ATLAS TDAQ DataCollection Software

Christian Haeberli, Andre dos Anjos, Hans Peter Beck, Andre Bogaerts, David Botterill, Szymon Gadomski, Piotr Golonka, Reiner Hauser, Micheal J. LeVine, Remigius Mommsen, Valeria Perez Reale, Stefan Nicolae Stancu, Jim Schlereth, Per Werner, Fred Wickens, and Haimo Zobernig

***Abstract*—The *DataCollection* (DC) is a subsystem of the *ATLAS* Trigger and DAQ system. It is responsible for the movement of event data from the *ReadOut* subsystem to the *Second Level Trigger* and to the *Event Filter*. This functionality is distributed on several software applications running on *Linux* PCs interconnected with *Gigabit Ethernet*. For the design and implementation of these applications a common approach has been adopted. This approach leads to the design and implementation of a common *DC* software framework providing a suite of common services.**

***Index Terms*—ATLAS, Dataflow, Event Filter, Large Hardron Collider (LHC), object oriented application framework, Second Level Trigger, TDAQ.**

I. Introduction

THE Large Hardron Collider (LHC), which is currently under construction at CERN and will be operational in 2007, will collide 7 TeV protons. ATLAS is a multipurpose experiment dedicated to exploit the discovery potential of the LHC.

Bunches of 10^{11} protons will collide at periods of 25 ns at the interaction point. This will result in $\sim$25 proton-proton collisions per bunch crossing and a total interaction rate of $\sim$1 GHz. An online selection needs to be applied in order to select the 100 events per second with the highest discovery potential and to reduce the event rate by a factor of 10^7. This reduction allows to store only a manageable and analyzable amount of data.

ATLAS wants to achieve the online event selection with a three level trigger system. The first level trigger is custom build hardware, whereas the High Level Triggers (HLT), which comprises the second level trigger (LVL2) and the Event Filter, are software algorithms running on commodity PCs.

The Dataflow system is responsible for the readout of detector data, the serving of data to the HLT system. The Dataflow system also moves data accepted by the HLT to mass storage. The ReadOut subsystem (ROS) groups multiple ReadOut links from the detector, buffers the incoming data and makes it available for the DataCollection (DC) subsystem which transports the data to the LVL2 and the Event Filter. A more detailed description of the Dataflow system is given in [2].

The DC subsystem transports regions-of-interest (RoI) data, as identified by the first level trigger system (LVL1), from the ROS to the LVL2 and full events to the Event Filter. In addition, it transports event data from the Event Filter to mass storage.

The basic requirements for the DC subsystem are to transport RoIs of an average size of 16 kB at the LVL1 accept rate of 100 kHz to the LVL2 processors and to build full sized events of 1.2 MB at a LVL2 accept rate of $\sim$3 kHz for the LHC startup luminosity. Therefore, the bandwidth into LVL2 is 1.6 GB/s and the bandwidth into the Event Filter is 3.6 GB/s.

The functionality of DC is distributed to six software applications running on Linux PCs interconnected with gigabit Ethernet. A common approach to design and implementation was chosen, which leads to the design and implementation of a common DC software framework, providing a suite of common services (e.g., Message Passing and Application control). The design and the implementation of the DC framework are based on C++ and the standard template library (STL). The DC applications are multithreaded and built on top of the framework.

II. Distribution of Functionality

The DC functionality is distributed to six software applications: the LVL2 Supervisor (L2SV), the LVL2 processing unit (L2PU), the pseudo ROS (pROS), the Dataflow manager (DFM), the Subfarm-input (SFI), and the Subfarm-output (SFO). An overview of the DC applications and their interactions is shown in Fig. 1.

A. LVL2 Supervisor

The L2SV receives a LVL1 decision containing geometry information of RoIs from the *region-of-interest builder*. It load balances a part of the LVL2 farm consisting of many L2PUs while assigning events to L2PUs chosen according to a load-balancing algorithm. After processing at the L2PU the L2SV receives the LVL2 *decisions*, which it forward to the DFM.

Manuscript received June 5, 2003; revised November 24, 2003. This work was done on behalf of the ATLAS TDAQ DataFlow community.

C. Haeberli, H. P. Beck, and V. Perez Reale are with the Laboratory for High Energy Physics, University of Bern, 3012 Bern, Switzerland (e-mail: christian.haeberli@cern.ch).

A. dos Anjos is with the Universidade Federal do Rio de Janeiro, COPPE/EE, 21945-970 Rio de Janeiro, Brazil.

A. Bogaerts, P. Golonka, S. N. Stancu, and P. Werner are with CERN, CH-1211 Geneva 23, Switzerland.

D. Botterill and F. Wickens are with the Rutherford Appleton Laboratory, Chilton, Didcot OX11 0QX, U.K.

S. Gadomski is with the Laboratory for High Energy Physics, University of Bern, 3012 Bern, Switzerland, and also with the Henryk Niewodniczanski Institute of Nuclear Physics, 31-342 Cracow, Poland.

R. Hauser is with the Michigan State University, Department of Physics and Astronomy, East Lansing, MI 48824, USA.

M. J. LeVine is with the Brookhaven National Laboratory (BNL), Upton, NY 11973 USA.

R. Mommsen is with the University of California, Irvine, CA 92697 USA.

J. Schlereth is with the Argonne National Laboratory, Argonne, IL 60439 USA.

H. Zobernig is with the Department of Physics, University of Wisconsin, Madison, WI 53706 USA.

Digital Object Identifier 10.1109/TNS.2004.828601

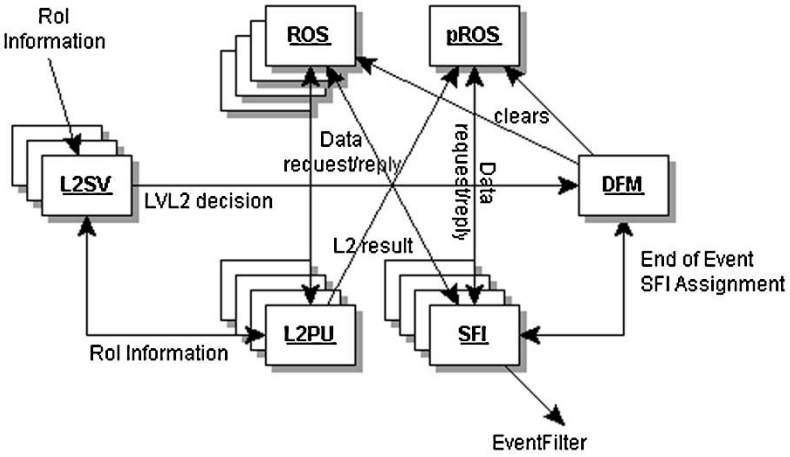

Fig. 1. Overview of the DC applications and the interactions inbetween them.

B. LVL2 Processing Unit

The L2PU receives a LVL1 *decision* from the L2SV. It requests the RoI data it needs for processing from many ROSs. After the LVL2 *decision* is taken it reports it to the *L2SV.* In addition, it sends a more detailed LVL2 *result* to the *pROS*.

The DC framework provides the basic services needed by the LVL2 algorithms. The decision taking LVL2 *algorithms* are beyond the scope of the DC software, but it builds the basis for them.

C. Pseudo ROS

The pROS is the interface between the LVL2 and the Event Filter. It receives detailed *LVL2 results* from the L2PUs and takes part in the event building as a common ROS. It ensures that the detailed *LVL2 result* becomes a part of the fully built event and therefore is accessible for the algorithms running in the Event Filter.

D. Data-Flow Manager

The DFM is responsible for the load balancing and the bookkeeping for the event building. It receives the *LVL2 decisions* from the L2SV and forward the *LVL2 rejects* to all instances of the ROS to ensure the deletion of these events. For every *LVL2 accept* the DFM assigns an SFI for event building.

E. Subfarm-Input

The SFI is fulfilling the major part of the event building functionality. It gets an event assigned by the DFM and requests the event fragments from all instances of the ROS. As soon as all fragments of a certain event arrived at the SFI an *end-of-event* message is sent to the DFM. The fully built event is kept and made accessible to the *Event Filter*.

F. Subfarm-Output

The SFO receives events accepted by the *Event Filter.* It buffers these in files held on a local disk and makes the files available to the *mass storage system*.

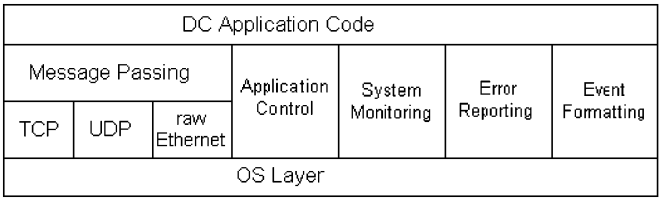

Fig. 2. Architectural view of the DC framework.

III. Framework Functionality

The framework provides a suite of common functionalities to the DC applications. These services include MessagePassing, Application Control, Error Reporting, Configuration Database, System Monitoring, OS Abstraction Layer, Event Formatting, and Time Stamping. A part of these framework packages (e.g., Application control) need to interface to the common TDAQ control software services, provided by the *OnlineSW suite* [3]. An architectural view of the DC framework is shown in Fig. 2.

A. Message Passing

The message passing layer is responsible for the transfer of event data and control messages between Dataflow components. The latter ensure the proper movement of the data. It imposes no structure on the data, which is to be exchanged, except a four-byte alignment of the byte-stream. It allows the transfer of data blocks of up to 64 kB size with a best-effort guarantee. For efficiency reasons this layer does no retransmission or acknowledgment of data. This choice has allowed the API to be implemented over a range of technologies without imposing an un-necessary overhead or the duplication of existing functionality, e.g., in the case of TCP/IP. The API supports the sending of both unicast and multicast messages. The latter may be emulated to the DC application code in case the underlying protocol does not support multicast, e.g., for TCP/IP.

The design of the message passing layer defines classes that allow the sending and receiving of messages. The *Node*, *Group*, and *Address* classes are used at configuration time to setup all the necessary internal connections. The *Port* class is the central interface for sending data. All user data has to be part of a *Buffer* object to enable it to be sent or received from a *Port*.

The *Buffer* interface allows for addition of user defined memory locations without copying. The Provider class is an internal interface from which different protocol and technology specific implementations inherit. Multiple Provider objects can be active at any given time allowing the concurrent use of different protocols. To date providers implementing TCP/IP, UDP/IP, and raw Ethernet exist. The design is open for more protocols to be added in the future.

As the message passing layer itself does not guarantee the reliable delivery of messages via the network (except if running a reliable underlying network protocol) the DC application code has to implement strategies to avoid message loss and to recover from packet loss.

We have mainly observed message loss in input buffers of switches and in the Linux kernel input buffer, despite the kernel buffer size was increased to 8 MB. A large fraction of this loss can be avoided by limiting the number of outstanding requests in a request-reply traffic pattern. This measure avoids any packet loss in the kernel input buffer because one can determine the maximum total size of buffered messages. In addition it decreases the probability for packet loss in the input buffers of the switches because one can determine the maximum number of messages travelling through the network at any time.

As limiting the number of outstanding requests can minimize the packet loss but not exclude it absolutely, some DC applications implement a re-ask mechanism for missing messages in request-reply traffic.

B. Application Control

The run control interface is responsible for translating state transition commands issued by a run controller application into commands internal to the DC application. The two applications communicate via a dedicated TCP/IP stream. Alternatively a keyboard interface has been defined, allowing for running DC applications in a local environment. The application control class is instantiated in the DC applications and provides virtual methods for all state transition to be implemented by the application developer.

C. Error Reporting

The error reporting package allows the logging of error messages either to standard output and standard error or to *message reporting service* (MRS) provided by the *OnlineSW*. Each DC software package can define its own set of error messages and error codes. Error logging can be enabled/disabled on a per package basis. Furthermore debug messages and error messages are treated logically differently, so the debug message could go to standard output while all normal application logs go to MRS. The user only interfaces via a set of macros to the Error-Reporting system allowing optimization of the applications at compile time.

D. Configuration Database

All DC applications obtain their configuration parameters from the configuration database provided by the *Online SW*. The access layer to the configuration database allows the underlying implementation to change without implying changes to the application. The application's view of the database is hidden by configuration objects, which access the database, providing a more convenient way to access configuration information. The configuration objects themselves are created by a code generator, which parses the configuration database schema file.

E. System Monitoring

This part of the framework allows every DC application to make arbitrary information available to some outside client. In practice this is used to publish statistics like counters or rates. The package makes this information available in various different ways, including the *OnlineSW*.

F. OS Abstraction Layer

The OS abstraction layer consists of packages hiding all OS specific interfaces. For example, the *threads* package hides the details of the underlying *POSIX* thread interface, which could in the future be replaced by another implementation.

G. Event Format

This package allows to format event data according to [4], to navigate through this structure and to store event data in dedicated storage types. One important example is the I/O vector storage type, which holds a vector of pointers to event fragments. This storage type is used to avoid copying if data fragments of a specific event are distributed to many places in memory and thus allows for scatter-gather send operations avoiding memory copies when shipping out data.

H. Time Stamping

The Time stamping library provides hooks to resolve the timing behavior of the software. Two dedicated implementations exist. The first one is based on *NetLogger* [5] and used to understand the timing behavior of a distributed system; the second provides fine-grained time resolution to understand the timing behavior inside a single application. The output of both implementations satisfies the *NetLogger* format and therefore the *NetLogger visualization tool* [5] can be used to analyze the data generated by both of them.

IV. Performance Requirements for DC Applications

There are two kinds of I/O performance requirements for the DC subsystem: message rates and bandwidth. Table I summarizes rate and bandwidth requirements for the different transactions between the DataCollection applications and the ROS. The SFI has to fulfill the most demanding I/O requirements of all the applications.

V. DC Framework Performance

The performance of the DC software was validated in an extensive measurements program. Its detailed results are presented in [6].

TABLE I
MESSAGE RATES AND BANDWIDTHS OF CONTROL AND DATA MESSAGES BETWEEN THE DC COMPONENTS AN THE ROS

	Message Type		Sender Rate	Sender Bandwidth	Receiver Rate	Receiver Bandwidth
L2SV → L2PU	RoI information		7.5 kHz	5 MB/s	0.5 kHz	0.3 MB/s
L2PU → ROS	Data requests	b)	6 kHz	0.6 MB/s	16 kHz	1.6 MB/s
		s)	11 kHz	1 MB/s	6 kHz	0.6 MB/s
ROS → L2PU	Event data	b)	16 kHz	32 MB/s	6 kHz	11 MB/s
		s)	6 kHz	6 MB/s	11 kHz	11 MB/s
L2PU → L2SV	LVL2 decisions		0.5 kHz	50 kB/s	7.5 kHz	750 kB/s
L2SV → DFM	LVL2 decision[g)]		75 Hz	40 kB/s	750 Hz	400 kB/s
DFM → SFI	Assign event		3 kHz	0.3 MB/s	30 Hz	3 kB/s
SFI → ROS	Data request	b)	4 kHz	0.4 MB/s	3 kHz	0.3 MB/s
		s)	48 kHz	4.8 MB/s	3 kHz	0.3 MB/s
ROS → SFI	Event data	b)	3 kHz	36 MB/s	4 kHz	48 MB/s
		s)	3 kHz	3 MB/s	48 kHz	48 MB/s
SFI → DFM	Finished Event		30 Hz	3 kB/s	3 kHz	0.3 MB/s
DFM → ROS	Clear buffers[g)]		250 Hz	0.4 MB/s	250 Hz	0.4 MB/s

b) bus-based ROS
s) switch-based ROS
g) message contents grouped for multiple event identifiers

A. Aim of the Performance Tests

A DC application will not use all framework services when running, but rather at earlier states, e.g., configuration. Another part of the framework functionality is used in the running state, but at a low rate only (e.g., *error reporting* and *system monitoring*). These parts of the framework are not critical for the performance of the running system. In a stably running setup error messages are occurring very rarely only (less than one per second). The monitoring service is running at a rate of $\sim$1 Hz in a separate thread. This thread is flushing counter values which are continuously updated by the performance critical working threads to the monitoring service of the Online Software. All measurements provided with the DC software were obtained with the error reporting and system monitoring facilities enabled. The measurements even rely on the system monitoring as the service provides the information which is needed to obtain the results of the measurements.

The framework functionalities, which are used at a high rate in the running state, are the *message passing*, the *event formatting* and the *OS services*. Therefore measuring the performance of a DC application implies also to validate the framework performance. To achieve this validation it is sufficient to demonstrate that the DC application with the most demanding I/O performance meets its requirements. Therefore a measurement using the SFI, which is the DC application with the most demanding I/O performance (see Table I), is the validation of the DC framework performance and is presented in this paper.

B. Setup of the Permance Tests

The tests were provided on the Dataflow performance testbed which is described in [2]. The SFI application was running on a dual Intel Xeon 2.4 GHz CPU PC with 1 GB RAM. The operating system was the CERN certified version of Redhat Linux 7.2.

The SFI was interconnected to 60 FPGA ROS emulators via two layers of Ethernet Switches: the first layer switch was a 31 ports Gigabit Ethernet BATM T6 which connected the SFIs to two concentrator switches. The concentrator switches were 32 ports Fast Ethernet BATM T5 with two Gigabit Ethernet uplink connections. These connected each 30 FPGA ROS emulators to the central switch using one of the uplinks. The measurement in Fig. 3. shows the input bandwidth of the SFI performing event building from 60 *FPGA ROS emulators* [7] simulating 1600 data sources.

The SFI was operating in pull mode. It was sending 1600 unicast requests in a round robin pattern to the FPGA ROS emulators. Raw Ethernet was applied as network protocol, because it is implemented in the FPGA ROS Emulators exclusively.

C. Results and Discussion

The plot in Fig. 3 shows that the SFI total message rate depends very little on the fragment size (131 kHz message rate for 160 Byte fragments versus 114 kHz for 1456 Byte fragments). The SFI application is running in the CPU limited domain and it is exploiting two third of the Gigabit Ethernet bandwidth. As the SFI performance is purely CPU limited, the bandwidth the SFI can drive is expected to scale linearly with the CPU speed of the SFI machine. Therefore it is expected that the application will be able to drive the full Gigabit Ethernet bandwidth on a 3.6-GHz CPU platform, which will become available well before the startup of LHC. In addition we expect performance gains from further optimizations of the DC code. The SFI is building events exclusively and is not processing any event data. Therefore it is not critical to use the full or a large fraction of its

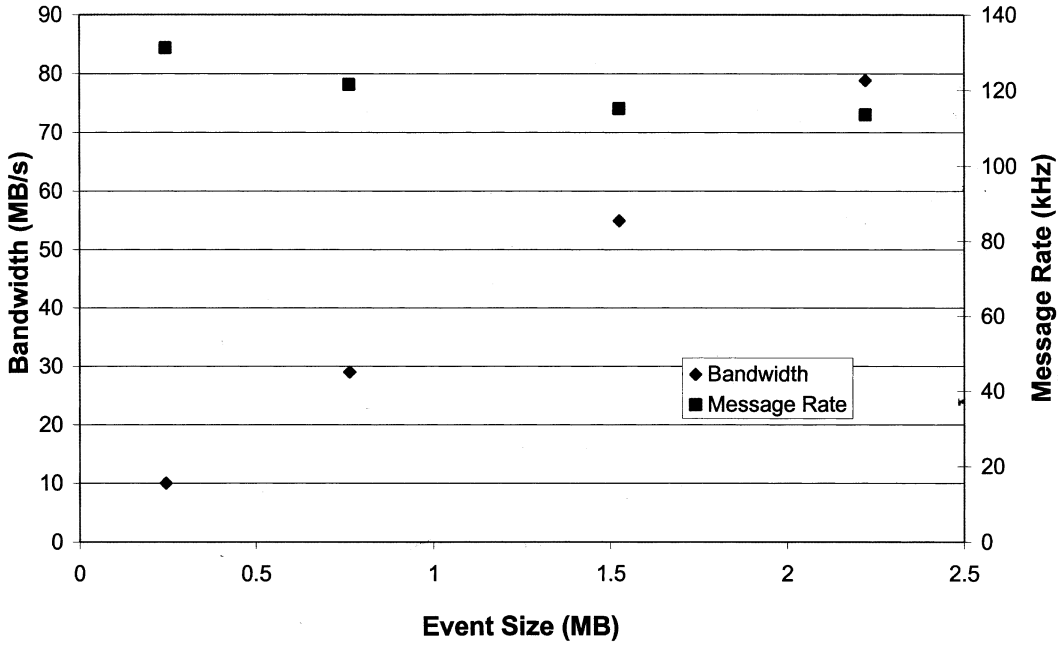

Fig. 3. The SFI bandwidth and message rate running against 60 FPGA ROS emulators acting as 1600 data sources. The fragment size per data source was varied from 160 to 1456 Bytes. Adding the header information of the DC raw Ethernet implementation this fragment size is a full Ethernet frame of 1500 Bytes. The variation of the fragment size means that the event size was varied from 0.2 to 2.2 MB.

CPU capacity for data movement. The SFI meets the message rate requirement of 96 kHz and the bandwidth requirement of 48 MB/s.

It is straightforward to translate the SFI performance results to other applications e.g., the L2PU. Given the fact that the SFI needs almost all its CPU resources to drive 80 MB/s input bandwidth, the L2PU will need 15% of its CPU resources for I/O, the remaining resources will be available for processing the decision taking algorithms for LVL2. This estimation is pessimistic, as in opposite to the SFI the L2PU does not need to do any event building operations. Measurements with the L2PU are indicating a worst case value of 10% [8].

All the other DC applications have much lower rate and bandwidth requirements than the SFI (see Table I). Therefore the performance of the DC framework suits the needs of these applications.

VI. Network Protocols

As for most of the DC performance measurements ROS emulators were used [7] and as those have only the raw socket protocol implemented, the DC group is mainly experienced with this protocol. We expect the UDP/IP protocol to behave similarly as raw Ethernet, as both are connectionless protocols. Measurements showed that the raw Ethernet protocol was faster for single frame messages than UDP/IP [9]. For multiframe messages the opposite applies.

The application of TCP/IP for RoI collection and event building is not appropriate due to the following reasons [10].

1) TCP/IP is a connection oriented protocol, every application has to hold a connection to each partner application it communicates to. For example, the SFI needed to hold up to 1600 connections. This may lead to a scalability problem.
2) The *sliding windows* mechanism is not appropriate for event building. If a package gets lost, no data can be exchanged between two communication partners. This means that no other event can be completely built before the TCP recovery mechanism took place and the lost package arrives at the SFI. This leads to a latency problem.
3) TCP/IP is a reliable protocol. In order to achieve the reliability the protocol issues acknowledgment messages to confirm the reception of a data packet. Depending on the round trip time of the network, TCP/IP can increase the message rate in the network up to 50% due to acknowledgment messages. This takes network and CPU resources.
4) Long-term tests, running event building with a connectionless protocol showed a rate of package loss of 10^{-9}. This minimal loss is caught by a recovery mechanism on the level of the SFI application. This means that there is no need for a reliable protocol and its potential drawbacks.

Therefore, TCP/IP will not be used for data traffic (event building and RoI collection). However, the protocol could be used for control messages like LVL2 decisions or end of event notifications.

VII. Conclusion

Thanks to the framework approach it was possible to develop the DC software within two years. In addition, this approach will ease future maintenance of the subsystem.

The performance of the applications built on top of the framework is promising and sufficient for ATLAS even with today's hardware. For example, a DC application running on a 2.4-GHz dual CPU machine, can handle a total message rate of 114 kHz in a request-reply traffic pattern (sending small request

messages, receiving event data). Therefore the choice of the framework approach, Linux, C++, STL, and multithreading was proved to be viable.

ACKNOWLEDGMENT

C. Haeberli would like to thank M. Zurek for the system administration of the testbed at CERN, which was the key facility to evaluate and improve the performance of the DC software.

REFERENCES

[1] The ATLAS TDAQ DataFlow community [Online]. Available: http://atlas.web.cern.ch/Atlas/GROUPS/DAQTRIG/DataFlow/DFlowAuthors.pdf

[2] H. P. Beck *et al.*, "The base line DataFlow system of the ATLAS trigger and DAQ," *IEEE Trans. Nucl. Sci.*, vol. 51, pp. 470–475, June 2004.

[3] I. Alexandrov *et al.*, "Online software for the ATLAS test beam data acquisition system," *IEEE Trans. Nucl. Sci.*, vol. 51, pp. 578–584, June 2004.

[4] C. Bee *et al.*, "The raw event format in the ATLAS Trigger and DAQ,", ATLAS Internal Note ATL-DAQ-98-129, Oct. 2002.

[5] B. Tierney and D. Gunter, "NetLogger: A toolkit for distributed system performance tuning and debugging,", LBNL Tech. Rep. LBNL-51 276, Dec. 2002.

[6] *ATLAS High Level Trigger, Data Acquisition and Controls*.

[7] M. LeVine *et al.*, "Network performance investigation for the ATLAS trigger/DAQ," in *Proc. 13th IEEE NPSS Real Time Conf.*, Montreal, QC, Canada, 2003.

[8] M. Abolins *et al.*, "The second level trigger of the ATLAS experiment at CERN's LHC," presented at the Proc. Nuclear Science Symp. 2003, Portland, OR, 2003.

[9] Linux network performance study for the ATLAS Data Flow System, P. Golonka. (2003). [Online]. Available: https://edms.cern.ch/document/368 844

[10] The Use of TCP/IP for real-time messages in ATLAS Trigger/DAQ, R. Hughes-Jones. (2003). [Online]. Available: https://edms.cern.ch/document/393 752