The DAQling open source data acquisition framework

M. Boretto, W. Brylinski, G. Lehmann Miotto, E. Gamberini, R. Sipos, V. Sonesten

CHEP 2019 - Adelaide, Australia
4-8 November 2019
“DAQling”

- Software framework providing a generic data acquisition ecosystem
- Key features:
  - Lightweight dependencies \(\Rightarrow\) header-only where possible
  - Processing and data movement performance \(\Rightarrow\) C++17 and ZeroMQ
  - Extensible control and monitoring \(\Rightarrow\) Python
  - Human-readable and structured configuration \(\Rightarrow\) JSON
  - Easy deployment and build \(\Rightarrow\) Ansible automation
- Designed to scale to distributed systems
- Open-source at [gitlab.cern.ch/ep-dt-di/daq/daqling](https://gitlab.cern.ch/ep-dt-di/daq/daqling)
- Project started in 2019, but leveraging on third-party tools and libraries allowed for fast development time
Overview

- **“Core” (C++17):**
  - Backbone of the DAQling processes

- **“Modules” (C/C++):**
  - Wrapping user code
  - Loaded as shared libraries

- **DAQ control (Python):**
  - Launches the processes
  - Distributes commands and configurations
  - Polls the health/status of processes
- The Core enforces the use of **base features** provided by the framework:
  - Module loading, Communication, Configuration, Logging, Monitoring, etc.
  - User Modules inherit functionalities and standard methods from the “DAQ Process” base class

- **Module loading:**
  - Module libraries are dynamically loaded into the barebone Core application

```
JSON
"type": "ReadoutInterface"
```

```
C++
load("libDaqlingModule"+type+.so")
```
Modules

- Module developer implements standard commands provided by DAQ Process:
  - `configure()` ⇒ initialization of module
  - `start()`, `stop()`, and `runner()` ⇒ control data flow and runner thread
  - Custom commands can be registered (e.g. `pause()`, `resume()`)

```cpp
registerCommand("pause", "paused", &ReadoutInterfaceModule::pause);
registerCommand("resume", "running", &ReadoutInterfaceModule::resume);
```

- Implementation of specific roles depends on the project; in general a data acquisition system needs (Readout Interfaces, Event Builders, File Writers, and Online Monitoring)
- Freedom on internal structure and flow
- Example modules for basics data acquisition chain are provided
Core in detail: Communication

- Configurable connections for control and data
- ZeroMQ TCP/IP and IPC transport, with Pair and Publish/Subscribe patterns support
- Messages are raw binary structures (Module developer responsible for data interpretation)
- `zeromq/libzmq.git`, `zeromq/cppzmq.git`
- Data channels implemented as queue system
- Folly SPSC queue `facebook/folly` (header only)
Core in detail: Configuration

- Based on nlohmann/json (header only)
- The utility parses the configuration string into a JSON structure, easily accessible in Core and Modules

```
Core in detail: Configuration

- Based on nlohmann/json (header only)
- The utility parses the configuration string into a JSON structure, easily accessible in Core and Modules

```
Core in detail: Logging

- Based on gabime/spdlog (header only)

- Log messages are formatted and sent to one or multiple sinks:
  - stdout sink available ⇒ log file
  - ZMQ publisher sink coming soon ⇒ log collector

```json
"loglevel": {"core": "INFO", "module": "DEBUG"}
```

```cpp
INFO("run started");
ERROR("component X crashed: " << msg);
WARNING("queue filling up!");
```

Log sink

```
[16:20] [core] [info] [Core::start()] run started
[16:20] [core] [error] [Core::bla()] component , X crashed: msg
[16:20] [module] [warning] [SomeModule::run()] queue filling up!
```
Core in detail: Operational Monitoring

- (optional) configurable POST (HTTP) or ZMQ publishing
- cURL wrapper whoshuu/cpr.git

```cpp
registerMetric<std::atomic<size_t>>(&m_eventmap_size, "EventMap-Size", LAST_VALUE);
```

Also available:
ACCUMULATE, AVERAGE, RATE

---

Diagram:
- Core
- Monitoring
- POST
- influxdb
- Grafana
- ZMQ
- Python broker
- redis
- Custom client

04/11/2019

Enrico Gamberini
**Control library**

- Written in Python

- **Process management** based on “Supervisor” [supervisord.org](http://supervisord.org)
  - Multi-host process supervision (spawning, status checking, automatic restart, etc.)

- Control channel implemented with ZMQ:
  - Commands, configuration and processes’ status polling

- Configuration based on JSON:
  - Enforced structure ⇒ JSON schema(s) + parser
  - Topology of data acquisition system (name, host, port, communication channels, etc.)
  - Module specific settings

- The Control library can be used:
  - in a command-line python script (“daqpy”)
  - in a Web GUI (developed by FASER)
  - in support tools (e.g. error recovery manager)
Demonstrator

- DAQling is shipped with an example application showcasing its main features
Deployment and build system

- DAQling is supported on CentOS 7
- Few Ansible playbooks for host set-up (tools and build environment)
  - Optional playbooks allow to add more tools/libraries
  - Debian playbook coming soon...
- The build system is based on CMake
  - Incremental build
  - Configurable options
- Docker images coming soon...
- New projects can fork from the daqling repo or from the daqling_top top-level repository
- Documentation available in repos

gitlab.cern.ch/ep-dt-di/daq/daqling_top

Enrico Gamberini
Projects

• FASER at CERN:
  ○ Main user at the moment. More details in next slide...
  ○ First application ⇒ useful suggestions, requests, and feedback
  ○ FASER will acquire its first data in 2021, after the LHC LS2.

• RD51 collaboration:
  ○ Laboratory setup for SRS readout + VMM3 ASIC
  ○ Raw UDP dump to file + decoder for monitoring/file writing
  ○ Possibility to scale up to test beam

• NA61/SHINE at CERN:
  ○ Use of significant part of DAQling for its DAQ upgrade
● Overview:
  ○ 1x Trigger Logic Board (~ 25 B fragments)
  ○ 9x Tracker readouts (>~ 250 B fragments)
  ○ 1x Digitizer (~ 15 kB fragments)
  ○ Trigger rate ~ 500 (peak 2k) Hz
  ○ Expected data on disk ~ 9 (peak 70) MB/s

● Successfully tested emulated full data flow on 2 servers

● Integration of detector readouts ongoing

● Automatic recovery manager and alerting under development
  ○ exploiting Python Control library
Web GUI

- Basic example developed by a FASER student:
  - Python web server based on Flask
  - Integration with op. monitoring display (Highcharts)
  - Configuration GUI based on JSON schemas

```json
{name: "frontendemulator01",
 host: "localhost",
 port: 5541,
 type: "FrontEndEmulator",
 loglevel: {
 core: "INFO",
 module: "DEBUG"
 },
 settings: {
 meanSize: 25,
 rmsSize: 0,
 fragmentId: 100000001,
 probMissingTrigger: 0,
 probMissingFragment: 0,
 probCorruptedFragment: 0,
 monitoringInterval: 1.5,
 triggerPort: 17001,
 daghost: "localhost",
 dagPort: 13001
 },
```

- Generalized version to be soon merged to DAQling

Courtesy of FASER
(Elham Amin Mansour)

Enrico Gamberini

04/11/2019
Summary

- DAQling provides a software ecosystem for distributed generic data acquisition systems
- C/C++ user code in Modules
- Configurable topology
- Integrated operational monitoring
- Python Control library
- Examples and documentation to help new developers
- Few projects at CERN already use DAQling

Please check the repository and documentation!
Contact us if interested (daqling-developers@cern.ch)