A distributed data acquisition system for nuclear detectors

Cristiano L. Fontana*, Alberto Carnera, Marcello Lunardon, Felix E. Pino, Cinzia Sada, Francesca Soramel, Luca Stevanato and Sandra Moretto

Department of Physics and Astronomy “Galileo Galilei”
University of Padua, Via Marzolo, 8
I-35131, Padova, Italy
*cristiano.fontana@pd.infn.it

Published 10 July 2018

Nowadays, many examples of data acquisition (DAQ) software for experimental nuclear physics are monolithic processes that run on a computer attached to the DAQ hardware. In this article we present a distributed DAQ system developed for the C-BORD project. With our system, we propose a novel approach, in which each task related to the different DAQ parts (acquisition, pre-process, analysis, etc.) runs in a separate process. In particular, the system is composed of a set of servers that exchange information through dedicated communication sockets. Therefore, with this architecture, an important advantage is the possibility to run the processes on different computers to distribute the computational load. The initial tests of the system have been giving excellent results, both in terms of performance (i.e., maximum acquisition rates) and stability. The project entitled “Effective container inspection at BORDer control points” (C-BORD) is funded by the European H2020 programme. Its aim is to develop a comprehensive set of technologies for the generalized non-intrusive inspection (NII) of containers and large-volume freight at the European Union border.

Keywords: Data acquisition system; distributed system; C-BORD.

PACS numbers: 07.05.Hd, 29.85.Ca

1. Introduction

The whole process of reading and digitizing the signals generated by nuclear detectors is usually referred to as data acquisition (DAQ), and the software that performs such a task is commonly called DAQ software, or simply DAQ. A DAQ is obviously in charge of reading out the analog-to-digital converters (ADC) results but often also manages the auxiliary tasks related to the nuclear detector systems (e.g., high-voltage power supplies, temperature measurements, and so on). Fast, modern ADC systems are called “digitizers” because they can digitize a full waveform, much like what oscilloscopes do. Here we will focus on the point of view of the development of the DAQ software.
Many examples of DAQ software for nuclear detectors are monolithic processes that run on a computer attached to the DAQ hardware (e.g., CAEN MC²Analyzer or ORTEC Maestro). In some cases, the graphical user interfaces and data display run on a different process than the DAQ server (e.g., FAST ComTec MCA4 or CAEN WaveDump) to minimize the interference between data visualization and data acquisition. In some other cases a network support was added on later versions (e.g., ORTEC Maestro) that allows remote control and monitor. We propose here a fully distributed approach, in which all the tasks related to the DAQ are split over different processes. The interprocess communication is obtained through network sockets. Each process is dedicated to a simple purpose, following the ideas of the UNIX Philosophy. For instance, one process reads only the ADCs data, another process manages HVs, and another one preprocesses the data for visualization. With this distributed architecture, we were able to isolate, test, and debug each part very easily. Our field tests of this system demonstrated that the whole network becomes very robust and reliable. Especially during the early testing stages, we were able to update and modify the single processes on an already deployed system without disturbing the overall functionality.

This DAQ system was developed for the European H2020 project entitled “Effective container inspection at BORDer control points” (C-BORD). The C-BORD project’s aim is to develop a comprehensive set of technologies that can be employed together for the non-intrusive-inspection (NII) of commercial freight containers. The need for a new DAQ was justified by the project requirements: the DAQ is supposed to interoperate with the project’s software infrastructure. Moreover, the performance of the existing DAQ was not sufficient. This system was developed to acquire data from CAEN digitizers and control CAEN HV power supplies.

2. DAQ General Architecture

Figure 1 shows a diagram of the general DAQ architecture. In the figure, each box represents a process (also called server) that runs independently from the others. Arrows indicate the communication channels between the processes. The servers are implemented following the finite-state machines paradigm. We employed a single-threaded approach: only one main loop reads the current status, performs some actions, and determines what the next state is according to the inputs. This single-threaded approach greatly simplifies the development phase as it is much easier to follow the program flow and debug the software. The high-performance servers were developed in C99 and C++11, auxiliary tools and scripts in Python, and the user interface as a Web service based on Node.js. Some scripts were prepared to fully control the DAQ system from the command line (e.g., while connected through an SSH connection). The processes are very lightweight and were tested on a varied set of hardware. On legacy systems, the new DAQ was able to outperform the former DAQ that we were employing.
The communication between processes is implemented through the optimum ZeroMQ messaging library. The ZeroMQ library allows communication through dedicated sockets, by means of sending content agnostic messages. We opted to use a publish–subscribe (PUB–SUB) pattern for data and status streams, and a push–pull pattern for command delivery. The PUB–SUB pattern allows multiple subscriber clients to be connected to a single publisher server. The messages are published by the publisher (i.e., are broadcast to all the subscribers) with no real knowledge if any subscriber is listening. This pattern allows several processes to be reading the same data streams, receiving all the same messages, and working in parallel. All the processes statuses and raw data are published through a PUB socket, as this information can be used by many consumers. The push–pull pattern allows several pushers to send messages to a single receiver. Commands travel through push–pull sockets to guarantee that the messages will be delivered. Processes exchanging messages can be hosted in the same machine or can communicate through a network. The rationale for hosting the processes on different computers is to share the workload of some computational intensive tasks.

Serialization is defined as the process that translates information to a format that can be stored in memory and/or transmitted. We are using two formats for data serialization: JSON and custom binary formats. Status data and commands are serialized as JSON strings, as it is a very common standardized format. It can be easily parsed from software (with many examples of libraries) and can be easily read by human beings. Configuration files are written as JSON objects as well in order to simplify their parsing and network publication. Raw data is serialized with a custom binary format that reflects the digitizer's data structures. Two kinds of raw data are foreseen at the moment: signal waveforms acquired by the digitizer and the on-board processed parameters of the radiation interactions (e.g., timestamps, energy, etc.).
We are not enforcing any kind of security in the DAQ, as it is designed to run inside protected networks of laboratories. On the other hand, some workarounds can be employed. The ZeroMQ library can use local inter-process communication (IPC) transports. The IPC transport is based on UNIX domain sockets that allow communication inside the host only; therefore, the DAQ could be controlled only from that host. If TCP sockets are to be used, Virtual Private Networks could isolate the system from malicious users.

3. Hardware Interfacing Servers

In our system, two servers are dedicated to the interface with the experimental hardware: the high-voltage- (HV-) management server (Fig. 2) and the acquisition (ACQ) server (Fig. 3). In the C-BORD project, we have been using CAEN digitizers and modules that support several kinds of communication interfaces: USB and optical fibers. The DAQ has been tested both with USB communication and optical fibers.

Fig. 2. Simplified state diagram for the high-voltage-management server. Normal transitions are triggered by internal events of the server; command transitions are triggered by commands received through the command sockets.
The HV manager has a very simple internal structure (Fig. 2), with a main loop that polls the hardware status. Periodically the HV statuses are published through the PUB status socket. If a command is received through the command socket, the server initiates a reconfiguration of the HV modules. The file with the initial configuration follows the JSON format.

The kernel of the system is the data readout from the digitizers, as its purpose is to acquire the physical data. The readout server has two modes of operation (Fig. 3): an idle mode, in which it monitors the digitizer status, and an acquisition mode, in which it activates the data acquisition and continuously polls the digitizer to obtain data. Being executed in a single thread, the repetition rate of the loops depends on a delay that can be set during the start up. Raw data is buffered in a queue, and when a timeout occurs, or the buffer is full, it is published through the PUB data socket. Periodically the digitizer status is published through the PUB status socket. Having such a simple structure, we were able to achieve excellent performance and stability compared to our previous DAQ system. The system stability is being tested on a very long acquisition; up to the moment of this writing, the acquisition has not shown instabilities and has been running for almost three months.
Performances can be measured by varying the rate of incoming signals and measuring the acquired rate and **dead time**. The dead time is the time in which the system is busy and is not able to acquire data; it can be expressed as the portion of time that the system is busy:

\[
\tau = \frac{\text{Busy time}}{\text{Acquisition time}} \times 100\% 
\]

Dead time is caused by a combination of the digitizer’s intrinsic time required to process the events and the ability of the DAQ to read the data in a timely manner from the digitizer. We will refer to the rate as the average rate of the incoming signals over a time of the order of minutes and measure it in events (or samples, S) per second (S/s). The tests were performed on a desktop computer with an Intel i7 processor and a CAEN A3818 PCI OpticalLink interface for communications. The digitizer used was a V1730 connected with the optical fiber interface. With this new DAQ, we were able to acquire data at a rate of more than 1 MS/s. The dead time stays below 1% up to a rate of 600 kS/s; after that rate, the dead time starts to increase up to about 50% at 1.2 MS/s. At 1.2 MS/s, the physical limit of the communication channel is reached. On the other hand, with the old system we were able to acquire up to a rate of 340 kS/s with a similar computer and the same A3828. The old system is monolithic software, developed in-house, that is multithreaded. The inability to achieve higher rates was due to instabilities and crashes induced by the interaction of the GUI thread and acquisition thread.

### 4. On-Line Data Processing

We developed a server dedicated to the on-line generation of the energy spectra of the detectors and of other important parameters for pulse shape discrimination. It reads and analyzes the binary data from the SUB sockets, and the resulting histograms are then published on a new PUB socket. The processed data are serialized as JSON objects to ease their interpretation and parsed with subsequent software. The processed data can be analyzed by other high-level analysis software that is usually tailored to the particular experiment.

### 5. Data Logging and Simulation

A data logger reads from the data and statuses sockets and saves all the raw messages to a file, which can be read from a simulator. The simulator substitutes the HV manager and ACQ server and generates simulated messages that fully reproduce the saved session. With the simulator, we are able to test new features without using real hardware. The digitizer data can be optionally saved to dedicated binary files to simplify the off-line analysis.
6. User Interface and Manager

To improve the accessibility of the DAQ for the users, we decided to develop its graphical user interface (GUI) as a Web service. With this approach, the system can be controlled by any computer that is connected to the DAQ host without the need to install dedicated clients. A Node.js instance runs an application that reads the published messages from the data and status sockets. This application then performs some minor processing and serves a Web page with controls and displays. Real-time communication with the browser is obtained through the Socket.IO library\textsuperscript{9} that employs WebSockets for fast communication.

The GUI application is foreseen as a DAQ manager, as well. The event-driven programming model of the Node.js platform is well adapted to the needs of a manager. Some actions may be programmed when particular events occur, such as emergencies from the HV modules or external triggers that can start or stop the acquisitions. The manager is not mandatory, as an expert user can execute the necessary steps to run an acquisition. On some experiments, though, the acquisitions may be automated by adapting the manager.

7. Conclusions

In this article we present a new, distributed DAQ system. It is based on a novel approach, in which each task related to the DAQ runs as a server in a separate independent process. Every server communicates with the rest through network sockets that are dedicated to data streams, status feeds, and command delivery. The processes are light-weight and were tested on a varied set of hardware. The performances of the new system are excellent, both in terms of acquisition rate and stability. Compared to the generally available DAQs, our approach allows for a very robust, yet flexible, system to be had that can be reconfigured on-line and remotely. The GUI was developed as a web service; therefore any computer can monitor and control the system through a network without the need of a dedicated client.

References

1. E. S. Raymond, \textit{The Art of UNIX Programming} (Pearson Education, 2003).
2. C. L. Fontana \textit{et al.}, \textit{Physics Procedia} \textbf{90}, 279 (2017).
3. CAEN s.p.a. http://www.caen.it/ (Retrieved June 2017).
4. P. Hintjens, \textit{ZeroMQ: The Guide}, http://zguide.zeromq.org/page:all (Retrieved June 2017).
5. ECMA, \textit{The JSON Data Interchange Format}, Tech. Rep. Standard ECMA-404 1st edn. (2013), Available: http://www.ecma-international.org/publications/files/ECMAST/ECMA-404.pdf
6. L. Stevanato, D. Cester, G. Nebbia, G. Viesti, F. Neri, S. Petrucci, S. Selmi and C. Tintori, \textit{Nucl. Instr. Meth. Phys. Res. A} \textbf{678}, 83 (2012), https://doi.org/10.1016/j.nima.2012.03.016.
7. D. Cester, M. Lunardon, G. Nebbia, L. Stevanato, G. Viesti, S. Petrucci and C. Tintori, \textit{Nucl. Instr. Meth. Phys. Res. A} \textbf{748}, 33 (2014), https://doi.org/10.1016/j.nima.2014.02.032.
8. F. Pino, L. Stevanato, D. Cester, G. Nebbia, L. Sajo-Bohus and G. Viesti, \textit{Appl. Radiat. Isot.} \textbf{89}, 79 (2014), https://doi.org/10.1016/j.apradiso.2014.02.016.
9. Socket.IO https://socket.io/ (Retrieved June 2017).