A C++ Shared-Memory IPC Framework for High-Throughput Data Acquisition Systems

Rolando Inglés, Mariusz Orlikowski, and Andrzej Napieralski

***Abstract*—High-Throughput Data Acquisition Systems are an essential part in many scientific experiments, and the processing of the large amount of data, represents a challenge in designing systems capable of managing such volume of data. Owing to the nature of this type of experiments, the processes responsible for gathering the data from devices that measure real-world phenomena, and those processes in charge of distributing the data to monitoring and/or controlling systems, shall communicate with accuracy and reliability. By running those processes concurrently in a multi-processor computer system, such requirements of accuracy and reliability can be achieved. In this paper, we present the design of a C++ framework, which implements a ring-buffer by using shared-memory as a fast mechanism of data communication among processes. Likewise, the framework controls the access to data in the shared ring-buffer by implementing inter-process synchronization objects in shared-memory. The effectiveness of the proposed solution has been evaluated by evaluating the latency time from when a new data is written into the shared ring-buffer and the longest instant when such a data is gathered. After the experimental test, the results show that it is possible to develop a C++ framework for helping programmers to create data acquisition system when a high-throughput data-stream is involved, getting suitable performance by using shared-memory.**

***Index Terms*—inter-process communication; ring-buffer; data acquisition system; high-throughput; condition variables; futex; atomic variables.**

I. Introduction

THE process of monitoring real-world phenomenas by means of specialized devices produce a large amount of data, such Data Acquisition (DAQ) process, as it is depicted in the Figure 1, it is defined as the process of mesuaring real-world phenomena and transform such measurements into a standardized signal format that can be processed by monitoring applications and control systems [1] [2].

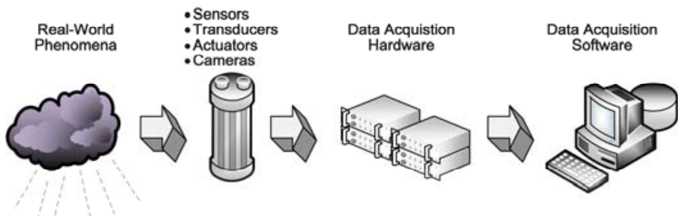


Fig. 1. Data acquisition process.

R. Inglés, M. Orlikowski and A. Napieralski are with the Department of Microelectronics and Computer Science, Lodz University of Technology, ul. Wolczanska 221/223, 90-924 Lodz, Poland (e-mails: {roling, mariuszo, napier}@dmcs.pl).

Such complex monitoring systems require to spread the adquired data toward different sub-systems quickly and with the appropiate of reliability and safety. As seen in Figure 2, the process of gathering the raw data from hardware devices and propagation of them toward the specialized sub-systems of control and archiving, is considered in the ITER project as Data Acquisition System.

The experimental assesment contained in this paper is focused on presenting an C++ Shared-Memory Ring-Buffer Framework. Such solution has been designed based on the ITER CODAC framework [3]; those specifications set forth the design of a generic C++ framework to help to programmers with the developing of diagnostics, monitoring and control applications for the TOKAMAK device [4] referenced as experimental fusion device from this point forward.

One particular use-case for monitoring the experimental fusion device implies the management of data in the form of images; these images are captured by high-speed cameras and acquired by using high-speed digital interfaces, e.g., Camera Link, PCIe or GbE [5]. Subsequently, such images are digitized through the use of frame grabber boards [6]. Since the images are used for monitoring and controling the experimental fusion device, the images must be transferred in the fastest manner posible toward the specialized sub-systems, particularly those sub-systems where the images are processed in the precise moment when they are grabbered, these sub-systems are represented in the Figure 2 as *Critical Systems*.

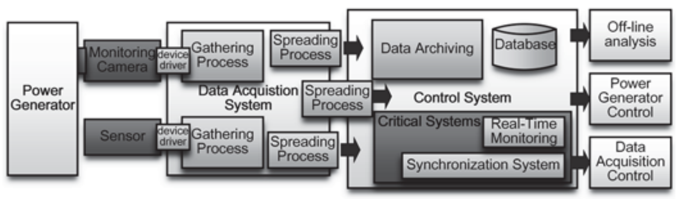


Fig. 2. Monitoring process data workflow

In order to achieve an efficient mechanism to transfer data between process responsible to gather data from the hardware devices and those processes responsible to spread the data toward the monitoring and controlling sub-systems, several methods has been proposed in the past [7] [8] [9]. In this paper, the shared-memory approach has been adopted because all the independent system processes involved in the DAQ and the Control System, require a robust inter-process communication (IPC) procedure among them.

The shared-memory approach is considered the fastest IPC mechanism, since only one copy of the data lies in the memory and it is available for all process once the shared-memory segment is mapped properly for each process [10]. Certainty, the primary envisioned use for shared-memory it was to provide high-speed inter-process communication among cooperating processes [11] [12] [13].

To achieve high-rate transfer in the dissemination of the data to the processing sub-systems, each process bound to each sub-system must to run concurrently and ideally running is its own CPU [14]. Notwithstanding, concurrent access to specific shared-memory segment it requires to use special synchronization techniques to control the access to the shared data and thus avoid the occurrence of race-conditions [15] [16].

II. The Data Acquisition System

A. *The Raw Data Source*

One of the sytems for monitoring the experimental fusion device is by means of high-speed cameras, specificlly the *EoSens® 3CL Full CL MC3010* model [3]. The Figure 3 depictes the information provided by the manufacturer related with the maximun frame rate for a given resolution [17].

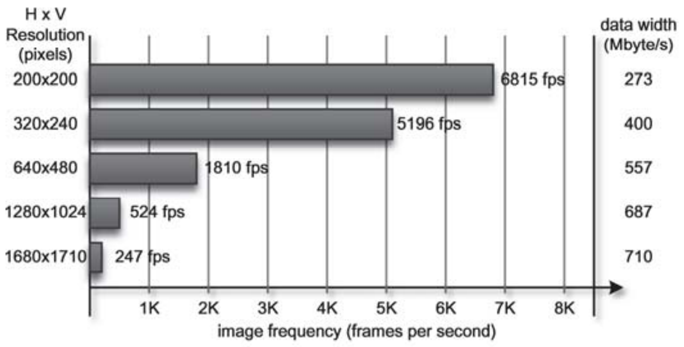


Fig. 3. Frameset factory profiles of EoSens® 3CL Full CL MC3010 high-speed camera [17]

Taking into account the above-mentioned information, the cameras are capable of producing *710 Mbytes* of raw data when are configured in the highest resolution *1680x1710 pixels*. And besides, according to thechnical documentation the cameras are able to produce up to *6815 frames per second* when the reduced resolution is set [17].

B. *The Control System*

The Control System executes the process for controlling and monitoring the operation of the power-generator [3]. Such controlling tasks are dependent on the physical time when results are produced, for this reason the DAQ and Control System operate in a real-time environment [18]. To conduct the controlling/monitoring tasks, the acquired images must be sent toward two sub-systems namely, *Data Archiving* and *Synchronization System*.

The block diagram in the Figure 4 outlines how the acquired data is acquired and transferred to the two above mentioned sub-systems.

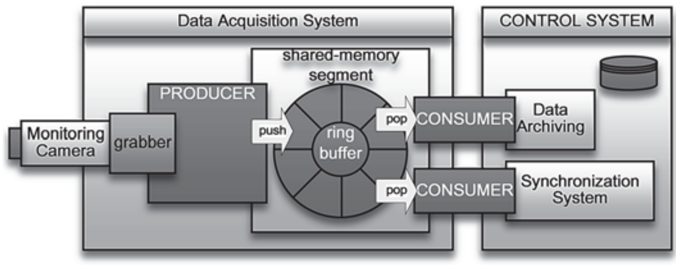


Fig. 4. Data acquisition system block diagram

The *Data Archiving* sub-system is devoted for persistent data storage, the storing tasks undertaken by this sub-system are considered no critical, since there is no specific time when they must be completed mainly because they are not attached with the control operations undertaken over the experimental fusion device. Nonetheless, the operations effected by the *Synchronization System* are extremely critical for the process of controlling the experimental fusion device [18].

In order to monitor the effective functioning of the experimental fusion device, the sensed variables must be measured continuously and the data proceeding from these capturing procedures must be processed promptly. This with the objective of supporting the early detection of any fail in the operation of the whole-system. As a result of this fact, a hundred percent of *accuracy* of the data transfer it is mandatory. This means that for the critical sub-systems where images are captured, all acquired images must be transferred pertinently to detect any failure in real-time. Likewise a hundred percent of *reliability* is needed to evaluate permanently the quality and consistency of the transferred data. The developed Data Acquisition System must be able to fulfill these requirements by transferring the frames from the cameras toward the Control System in the same rate that they are produced.

C. *Data Acquisition System Operation*

The Figure 4, presented above, outlines a generic schema of the Data Acquisition System. One *producer* and two *consumers* form the depicted schema. The *producer* process is in charge of gathering the message from the source device, and ask for an available slot in the shared ring-buffer, if there is no any available slot, the *producer* process become into waiting state until some signal is received from the consumers. The meaning of such signal is connected with the fact that there is room for a new incoming data.

The *consumer* processes run independently of one another and the *producer* process. Ideally and directly linked with the operating system scheduler algorithm, each *consumer* process should run in its own CPU.

Specifically, one consumer is aimed to manage complete data messages and transfer them toward the archiving sub-system. In addition, the other *consumer* process is in charge of managing the synchronization data to confirm the data accuracy of each event.

D. *The Transferred Data*

The images captured by the high-speed cameras are grabbed by the producer process and placed them into the shared ring-buffer, one per slot. Each image has by definition the TIFF standardized format [19], what it means is that each image is equivalent to the intricate pair *meta-data* + *raw-data*. The *raw-data* constitute the image itself and the *meta-data* represents the identification of each frame named *image-header*.

III. Related Works

An important component of all modern operating systems is the implementation the well-defined inter-process communication (IPC) mechanisms, both for communicating data among processes and for synchronizing the access to the data itself.

Notwithstanding, for the rapid developing of applications where inter-communication among processes is required, it is necessary a higher level of abstraction in order to encapsulate the low-level calls of the operating related with the services of shared-memory and IPC. This is the principal reason for developing libraries that enclose the details of the low-level operating systems calls and to provide a user-friendly (developer) interface that meets with applications requirements [20]. Hereafter, some libraries are described briefly.

The *FastFlow* is a *C++* framework for parallel processing advocating high-level and pattern-based parallel programming [21]. This framework has been developed to provide a parallel programming abstraction and simplify the development of multi-core platforms. Even though is a complete solutions, it is multi-threading oriented, and one the requirements to develop the DAQ for the power generator is to use independent processes.

The *ADAPTIVE Communication Environment* (ACE) is a toolkit that implements patterns for communication software [22]. This framework is object-oriented and aimed to develop network applications involving concurrency and inter-process communication. However, although is a complete solution based on C++, object-oriented and with IPC interfaces, the main disadvantage is that it is based on Microsoft Windows ® inter-process communication technology, and the high-performance DAQ must be based on Linux Operating System.

The *Dynamic Shared-Memory Allocator* (DSMA) has been developed to provide IPC mechanisms to ensure communication, data transfer and synchronization among different processes [23]. This solution is based on the use of virtual-memory paging mechanism, which is implemented in most operating systems and allows using a virtual-memory space larger than the available physical system memory. While this solution is simple and intuitive, it also relies on the multi-threading technology, which it is no suitable with the requirements of the DAQ.

IV. Proposed Solution

A. General Overview

Notwithstanding the use-case presented in this paper is related with high data-stream production by high-speed cameras, the C++ framework has the capacity of managing generic data types by using the C++ templates approach. This means that in order to create an application software based on the framework, the programmers must declare and create an object of the producer and consumer classes specifying the data-type as a template parameter. Indeed, specified data-type is the expected data to be transferred among the processes.

Three main classes compose the framework, as follows: the first class is in charge of managing the shared-memory segment for a given data-type whether data itself or an IPC synchronization objects. The second one, named *channel*, implements different forms of communication among processes by using synchronization-objects. Finally, the third class implements a ring-buffer data-structure and includes the algorithms of reading and writing from and into this shared-memory data-structure.

B. The Logical-Design

The Figure 5 presents schematically the logical structure of the framework in accordance how the components work. The synchronization objects are encapsulated in a series of classes, which place such the IPC in shared-memory. This means that each synchronization object is allocated in shared-memory making it accessible for all processes that have mapped such shared-memory segment.

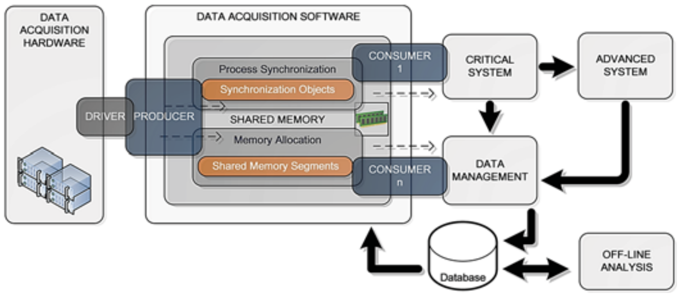
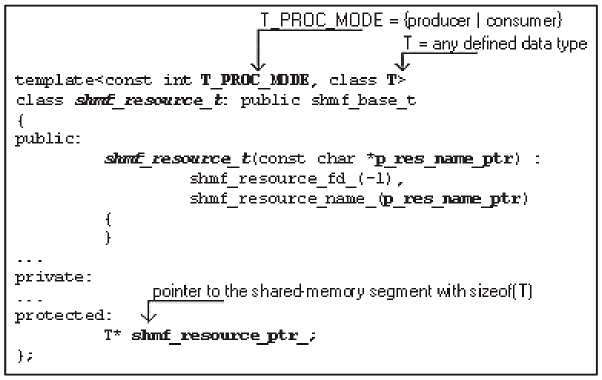


Fig. 5. C++ shared-memory ring-buffer logical design

Since by design the framework is oriented to provide the appropriate interfaces for developing final multi-process monitoring applications, all the classes have been developed using C++ templates. Concretely the shared-memory resource class allows creating shared synchronization objects and the specific message data type to be transferred as is shown in the Figure 6.


T_PROC_MODE = {producer | consumer}
T = any defined data type
pointer to the shared-memory segment with sizeof(T)

```
template<const int T_PROC_MODE, class T>
class shmf_resource_t: public shmf_base_t
{
public:
         shmf_resource_t(const char *p_res_name_ptr) :
                  shmf_resource_fd_(-1),
                  shmf_resource_name_(p_res_name_ptr)
         {
         }
...
private:
...
protected:
         T* shmf_resource_ptr_;
};
```

Fig. 6. Shared-memory class interface.

Additionally, it is important to underline that for this paper the use-case is oriented to evaluated the effectiveness of the framework when high-speed cameras are used. The data transferred among process were format *meta-data* + *raw-data* as has been outlined previously.

C. The Physical Design

The Figure 7 details the relationship among the classes that forms the framework. All of them use templates making possible to share any synchronization object among processes and to communicate any data as well. The framework is designed over the classes framework is designed over three base classes, namely *shmf_resource_t* (shared memory resource), *shmf_channel_t* (channel) and *shmf_proc_t* (process).

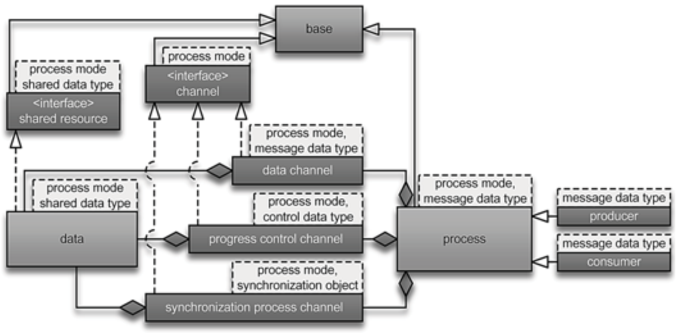


Fig. 7. C++ Shared-memory ring-buffer physical design

The framework provides two classes as interfaces to developing complete monitoring applications. In a typical application, the program responsible for grabber the data from monitoring devices and populate the shared ring-buffer with such data, should create an object of the type *shmf_producer_t* (producer) and any program which is going to grabber data from the shared ring-buffer should create an object of the type *shmf_consumer_t* (consumer).

V. Experimental Evaluation

A. The Testing Enviroment

The evaluation has been performed in a system with sixteen CPUs Intel® Xeon® CPU C5549 @ 2.53GHz, 8192 KB of cache size running the operating system Red Hat Enterprise Linux Workstation release 6.5 based on the Linux kernel version 2.6.32-431.20.3.el6.x86_64. Furthermore, the system is equipped with 24GB of RAM memory.

This Linux distribution is provided with the *gcc* (GCC) 4.4.7 20120313 (Red Hat 4.4.7-4) compiler and all source code has been compiled using the flag -O3 to take advantage of all the optimization improvements provided by this compiler.

The data grabbing process has been simulated by means of a data file produced by the cameras, which it was mapped into memory in order to make the data accessible during the whole experiment in the fastest way available. It is important to point out that the experiment was conducted by using only *6815 frames* from the data file, which corresponds to the cameras resolution setting of *200x200 pixels*.

B. The Timing Process

In accordance with the requirements stated by the project for which this research has been conducted, the data should be taken from a shared-memory segment to simulate the frame production from the cameras. This has been achieved by mapping the file into memory with the mapping memory mechanism provided by all of *NIX like operating systems.

Following the execution logic of the timing algorithm depicted in Figure 8, the *producer* process maps the sample data file into memory by means of *mmap* system call [24]. Then each frame is read sequentially and if there is room in the shared ring-buffer for allocate the new read frame, the available slot is fills with the latest read frame. Before pushing the new message in the shared ring-buffer, each message is identified with an integer number to provide a control method to calculate latency and to checking the accuracy of the all message in the appropriate sequence.

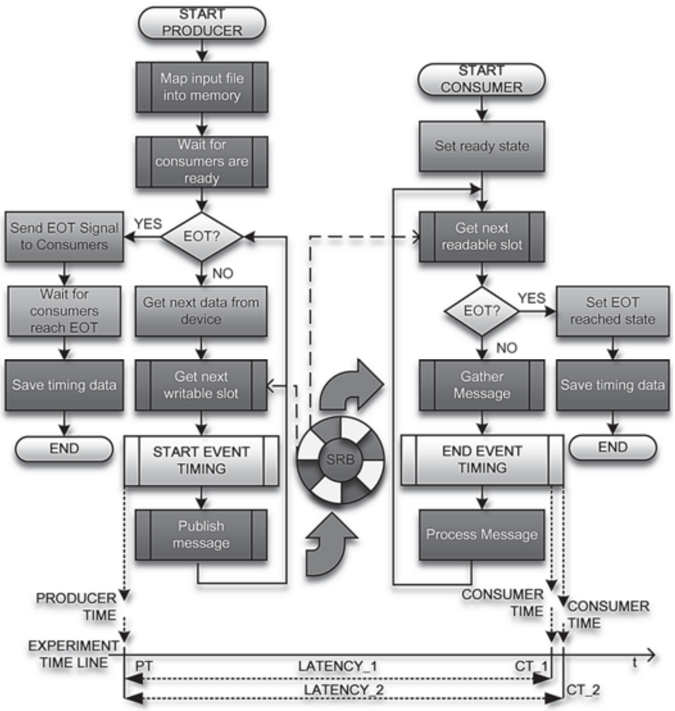


Fig. 8. Timing algorithm

The sub-routine START EVENT TIMING is executed just before the event is pushed into the next available slot in the shared ring-buffer. Internally the start measurement time is gathered in this sub-routing by calling the system function *clock_gettime(CLOCK_REALTIME, &timespec)*, where CLOCK_REALTIME refers to the system-wide real-time clock and &timespec to the data structure where the clock value are stored.

In the meantime and concurrently, the *consumer processes* check if there is a new data arrival in the shared ring-buffer to be gathered. If any, the *consumer* process takes the message and the sub-routine END_EVENT_TIMING is called, which implies to calling the *clock_gettime* system function in order to take the stop transmission time for a specific message.

When the *producer* process reach the end of the input file, it sets the *end-of-transmission* (EOT) flag into the next slot and it dumps the gathered timing data into a binary file. Similarly, when the *consumer processes* detect the EOT flag, each consumer dumps the stop time snapshots into a separated binary file.

The *producer* process is responsible to close all shared resources created during the timing process, for this reason, the *producer* process must wait for the appropriate ending of all *consumer* processes. The *consumer* processes notify the producer about the last message has been reached by means of setting the *EOT* flag.

C. The Timing Data Processing

The timing process was executed for one million of message transmitted and with different number of slots for the shared ring-buffer, beginning with 256 and being increased by twice up to 8192 slots.

The next stage is to process the timing files in batch mode by verifying the correspondence of each measurement by using the identification of each message. The start time is subtracted from each stop time and the greatest difference is taken as the latency time for the message under consideration. The results analysis is based on this latency time calculation.

VI. Results

A. IPC Mechanism Comparison

By means of the effectiveness rates reported in Table I it can be established that by using the *batch-consuming* method have an important impact for the improvement of the performance. This is explained by the fact that the *consumers* processes become into wait mode only when the first unread slot in the *ring-buffer* is reached. By contrast, with the *per-slot* method every slot in the ring-buffer is checked during the consuming process. The column *No. Slots* represents the number of slots to which the ring-buffer was configured and it can noticed that by increasing the number of slot do not make an impact in the reduction of latency itself.

TABLE I.
Effectiveness rates of C++ shared ring-buffer

# Slots	*PER-SLOT*			*BATCH MODE*		
	POSIX	*FUTEX*	*ATOMIC BUILT-IN*	*POSIX*	*FUTEX*	*ATOMIC BUILT-IN*
256	92.74%	92.39%	99.64%	98.49%	81.75%	99.66%
512	62.66%	97.78%	96.37%	98.57%	92.15%	99.28%
1024	86.33%	98.25%	68.41%	97.59%	62.98%	98.67%
2048	67.00%	88.77%	70.24%	98.19%	92.45%	98.15%
4096	57.30%	84.10%	71.64%	96.49%	95.72%	98.22%
8192	14.71%	85.80%	72.04%	96.42%	83.77%	98.79%

B. Latency

The Figure 9(f) reflects the latency measurements for the shared ring-buffer, which use atomic built-in functions. This approach reports the expected behavior for a message latency because the 94% of the messages were sent in two hundred nanoseconds or less. Furthermore, this behavior is constant for all configurations of slots in the ring-buffer evaluated. Nevertheless, the same approach but signaling by per-slot does not produce suitable results as is depicted in the Figure 9(e). Finally, by using the batch consuming approach incorporates a substantial enhancement aside of the synchronization method used.

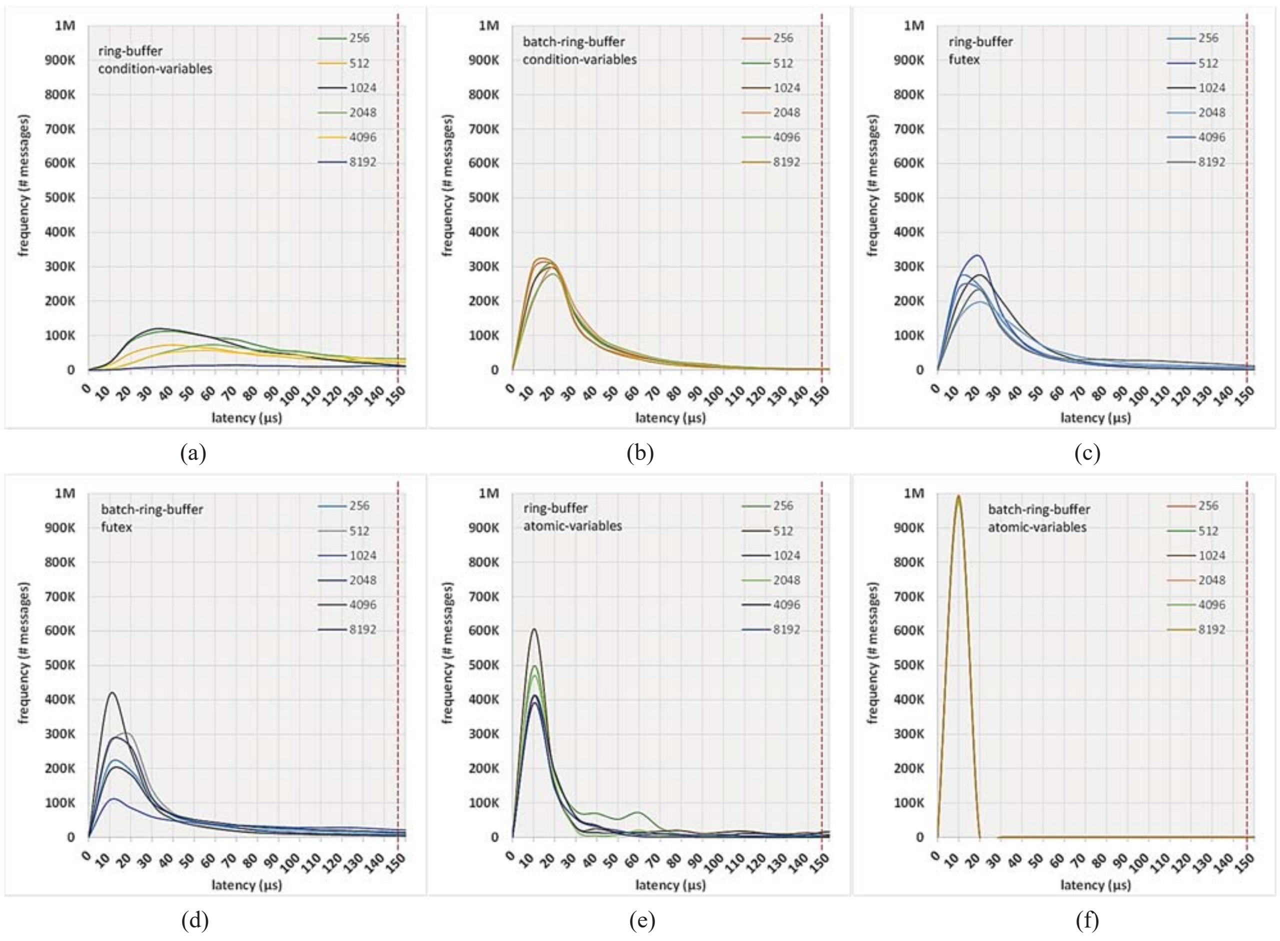


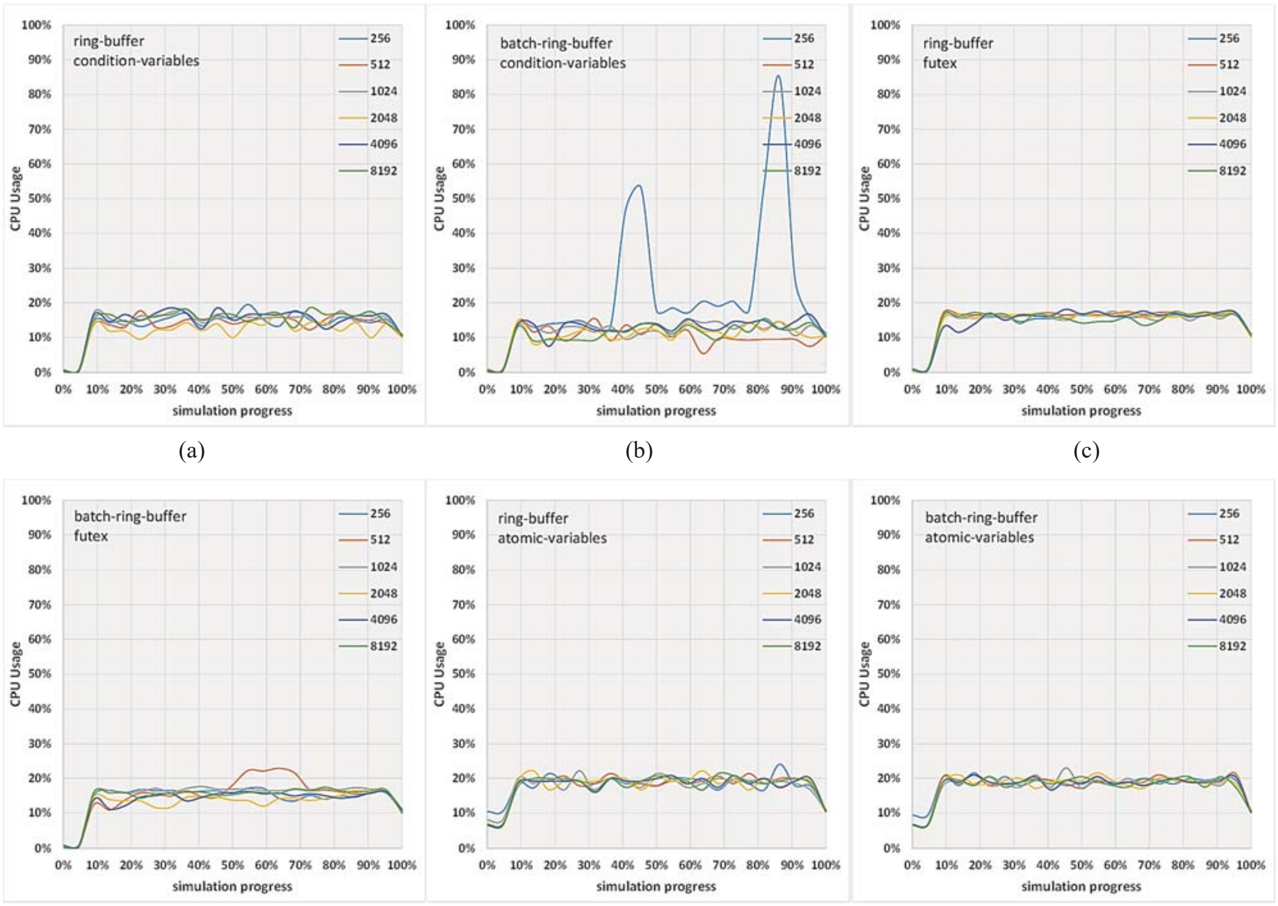
Fig. 9. Latency measurements.

C. *CPU-Usage*

The Figure 10 reflects the best CPU usage for all of experiments conducted during the approaches evaluation. When condition-variables are used as mechanism of signaling, the CPU usage average is 15% as is depicted in the Figure 10(a). By using the same signaling mechanism but with batch consuming mode, an improvement in the CPU usage were reached as illustrates the Figure 10(b). Nevertheless, an unexpected behavior was presented when 256 slots were used in the ring-buffer.

Concerning with the CPU usage associated with the FUTEX signaling mechanism, the Figure 10(c) presents a steady CPU usage around of 16% and 17% for the consume of data in batch mode as illustrates the Figure 10(d). Both results are similar except a suddenly increase for 512 slots in batch-mode.

The CPU usage for atomic-variables presented in Figure 10(e) reflects an expected behavior since the atomic-variable do not provide an waiting mechanism associated with the operating system, it means that an infinite loop is used for waiting for the expected value; hence, the substantial increase in the CPU usage reaching the 20%. The Figure 10(f) presents similar CPU usage when the ring-buffer is consumed in batch mode, meaning that there was no a decrement in the CPU usage as expected.



(a) (b) (c)

(d) (e) (f)

Fig. 10. CPU usage.

VII. Conclusions

In this paper, the experimental evaluation for inter-process communication for high-throughput data acquisition systems was discussed. Our proposed C++ framework design was evaluated by using *POSIX* condition-variables, *FUTEXES* and *atomic-variables* as IPC mechanisms, Likewise, the framework implements two different algorithms of controlling the manner to settle when the producer process can publish a new data and when the *consumer* processes can grab such data.

Based upon the experimental results, the use of *atomic-variables* is a feasible technique for the synchronization of processes when shared-memory is used for transferring data among them. Notwithstanding, in order to obtain better results, the *consumer* processes should use a batch mode for read the new arrivals placed in the shared ring-buffer. With this approach, significant amount of waiting time is saved because there is no over checking if new data has been published.

It is important underline that the use of POSIX *condition-variables* and FUTEX are not suitable with the requirements of the project. In the case where high-rate data transfer is required, the use of atomic variables is recommended, but a high CPU usage must be taken into consideration.

Clearly additional IPC mechanism should be evaluated in order to establish which can manage latency levels similar to atomic variables but with a moderate usage of CPU.

The research conducted in this paper, is the start point for developing a solution that fit with the 100% of accuracy required and that incorporates the 100% of reliability of messages transferred.

References

[1] J. Park and S. Mackay, Practical Data Acquisition for Instrumetnation and Control Systems, Newnes, 2003.

[2] M. Alam and M. Azad, "Development of biomedical data acquisition system in Hard Real-Time Linux environment," in *Biomedical Engineering (ICoBE)*, Penang, 2012.

[3] F. Di Mario, M. Park and A. Wallander, "ITER - CODAC Core System Overview," 8 February 2007. [Online]. Available: http://static.iter.org/codac/cs/CODAC_Core_System_Overview_34SDZ5_v5_4.pdf. [Accessed 23 April 2017].

[4] ITER, "The ITER Tokamak," ITER, [Online]. Available: https://www.iter.org/mach/tokamak. [Accessed 23 04 2017].

[5] J. Y. Journeaux, "Plant Control Design Handbook," 08 April 2013. [Online]. Available: http://static.iter.org/codac/pcdh7/Folder%201/1-Plant_Control_Design_Handbook_27LH2V_v7_0.pdf. [Accessed 09 June 2016].

[6] A. Mielczarek, P. Perek, D. Makowski, M. Orlikowski, G. Jabłoński and A. Napieralski, "AMC frame grabber module with PCIe interface," in *20th International Conference Mixed Design of Integrated Circuits and Systems*, Gdynia, Poland, 2013.

[7] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," in *Principles and Practice of Parallel Programming*, Shenzhen, China, 2013.

[8] M. Torquati, "Single-Producer/Single-Consumer Queues on Shared Cache Multi-Core Systems," Computer Science Department, University of Pisa, Italy, 2010.

[9] A. Alexandrescu, A. Stan, N. Botezatu and S. Caraiman, "Real-Time inter-process communication in heterogeneous programming enviroments," in *20th International Conference on System Theory, Control and Computing (ICSTCC)*, Sinaia, Romania, 2016.

[10] R. Stevens, UNIX Network Programming Volume 2, Second Edition, Interprocess Communications, Prentice Hall, 1999.

[11] M. McKusick and M. J. Karels, "A New Virtual Memory Implementation for Berkeley UNIX(r)," Universidad of California, Berkeley.

[12] M. D. Cvetkovic and M. S. Jevtic, "Interprocess communication in real-time Linux," in *Telecommunications in Modern Satellite, Cable and Broadcasting Service, 2003. TELSIKS 2003. 6th International Conference on*, 2003.

[13] F. Hijaz, B. Kahne, P. Wilson and O. Khan, "Efficient parallel packet processing using a shared memory many-core processor with hardware support to accelerate communication," in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, Boston, 2015.

[14] A. Williams, C++ Concurrency in Action, New York: Manning Publicaitons Co, 2012.

[15] K. Wall, M. Watson and M. Whitis, Linux Programming Unleashed, Sams Publishing, 1999.

[16] B. Gamsa, O. Krieger and M. Stumm, "Optimizing IPC Performance for Shared-Memory Multiprocessors," in *International Conference on Parallel Processing, 1994. Vol. 1. ICPP 1994.*, 1994.

[17] *EoSens® 3CL Camera Manual, Rev. 1.01, Copyright ©,* Mikrotron GmbH, 2010.

[18] H.Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications. Second Edition, Springer, 2011.

[19] A. D. Association, *TIFF Revision 6.0,* 1992.

[20] A. Alexandrescu, A. Stan, N. Botezatu and S. Caraiman, "Real-time inter-process communication in heterogeneous programming environments," in *20th International Confrence on System Theory, Control an dComputing (ICSTCC), October 13-15*, Sinaia, Romania, 2016.

[21] M. Aldinucci, M. Danelutto, P. Kilpatrick and M. Torquati, "FastFlow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, Wiley, 2014.

[22] D. C. Schmidt, "The ADAPTIVE Communication Environment An Object-Oriented Network Programming Toolkit," in *12th Sun User Group conferences*, San Francisco, 1993.

[23] R. Benosman, K. Barkaoui and Y. Albrieux, "A New Dynamic IPC-Memory Allocator Based on a Paging Approach," in *High Performance Computing and Simulation (HPCS)*, Paris, 2013.

[24] M. Mitchel, J. Oldham and A. Samuel, Advanced Linux Programming, New Riders Publishing, 2001.

Rolando Inglés was born in 1972. He earned the B.Eng. degree in System Engineering and Computer Science in the University of Technology of El Salvador in 1996. He received the M.Sc. degree in Systems Engineering in the Engineering Institute of the Autonomous University of Baja California, Mexico; in 2003. He is currently a full-time Ph.D. student in the Department of Microelectronic and Computer Science, Lodz University of Technology. His research interests include object-oriented programming, multi-threading and multi-processes concurrency issues and inter-process communication for large-scale data processing systems.

Mariusz Orlikowski was born in 1971. He received MSc and PhD degrees in electrical engineering from Lodz University of Technology in 1995 and 2000 respectively. He is currently an Associate Professor in the Department of Microelectronics and Computer Science Lodz University of Technology. His research interests include behavioral modelling using Hardware Description Languages, object oriented programming, distributed programming of data acquisition and processing systems.

Andrzej Napieralski received the M.Sc. and Ph.D. degrees from the Lodz University of Technology (TUL) in 1973 and 1977, respectively, and a D.Sc. degree in electronics from the Warsaw University of Technology (Poland) and in microelectronics from the Université de Paul Sabatié (France) in 1989. Since 1996 he has been a Director of the Department of Microelectronics and Computer Science. Between 2002 and 2008 he held a position of the Vice-President of TUL. He is an author or co-author of over 900 publications and editor of 18 conference proceedings and 12 scientific Journals. He supervised 44 PhD theses; five of them received the price of the Prime Minister of Poland. In 2008 he received the Degree of Honorary Doctor of Yaroslaw the Wise Novgorod State University (Russia).