DAQling: an open-source data acquisition framework

Marco Boretto[1,2,3], *Wojciech* Brylinski[1,4], *Giovanna* Lehmann Miotto[1], *Enrico* Gamberini[1,*], *Roland* Sipos[1], and *Viktor Vilhelm* Sonesten[1]

[1]CERN, Geneva, Switzerland
[2]INFN Torino, Torino, Italy
[3]University of Torino, Torino, Italy
[4]Warsaw University of Technology, Warsaw, Poland

Abstract. The Data AcQuisition (DAQ) software for most applications in high energy physics is composed of common building blocks, such as a networking layer, plug-in loading, configuration, and process management. These are often re-invented and developed from scratch for each project or experiment around specific needs. In some cases, time and available resources can be limited and make development requirements difficult or impossible to meet. Moved by these premises, our team developed an open-source lightweight C++ software framework called DAQling, to be used as the core for the DAQ systems of small and medium-sized experiments and collaborations. The framework offers a complete DAQ ecosystem, including a communication layer based on the widespread ZeroMQ messaging library, configuration management based on the JSON format, control of distributed applications, extendable operational monitoring with web-based visualisation, and a set of generic utilities. The framework comes with minimal dependencies, and provides automated host and build environment setup based on the Ansible automation tool. Finally, the end-user code is wrapped in so-called "Modules", that can be loaded at configuration time, and implement specific roles. Several collaborations already chose DAQling as the core for their DAQ systems, such as FASER, RD51, and NA61/SHINE. We will present the framework and project-specific implementations and experiences.

1 Introduction

Software Data AcQuisition (DAQ) systems are comprised of building blocks that are common to different applications, such as data flow, storage, control, configuration and operation monitoring. Even though a laboratory setup can be easily read-out using a single personal computer, from test beam setups to medium-sized experiments the acquisition system is distributed on multiple readout devices, Commercial-Off-The-Shelf (COTS) components, and commodity servers. Therefore the scalability to distributed systems is a necessary feature of a modern software data acquisition framework targeting small to medium-sized systems.

DAQling [1] is an open-source lightweight, yet complete, software framework for data acquisition. Its main components are developed in C++ and Python. The development has started in 2019, but leveraging on third-party tools and libraries allowed for a reduced development time.

*e-mail: enrico.gamberini@cern.ch

2 Overview

The framework is composed of three main elements: `Core`, `Modules`, and `DAQ control`. The `Core` is the backbone of the DAQling processes and it is developed in C++ using standard features up to C++17. `Modules` wrap user code written in C/C++ and are loaded by the `Core` as shared libraries. Finally, the `DAQ control` is a set of Python 3.6 libraries, handling the execution of processes, distributing commands and configurations and polling the status of processes. Figure 1 shows the main elements and their interconnection.

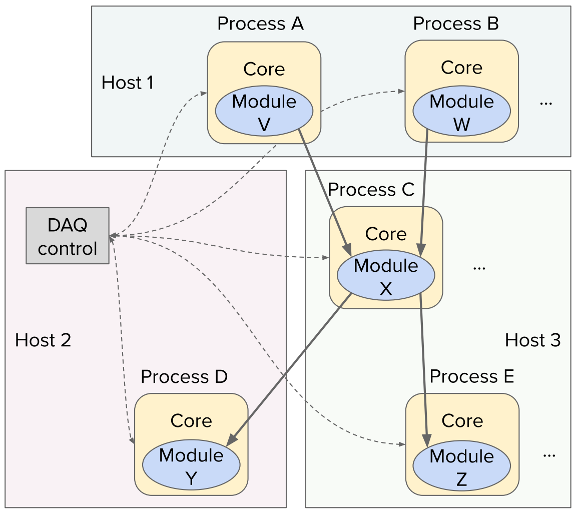

Figure 1. Example scheme of the main elements distributed on multiple hosts.

2.1 Core

The `Core` enforces the use of base features provided by the framework, such as `Module` loading, communication, configuration, logging, and monitoring as summarised in Figure 2. `Modules` acquire these functionalities and standard methods through the inheritance from the `DAQProcess` base class.

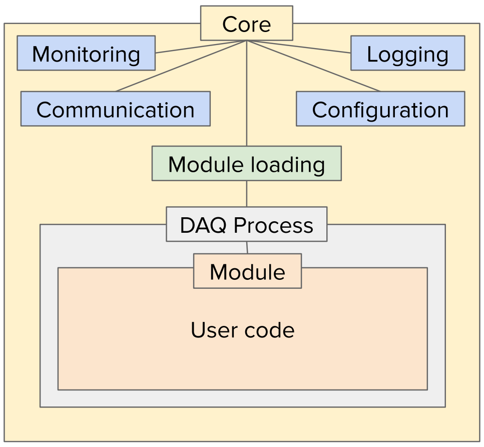

Figure 2. Diagram showing a simplified include tree of the DAQling `Core`.

The `Module`-loading utility is part of the bare-bone `Core` application; this utility loads dynamically a shared library, whose name is specified in the configuration provided to the application. The shared library is based on the `Module`'s class, which is inheriting from the common `DAQProcess` interface class.

The entry point for the `Core` class is the `daqling` executable, which takes a few initial parameters such as control communication port and log levels. The executable launches the `Core` and initialises all the base features. After the initialisation is complete, it then waits for further commands.

Communication

The communication utility provides configurable channels for control, data, and monitoring, based on the *ZeroMQ* [2] messaging library. The data channels are implemented as a queue system in which the *ZeroMQ* sockets are connected to a user-space Single-Producer/Single-Consumer queue (from *Folly* library [3]). The user-space queue allows monitoring if back-pressure is generated in the data-flow system.

The TCP/IP and inter-process shared memory transports are supported, respectively for distributed and local systems. For what concerns communication patterns, the *ZeroMQ* exclusive pair and publish/subscribe are supported, the latter with the possibility to filter the subscribed data on a topic represented by a programmable number of leading bytes of the message.

The channel for control is initialised at the startup of the `daqling` application through the `Core`, and it is implemented using the Request/Reply pattern. On the `Core` side, the channel is polled continuously waiting for requests from the `DAQ control`, such as status request or commands, and after execution, replies with the result through the same channel.

The data-flow channels are instead initialised during configuration, specifying communication transport and pattern, channel identifier, and address. Messages passed through these channels are exposed to the user code as raw binary structures. This design choice leaves the responsibility on the interpretation of the data to the user, and therefore also allows for maximum flexibility.

The monitoring channels are initialised at startup and during configuration, depending on whether the metrics to monitor are defined in the `Core` or the `Module`. They are implemented as Publishers.

Configuration

The configuration utility is a simple interface allowing the configuration's JSON structure to be accessible inside the application, both in the `Core` and `Module` parts. The JSON handling implementation is based on the *nlohmann/json* [4] header-only library. The default `Core` configuration fields include name, host/port pair, `Module` type, logging level, connections, and `Module`-specific settings that can be accessed via the `getConfig` and `getSettings` methods.

Logging

The logging utility is wrapped around the *gabime/spdlog* [5] header-only library. Messages are entered in the code utilising methods named according to their severity level, from `DEBUG` to `ERROR`. Messages are automatically formatted adding time, severity, location in code, and customised message. The minimum severity of messages to be collected can be configured separately for the `Core` and `Module` part, to avoid flooding of `DEBUG` messages from `Core` while debugging the user code in `Module`.

The library provides a selection of standard sinks, between which the *stdout* is used as the baseline for DAQling. The library also allows to add custom sinks; DAQling implements a *ZeroMQ* sink, publishing log messages towards a centralised log collector (implemented as a subscriber).

Monitoring

Operational monitoring is a crucial feature that allows to continuously check metrics regarding the data flow for signs of back-pressure or problems in general. DAQling exposes a `registerMetric` method that allows the user to register variables to be published, specifying a string name and which operation should be applied to it. The supported variables types are `int`, `unsigned`, `bool`, `float`, and `double`, while the supported operations are `LAST_VALUE`, `ACCUMULATE`, `AVERAGE`, and `RATE`. The monitored metrics are published via *ZeroMQ* towards a Python broker (called *Metrics manager* in DAQling framework). The latter is subscribed to all metrics publishers and can be configured to route the values from each source to one of the two supported destinations: *InfluxDB* [6] and/or *Redis* [7]. In the setup provided by DAQling, *InfluxDB* is used as the source by the *Grafana* [8] front-end for time-series visualisation. The *Redis* in-memory cache can be easily used by any custom client to display polled values.

2.2 Modules

The `Modules` are wrappers for the user code and are built as shared libraries to be loaded by the `Core` application. A `Module` inherits the standard commands provided by `DAQProcess` in the form of a set of methods that are re-implemented to specify functionalities to the `Module`. `DAQProcess` provides a few basic methods that are expected to be used in a data acquisition system such as `configure()`, `start()`, `stop()`, and a `runner()` that is launched as a runner thread. In addition to these generic methods, a `registerCommand` is also provided, allowing the user to expand the Finite State Machine (FSM) with custom commands and states.

The inherited class is, therefore, a minimal implementation that allows complete freedom to the user in the matter of data flow, internal structure, and logic while being able to exploit the `Core`'s functionalities. The communication channels that are set-up during the configuration step can be accessed from the `Module` code via the configuration's *JSON* structure. Logging messages can be registered to specify the desired log level. Finally, metrics can be registered via the monitoring functionality.

DAQling includes few `Modules` that together with a ready-to-use configuration file allow running a demonstrative data flow system. The `Modules` include a "Readout interface", an "Event builder", and a "File writer". While the latter is a complete configurable binary file writer that can be used in its current state or adapted, the first two are simple examples with an event data generation and an event-number-based building. Nonetheless, the `Modules` show how to develop user code and show the application of all the features in DAQling `Core`, including operational monitoring.

2.3 Control library

The control of the processes is entrusted to a Python library handling the supervision of processes, distribution of commands and configurations, and status checking. The library is employed in an example command-line Python script (shipped with DAQling) and can be used in applications such as web graphical user interfaces (see Subsection 3.1). The library can be used in combination with the DAQling monitoring and logging utilities to develop support tools (e.g. error recovery manager).

The overall configuration of the system is described in JSON format and it is checked against a schema for field correctness and completeness. The use of different schemas (potentially one for every different `Module` type in the system), allows using a web interface to insert or modify values into pre-compiled fields based on the schema itself. In general, the overall configuration allows describing the topology of the data acquisition system and its data flow. In particular, it is possible to spawn applications (loading the specified DAQling `Module`) on different hosts, define their interconnections and `Module`-specific settings.

The process management is based on *Supervisor* [9] and its *mnaberez/supervisor_twiddler* extension [10]. The supervised hosts run a daemon listening on a known port for XML-RPC [11] requests, which can trigger the spawning, automatic restart, status checking, etc. of processes. The DAQling control library wraps a class handling the XML-RPC requests for several supported use-cases.

The communication between the control library and the applications is handled via *ZeroMQ* "control" channels. These are exclusive Request/Reply pairs towards each DAQling application. The commands sent through these channels will call the corresponding methods defined in the DAQling `Modules` (`configure`, `start`, etc.), including custom commands defined through `registerCommand` (see 2.2). In particular, for the *configure* command, the JSON configuration corresponding to the targeted application/`Module` is transmitted. Also, the control channel allows polling the `Module`'s status as defined in the FSM.

2.4 Deployment and build system

The deployment of the framework is handled by the Ansible [12] automation tool with a selection of "playbooks" provided with DAQling. The minimal dependencies and required tools for building and running DAQling are installed via a single "host-setup" playbook, while optional playbooks take care of the installation of additional tools or libraries. The playbooks are supported on CERN CentOS 7, but installation on Debian-based OSs and cross-compilation on ARM architectures have been achieved. The Docker [13] file and corresponding image are available on the DAQling repository, allowing for containerised execution of the baseline DAQling framework.

The DAQling `Core` is written in using C++ revision 17. The building of the framework is handled with CMake (version 3) and GCC8. CMake allows for the selective and configurable building of the DAQling `Core` and `Modules`.

3 Applications

Few projects are developing their data acquisition systems using DAQling: FASER [14], NA61/SHINE [15], and RD51 [16]. More information on the FASER and NA61/SHINE usage are provided in the next Subsections. For what concerns the RD51 collaboration, a simple laboratory setup has been developed, performing the raw data dump, decoding and online monitoring of data coming from the VMM3 front-end ASIC [17] and read out through an FEC card [18]. This system can be easily scaled to test beam tests.

3.1 FASER

The new FASER experiment is expected to start operation in 2021, after the LHC Long Shutdown 2 [19]. The collaboration chose to rely on DAQling as the framework to develop their software data acquisition system. This choice allowed the FASER developers to quickly set up a simulated data flow with the expected topology and data rates. The development of

DAQling started shortly before the application to the FASER DAQ, therefore the feedback, feature requests, and suggestions proved helpful from the beginning of the collaboration.

The FASER DAQ system includes:

- 1× Trigger Logic Board (TLB) issuing hardware triggers and producing event fragments (~ 25 B) with an expected average rate of 500 Hz, peaking at 2 kHz;
- 9× Tracker Readout Boards (TRB), each receiving the hardware trigger and producing event fragments of ~ 250 B up to 1.5 kB;
- 1× Digitizer, from which data is pulled upon a trigger request and with event fragments of ~ 10-20 kB.

These hardware components are paired with corresponding "readout interfaces" which take care of the configuration of detector electronics and acquisition of data. The data from the readout interfaces account for an expected average rate of ~ 9 MB/s, peaking to 70 MB/s in the worst-case scenario. Event fragments from readout interfaces are sent to an "event builder" that assembles and checks the fragments from different sources. The event builder then distributes such complete events to the file writer and the online monitoring processes.

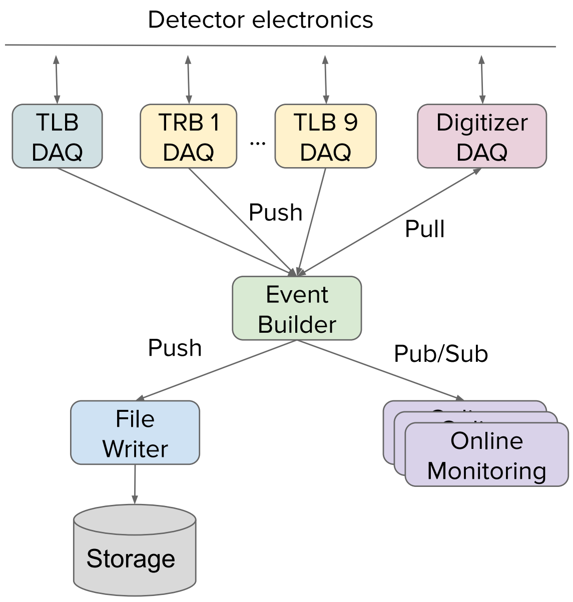

Figure 3. Overview of the FASER DAQ system.

The collaboration also developed a GUI based on a Flask [20] web server. The GUI allows the control of the DAQ system by means of the DAQling control library. It integrates the display of operational monitoring data polled from a *Redis* instance. Finally, a configuration GUI has also been developed, allowing to manipulate the JSON configuration for the different `Modules` using JSON schemas. The described GUIs are to be merged and become an integrating part of the DAQling framework.

The FASER experiment is expected to run nominally without 24/7 shifters. An automatic recovery and alarming system will, therefore, allow remote expert support only in case of problems. The automatic recovery system is planned to be based on DAQling's control library and exploiting its logging, monitoring utilities.

3.2 NA61/SHINE

NA61/SHINE (SPS Heavy Ion and Neutrino Experiment) [15] is a fixed-target experiment operating at the CERN SPS accelerator. The physics program of the experiment was recently extended and major upgrades of the experimental facility during the Long Shutdown 2 are required to perform the planned measurements. The event flow rate is planned to be increased from 80 Hz to 1 kHz using readout electronics obtained from the ALICE Collaboration [21]. This requires the development of a new data acquisition system.

The DAQ system of NA61/SHINE will consist of 12 readout nodes sending the data from different sub-detectors to one of 160 Event Builders for the online reconstruction and the noise rejection.

The Collaboration decided to use DAQling as a base framework for the local readout nodes as well as the Event Builders software. The communication layer, however, will be replaced by a custom library which will allow controlling the data flow by an Orchestrator (Acquisition Supervisor). The Orchestrator will decide about the destination Event Builder of the given sub-event and update the configuration of the nodes.

The DAQling framework was successfully used for the tests of the new Time Projection Chambers (TPC) readout electronics. The DAQ upgrade team developed a module for reading out the TPC data, decoding and online data reconstruction.

4 Summary

DAQling provides a generic software ecosystem for distributed DAQ systems. The C/C++ user code is contained in `Module`s enforcing the use of the framework's utilities while allowing for freedom on data format, data flow and processing choices. The topology of the DAQ system is configurable via JSON files, and the DAQ system itself can be controlled through an extensible control library and integrated monitoring and logging utilities.

The open-source approach of the DAQling project facilitates the continuous enhancement of the framework. Few projects already use DAQling for their DAQ systems and the user communities directly contribute to its growth by feeding back ideas and code.

References

[1] *DAQling*, `https://gitlab.cern.ch/ep-dt-di/daq/daqling`
[2] *ZeroMQ*, `https://zeromq.org`
[3] Facebook, *Facebook Open-source Library*, `https://github.com/facebook/folly`
[4] *nlohmann/json*, `https://github.com/nlohmann/json`
[5] *gabime/spdlog*, `https://github.com/gabime/spdlog`
[6] *InfluxDB*, `https://www.influxdata.com/products/influxdb-overview/`
[7] *Redis*, `https://redis.io/`
[8] *Grafana*, `https://grafana.com/`
[9] *Supervisor*, `http://supervisord.org/`
[10] *mnaberez/supervisor_twiddler*, `https://github.com/mnaberez/supervisor_twiddler`
[11] *XML-RPC*, `https://en.wikipedia.org/wiki/XML-RPC`
[12] *Ansible*, `https://www.ansible.com/`
[13] *Docker*, `https://www.docker.com/`

[14] A. Ariga, T. Ariga, J. Boyd, F. Cadoux, D.W. Casper, Y. Favre, J.L. Feng, D. Ferrere, I. Galon, S. Gonzalez-Sevilla et al., *FASER: ForwArd Search ExpeRiment at the LHC* (2019)

[15] N. Abgrall, O. Andreeva, A. Aduszkiewicz, Y. Ali, T. Anticic, N. Antoniou, B. Baatar, F. Bay, A. Blondel, J. Blumer et al., *NA61/SHINE facility at the CERN SPS: beams and detector system* (2014)

[16] *RD51 Collaboration*, `http://rd51-public.web.cern.ch/rd51-public/Welcome.html`

[17] G. Iakovidis, *VMM3, an ASIC for Micropattern Detectors* (2018)

[18] J. Toledo, H. Muller, R. Esteve, J.M. Monzó, A. Tarazona, S. Martoiu, *The Front-End Concentrator card for the RD51 Scalable Readout System* (2011)

[19] *LHC long term schedule*, `https://lhc-commissioning.web.cern.ch/lhc-commissioning/schedule/LHC-long-term.htm`

[20] *Flask*, `https://palletsprojects.com/p/flask/`

[21] *ALICE*, `http://alice.web.cern.ch/`