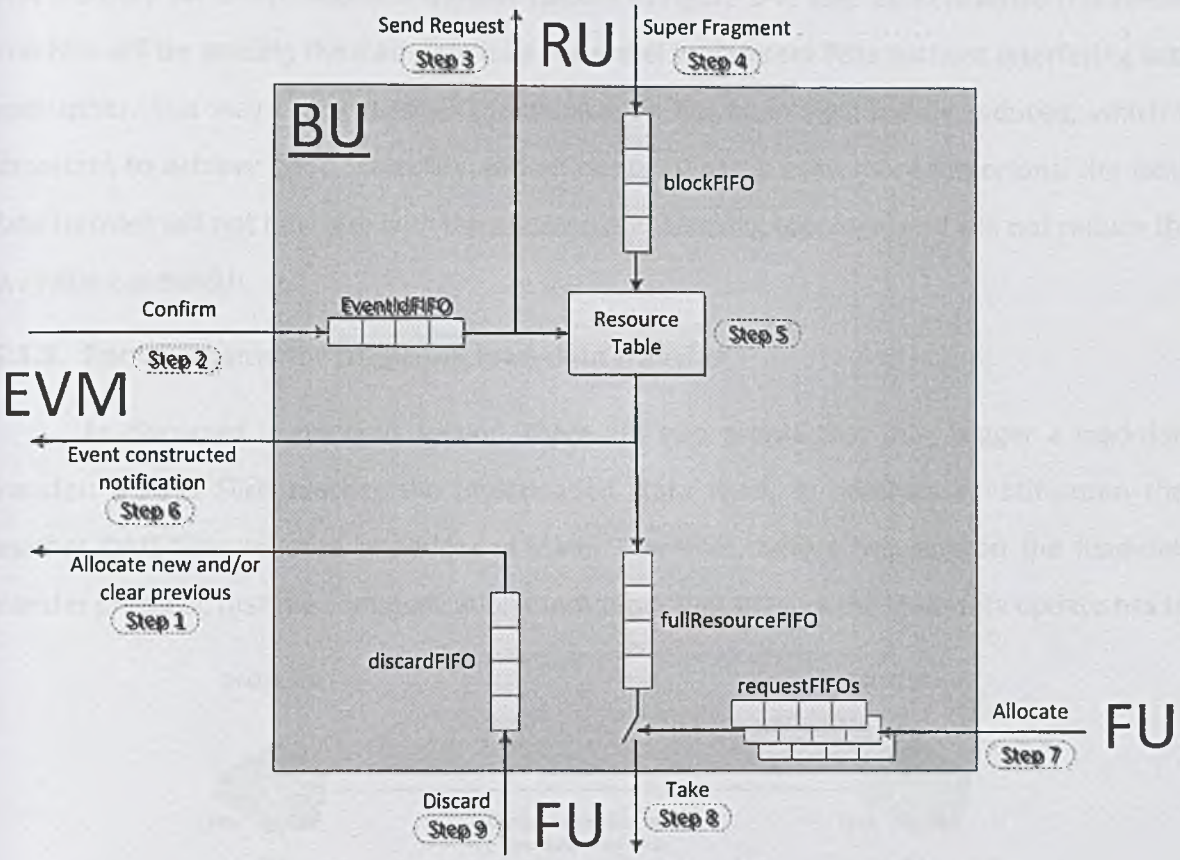


Figure 5-6 Builder Unit internal FIFOs, *Event constructed notification* message added, due to the load scheduling algorithm

5.3.2 Two-step load-data transfer

After reaching the underloaded state, the EVM needs to send the load-data to the FRLs. An EVM is directly connected to all FRLs through the non-blocking Myrinet [6] network. Unfortunately, the Myrinet network does not support multicasting, and implementing it using a multicast tree would interfere with the non-blocking properties of the network, and as a result would decrease the available throughput. On the other hand, the Terascale Force10 switch, which lies at the heart of each DAQ Slice, supports multicast data delivery. Moreover, the multicast facility is implemented in a very efficient way and scales very well. Therefore, the Force10 switch can be used for transferring the load-data to the readout nodes (shown in Figure 5-7, step 1). Since the readout nodes are receiving the event fragments from FRLs, each RU can transmit further the load-data to a subset of FRLs using the same non-blocking routs

that are used for event fragment transfer (shown in Figure 5-7, step 2). In practise this means that RUs will be sending the load-data fully in parallel to different FRLs without interfering with each other. This way the sequential communication has been significantly reduced, which is important to achieve good scalability and efficiency. What is even more important, the load-data transfer will not interfere with the adopted non-blocking topology, and will not reduce the available bandwidth.

5.3.3 Requirements for triggering load-data transfer

As discussed in previous section, there are two events that may trigger a load-data transfer: a DAQ Slice reaches the underloaded state itself, or receives a notification that another DAQ Slice reached underloaded state. Therefore, before focusing on the load-data transfer protocol, first the communication mechanism that triggers the load-data update has to

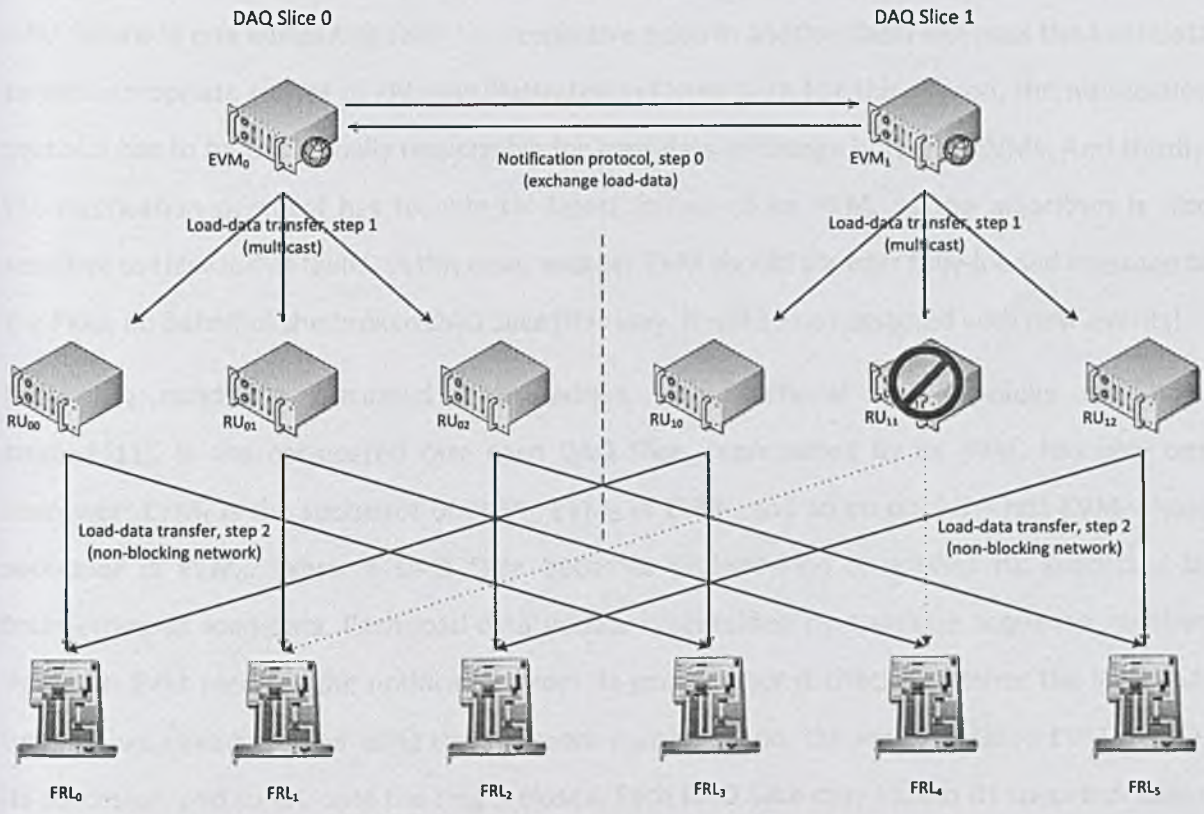


Figure 5-7 Load scheduling protocol - load-data redundancy. Step 0: EVMs exchange load-data. Step 1: EVMs multicast the redundant load-data to RUs (RU_{11} is faulty). Step 2: RUs transfer the load-data to FRLs via non-blocking network (FRL_0 and FRL_4 receive the load-data from DAQ Slice 1 via RU_{01})

be discussed (shown in Figure 5-7, step 0). For the purpose of this intercommunication a DAQ Slice is represented by its EVM, as it is its central point. The notification protocol has to be designed having in mind the essential system attributes, as well as that the subsequent load-data transfer will be carried out in two stages.

Firstly, in the standard production system all DAQ Slices have equal capacity (assuming that there were no fault occurrences), and are assigned with equal number of events for processing. Therefore, statistically the workload on each computing farm should be the same, and as a result, it is very likely that all DAQ Slices will be reaching the underloaded state in the same time. Secondly, since the load-data update is realized in two steps, it is sensitive to failures in readout nodes (they are responsible for passing the load-data further to FRLs). Therefore, there has to be a redundancy in the load-data, which means that in practise an EVM has to transfer load-data not only from its own DAQ Slice, but also from another DAQ Slice. This way, in case of a RU failure in one computing farm the respective node in another farm will pass the load-data to the appropriate subset of FRLs (as illustrated in Figure 5-7). For this reason, the notification protocol has to be additionally responsible for load-data exchange between EVMs. And thirdly, the notification protocol has to able to detect failure of an EVM, as the algorithm is also sensitive to this kind of faults. In this case, another EVM should transfer fully-loaded message to the FRLs, on behalf of the broken DAQ Slice (this way, it will be not assigned with new events).

Having in mind the discussed particularities, one-directional ring topology has been studied [11]. In the considered case each DAQ Slice, represented by its EVM, has only one successor: EVM_1 is the successor of EVM_0, EVM_2 of EVM_1, and so on until the last EVM whose successor is EVM_0. When a DAQ Slice becomes underloaded it notifies its successor by transferring its load-data. Each load-data update is identified by a unique sequence number. When an EVM receives the notification from its predecessor it checks whether the load-data update was already served using the sequence number. If no, the newly notified EVM notifies its successor, and so on, until the ring is closed. Each DAQ Slice may inform its successor about a given load-data update only once. If a DAQ Slice already triggered load-data update, because it became underloaded itself, and then was notified by its predecessor it will not pass this notification further. Transferring of the load-data to FRLs via RUs can only begin after receiving

the notification from the predecessor, as only then the EVM has the complete information about its and its predecessor workload. This way the necessary redundancy has been achieved by transferring load-data concerning each DAQ Slice to FRLs twice: firstly by the DAQ Slice itself, and secondly by its successor. As previously mentioned, this strategy has been designed keeping in mind that in the production system the DAQ Slices have the same capacity. Therefore, all DAQ Slices are expected to reach the underloaded state and send the notification to their successors at the same time. It can be also noticed that this strategy is convenient for detecting DAQ Slices that for some reason became slower, or excluding a broken processing farm from a data-taking run, at a very low network-traffic cost.

Let us consider an exemplary DAQ System of 3 DAQ Slices (Slice 0, Slice 1 and Slice 2), and let us assume that the notification message is sent exactly after reaching underloaded state and is delivered in time t_n. The load-data transfer to FRLs starts immediately after the notification has been received. The remaining assumptions from the previous examples remain unchanged. As

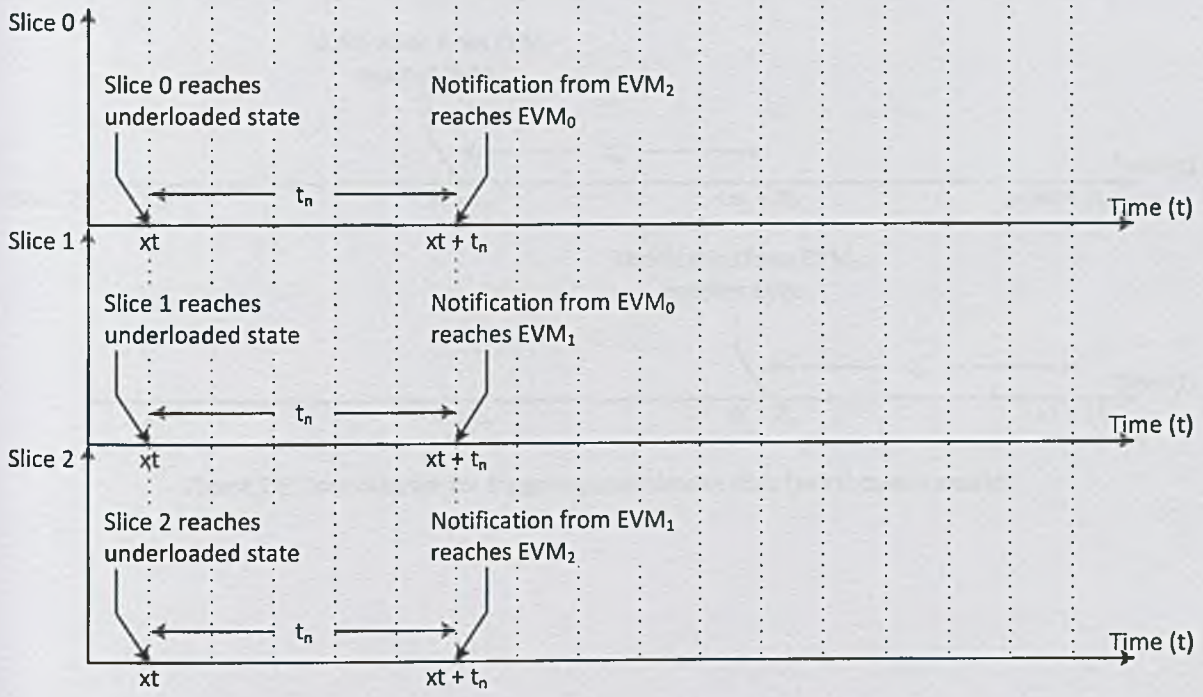


Figure 5-8 Time diagram for triggering load-data update (expected case scenario)

shown in Figure 5-8, in the expected case (all filtering farms are reaching the underloaded threshold at the same time), the load-data transfer to FRLs is only delayed by t_n time in all DAQ

Slice. At this point, it can be noticed that adding an additional DAQ Slice to the system does not introduce additional delay (good scalability). On the other hand, in the worst case, the load-data transfer to the data sources is delayed by $t_n \times k$, where k is the number of DAQ Slices (as shown in Figure 5-9). In this case, the time t_d after which the load-data are sent to FRLs grows linearly with the number of DAQ Slices ($t_d = t_n \times k$). Such behaviour is expected in a DAQ System where the processing farms have different or varying computing power. It is especially likely if there is one filtering farm that outperforms significantly the others. In this case, the more capacious slice will be the only one that reaches the underloaded state. The other slices will be only passing the underloaded notification further through the adopted ring network.

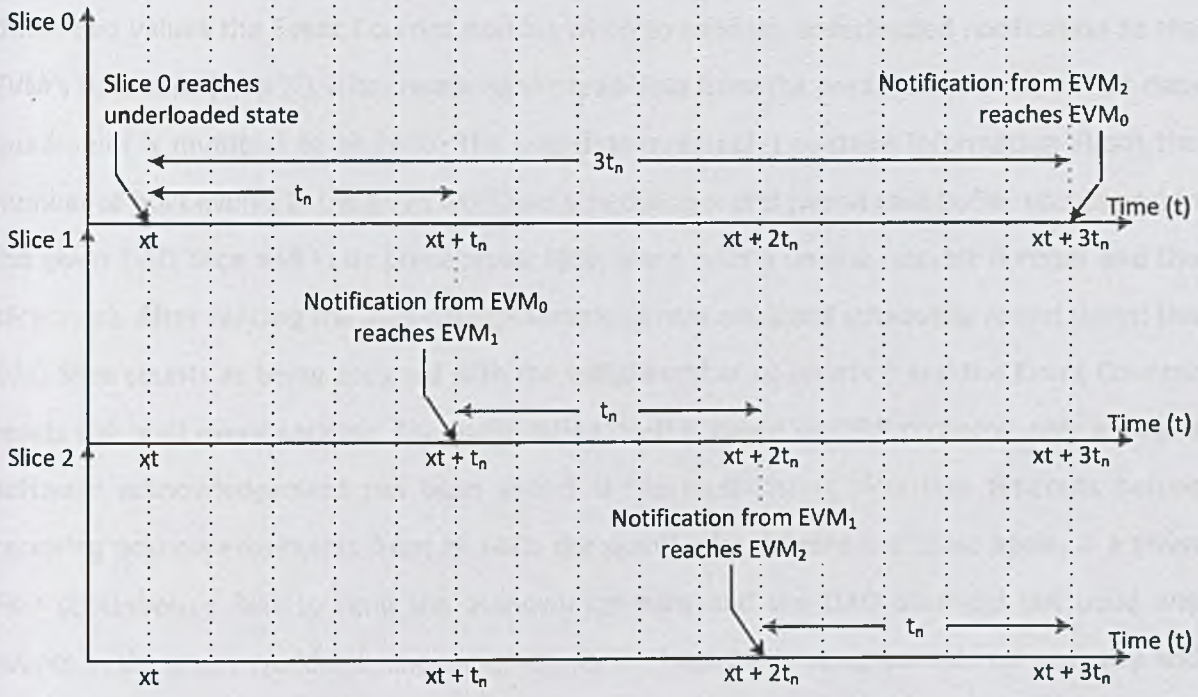


Figure 5-9 Time diagram for triggering load-data update (worst case scenario)

5.3.4 EVM's workload communication algorithm

In order to introduce the proposed method the Event Manager entity has been modified in respect to the standard system as shown in Figure 5-10. Firstly, the ability to process the event constructed notification send from BUs has been added (step 4). The notification is passed to Event Counter, which keeps track of the number of built events. Moreover, also the notifications from the EVM's predecessor have to be handled (step 5'). The underloaded notification consists of a load-data, request number and the slice mask triplet. The EVM updates its own slice mask accordingly to the one received in the triplet. The request number, similarly as the event constructed notification is passed to the Event Counter (step 6'). Based on those two values the Event Counter decides when to send an underloaded notification to the EVM's successor (step 7'). After receiving the load-data from the predecessor EVM, a load-data quadruplet is multicast to all FBOs. The load-data quadruplet contains information about the number of built events in the given workload scheduling round (worst case buffer occupancy) in the given DAQ Slice and in its predecessor Slice, along with a unique request number and the slice mask. After sending the load-data quadruplet a new workload scheduling round starts: the DAQ Slice counts as being assigned with the initial number of events *n* and the Event Counter resets the built event counter. The multicasting itself is based on UDP protocol; only a simple software acknowledgement has been added. If the multicasting algorithm timeouts before receiving acknowledgements from all FBOs the quadruplet is being multicast again. If a given FBO continuously fails to send the acknowledgement and the DAQ Slice did not build any events in the given workload scheduling round, the host RU is being considered as faulty and the EVM masks its own Slice in the DAQ Slice Mask, which will be propagated over the system during the next workload scheduling rounds, and will result in excluding the broken DAQ Slice.

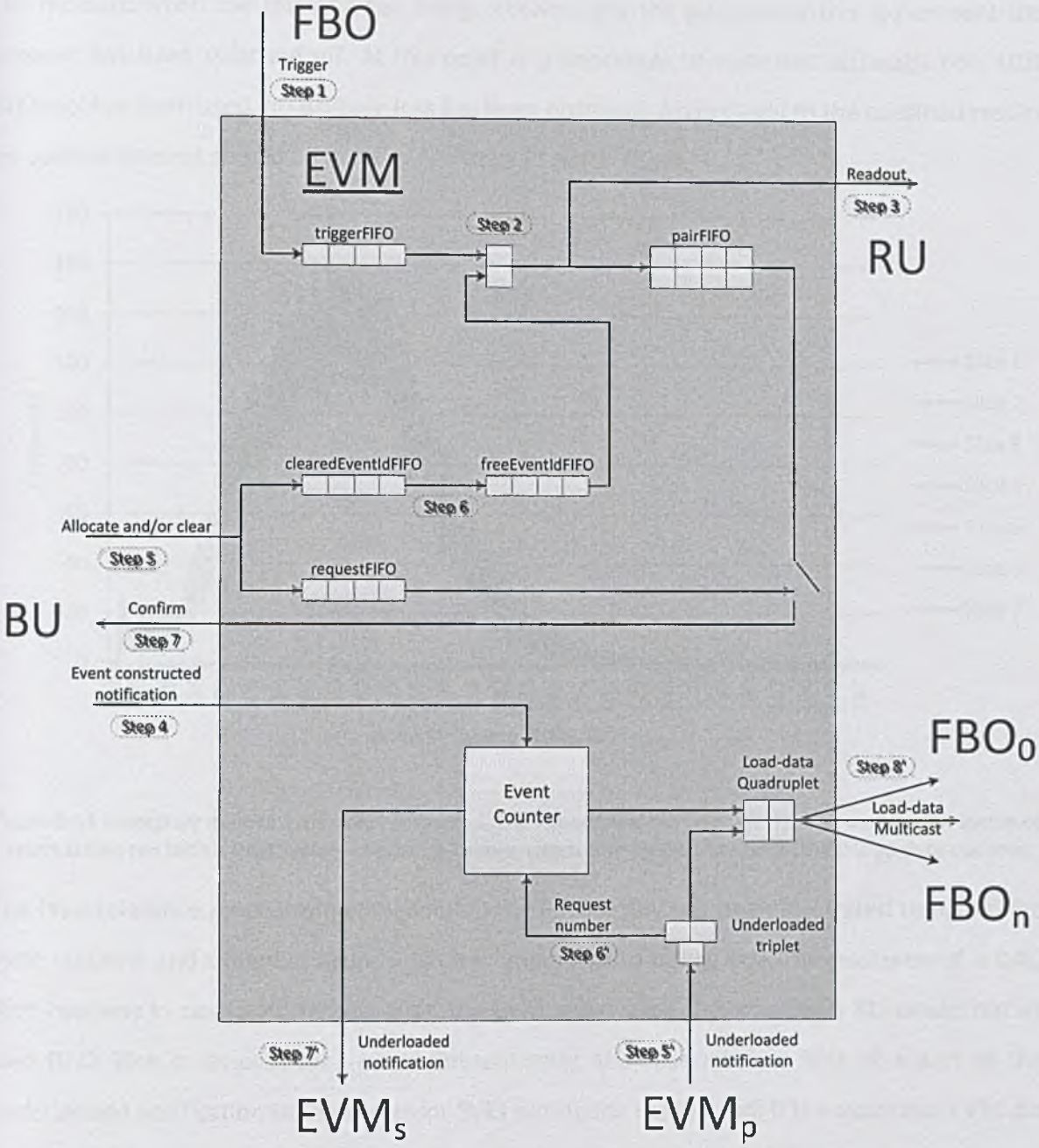


Figure 5-10 Event Manager internal FIFOs (Event Counter entity has been added, along with corresponding notification messages, due to the load scheduling algorithm)

The value of the timeout as well as the number of retries must be determined experimentally. Therefore, the distribution of round trip time has been studied (shown in figure 5-11). In this context by round trip time we mean the time from the point of multicasting the load-data until

the moment when the last ACK has being received. For the purpose of this experiment the timeout has been switched off. At this point it is important to note that although only UDP protocol has been used, no package loss has been observed. Accordingly to the obtained results an optimal timeout should be in the range from 12 ms to 15 ms.

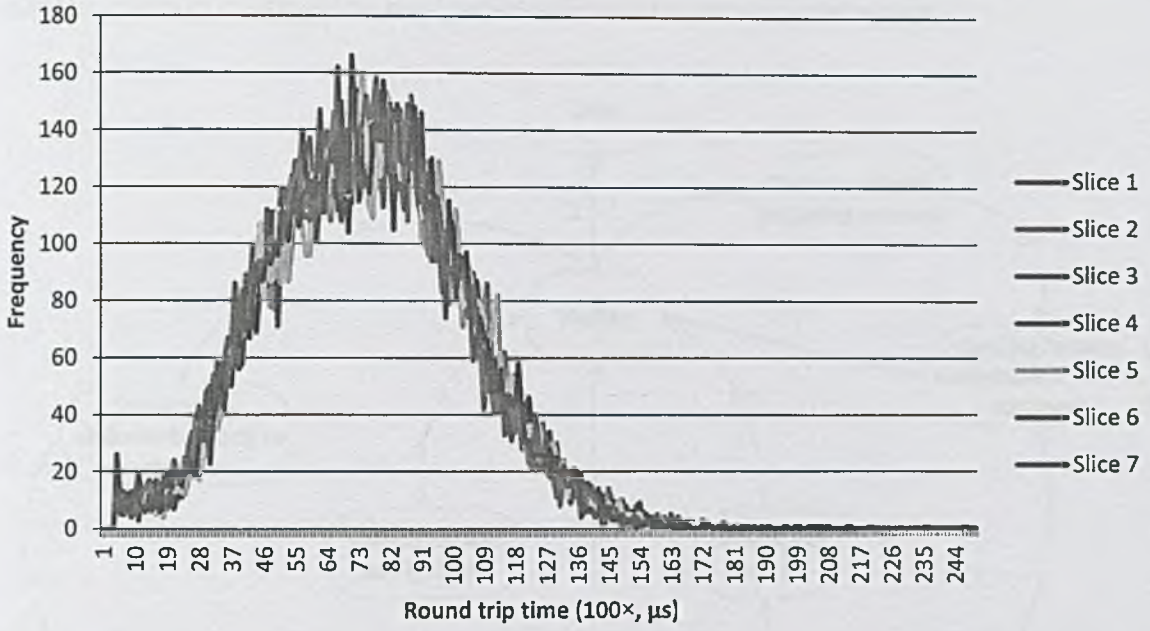


Figure 5-11 Exemplary multicast efficiency measured. The experiment has been carried out during 30 minutes of a data taking run with a DAQ System consisting 7 slices, which corresponds to about 10000 load-data updates.

The fault tolerance mechanism embedded in the EVM entity has been illustrated using a finite state machine and shown in figure 5-12. It is important to notice how the exclusion of a DAQ Slice happens in case a RU failure. First, the EVM supervising the erroneous RU masks out its own DAQ Slice in its own slice mask. Subsequently, the slice mask is sent as a part of the underloaded notification to the successor EVM during the next round. If the successor EVM did not send the underloaded notification to its successor yet, the slice mask will reach its successor in the same round. However, if the notification has been sent already the slice mask will be propagate further in the subsequent round, and so on. After receiving the slice mask the successor EVM will close the connection to the broken slice. It will however leave the respective port open. Until the slice mask reaches the predecessor of the broken slice, it will keep sending the underloaded notifications to the broken slice, which in turn will pass the notification further

(the connection will be reopened and then accordingly to the above described procedure closed again). When the mask will reach the predecessor EVM, it will close the connection to its broken successor, which will be effectively excluded from data-taking. Then a new successor will be resolved (the successor of the faulty slice) and an introduction message containing the predecessor slice number will be send, which will end the recovery procedure.

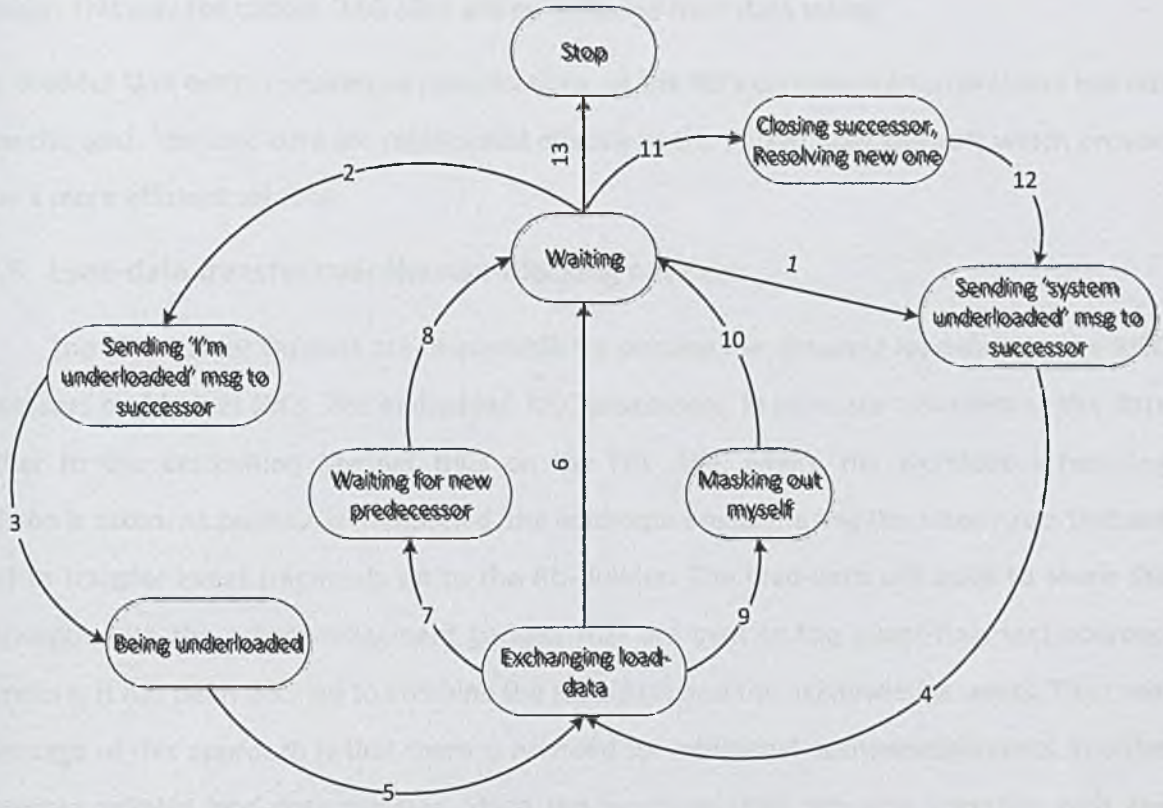


Figure 5-12 EVM's Finale State Machine, the transition explanation is given below:

1. **EVM received an 'underloaded' notification from its predecessor.**
2. **EVM became underloaded itself.**
3. **Automatically.**
4. **Automatically.**
5. **EVM received an 'underloaded' notification from its predecessor.**
6. **If requirements '7' and '9' are not met.**
7. **EVM's predecessor was masked out.**
8. **A new predecessor introduced himself.**
9. **EVM – RU communication timed out**
10. **Automatically.**
11. **EVM received an 'underloaded' notification from its predecessor (the successor has been masked out).**
12. **Automatically**
13. **After receiving the 'close' message from predecessor.**

The fault tolerance mechanism for an EVM failure is not yet fully implemented. Nevertheless, it is planned that the predecessor and successor will first detect the erroneous state using standard TCP features (Keep-Alive [75]), and subsequently they will close the connection. Then, the successor will start listening for a new predecessor. The predecessor, in turn, will introduce itself to the successor of its current, broken successor by sending a short message with its slice number. This way the broken DAQ Slice will be excluded from data taking.

The Readout Unit entity required no modifications, as the RU's communication protocol has not been changed. The load-data are multicasted directly to the FED-Builder Outputs which proved to be a more efficient solution.

5.3.5 Load-data transfer over the non-blocking network

The FED-Builder Outputs are responsible for passing the obtained load-data to the RISC processors on Myrinet NICs. The embedded RISC processors, in turn, are transmitting the data further to the destination Myrinet NICs on the FRL side, where the workload scheduling decision is taken. As previously mentioned, the load-data are sent using the same routs that are used to transfer event fragments up to the RU-Builder. The load-data will have to share the bandwidth with the acknowledgement packets that are sent to the event-fragment sources. Therefore, it has been decided to combine the load-data and the acknowledgements. The main advantage of this approach is that there is no need for additional acknowledgements in order to ensure reliable load-data transfer. Since the workload-data are sent together with the standard acknowledgement used by the event-fragment transmission protocol, in case of an acknowledgement loss, the event-fragment source will stop sending event-fragments as soon as it runs out of packet credits (previously described in section 3.1.1), and will resume the even-fragment transmission only if it receives a retransmitted acknowledgement (together with the load-data). This strategy implies of course that there is a small delay between receiving the load-data from the FBO and transferring them to FRLs. Although, the frequency of load-data-updates is much lower than the frequency with which the acknowledgements are sent, it has to be ensured that the FBO overwrites the load-data structure in the Myrinet RISC memory, only after the previous load-data have been transferred to all FRLs corresponding to the given RU.

Otherwise, in case when one load-data-update is triggered immediately after another (which could happen because of some temporary network issues in one DAQ Slice) the load-data could be overwritten by their successors before they were actually transferred to the FRLs. As a result, a FRL could obtain event counters from different load-data updates, which would lead to stopping the whole data acquisition process.

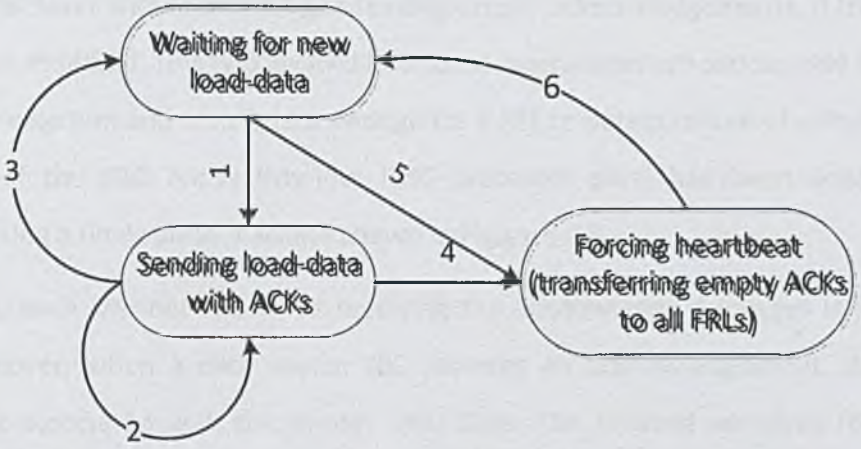


Figure 5-13 FBO's Finale State Machine, the transition explanation is given below:

1. **FBO received new load-data.**
2. **Some ACK were sent but not all FRLs of interest were served.**
3. **ACK were sent to all FRLs of interest.**
4. **Keep-alive timeout has been reached and keep-alive variable has been modified.**
5. **Keep-alive timeout has been reached and keep alive variable has been modified.**
6. **Automatically.**

A second function of the protocol, besides transporting load-data, is to ensure that the data source will be able to detect failure of one of the corresponding Readout nodes. In order to meet this requirement a heartbeat mechanism has to be applied. Since the acknowledgements are sent very often in respect to the frequency of load-data updates, they are strong candidates for also being used as the *keepalive* messages. However, if the efficiency of one DAQ Slice drops drastically, the ratio of the number of acknowledgements sent in this particular DAQ Slice to the number of load-data updates (which are triggered by fully operational DAQ Slices) will decrease as well. Therefore, a timeout should be added that triggers sending empty acknowledgements (no credits for new event fragments) to FRLs , if no acknowledgement has been send for longer than the minimum time that guaranties that the ACK packets can be used as the *keepalive* message. On the other hand, since the timeout would be implemented inside

of the Myrinet NIC's RISC processor, it would be insensitive to failures of other RU software or hardware components (it actually would cover only the power cat case). Therefore, in order to adapt this solution, another heartbeat layer is required. The second heartbeat can be realized in a very simple way, using the shared memory. Namely, the RU's software only has to modify (e.g. increment) periodically a *keepalive* variable in the shared memory. In turn, the timeout in the first heartbeat layer should only trigger sending empty acknowledgements, if the *keepalive* variable has been modified. This two layered heartbeat mechanism introduces only a very small overhead into the system and is sufficient enough for a FRL to detect failure of a Readout node. The behaviour of the FBO entity (Myrinet RISC processor part) has been illustrated in a simplified way using a finale state machine shown in Figure 5-13.

On the FRL site, each Myrinet NIC while receiving the acknowledge packages is updating its load-data. Moreover, when a data source NIC receives an acknowledgement, it resets the *timeout variable* associated with the sender DAQ Slice. The *timeout variables* (there is one corresponding to each filtering farm) are incremented continuously every 64 µs. If a variable reaches the threshold value it is assumed that the corresponding RU failed and therefore all pending packages that are queued, and were assigned to the particular DAQ Slice containing the broken RU are discarded. Also from this moment on, all packages that will be assigned to the particular DAQ Slice will be discarded as well. This is a necessary loss, because in case of a Readout node failure, the packages assigned to it would dominate the *ready-to-send* queue, which would result in stopping the whole data taking process. It is also not possible to transfer the event fragments to other DAQ Slices, because until new load-data will arrive other data source will be transferring the corresponding event fragments to the broken DAQ Slice (only the data sources that are sending the event fragments to the broken RU are able to detect the malfunction), which would lead to a mismatch error. After receiving the load-data from a Readout Unit, firstly it is ensured that the sender's DAQ Slice is not masked out in the currently used slice mask. Although, the information concerning the sender slice itself cannot do any harm since the slice is masked out, it is important not to take the load-data into account because of the additional data about the predecessor slice. Secondly, the request number, which identifies each load-data update, is checked. This way it is ensured that the load-data are

only used if they were received in the right sequence (the request number 2 should be received after 1, 3 after 2, and so on). If these two requirements have been met, the event counter of the sender slice is written into a circular buffer, also the slice mask that will be used with the newly obtained data is being updated (each set of event counters corresponding to each load-data update, has its own slice mask created using load-data only from this single update). Moreover, there is a second *slice ready mask* assigned to each update. Every time a counter is written into the buffer, the respective slice is market as ready. All DAQ Slices that have been masked out are considered as ready as well. Only if the given *slice ready mask* contains all DAQ Slices the corresponding set of counters is ready to be used (the scheduling decision must be based on counters from all filtering farms that are taking part in the data acquisition process). The predecessor's event counter that is also delivered in the load-data is processed exactly in the same way as the sender's event counter. If it has not been received earlier from the predecessor slice, it will be written into the circular buffer into the proper slot. It is important to note that after an event counter has been written into the circular buffer all other event counters corresponding to the same DAQ Slice and having the same request number will be ignored regardless of whether they have been received from the slice itself or from its successor.

In parallel, the FRL is receiving event fragments in form of smaller packages from the Front End Drivers. Each time an event fragment is completed, it is pushed into a dedicated queue. The content of the queue is check periodically. First it is determined whether there are any free credits (the value of at least one event counter has to be greater than 0), if yes a slice is assigned to the event fragment that has been retrieved from the front of the queue (the event fragment allocation algorithms will be discussed in detail in the next section). In the case when the allocated DAQ Slice is marked as broken the event fragment is discarded, otherwise it is pushed into another (*ready-to-send*) queue where it awaits transferring to the respective filtering farm.

A new set of counters may be used if, and only if all the counters in the currently used set are equal to 0 and the *slice ready mask* of the set that is next in turn contains all DAQ Slices. Therefore, these two conditions must be check always after a filtering farm has been allocated

to an event fragment or after new load-data have been received. In the case when the discussed conditions are fulfilled, all the event counters in the current set as well as the current *slice mask* are assigned with default values, and are marked as ready for reuse in the circular buffer. Afterwards, the next set of event counters becomes the current one, and is used by the allocation algorithm to assign event fragments to DAQ Slices.

The behaviour of the FRL entity (Myrinet RISC processor part) has been illustrated in a simplified way using a finale state machine shown in Figure 5-14.

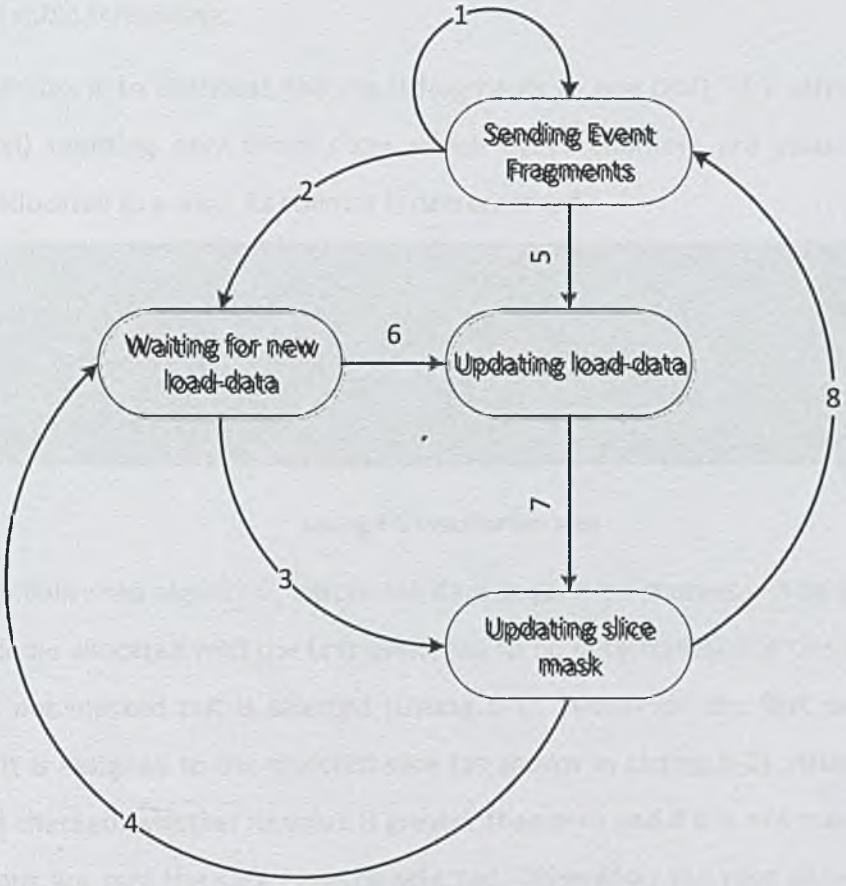


Figure 5-14 FRL's Finale State Machine, the transition explanation is given below:

1. **All event counters from the current set have been used, the new set of event counters is ready to use.**
2. **All event counters from the current set have been used, the new set of event counters is not yet ready.**
3. **Timeouted on slice's heartbeat.**
4. **All event counters from the current set have been used, the new set of event counters is not yet ready.**
5. **Received an ACK with piggybacked load-data.**
6. **Received an ACK with piggybacked load-data.**
7. **Automatically.**
8. **There are credits in current event counter set or the new event counter set is ready to use**

5.4 Event-fragment allocation algorithm

An event-fragment allocation algorithm is responsible for assigning an event fragment to a destination DAQ Slice. The algorithm is meant as a replacement for the currently used method (previously described in 3.1.1). Since the algorithm has to be implemented inside of the Myrinet NIC, and will be executed for each event fragment separately (on average once per 10 μs), it is crucial to keep its computational complexity as low as possible. Moreover, in a case when all DAQ Slices reached the underloaded state at the same time, the algorithm should mimic round robin scheduling.

Basically, the idea is to distribute the event fragments to one DAQ Slice after another (round robin fashion) omitting only those slices which event counters are zero. After an event fragment is allocated to a slice, its counter is decremented.

```
int alloc_slice;
for (alloc_sllice = 0; slice < NMB_SILCE; alloc_slice++) {
        if ( ! isMaskedOut(alloc_slice) ) break;
}
```

Listing 5-1 Initialization step

In order to initialize the algorithm, before the data acquisition process will be started a filtering farm that will be allocated with the first event has to be determined. For this purpose the first slice that is not masked out is selected (Listing 5-1). Whenever the first event fragment is completed, it is assigned to the selected slice (as shown in Listing 5-2). Afterwards, the next slice is being checked: whether its value is greater than zero and if it is not masked out. If those two conditions are met the slice is being selected. Otherwise, the next slice in turn is being checked, and so on, until a slice that meets the selection conditions is found. This procedure is repeated for all upcoming event fragments until it is not possible to find a slice that fulfils the requirements. In this case the next DAQ Slice after currently selected slice that is not masked out (the one that would be normally used if the counters were not zeros) is being selected. This way it can be easily determined that all event counters in the currently used set are zeros (the

selected counter is 0). Whenever the *slice ready mask* of the next set contains all slices, it will be used.

```
slice_index = alloc_slice;
zeroed_counters = 0;
--evt_counter[alloc_slice];
do {
        alloc_slice = alloc_slice + 1 == NMB_SLICE ? 0 : alloc_slice + 1;
} while(
        (evt_counter[alloc_slice] <= 0 &&
        ++zeroed_counters < NMB_SLICE + 1) ||
        isMaskedOutt(alloc_slice)
);
```

Listing 5-2 Workload allocation step

After the event counter sets have been swapped, it is important to double check whether the selected slice is not masked out or its value is not 0. The first slice in turn that meets those conditions is being selected (Listing 5-3).

```
for (int i = 0; i < NMB_SLICE; i++) {
        alloc_slice = alloc_slice + i == NMB_SLICE ? 0 : alloc_slice + 1;
        if (evt_counter[alloc_slice] >0 && ! isMaskedOut(alloc_slice))
                break;
}
```

Listing 5-3 Swap event counter sets step

Let us now consider the computational complexity of the algorithm 1. The *'initialization'* step of the algorithm is executed only once before the data-taking process starts so it has no impact on the system capacity. The *'swap event counter'* step is expected to be executed approximately once per 10 ms. The most expensive operations in this step are off course the *'for'* and *'if'* statements. Certainly, each of them will be executed at least once. Whether they will be executed more than once depends on whether the selected slice will send a zero-counter, or will be masked out. The probability of sending a zero-counter or masking out a DAQ Slice is very low since it is equal to the probability that the DAQ Slice in question is broken. Moreover, in the

case when a DAQ Slice is faulty, the system running with workload scheduling significantly outperforms the standard system. As a result, the overhead introduced by the discussed step is negligible and therefore not worthy analysing. The *'workload allocation'* step, however, will be executed at least once per 10 µs, and therefore it can have significant impact on the overall capacity. In this case, as well as in the previous one, the most expensive is the loop-related operation (the '*while*' statement). It will be executed at least one time per allocation step. Whether it will be executed more times depends on values of the load data. Let us consider a DAQ System of *m* DAQ Slices and an initial number of events *n*, allocating events accordingly to the following set of counters:

$$c_0 \geq c_1 \geq c_2 \geq c_3 \geq c_4 \geq \cdots \geq c_{m-1}$$

and

$$c_0 = \frac{n}{2}$$

The 'while' statement will be executed once per allocation step for the first $m \times c_{m-1}$ event-fragments (until c_{m-1} is greater than zero the loop's condition is never meet). Afterwards, it will be executed twice for event-fragments that will be assigned to the slice corresponding to c_0 counter, and once for other slice until c_{m-2} reaches zero, and so on. Therefore, the sum *S* of all '*while*' statements executions during the given workload scheduling cycle can be written as:

$$m \times c_{m-1} +$$

$$(m-1) \times (c_{m-2} - c_{m-1}) + 1 \times (c_{m-2} - c_{m-1}) +$$

$$(m-2) \times (c_{m-3} - c_{m-2}) + 2 \times (c_{m-3} - c_{m-2}) +$$

$$\ldots +$$

$$1 \times (c_0 - c_1) + (m-1) \times (c_0 - c_1)$$

the above equation can be easily transformed into:

$$m \times c_{m-1} +$$

$$m \times (c_{m-2} - c_{m-1}) +$$

$$m \times (c_{m-3} - c_{m-2}) +$$

$$\ldots$$

$$m \times (c_0 - c_1)$$

as a result:

$$S = m \times c_0$$

$$S = \frac{m \times n}{2}$$

Thus, the sum S is constant and does not depend on the counter's values and the average number of '*while*' operations per allocation step equals:

$$T_{avg} = \frac{m \times n}{2 \sum_{i=0}^{m-1} c_i}$$

It can be easily noticed that in the worst case scenario ($c_0 = \frac{n}{2}, c_1 = 0, \dots, c_{m-1} = 0$):

$$T_w = m,$$

while in the best case scenario ($c_0 = c_1 = \cdots = c_{m-1} = \frac{n}{2}$):

$$T_b = 1$$

It is important to notice that the algorithm 1 distributes the event-fragments in round robin fashion between those DAQ Slices whose counter values are greater than zero. This means in practise that if one of the slices builds more events than the others, the surplus will be sent at the very end of the scheduling round and as a result the most efficient slice will be disfavoured. Since the distribution of load-data counters has a serious impact on the algorithm's complexity, it is important to determine experimentally whether the algorithm 1 boosts the differentiation in the counter's values (significant variation of event counters' values may reduce the overall capacity of the DAQ System).

Let us now consider the size of the circular buffer. An event counter may be written into the buffer only if there is a free slot that has been marked as ready to reuse. If some load-data have been received but there is no free space for them, they will be ignored. Of course, as it has been described before, the load-data will be sent again to the data source together with all the upcoming acknowledgements. Therefore as soon as a slot is available the load-data will be written into the buffer. There is a risk however, that before there is a free slot another load-data update will be triggered. In this case, the new load-data will be transferred along with ACK

packages and the previous load-data will be lost. The new load-data will not be accepted even if there is free space in the buffer because of its sequence number (only load-data that have been received in the right sequence are accepted). Therefore, it is crucial to ensure that there is always a free slot in the circular buffer for the upcoming load-data. Thus, let us analyse the worst case scenario (Figure 5-15). Suppose there is a DAQ System for which the initial number of events equals 2000 (*n = 2000*) and the underloaded state is reached if 1000 events have been built. The system consists of four DAQ Slices and the state of the currently used set of event counters is 100, 200, 300 and 400 respectively for slice 0, slice 1, slice 2, and slice 3 (as shown in Figure 5-15, Step 1). The state of the next set of event counters that will be used is also known, it is 1000 for each of them. After distributing 100 event fragments to each slice (Figure 5-15, Step 2), there are 0 events in the current set of counters and 1000 events in the next set of counters corresponding to DAQ Slice 0. Since each slice is allocated with 2000 events, it is clear that 1000 events have been already sent to the slice itself. As a result it is entirely possible that DAQ Slice 0 will reach the underloaded state and trigger a load-data update. Suppose now, that there is a network problem in other DAQ Slices and all *event constructed notifications* have been delayed and therefore Event Counters will state that no events have been built. As a result a third set of counters will be created: 1000, 0, 0 and 0. Afterwards next 100 events will be sent to each slice, beside slice 0 which will be omitted because the respective event counter is 0 (Figure 5-15, Step 3). At this point there will be 1000 events inside of DAQ Slice 1, and if the network problem does not persist any longer in this slice, the underloaded state may be reached and a new load-data update will be triggered. In this case a fourth set of event counters will be created: 0, 1000, 0 and 0. Analogically, after sending further 100 events to slice 2 and 3 (Figure 5-15, Step 4), a fifth set of event counters will be composed: 0, 0, 1000 and 0. Let us now assume that transferring the final 100 events to slice 3 will result in another load-data update: 0, 0, 0 and 1000 (Figure 5-15, Step 5). At this

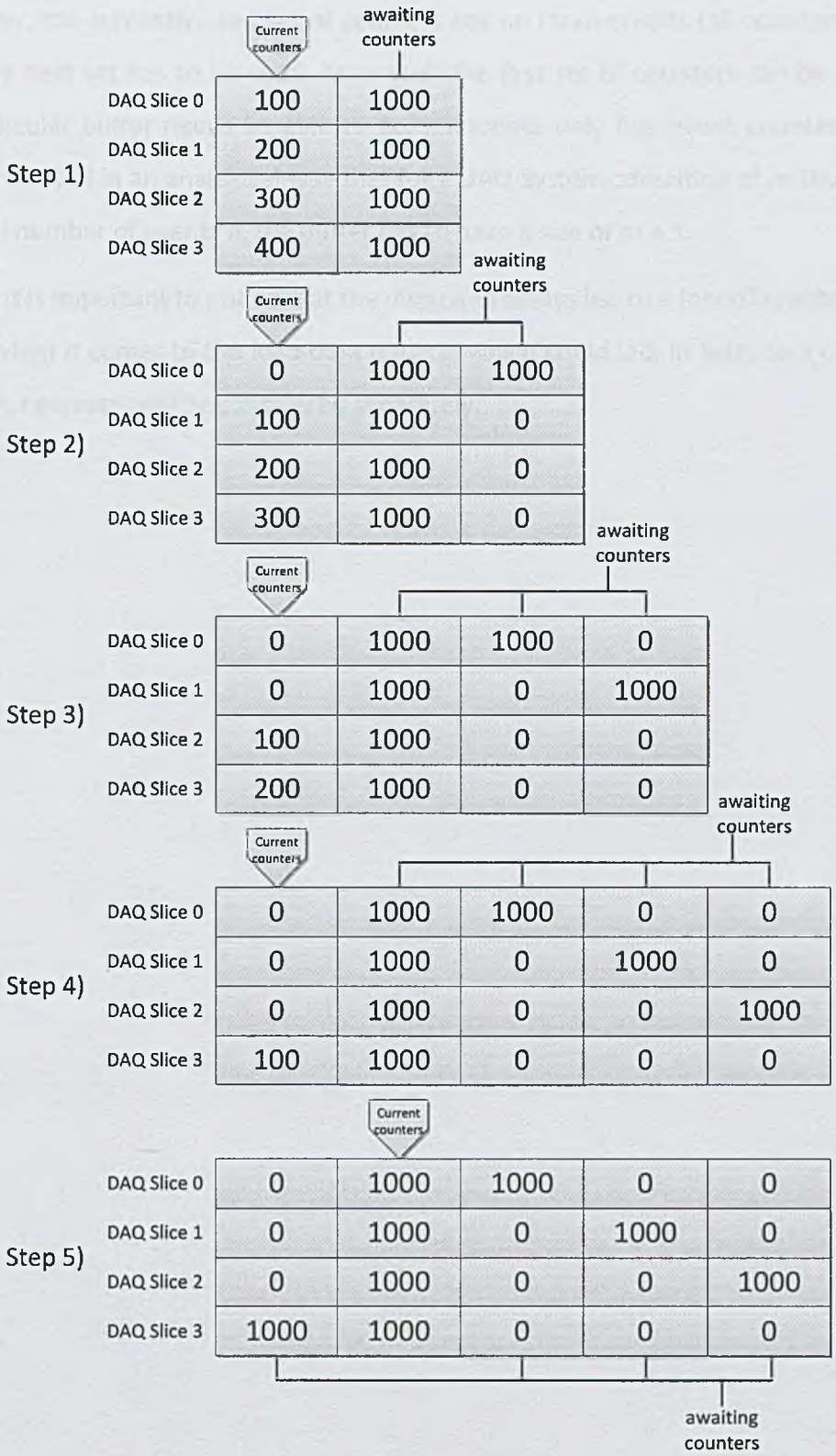


Figure 5-15 Circular buffer size – the worst case scenario

point however, the currently used set of counters has no more credits (all counters are 0) and therefore the next set has to be used. As a result the first set of counters can be reused, and hence the circular buffer needs be able to accommodate only five event counters. It can be easily demonstrated in an analogical way that for a DAQ System consisting of *m* DAQ Slices and for any initial number of events *n*, the buffer has to have a size of *m + 1*.

At this point it is important to notice that the discussed delays led to a loss of synchronization in the system when it comes to the load-data update, which could led, in turn, to a capacity loss. This problem, however, will be considered separately.

6 Results

The workload scheduling algorithm proposed in chapter 5 has been implemented in C/C++ and studied in the CMS testing environment as well as in the production Data Acquisition System. In this chapter we aim to give a complex overview of the conducted experiments and obtained results. First of all it will be measured how the algorithm affects the throughput of the FED-Builder network. The throughput of the standard system and the one running with the load scheduling algorithm will be compared. Also, the obtained results will help us to understand if a system that employs load balancing is meeting the CMS experiment's requirement, namely if it is able to sustain an average input of about 100 GB/s, which corresponds to 200 MB/s per single Readout Unit (RU). Then we will focus on the event building and reconstruction efficiency and the impact of the studied algorithm on this part of the data acquisition process. Afterwards, experiments on fault tolerance capabilities will be conducted and the respective results will be discussed. The behaviour of the system in case of software and hardware (both network links and computing nodes) failures will be studied. Finally, long data taking runs will be investigated in order to demonstrate the algorithms stability and robustness (in particular we will be investigating the system for run condition occurrences). During all those experiments we will be also monitoring whether the event allocation method assigns all the event fragments corresponding to a single event to the same filtering farm (DAQ Slice).

6.1 Network throughput

The prototype has been studied in the CMS DAQ production system during a technical stop in August 2011. Each of the computing farms participating in those tests was having 1 EVM, 63 RUs, 82 BU-FUs and 2 SMs. The goal of our experiments was to verify if the prototype meets the requirements of the CMS experiment and to investigate the overhead of the dynamic workload scheduling algorithm in respect to the standard static load allocation method. Moreover we studied the response of the system in case of network throughput degradation caused by a failure of a single network-link. The measurements have been carried out for a setup of 8 filtering farms (DAQ Slices). The throughput was measured for expected

event fragment sizes in the range from 128 B to 10 kB. First, the firmware running in all the Front-Ends was set to emulation mode so the DAQ system would be provided with artificial event-fragments. Then the event-fragment size has been set manually first to 128 B, then 256 B and so on until 10 kB. For each event-fragment size a data-taking run has been started and the obtained data acquisition rate has been noted. Those values have been then used to calculate the system's throughput. All the data taking runs were configured to drop the event data when they reach Builder Units (the '*drop at BU*' option has been used), meaning that the network was the only limiting factor. As a result the maximum throughput for each event-fragment size has been reached. The measurements were carried out for constant and variable event fragment sizes from log-normal distribution (stdev = 0.5 and stdev = 1.0). The initial number of events n was set to 3000.

The obtained results (shown in Figures 6-1, 6-2 and 6-3) confirm that the algorithm meets the design specifications of the CMS DAQ system, meaning that for 2 kB event fragments the throughput is greater than 200 MB/s (the CMS DAQ working point lays below the red curve). The CMS DAQ requirement is met for both the constant and the variable (stdev = 0.5 and stdev = 1.0) event fragment sizes. The actual conditions during a production data-taking run are most similar to those of Figure 6-2 where the standard deviation has been set to 0.5. Nevertheless, it is also important to prove that the system running with dynamic load scheduling can adapt to bigger or lower event size fluctuation as easily as the standard system and as a result that it can be also utilized if the experiment's nominal conditions change. Similarly, the expected event fragment size for CMS detector is 2 kB, however the measurements were carried out in the range from 128 B to 10 kB in order to demonstrate the flexibility of the algorithm (e.g. the expected event-fragment size in case heavy ion runs is much bigger than 2 kB). Although, the overhead of the scheduling mechanism is substantial, it is acceptable because during production data taking the limiting factor lies in the available computing power and not in the network throughput. The overhead is caused by the event allocation procedure (that is executed for each event fragment separately), which can be further optimized. The system running with dynamic workload scheduling similarly as the

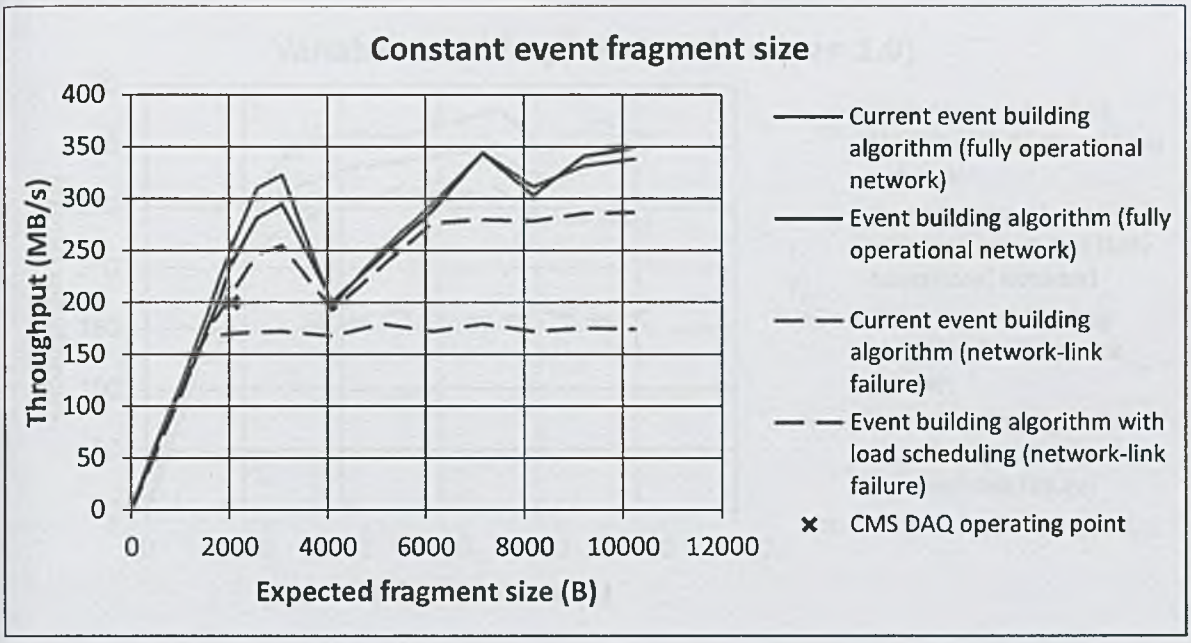


Figure 6-1 The available throughput in case of a fully operational network and in case of a network-link failure in one of the readout nodes for the static and dynamic scheduling mechanism, for constant event fragment size

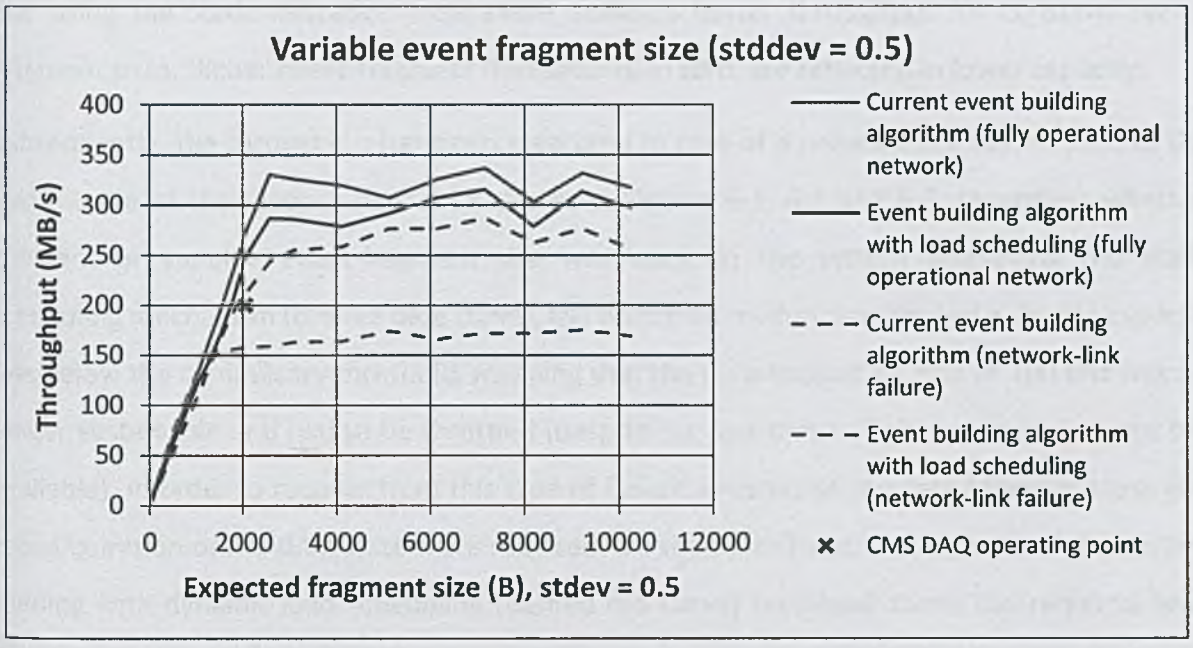


Figure 6-2 The available throughput in case of a fully operational network and in case of a network-link failure in one of the readout nodes for the static and dynamic scheduling mechanism, for variable event fragment size (stdev = 0.5)

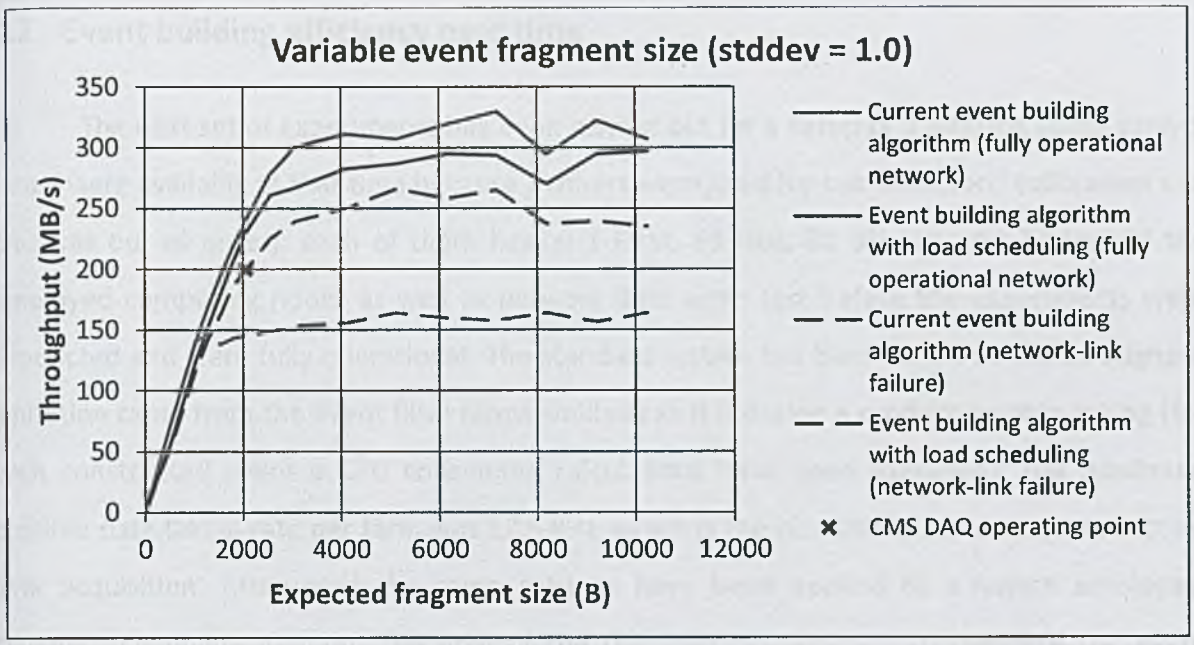


Figure 6-3 The available throughput in case of a fully operational network and in case of a network-link failure in one of the readout nodes for the static and dynamic scheduling mechanism, for variable event fragment size (stdev = 1.0)

one using the static allocation mechanism achieves better throughput for constant event-fragment sizes. Higher event-fragment fluctuations, in turn, are reflected in lower capacity.

Subsequently, the throughput has been measured in case of a network-link failure (one of the two) in one of the readout nodes. As shown in Figures 6-1, 6-2 and 6-3, regardless whether constant or variable event-fragment size was used, in the system employing the static scheduling mechanism (dashed blue curve), the discussed malfunction caused a drastic capacity loss below the compulsory threshold, meaning that the data acquisition rate of 100 kHz was no longer sustainable and had to be throttled (despite the fact that sufficient resources were still available). In order to recover from this type of failure a restart of the data taking process and reconfiguration of the DAQ system are required. On the other hand, the capacity of the system running with dynamic load scheduling (dashed red curve) remained above the required level (even for stdev = 1.0 it is still acceptable) and as a result the data taking run has been continued. It is important to notice that in contrary to the standard system, the data acquisition process was not interrupted and therefore it has been possible to avoid a downtime.

6.2 Event building efficiency over time

The next set of experiments has been carried out for a setup of 5 filtering farms (only 5 farms were available at that time because 2 others were used for sub-detectors' calibration and one was out of order), each of them having 1 EVM, 63 RUs, 82 BU-FUs and 2 SMs. All the employed computing nodes as well as network links were test before the experiments were conducted and were fully operational. The standard system has been tuned so the throughput limitation came from the event filter farms similarly as it is during a production data-taking (for each constructed event a CPU consuming calculations have been executed). The maximum possible data taking rate per farm was 12.5 kHz, which is the nominal speed during production data acquisition. Afterwards the same settings have been applied to a system employing dynamic scheduling method. The goal of the discussed studies was to compere the event building efficiency as well as the data acquisition rate for the standard system and the system running with dynamic workload scheduling algorithm. Moreover, we aimed to verify if in case of an unbalanced system an appropriate (proportional to the capacity) fraction of load is assigned to each filtering farm.

In the first experiment a short data taking runs have been started for 200 seconds each in order to verify which system builds more events in the same period of time. As shown in Figure 6-4 the system running with dynamic workload scheduling slightly outperformed the standard system, which means that most likely the resource utilization in case of dynamic scheduling is more efficient. In order to further study this problem, long (1 - 2 hour) data-taking runs were analysed. First, we focused on the aggregated data taking rate for all the filtering farms that were participating in those runs (Figure 6-5). This experiment confirmed that the resource utilization is more efficient in case of the system which is dynamically allocating the workload (the data acquisition rate is about 3.5% higher) for both constant and variable (stdev = 0.5 and stdev = 1.0) event fragment size. In contrary to the short runs from the previous experiment, in case of longer runs no differences in data taking rate has been observed between the runs that were made for constant and variable event fragment sizes. It

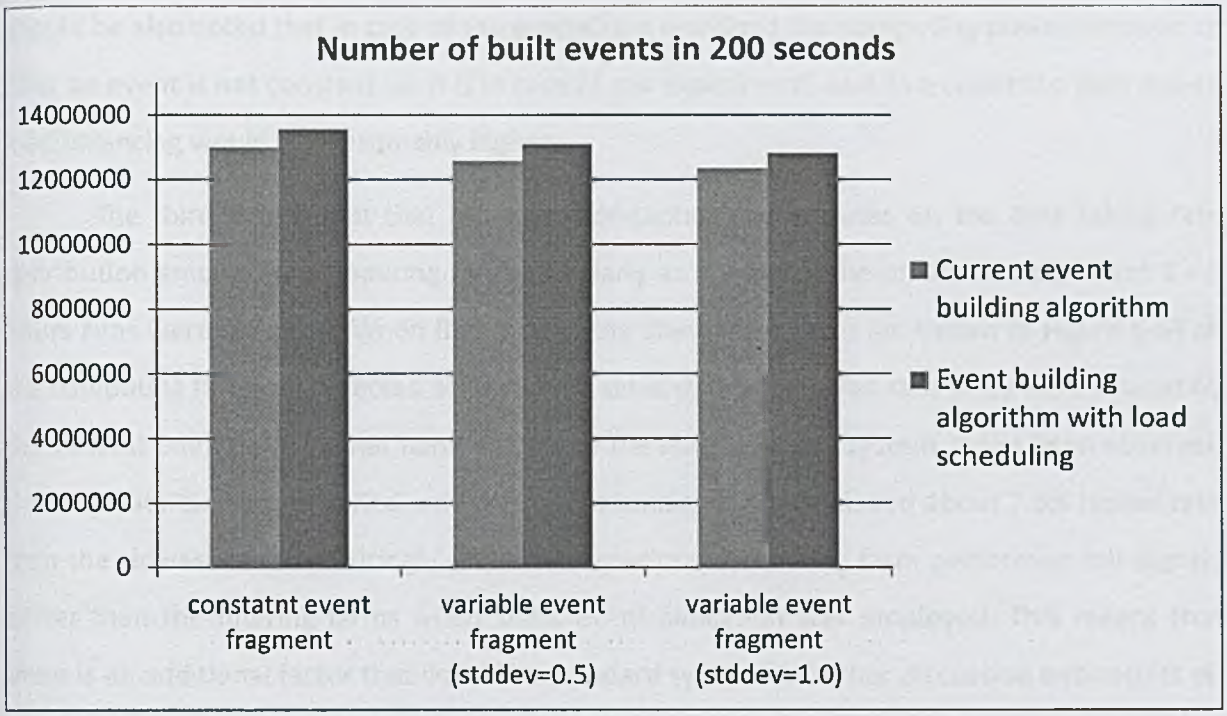


Figure 6-4 Number of built events during a 200 s run for the static and dynamic scheduling mechanism

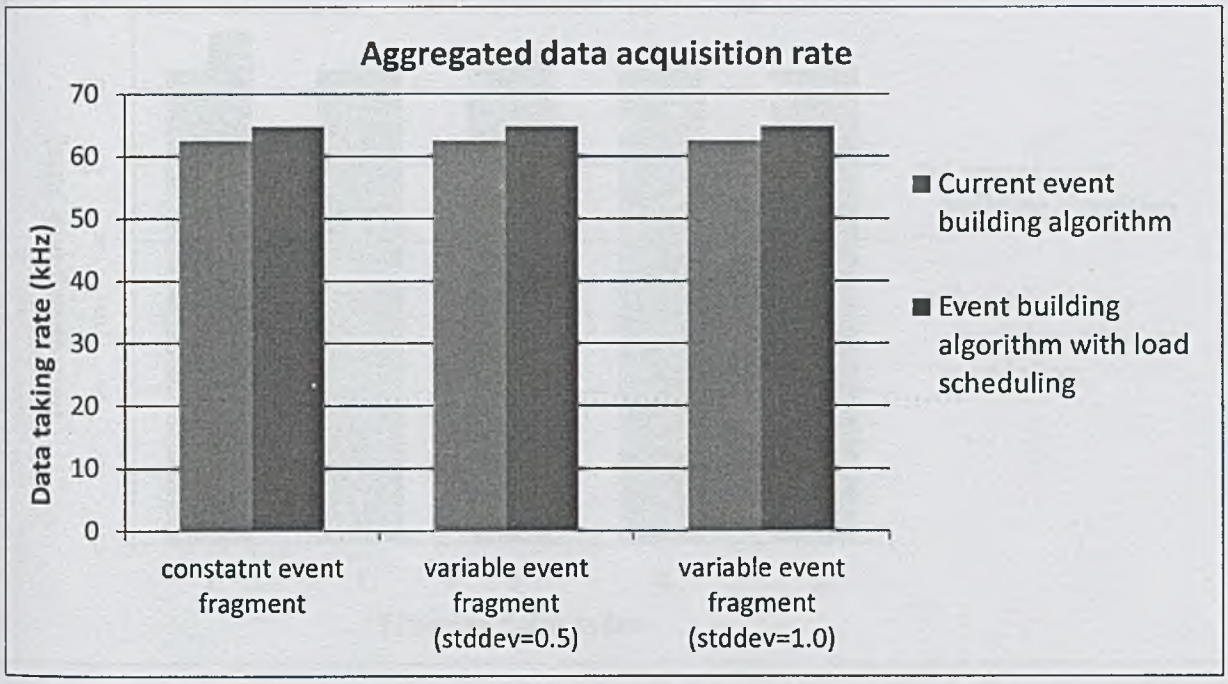


Figure 6-5 Aggregated data acquisition rate for constant and variable event fragment size for 1 hour data-taking run

should be also noted that in case of the production workload the computing power required to filter an event is not constant (as it is in case of our experiment) and as a result the gain due to load balancing would be presumably higher.

The third experiment that has been conducted concentrates on the data taking rate distribution among the computing farms. Similarly as in case of the previous experiment 1 – 2 hours runs were analysed. When it comes to the standard system (as shown in Figure 6-6) all the computing farms as expected achieved the same data acquisition rate (they were limited by the slowest one). On the other hand, in case of the load balanced system, it has been observed that the data taking rate varied, and the most capacious farm achieved about 7.5% higher rate than the slowest one. Surprisingly, the least capacious computing farm performed still slightly better than the filtering farms when static event allocation was employed. This means that there is an additional factor that limits the standard system. A further discussion and results on this topic will be presented in the following section.

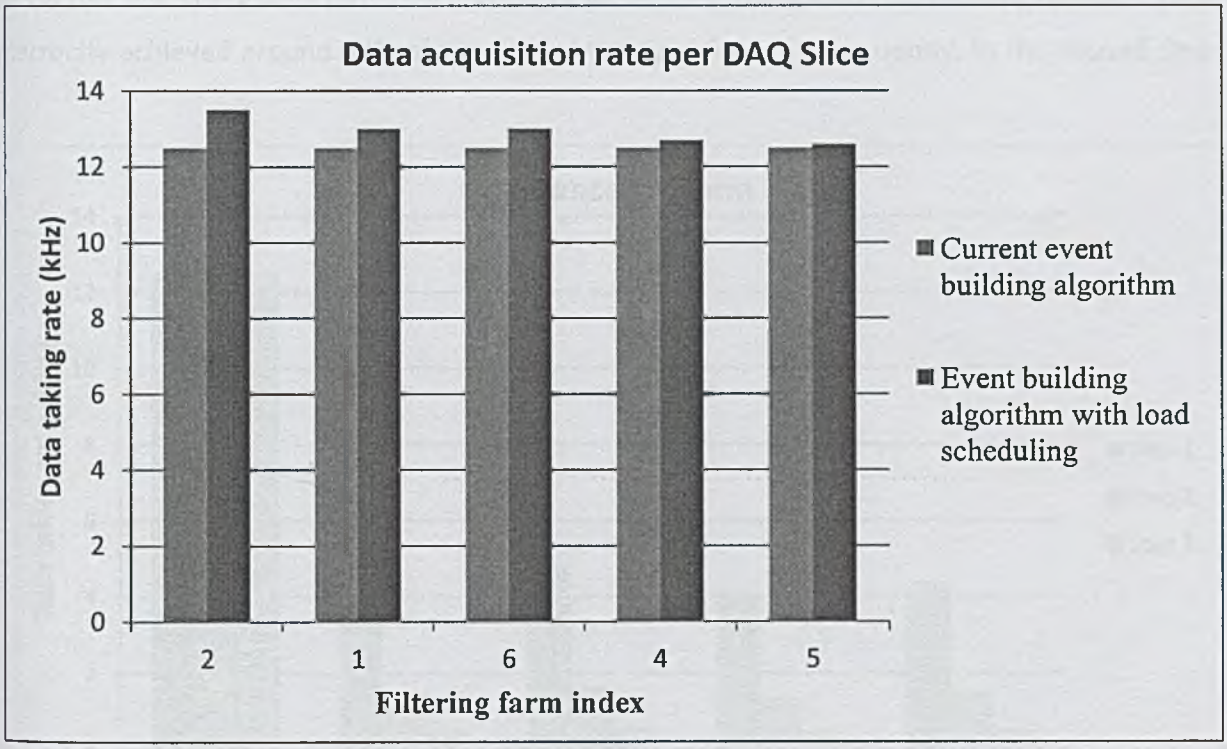


Figure 6-6 Data acquisition rate per DAQ Slice for variable event fragment size (stdev = 0.5) for 1 hour data-taking run

Subsequently, we conducted further studies to verify if the fraction of workload allocated to each filtering farm is proportional to its capacity. There is a particular use case that the dynamic load scheduling aims to address, namely a system consisting of a single filtering farm that can sustain the whole incoming workload with additional small computing farms that are used for monitoring purposes (real-time monitoring of the quality of the selected events). Moreover, the capacity of the monitoring farm should be reducible on demand (if the event quality is high only a very small fraction of the events should be consumed by the monitoring farms). In order to investigate the discussed use case as well as the general ability to allocate a correct fraction of load to filtering farms the following experiment has been conducted. A similar setup of 5 filtering farms as in previous experiments has been used, and the system has been also tuned so the maximum possible data taking rate per farm was 12.5 kHz. However, in 4 of them a substantial number of BU-FU nodes have been switched off (55 of 82, around 67%). Afterwards a data taking run has been started. As shown in Figure 6-7, in the first step, the *'big'* computing farm reach 100% of its capacity (12.5 kHz). The smaller farms, in turn, correctly achieved around 33% of the rate of the *'big'* farm. Subsequently, in the second step,

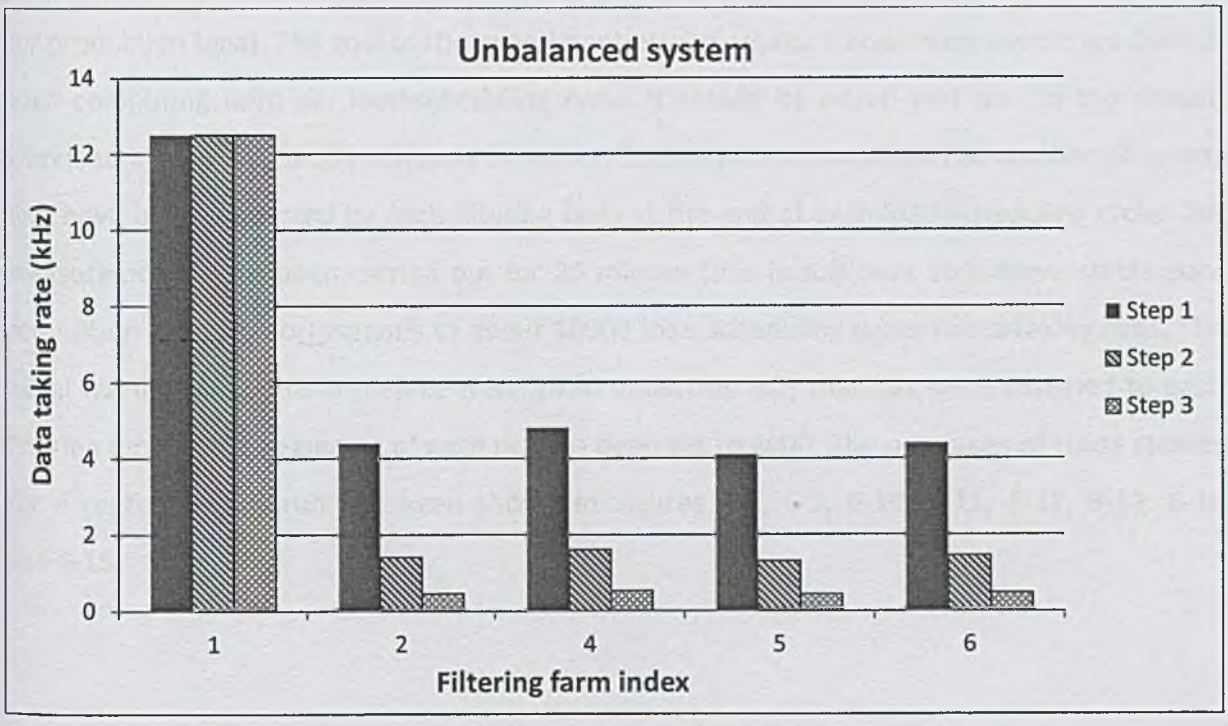


Figure 6-7 the *'big'* filtering farm use case

two thirds of the BU-FU nodes in the monitoring farms were switched off. This action, which is very important, had no impact on the data taking rate of the *'big'* farm. The data acquisition rate of the monitoring farms as expected has been reduced by two thirds. Finally, in the third step, again two thirds of the remaining BU-FU nodes in the small farms were switched off, and again positive results were obtained. There was no impact on the *'big'* farm and the data acquisition rate of the other computing farms dropped as expected by 66%. This way it has been demonstrated that the algorithm can cope with a heavily unbalanced system and that it allocates the workload proportionally to the capacity of the consumers.

6.3 Event building efficiency per load-scheduling cycle

In order to understand better the nature of the network throughput overhead as well as to further analyse the reason for the increased data acquisition rate, which can be reached, the event building efficiency per load-scheduling cycle has been studied. Again the production system (a setup of 8 filtering farms) has been used. One of the participating computing farms (DAQ Slices) was experiencing problems at that time and therefore its capacity was significantly smaller. Each farm was having 1 EVM, 63 RUs, 82 BU-FUs and 2 SMs (same configuration is used for production load). The goal of the experiment was to measure how many events are built in each computing farm per load-scheduling cycle. It should be noted that due to the chosen workload index (described previously in section 5.2) this corresponds to the number of events that have been requested by each filtering farm at the end of each load-scheduling cycle. The measurements have been carried out for 25 minute (this is sufficient to achieve stable data acquisition rate and corresponds to about 10000 load scheduling cycles) data-taking runs. The initial number of events n (details were given in section 5.2) that has been assigned to each filtering farm at the beginning of each run has been set to 4000. The outcomes of those studies for a representative run has been shown in Figures 6-8, 6-9, 6-10, 6-11, 6-12, 6-13, 6-14 and 6-15.

Figure 6-12 Number of events per cycle for

Figure 6-10 Number of events per cycle

Figure 6-8 Number of events per cycle for

Figure 6-13 Number of events per cycle for

Figure 6-11 Number of events per cycle

Figure 6-9 Number of events per cycle for

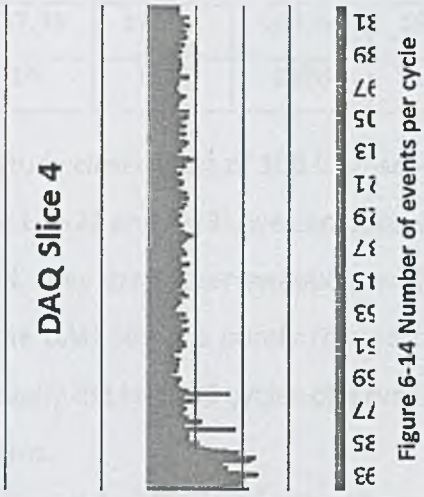


Figure 6-14 Number of events per cycle

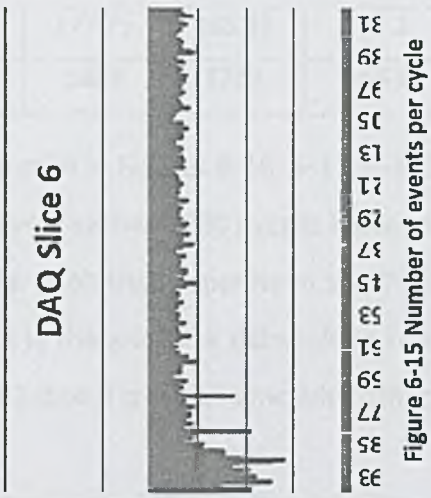


Figure 6-15 Number of events per cycle

It has been observed that there are fluctuations in the event building efficiency per load-scheduling cycle in each of the participating computing farms. Moreover, it has been noticed that it happens very rarely (almost never) that more than one filtering farm reaches the expected 2000 events in the same cycle. The fluctuations are bigger at the beginning of the run and are getting more stable after about 1500 cycles. In order to analyse the fluctuations the average (avg.) and standard deviation (stdev1 and stdev2) have been calculated (as shown in Table 6-1). In case of standard deviation (1) for the purpose of calculation it has been assumed that the expected value is the average. In case of standard deviation (2), in turn, it has been assumed that the expected value is 2000. Last row gives us the number of load-data transfers that were triggered (trg.) by the given DAQ Slice. It can be easily noticed that DAQ Slices 1, 2, 3, 5, 6 and 7 have almost the same capacity. Although, Slice 4 is just slightly less efficient (around 1%), it triggers significantly less load-data transfers (about 23%). This behaviour is understandable because in order to trigger load-data transfer a filtering farm has to build exactly 2000 events. Therefore, a small decrease in the capacity will result in a drastic reduction in the number of load-data transfer that a DAQ Slice triggers. The malfunction of DAQ Slice 0 is reflected in the considerably smaller capacity of the farm.

Table 6-1 Summarized analysis of the studies on event building efficiency per load-scheduling cycle

	Slice 0	Slice 1	Slice 2	Slice 3	Slice 4	Slice 5	Slice 6	Slice 7
avg.	1479	1870	1872	1875	1857	1877	1872	1872
stdev (1)	135.58	133.3	133.71	135.25	132.92	129.17	135.66	133.85

stdev (2)	537.95	186.12	184.94	183.92	194.5	177.75	185.91	185.2
trg.	14	1611	1670	1700	1292	1687	1717	1663

If we study closer a set of 100 load-scheduling cycles (shown in Figures 6-16, 6-17, 6-18, 6-19, 6-20, 6-21, 6-22 and 6-23), we can observe that a farm never reaches 2000 events twice in a row. Instead, they are rather swapping with each other. The peak that happens in the 74th cycle for all the DAQ Slices is purely related to the fact that it is the end of a data-taking run (these are actually the last 100 cycles of a run) and that the DAQ Slice 0 is out of sync with other computing farms.

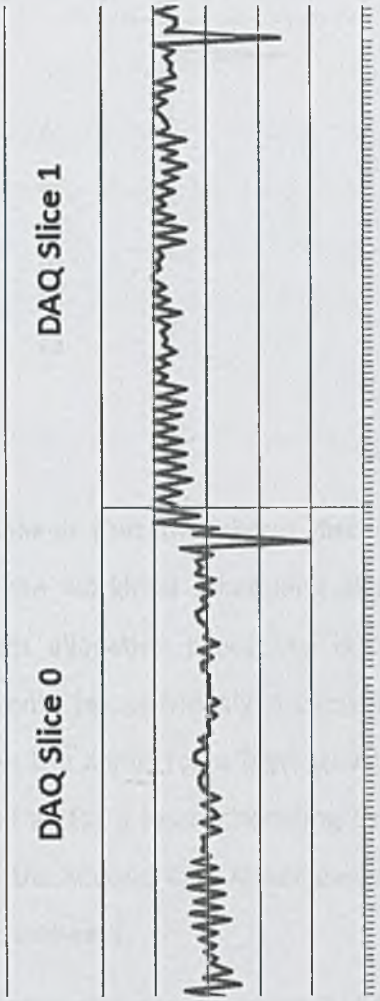
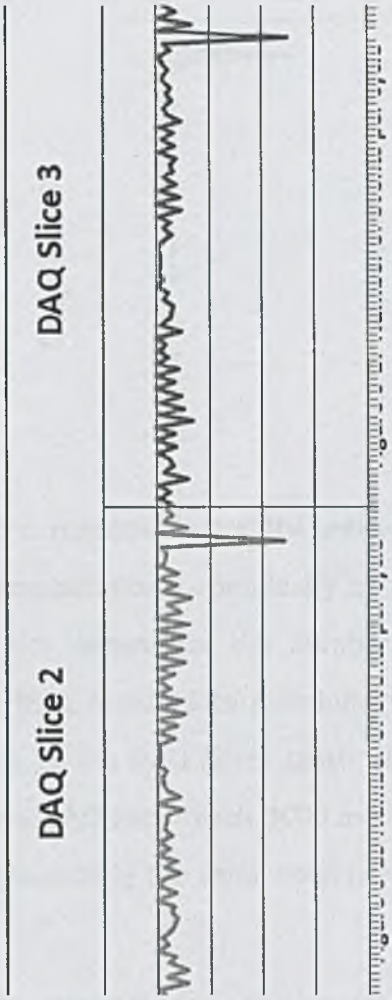


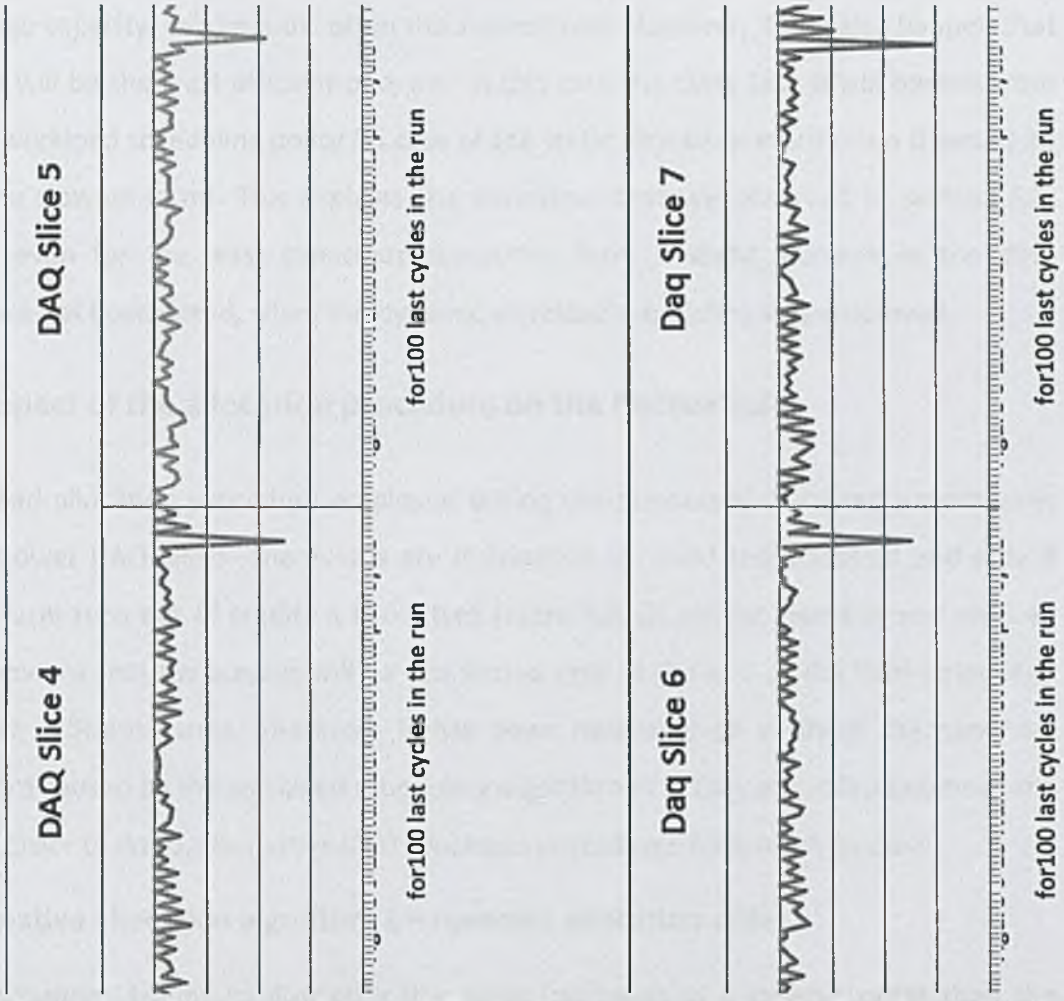
Figure 6-16 Number of events per cycle

Figure 6-18 Number of events per cycle



for100 last cycles in the run

for100 last cycles in the run



We believe that the above discussed fluctuations are responsible for the network overhead of the workload scheduling algorithm. Since the computational complexity of the event-fragment allocation procedure is constant and does not depend on the number of requested events (as previously proven in section 5.4), the time required to distribute the events will be the same for a load-scheduling cycle in which all the DAQ Slices reach 2000 events as well as for a load-scheduling cycle in which just one DAQ Slice reach 2000 events. Therefore, in the second case fewer events will be allocated in exactly the same time (which results in an overhead).

The computing farm's event building efficiency varies depending on the workload-scheduling cycle, as a result also the index of the slowest farm is not constant. For example in the first load-scheduling cycle the DAQ Slice 0 might be the slowest one, in the subsequent cycle DAQ Slice 4 might by the slowest one, and so on. Of course the DAQ Slice 0, which has the

lowest, average capacity, will be most often the slowest one. However, it will also happen that another farm will be the least efficient one, and in this case the DAQ Slice 0 will benefit from the dynamic workload scheduling policy (in case of the static allocation mechanism it would be limited by the slowest farm). This explains the behaviour that we observed in section 6.2, namely that even for the least capacious computing farm a slight increase in the data acquisition rare has been noted, when the dynamic workload scheduling was employed.

6.4 The impact of the allocation procedure on the fluctuations

The load allocation procedure employed during the previously discussed experiments favours the slower DAQ Slices. The events are distributed in round robin fashion and only if a computing farm runs out of credits it is omitted (more details can be found in section 5.4). This actually means that the surplus will be distributed only at the end of the load-scheduling cycle to more efficient farms. Therefore it has been necessary to verify if the observed fluctuations are caused by the workload scheduling algorithm or if they are independent of the algorithm. In order to do so, alternative load allocation procedures have been studied.

6.4.1 Alternative allocation algorithm 1 – reversed allocation order

The algorithm 1 aims to distribute the event-fragments in a reverse order than the previously employed algorithm (described in section 5.4). Thus, the first event fragment is allocated to a DAQ Slice with the highest event counter (event counters are populated with values obtained from requests received from respective filtering farms, as previously described in subsection 5.3.5), and then the counter is decremented. The second event is allocated once more to a slice whose counter has currently the highest value, and again the counter is decremented. This decision process is repeated afterwards for all subsequent event fragments. Since the scheduling decision has to be taken for each event fragment separately (once per 10 μs) it is crucial to keep the computational complexity of this step as low as possible. As a result, most of the logic has to be moved to the step preceding the workload scheduling where an appropriate data structure has to be prepared. It has been decided that the slices' indexes will be stored in a two dimensional $M \times M$ (where M is the number of DAQ Slices) array called

'slices'. The currently used row of the table will be pointed by *'curr_arr'* variable. An array *'slices_len'* of length *M* will store in each row the number of slices' indexes that are stored in a respective row in the *'slices'* array. Similarly, an array *'slices_credits'* will store number of event fragments that will be allocated using a respective row in the *'slices'* array. A variable *'arr_count'* will have the information about how many rows in the *'slices'* array are in use. Finally, a variable *'curr_index'* will point at the next slices' index in the current row of *'slices'* array that will be used.

In the initialization step (Listing 6-1) *'curr_arr'* and *'curr_index'* are set to zero (the event fragment allocation will start with first column of first row of the *'slices'* array). *'arr_count'* is set to 1, which means that only the first row of the *'slices'* array will be used. The first rows of *'slices_len'* and *'slices_credis'* are set respectively to *M* and $M \times n$ (where *n* is the initial number of events assigned to each slice), thus the first row of *'slices'* will be responsible for distributing all the initial events between all DAQ Slices. Finally, the first row of *'slices'* is initialized with slices' indexes in such a way that the initial events will be allocated in round-robin fashion. The first event fragment (Listing 6-2) is assigned to a slice whose index is stored in the first column

```
curr_arr = 0;
arr_count = 1;
curr_index = 0;
slices_len[0] = 0;
slices_credits[0] = NMB_SLICE × n/2;

int index = 0;
for (int i = 0; i < NMB_SLICE; i++) {
        if  (!isMaskedOut(i) {
                ++slices_len[0];
                slices[0][index] = i;
                ++index;
        }
}
```

Listing 6-1 Initialization step

of the first row of the '*slices*' array (according to the initialization step it will be the first slice that was not masked out) The second event fragment is assigned to a slice whose index is stored in the second column, and so on. After assigning an event fragment to a slice whose index is in the last column, again the first column is used, then the second, then the third and so on. Whenever en event fragment is assigned using the currently used row a corresponding row in the '*slices_credis*' is decremented. When the currently used row runs out of credits, the 'curr_arr' index is incremented, and the subsequent event fragments are distributed using next row. If all event fragments that were allocated for the given workload scheduling round have been distributed ('*curr_arr*' equals to '*arr_count*') the next event counter set (if available) has to be utilized.

```
int slice_index = slices [curr_arr][curr_index];
curr_index = (curr_index == slices_len[curr_arr]) ? 0 : curr_index + 1;
--evt_counter[index][curr];
--slices_credits[curr_a];
if (credits[curr_arr] ==
                        ++curr_arr;
```

Listing 6-2 Workload allocation step

Before the used set of counters can be swapped with a new set of counters, the '*slices*' array has to be prepared for another workload scheduling round (Listing 6-3). First, it has to be established how many distinct event counters values are there ('*arr_len*'). Next, all the distinct counter values have to be sorted in descending order. Due to the fact that the standard system

```
int counter_vals[9] = {0};
arr_len = 0;
for (int i = 0; i < NMB_SLICE; i++) {
        if (evt_counter[i][next] == 0) continue;
        recur = FALSE;
        for (int j = 0; j < arr_len; j++) {
                if (counter_vals[j] == evt_counter[i][next]) {
                        recur = TRUE;
                        break;
                }
        }

        if (!recur) {
                counter_vals[arr_len] = evt_counter[i][next];
                arr_len++;
        }
}

sort (counter_vals);

zero (slices_len);
for (i = 0; i < NMB_SLICE; i++) {
        for (j = 0; j < arr_len; j++) {
                if (evt_counter[i][next] >= counter_vals[j]) {
                        slices[j][slices_len[j]] = i;
                        slices_len[j]++;
                }
        }
}

for (i = 0; i < arr_len; i++) {
        slices_credits[i] = slices_len[i] * (counter_vals[i] - counter_vals[i+1]);
}
```

Listing 6-3 Swap event counter sets step

consists of 8 DAQ Slices at maximum, in the worst case there will be 8 distinct counter values. Since the number of elements is limited to 8, and also to keep the implementation simple, the bobble sort [76] algorithm has been used. Afterwards, the '*slices*' array has to be filled with respective slice's indexes. The first row will be populated with slice's indexes, whose event

counters are greater or equal to the first (the highest) of the sorted counters. Then, the second row will be populated with slice's indexes, whose event counters are greater or equal to the second of the sorted values, and so on. In the last step, each row in the '*slices_credits*' array has to be assigned with a number of credits that corresponds to the number of event-fragments, which will be distributed using respective row in the '*slices*' array.

Let us now consider the computational complexity of the alternative algorithm 1. The 'initialization' step of the algorithm 1is executed only once before the data-taking process starts and as a result it has no impact on the system capacity. The 'workload allocation' step, will be executed at least once per 10 μs and therefore it is the most crucial part of the algorithm. The most expensive operation is the 'if' statement, which will be executed only once per allocation step. On the contrary to the standard allocation algorithm (described in section 5.4), this does not dependent on the values of event counters, which is a desirable property. Most of the logic has been moved to the '*swap event counter*' step, which is expected to be executed approximately once per 10 ms. The most expensive operations in this step are off course the 'for' and 'if' statements. Certainly, each of them will be executed at least once. Whether they will be executed more than once depends on whether the selected slice (the one that has been selected to be the next destination) will send a zero-counter, or will be masked out. The probability of sending a zero-counter or masking out a DAQ Slice is very low since it is equal to the probability that the DAQ Slice in question is broken. Moreover, in the case when a DAQ Slice is faulty, the system running with workload scheduling significantly outperforms the standard system. As a result, the overhead introduced by the discussed step is negligible and therefore not worthy analysing.

6.4.2 Alternative allocation algorithm 2 – intermediate solution

The alternative algorithm 2 is an intermediate solution between the standard dynamic allocation procedure and the alternative algorithm 1. At the beginning of a load-scheduling cycle the computing farm that requested the highest number of events is fount. Subsequently, in the first round each DAQ Slice is assigned with one event, in addition the farm that requested the highest number of events is allocated with an extra event. This way, the most capacious

slice gets more events. It can be noted, however, that the event counter (event counters were previously described in subsection 5.3.5) corresponding to the most efficient slice will be decreasing much faster than the other counters. Whenever, another counter starts being equal to the counter of the most capacious filtering farm, it will be also getting an additional event (similarly as the most efficient farm) per round. In order to illustrate how the algorithm works, let us suppose that the algorithm is distributing the load between 3 filtering farms, and that it received the following set of counters: DAQ Slice 0 – 6, DAQ Slice 1 – 5 and DAQ Slice 2 – 4. In the first round slice 0 will be allocated with 2 events (it is the most efficient farm), slices 1 and 2, in turn, will be allocated with 1 event each. After the first round the state of event counters will be as follows: DAQ Slice 0 – 4, DAQ Slice 1 – 4 and DAQ Slice 2 – 3. In the subsequent round slice 0 will be again allocated with 2 events, this time, however, also slice 1 will be allocated with 2 events (its counter has the same value as the counter of the most efficient farm). Slice 2 will be allocated with 1 event. After the second round the state of event counters will be as follows: DAQ Slice 0 – 2, DAQ Slice 1 – 2 and DAQ Slice 2 – 2. In the third and also the last round all the DAQ Slices will be allocated with 2 events, because all the counters have the same value as the one corresponding to the most capacious farm.

A circular doubly linked list has been employed to store data that are used to evaluate which farm will be chosen as the next destination. The maximum number of events that can be requested by a filtering farm is equal to $^{n}/_{2}$, where n is the initial number of events assigned to each farm (see section 5.2 for more details). At the beginning of a load-scheduling cycle, all the DAQ Slices that requested $^{n}/_{2}$ events are being found. Then, each farm's index is being added to the linked list – the indexes of DAQ Slices that requested the maximum are added twice while the indexes corresponding to the other slices are added once. A marker will be used to store the current position in the list and initially it will be set to the head. The first event-fragment will be allocated to the farm that is pointed by the marker (the first index in the list), the respective event counter will be decremented and the marker will be moved to the next element in the list. Whenever, an event counter becomes equal to those that were set to $^{n}/_{2}$ at the beginning of the cycle, the respective index is being added to the linked list for a second time. On the other hand, whenever a counter becomes equal to 0, all the respective indexes are

removed from the list. When the list becomes empty, a new set of event counters has to be utilized if available. Since adding and removing an element from a doubly linked list are trivial operations the overhead of the algorithm is negligible.

6.4.3 Comparison of the event building efficiency per load-scheduling cycle for different event-fragment allocation methods

In order to compare the different event-fragment allocation methods, the event building efficiency per load-scheduling cycle for each of them have been studied. For the purpose of these experiments the CMS DAQ test environment has been used. At that time the CMS DAQ production system has not been available, and it has been found out that it is also possible to observe and study the fluctuations in the CMS DAQ test system. A four DAQ Slice setup with 1 EVM, 3 RUs and 4 BU-FUs per DAQ Slice have been used. One of the computing nodes in the DAQ Slice 0 was experiencing problems at that time and therefore the capacity of the farm was reduced in respect to the three other farms. The system was tuned so that the throughput limitation would come from event filter farm and the maximum data taking rate would be 50 kHz (12.5 kHz per DAQ Slice, which is the nominal speed for the production system). The initial trigger rate was set to 50 kHz. The measurements have been carried out for 50 minute (since we had a much smaller system available for this set experiments longer runs were studied) data-taking runs. The initial number of events *n* (see section 5.2 for more details) that has been assigned to each filtering farm at the beginning of each run has been set to 4000 (similarly as in case of the production system). The outcomes of those studies for representative runs have been summarized in Tables 6-2, 6-3, 6-4 and 6-5. It should be noted the results obtained for the standard dynamic load allocation method in the CMS DAQ test system converge with the results that were obtained for the CMS DAQ production system (presented in section 6.3). In case of the system running with the static allocation mechanism the average number of events built per load-scheduling cycle is much closer to the expected 2000 events. This behaviour, however, is caused by the static allocation mechanism, which enforces synchronisation between the filtering farms. A similar property was observed in the first cycles of the load balanced runs where the synchronization was enforced by the initialization step of the dynamic workload scheduling algorithm. Although in case of the static

allocation policy the synchronization between computing farms is much stronger (which results in equal event building efficiency), the farms still almost never reach the expected 2000 events in the same load-scheduling cycle. When it comes to the experiments conducted with the dynamic allocation policies, the differences between the outcomes obtained for the standard dynamic allocation algorithm and the alternative algorithms are negligible. This leads as, in turn, to the conclusion that the fluctuations in the event building efficiency per load-scheduling cycle are not related to the event-fragment allocation method.

Table 6-2 Analysis of event building efficiency per load-scheduling cycle for the static allocation mechanism

	DAQ Slice 0	DAQ Slice 1	DAQ Slice 2	DAQ Slice 3
avg.	1957.35	1957.37	1957.37	1957.37
stdev (1)	73	75	74	74
stdev (2)	85	86	86	85
trg.	2730	3007	2986	2966

Table 6-3 Analysis of event building efficiency per load-scheduling cycle for the standard dynamic allocation method

	DAQ Slice 0	DAQ Slice 1	DAQ Slice 2	DAQ Slice 3
avg.	1584	1866	1890	1995
stdev (1)	158	142	129	137
stdev (2)	445	195	169	179
trg.	42	3264	4223	4029

Table 6-4 Analysis of event building efficiency per load-scheduling cycle for the alternative dynamic allocation algorithm 1

	DAQ Slice 0	DAQ Slice 1	DAQ Slice 2	DAQ Slice 3
avg.	1590	1876	1891	1882
stdev (1)	153	133	133	138
stdev (2)	437	182	172	181
trg.	63	5685	7267	181

Table 6-5 Analysis of event building efficiency per load-scheduling cycle for the alternative dynamic allocation algorithm 2

	DAQ Slice 0	DAQ Slice 1	DAQ Slice 2	DAQ Slice 3
avg.	1583	1875	1888	1871
stdev (1)	154	136	138	147
stdev (2)	444	185	177	195
trg.	32	2781	3465	2907

6.5 Fault tolerance

The response of the system to a fault occurrence in a particular filtering farm has been studied first in the CMS DAQ test environment and then in the CMS DAQ production system. The goal of those experiments was to verify if the fault tolerance of the system has been enhanced through the dynamic load scheduling. First, we conducted our research on a 4 DAQ Slice setup with 1 EVM, 3 RUs and 4 BU-FUs per DAQ Slice (as previously described in [11] and [4]). The system has been tuned so the throughput limitation would come from event filter farm and the maximum possible data taking rate would be 50 kHz (12.5 kHz per DAQ Slice, which is the nominal speed for the production system). The initial data acquisition rate was set to maximum (50 kHz). As shown in the Figure 6-24, after a certain period of time BU-FU nodes were killed one after another. It can be easily noticed that the capacity loss in the standard system was significantly greater. Moreover, loss of the entire processing power in one DAQ Slice stopped the data acquisition process. On the other hand, in case of the system running with dynamic load scheduling algorithm, data were distributed proportionally to the efficiency of the computing farms, and as a result the erroneous DAQ Slice has been excluded from the data taking run. This experiment was afterwards repeated, except that the faults were introduced to more than one filtering farm, and gave also positive outcomes.

In case of the production system a failure of a BU-FU node does not result in such a drastic capacity loss of the respective computing farm (a farm contains more than 80 BU-FU nodes). Nevertheless, even in this case the dynamic load scheduling algorithm will save some

computing power. Moreover, it is possible that a whole rack of BU-FU nodes fails due to a network problem and this case is fully comparable to the one described above.

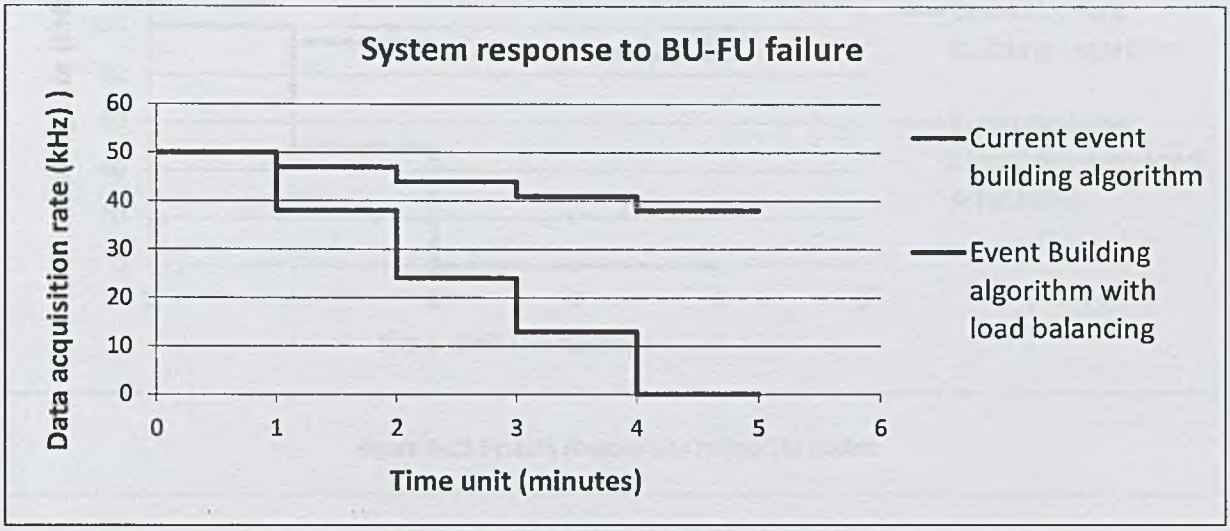


Figure 6-24 System response to failing BU-FU nodes

Subsequently, we continued our studies in the CMS DAQ production system (as previously described in [74]). A setup of 8 filtering farms has been used. Each of the computing farms participating in those tests was having 1 EVM, 63 RUs, 82 BU-FUs and 2 SMs. Both for the static and dynamic scheduling, the system was tuned so that the throughput limitation would come from event filter farms and the maximum possible data taking rate would be 100 kHz (12.5 kHz per farm). The initial data acquisition rate was set to maximum (100 kHz). In the first experiment shown in Figure 6-25, after a certain period of time SM nodes were powered off one after another. It can be noticed that the standard system lost 50% of its original capacity after switching off the first SM. Then, after the second one was turned off the whole data acquisition process was stopped. On the other hand, the system running with dynamic workload scheduling algorithm lost, as expected, only 6.25% of its capacity per SM node. In the second experiment shown in Figure 6-26, a RU node (which is a single point of failure for a computing farm) was powered off. In case of the standard system, the experiment resulted in an immediate termination of data taking. The system employing dynamic scheduling however lost only 12.5% of its capacity (which corresponds to one computing farm).

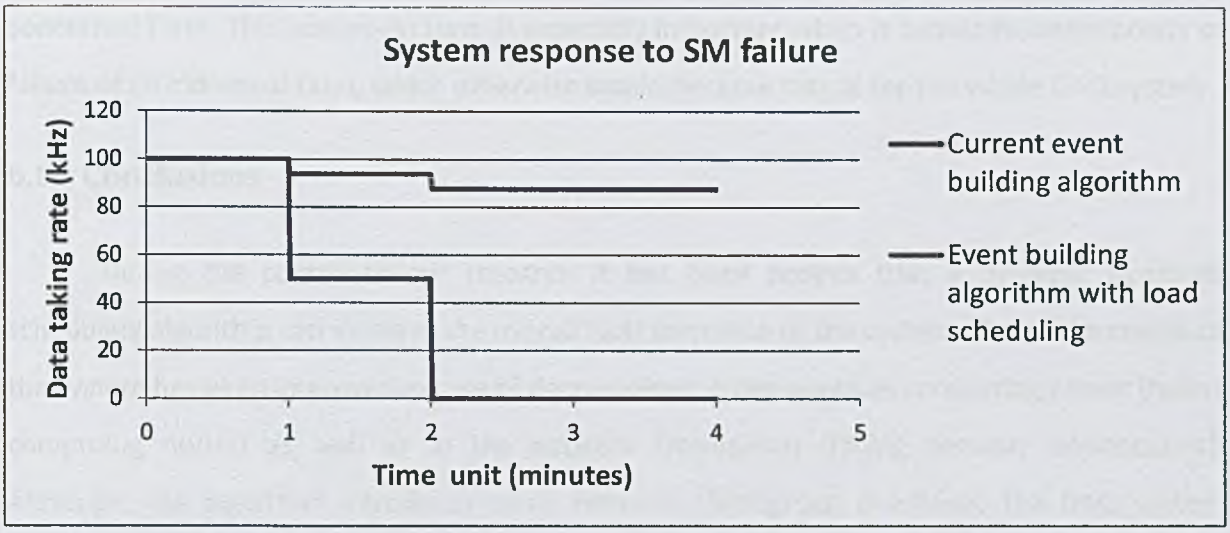


Figure 6-25 System response to failing SM nodes

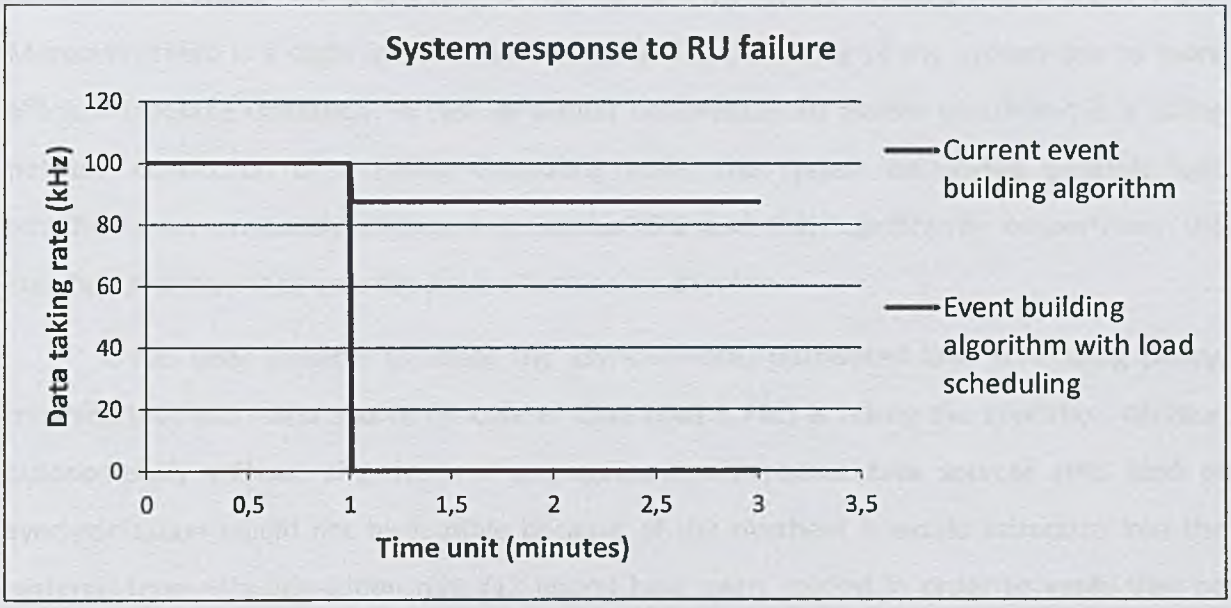


Figure 6-26 System response to failing RU node

The above discussed experiments were repeated with faults introduced into more than one filtering farm, and also gave positive outcomes and likewise confirmed the robustness of the dynamic load scheduling. During the course of our studies, it has been proven that the performance of the system has been improved in case of degradation of one (or more) of the computing farms participating in a data taking run. It has been also possible to decouple the farms from each other and as a result to limit the effects of some fault occurrences just to the

concerned farm. This feature, in turn, is especially important when it comes to single points of failure of an individual farm, which otherwise would became critical for the whole DAQ system.

6.6 Conclusions

During the course of our research it has been proven that a dynamic workload scheduling algorithm can increase the overall fault tolerance of the system. The performance of the system has been improved in case of degradations in the available computing power (failing computing nodes) as well as in the network throughput (failing network connections). Although, the algorithm introduces some network throughput overhead, the DAQ system running with dynamic load scheduling can easily sustain the requirements of CMS experiment, namely the nominal data taking rate of 100 kHz and the expected event size of about 1 MB. Moreover, there is a slight increase (about 3.5%) in the capacity of the system due to more efficient resource utilization. In case of a fault occurrence, no matter whether it is a failing network connection or a failing computing node, the system employing dynamic load scheduling (as previously discussed in section 6.1 and 6.4) significantly outperforms the standard system, which uses the static allocation mechanism.

It has been possible to adopt the asynchronous, distributed load scheduling policy, meaning that each data source (in case of CMS DAQ a FRL) is taking the allocation decision autonomously without any need to communicate with other data sources (this kind of synchronization would not be feasible because of the overheat it would introduce into the system). Long data acquisition runs (12 hours) have been studied in order to verify that no event mismatch errors occur (this type of error could be caused by a run condition). We obtained positive results concerning the correctness of the event allocation process. Due to the employed scheduling strategy, it has been possible to avoid introducing an additional single point of failure into the system, which was also crucial. Since the proposed method relays on the fact that each data source is producing the event-fragments in the same sequence, before the algorithm can become a part of the production system it has to support the *'out of sync'* use case. In the production system it can happen that the data sources run out of sync for example because one of them skipped a single event fragment. The system has to be able to

recover automatically from this type of error. This actually means that the algorithm has to monitor that the event-fragments are provided in the right sequence and in case if a fragment is missing the allocation decision has to be made for the not-existing fragment in order to set the allocation procedure in the right state. It can also happen that the synchronisation is enforced by an operator, which means that the event-fragments will be indexed again from 0. In this case, a data source may skip more than one event-fragment before the zero-event-fragment arrives (depends on how many event-fragments were already in the buffers of a data source before the synchronization has been enforced). In order to avoid event mismatch error, when the zero-event-fragment arrives, each data source has to be reset to the initial conditions that are used at the beginning of each data taking run (each filtering farm is assigned with *n* events).

It has been demonstrated that the proposed method scales well. It was used in the small CMS DAQ test setup, as well as in the big CMS DAQ production system (about 1500 computing nodes) and in both cases all our experiments were likewise successful. The proposed load index and load-data communication pattern are robust, accurate and do not introduce additional overhead into the system. An obvious weakness of the proposed method is the EVM-EVM communication procedure (one directional ring topology, described previously in subsection 5.3.4). Due to the behaviour described in section 6.3 (filtering farms are never reaching the under-loaded state at once), the communication will be always sequential. However, if the algorithm would become part of the production system most likely a dedicated network for the EVM-EVM communication would be added (so far we had to use the control network), which would give us much more flexibility in developing a robust and scalable communication solution (e.g. multicasting could be utilized). Also, the system is not yet fully tolerant to failing EVM nodes and there are plans to employ the Keep-Alive mechanism of the TCP protocol [75] in order to address this shortcoming.

Another weakness of the studied workload scheduling algorithm is the computational complexity of the allocation procedure (described in 5.4), which is the reason for the network throughput overhead that has been discussed in section 6.1. This, however, could be easily improved by using a circular doubly linked list similarly as in the alternative algorithm 2

(described in 6.4.2). Unfortunately, we had not enough time to conduct respective experiments on the production system. The time when the studies can be carried out on the CMS DAQ production system is limited only to the technical stops of LHC.

The work discussed in this thesis was conducted as a prove of concept in order to demonstrate that the CMS online DAQ system running with dynamic load scheduling can sustain the production workload of CMS experiment, is more robust and more reliable in case of software and hardware failures. Whether the dynamic workload scheduling algorithm will be integrated into the CMS online DAQ production system depends on the changes that will be introduced into the system during the upcoming upgrade (mainly, it depends on how will the network evolve). However, some concepts contained in this thesis have been already implemented at a different level and included in the current CMS Run Control and Monitoring system.

6.6.1 Future work

In the future we intend to replace the one-directional ring topology with a more efficient solution, which would allow us to parallelize the communication process between EVMs and thus speed it up. We are planning to evaluate multicasting solutions. The switch, which will be used to implement the dedicated EVM – EVM network, will presumably support multicasting. If we decide to take advantage of this feature however, we will be forced to use UDP protocol. This, in turn, would mean that we could not benefit from the built-in TCP Keep-Alive. In order to address the failing EVM use case we would need to provide our own implementation of this mechanism. On the other hand, the EVM – EVM communication could be implement using a multicast tree. In this case we could fully benefit from all the TCP features including reliable in-order packet delivery and Keep-Alive.

Further research will be conducted on the event-fragment allocation procedure. We are planning to reduce its complexity by employing a circular doubly linked list similarly as proposed in the alternative algorithm 2 (more details can be found in subsection 6.4.2). Also, we are planning to study the possibility of increasing the synchronization between filtering farms by adding up all the sets of counters that have been already received from computing

farms and are not currently in use (more details can be found in section 5.4). This topic has to be carefully investigated since there is a possibility of introducing a run condition into the system.

7 Bibliography

[1] William R Leo, *Techniques for nuclear and particle physics experiments.*: Springer, 1994.

[2] The CMS collaboration et al, "The CMS experiment at the CERN LHC," *JINST*, vol. 3, pp. 1-5, 261-282, August 2008.

[3] Simon S. Young, *Computerized Data Acquisition and Analysis for the Life Sciences.*: Cambridge University Press, 2001.

[4] Michał Kamil Simon, "Fault tolerant data acquisition through dynamic load balancing," in *IPDPSW*, Anchorage, 2011.

[5] Charles Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, pp. 406-424, October 1952.

[6] Nanette J. Boden et al., "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29-36, Februar 1995.

[7] Frederica Darema, "The SPMD Model: Past, Present and Future," in *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, London, 2001, p. 1.

[8] Brian Randell, *Reliability Issues in Computing System Design.*: Ass. Computing Machinery, 1978.

[9] Peter J. Denning, "Fault Tolerant Operating Systems," *CSUR*, vol. 8, no. 4, December 1976.

[10] Behrooz A. Shiraz, Ali R. Hurson, and Krishna M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.*: Wiley-IEEE Computer Society Press, 1995.

[11] Michał Kamil Simon, Hannes Sakulin, and Stanisław Kozielski, "Studies on load metric and communication for a load balancing algorithm in a distributed data acquisition system," in *Journal of Physics: Conference Series*, Taipei, 2010.

[12] Kristian Paul Bubendorfer, *Resource Based Policies for Load Distribution.* Wellington: Victoria University of Wellington, 1996.

[13] A. Osman and H. Ammar, "Dynamic Load Balancing Strategies for Parallel Computers," *Sci*

Ann Comput Sci, vol. 11, pp. 110-120, 2002.

[14] P. Chretienne, "Task Scheduling over Distributed Memory Machines," in *Proc. of the Inter. Workshop on Parallel and Distributed Algorithms*, North Holland, 1989.

[15] Christos H. Papadimitriou and Mihalis Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM J. Comput*, vol. 19, no. 2, pp. 322-328, 1990.

[16] V. Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," in *The MIT Press*, 1989.

[17] L. Borzemski, "Load balancing in parallel and distributed processing of tree-based multiple-task jobs," in *Euromicro Workshop on Parallel and Distributed Processing*, 1995, pp. 98-105.

[18] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton, "Preemptable remote execution facilities for the V-system," *ACM SIGOPS Operating Systems Review*, vol. 19, no. 5, December 1985.

[19] L. Borzemski, A. Zatwarnicka, and K. Zatwarnicki, "Method and algorithms of broker-based HTTP request global distribution," *Theoretical and Applied Informatics* , vol. 20, no. 1, pp. 15-27, 2008.

[20] Paul Werstein, Hailing Situ Situ, Zhiyi Huang, and Zhiyi Huang, "Load Balancing in a Cluster Computer," in *International Conference Parallel and Distributed Computing Applications and Technologies*, Taipei, 2006, pp. 569-577.

[21] C Fonlupt, P Marquet, and J Dekeyser, "Data-Parallel Load Balancing Strategies," *Parallel Computing*, vol. 24, pp. 1665-1684, 1996.

[22] Castelo K R L J Regina and Moreno E D Ordonez, "Load Indices - Past, Present and Future," in *International Conference Hybrid Information Technology*, Cheju Island, 2006, pp. 206-214.

[23] L. Borzemski and K. Zatwarnicki, "A fuzzy adaptive request distribution algorithm for cluster-based Web systems," in *Proceedings. Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2003, pp. 119-126.

[24] M. Andreolini, C. Canali, and R. Lancellotti, "Impact of request dispatching granularity in

geographically distributed Web systems," in *IEEE International Symposium on Network Computing and Applications*, 2007, pp. 45-52.

[25] Li YunFeng, Zhu Qingsheng, and Cao YuKun, "A request dispatching policy for Web server cluster," in *Proceedings. The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*, 2005, pp. 391-394.

[26] Jingxi Jia, Bharadwaj Veeravalli, and Jon Weissman, "Scheduling Multisource Divisible Loads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 99, pp. 520-531, 2009.

[27] P. Byrnes and L. A. Miller, "Divisible Load Scheduling in Distributed Computing Environments: Complexity and Algorithms," Univ. of Minnesota, Technical Report MN ISYE-TR-06-006, 2006.

[28] Dantong Yu and Thomas Robertazzi, "Multi-Source Grid Scheduling for Divisible Loads," *Proc. Conf. Information Sciences and Systems*, pp. 188-191, March 2006.

[29] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G. Robertazzi, "Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems," *Cluster Computing*, vol. 6, no. 1, pp. 7-17, January 2003.

[30] Charu C. Aggarwal, *Data streams: models and algorithms*. IBM T. J. Watson Research Center, Yorktown Heights, NY: Kluwer Academic Publishers, 2006.

[31] James A. Broberg, Zhen Liu, Cathy H. Xia, and Li Zhang, "A multicommodity flow model for distributed stream processing," *SIGMETRICS Proceedings of the joint international conference on Measurement and modeling of computer systems*, vol. 34, no. 1, June 2006.

[32] Marcin Gorawski and Pawel Marks, "Fault-Tolerant Distributed Stream Processing System," in *17th International Workshop on Database and Expert Systems Applications*, 2006, pp. 395 - 399.

[33] Marcin Gorawski and Pawel Marks, "Towards Reliability and Fault-Tolerance of Distributed Stream Processing System," in *2nd International Conference on Dependability of Computer Systems*, 2007, pp. 246 - 253.

[34] Robert Wrembel and Christian Koncilla, *Data Warehouses and OLAP: Concepts,*

Architectures and Solutions.: IGI Global, 2007.

[35] Marcin Gorawski and Robert Chechelski, "Parallel telemetric data warehouse balancing algorithm," in *International Conference onIntelligent Systems Design and Applications*, 2005, pp. 387-392.

[36] Xuan Lin, Ying Lu, J. Deogun, and S. Goddard, "Real-Time Divisible Load Scheduling for Cluster Computing," *IEEE Real Time and Embedded Technology and Applications Symposium*, pp. 303-314, April 2007.

[37] V. Thome, D. Vianna, R. Costa, A. Plastino, and O.T. Filho da Silveira, "Exploring load balancing in a scientific SPMD parallel application," in *International Conference on Parallel Processing Workshops, 2002. Proceedings.*, Vancouver, 2002, pp. 419 - 426.

[38] A. Osman and H. Ammar, "Designing a scalable dynamic load-balancing algorithm for pipelined single program multiple data applications on a non-dedicated heterogeneous network of workstations," in *Doctoral Dissertation. West Virginia University Morgantown*, Morgantown, 2003.

[39] Mitch Cherniack et al., "Scalable distributed stream processing," in *CIDR 2003*, Asilomar, 2003.

[40] F. Alessio et al., "The LHCb Readout System and Real-Time Event Management," *IEEE Transactions on Nuclear Science*, vol. 57, no. 2, pp. 663-668, April 2010.

[41] W. Vandelli et al., "The ATLAS Event Builder," *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3556 - 3562, December 2008.

[42] R. Angstadt et al., "The DZERO level 3 data acquisition system," *IEEE Transactions on Nuclear Science*, vol. 51, no. 3, pp. 445 - 450, June 2004.

[43] R. Carlin, W. H. Smith, K. Tokushuku, and L. W. Wi, "The trigger of zeus, a flexible system for a high bunch crossing rate collider," in *Nucl. Instrum. Methods A 379*, 1996, p. 542.

[44] K. Anikeev et al., "Event builder and level 3 trigger at the CDF experiment," *Computer Physics Communications*, vol. 140, no. 1-2, pp. 110-116, 2001.

[45] D. Sevilla, J. M. Garcia, and A. Gomez, "Aspect-Oriented Programing Techniques to support

Distribution, Fault Tolerance, and Load Balancing in the CORBA-LC Component Model," in *Sixth IEEE International Symposium on Network Computing and Applications, 2007. NCA 2007*, Cambridge, 2007, pp. 195-204.

[46] P. M. Melliar-Smith and Louise E. Moser, "O-Ring: A Fault Tolerance and Load Balancing Architecture for Peer-to-Peer Systems," in *International Conference of the Chilean Computer Science Society (SCCC), 2009*, Santiago, 2009, pp. 25-33.

[47] M. Flatebo, A. K. Datta, and B. Bourgon, "Self-Stabilizing Load Balancing Algorithms," in *IEEE 13th Annual International Phoenix Conference on Computers and Communications*, Phoenix, 1994, pp. 303-308.

[48] Edsger W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, pp. 643-644, November 1974.

[49] M. Votava, "BTeV Trigger/DAQ Innovations," *IEEE Transactions on Nuclear Science*, no. 4, pp. 2124-2130, August 2006.

[50] J. C. Oh, M. S. Tamhankar, and D. Mosse, "Design of Very Lightweight Agents for reactive embedded systems," in *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2003. Proceedings.*, Huntsville, 2003, pp. 149-158.

[51] K. Whisnant, S. Bagchi, B. Srinivasan, Z. Kalbarczyk, and R. Iyer, "Incorporating reconfigurability, error detection and recovery into the chameleon armor architecture," UIUC, UILU-ENG-98-2227 1998.

[52] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, no. 1, pp. 14-23, March 1986.

[53] G. Bauer, B. Beccati,..., M Simon et al., "Journal of Physics: Conference Series," *Journal of Physics: Conference Series*, vol. 331, no. 2, pp. 1-7, 2011.

[54] G Bauer, B Beccati,..., M Simon et al., "The LHC Compact Muon Solenoid experiment Detector Control System ," *Journal of Physics: Conference Series*, vol. 331, no. 2, pp. 1-7, 2011.

[55] G. Baue et al., "Dynamic configuration of the CMS Data Acquisition cluster," *Journal of*

Physics: Conference Series, vol. 219, no. 2, pp. 1-9, 2010.

[56] Alexander Oh, "The CMS DAQ and run control system," *Journal of Physics: Conference Series*, vol. 110, no. 092020, pp. 1-4, 2008.

[57] G. Bauer,..., H. Sakulin, M. Simon et al., "First operational experience with the CMS run control system," in *Real Time Conference (RT), 2010 17th IEEE-NPSS*, Lisbon, 2010, pp. 1-5.

[58] Gerry Bauer et al., "The Terabit/s Super-Fragment Builder and Trigger Throttling System for the Compact Muon Solenoid Experiment at CERN," *IEEE Transactions on Nuclear Science*, vol. 55, no. 1, pp. 190-197, Februrar 2008.

[59] G. Bauer et al., "Effects of Adaptive Wormhole Routing in Event Builder Networks," in *Real-Time Conference, 2007 15th IEEE-NPSS*, Batavia, 2007, pp. 1-7.

[60] "IEEE Standard for Heterogeneous InterConnect (HIC) (Low-Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)," The Institute of Electrical and Electronics Engineers, Inc., New York, ISBN 1-55937-595-7, 1995.

[61] Pablo Molinero-Fernández and Nick McKeown, "The performance of circuit switching in the internet," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 3, p. 12, July 2002.

[62] V. Brigljevic et al., "The CMS Event Builder," in *CHEP03*, La Jolla, 2003.

[63] M. Karol, M. Hluchyj, and S. Morgan, "Input Versus Output Queueing on a Space-Division Packet Switch," *IEEE Transactions on Communications*, vol. 35, no. 12, pp. 1347-1356, December 1987.

[64] R. Mandeville, "Benchmarking Terminology for LAN Switching Devices," *RFC 2285*, February 1998.

[65] T. Czachórski and F. Pekergin, "Diffusion Approximation as a modelling tool in congestion control and performance evaluation, tutorial," in *Proc. of HET-NETs '04, Performance Modelling and Evaluation of Heterogenous Networks*, 2004.

[66] T. Czachórski and F. Pekergin, "Modelling New Congestion Control Mechanisms TCP/IP Internet Protocols," in *Proc. of International Conference on Control AUTOMATICS 2001*,

Odessa, 2001.

[67] Larry L. Peterson and Bruce S. Davie, *Computer Networks: A System Approach*, 5th ed.: Morgan Kaufmann, 2000.

[68] B. Jouaber, T. Atmaca, M. Pastuszka, and T. Czachórski, "Modeling the sliding window mechanism," in *International Conference on Communications, ICC 98*, 1998, pp. 1749-1753.

[69] Douglas E. Comer, *Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architecture.*: Prentice Hall, 1995.

[70] R. Mommsen and S. Murray, "RU Builder User Manual," CERN, Geneva, EVB_D_18306, 2008.

[71] Kees Verstoep, Koen Langendoen, and Henri Bal , "Efficient Reliable Multicast on Myrinet," in *ICPP '96 Proceedings*, 1996, pp. 156-165.

[72] H. C. van der Bij, "S-LINK, a data link interface specification for the LHC era," in *Nuclear Science Symposium*, Anaheim, 1996, pp. 465 - 469.

[73] Gerry Bauer, Barbara Beccati,..., M Simon et al., "Studies of future readout links for the CMS experiment," *Journal of Physics: Conference Series*, vol. 331, no. 2, pp. 1-7, 2011.

[74] Michal Simon, Hannes Sakulin, and Stanislaw Kozielski, "Experimental Results of Dynamic Load Scheduling in the CMS Data Acquisition System," *Communications in Computer and Information Science*, in press.

[75] R. Barden, *Requirements for Internet Hosts - Communication Layers*, R. Braden, Ed. United States: RFC Editor, 1989.

[76] Donald Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 3rd ed.: Addison-Wesley, 1997.