SoC-DAQ Interface

Andrei Kazarov, NRC Kurchatov Institute – Petersburg NPI

Giuseppe Avolio, CERN

Outline

General topic: “phase II DAQ-SoC communication”

Today: **focus on requirements**, use cases, examples

SoC – DAQ software Interface (SoC-DAQI) Requirements

- Prerequisites and constraints
- Use Cases and Examples
- Functional requirements
- System integration requirements
- Performance and Resource utilization requirements
- Progress and plans

ATLAS DAQ s/w and Run Control

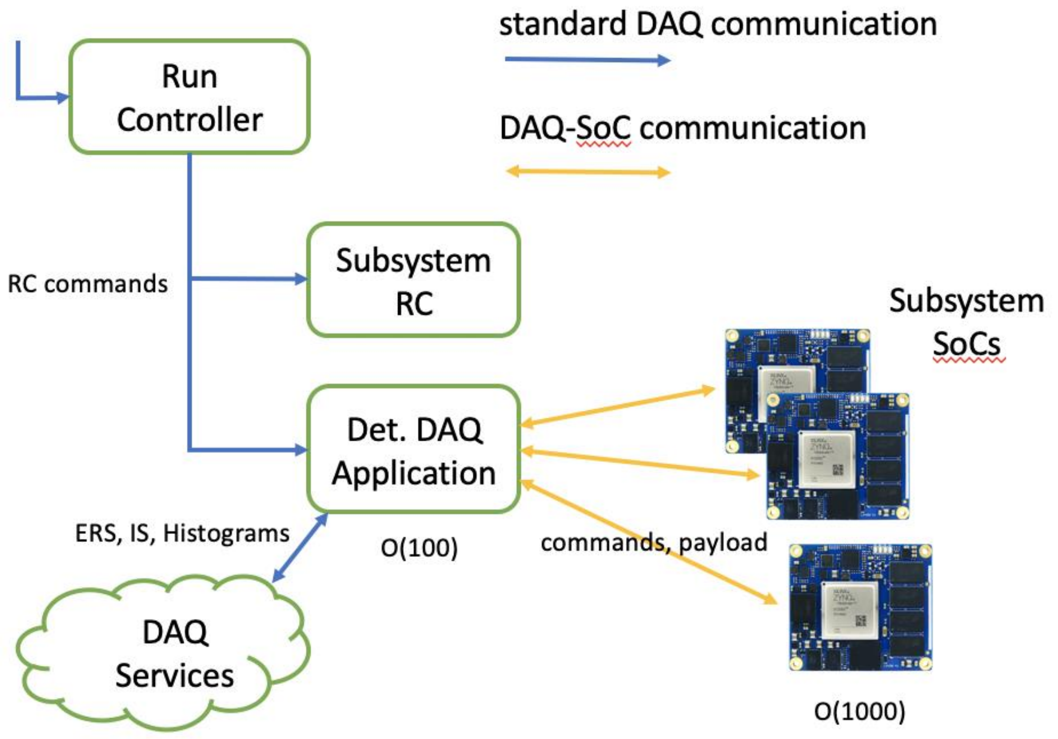
- All DAQ applications are organized in a Run Control (RC) tree (next slide) and share a FSM for synchronous command execution
- DAQ s/w provides a number of common services for DAQ applications
  - ERS (Error Reposting): report/subscribe of structured messages
  - IS (Information Sharing): publish/subscribe of structured information objects
  - Histogramming
  - …

Context

“It is anticipated that SoC will play a major role in configuration, control and monitoring of custom ATCA boards of the TDAQ Phase-II subsystems” (from SoC UR ATL-COM-DAQ-2019-144).

This implies **integration** of SoC **with TDAQ software infrastructure**, which provides common frameworks for all TDAQ subsystems participating in commissioning and ATLAS data-taking.

- Run Control for SoC: a number of System-on-CHIP devices are controlled by a custom subsystem DAQ or RC Application
- A protocol or an interface to communicate commands (and more generally, to **exchange information**) to a SoC system is to be implemented: SoC-DAQ Interface (**SoC-DAQI**)
- An DAQ application serves as a gateway from SoC eco-system to the DAQ services



accessing SoC over network in a similar manner as getting data from readout boards over VME bus by a application running on a PC or SBC

Examples

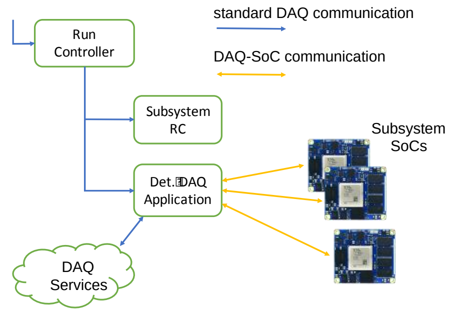
- RC application regularly gets a status from SoC and publishes it in DAQ IS, or in case of an reported error, issues an ERS message
- RC application receives Run Control transition command from its parent, distributes it (when necessary) to the controlled SoC systems, gets the responses and defines the overall RC state of the controlled domain
- RC application passes configuration data (e.g. a JSON string) to SoC
- RC application gets data from SoC and creates and publishes histograms in DAQ

the communication interface is designed to be very generic and may cover use cases not presently available in DAQ interfaces

Prerequisites and constraints

- It is assumed that SoC will host a standard, CERN/IT-certified Linux OS, like CentOS, and that a standard Ethernet-based network communication stack, including TCP/IP and HTTP libraries is available. A standard system gcc compiler and development kit is also available on SoC (or in a Linux cross-compilation environment).
- It is assumed that SoC is accessible from ATLAS Control Network (ATCN) and TDAQ Control networks and is addressable by its DNS name.
- It is assumed that a common middle layer s/w stack (like LCG s/w, providing widely used libraries like Boost Intel TBB, ROOT etc) and standard TDAQ s/w **may be not supported** by either SOC s/w or h/w platform
- Implementation of SoC-DAQI is foreseen to be lightweight and on the SoC (server) side to be independent from TDAQ s/w and to rely only on s/w packages available in standard repositories for SoC OS, thus **not imposing any additional requirements or restrictions** on the TDAQ s/w design coming from the SoC architecture.
- Implementation should be **stable and supported for decades** of the experiment lifetime

Use Cases



- A customized DAQ application serves as a bridge between DAQ data-taking services and a number of subsystem SoCs. It uses SoC-DAQI to send **commands** to applications running on SoCs, which process commands and return **results** to the DAQ application

Remark:

SoC – DAQ API does not define (or re-define) the specific interfaces to DAQ services, like access to configuration, histograms, archives etc. A generic command interface is offered instead, allowing subsystem DAQ applications to implement any kind of DAQ functionality with SoC, which may be even not implemented or foreseen in standard DAQ APIs. The closest implementation available in DAQ APIs are RC User Commands.

SoC-DAQI **commands** can be seen more generally like messages. It consists of a command name and a number of named parameters, and an arbitrary **payload**. A result of the command execution includes status code (or an exception) and also a returned payload. For example, using JSON in Payload gives users a possibility to pass around any kind of data.

Functional requirements

- SoC–DAQI shall implement the possibility to establish a client-server communication between ATLAS data taking control and SoC boards, i.e. to make possible for a client to send commands to a server application running on SoC.
  - **client**: typically a DAQ/RunControl application controlling some part of a subsystem
  - **server**: uses the server-side SoC–DAQI API and is responsible for performing corresponding actions on SoC side, specific to the subsystem, and returning a response to the client
- SoC–DAQI shall implement a possibility to execute arbitrary user code (C/C++) upon receiving a command.
- SoC–DAQI shall provide a possibility to send commands in asynchronous or synchronous manner.
- SoC–DAQI shall support for communicating messages to multiple server applications running in parallel on a SoC, and to multiple SoC systems from the same client. An addressing scheme or a kind of naming convention must be implemented.

Functional requirements (from SoC to DAQ)

- SoC–DAQI shall implement a possibility for a SoC application to report a status of execution (including an exception) of a command.
- It shall be possible for a DAQ application to be asynchronously notified about a change of a simple data variable on SoC, like a counter. Typically this means a possibility to subscribe to a data source by name, providing a callback function.

System integration requirements

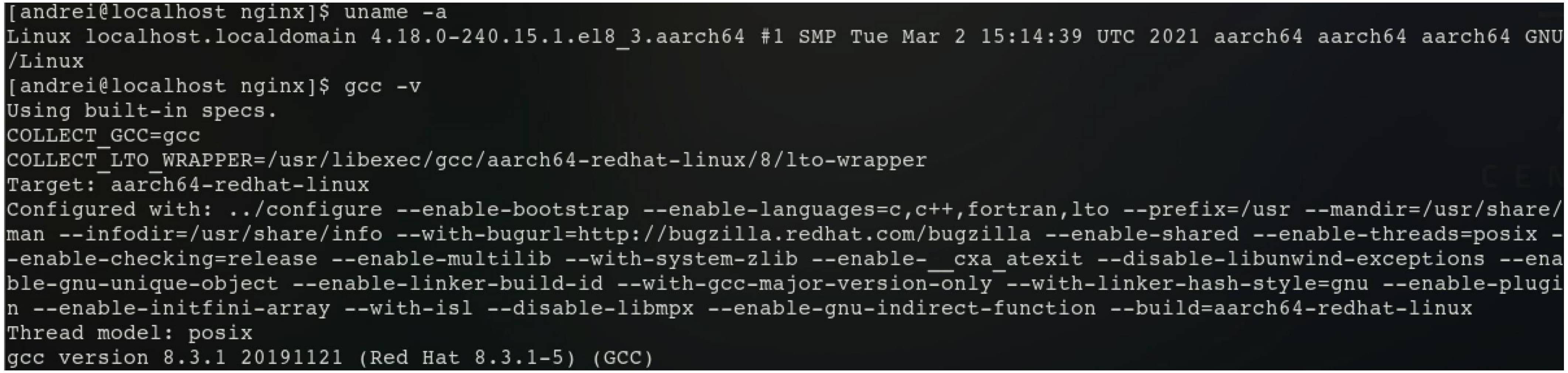
- SoC–RCI shall provide client and server side libraries, which is possible to use either from an application or from a plugin in some framework.
- SoC–RCI server side libraries and utilities shall be supported on variety of SoC s/w and h/w.
- The client libraries must be provided for the platforms supported by TDAQ s/w releases and use only packages provided TDAQ s/w stack.
- Server-side library shall be available in C/C++ language. Client-side library shall be implemented in languages supported by the DAQ framework, namely in C/C++, python and java.

Performance and scalability requirements

- The protocol implementation must support O(1000) of SoC servers
- The overhead coming from command handling shall be negligible with respect to typical latencies in DAQ Run Control
- Use of SoC resources (like CPU and RAM) by SoC–DAQI layer on the server shall not impact performance of SoC.

Progress

- Having the requirements finalized, the next steps are:
  - technologies evaluation
    - HTTP ← we are here, looks flexible and promising
    - gRPC ?
    - …
  - a toy example
  - a prototype implementation (due to Oct 2021)
- Development environment for prototyping and validation:
  - Centos8 ARM running in QEMU on CC7 x86_64 host
  - can compile, run and connect from the host



```
[andrei@localhost nginx]$ uname -a
Linux localhost.localdomain 4.18.0-240.15.1.el8_3.aarch64 #1 SMP Tue Mar 2 15:14:39 UTC 2021 aarch64 aarch64 aarch64 GNU
/Linux
[andrei@localhost nginx]$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/aarch64-redhat-linux/8/lto-wrapper
Target: aarch64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,fortran,lto --prefix=/usr --mandir=/usr/share/
man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-threads=posix -
-enable-checking=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --ena
ble-gnu-unique-object --enable-linker-build-id --with-gcc-major-version-only --with-linker-hash-style=gnu --enable-plugi
n --enable-initfini-array --with-isl --disable-libmpx --enable-gnu-indirect-function --build=aarch64-redhat-linux
Thread model: posix
gcc version 8.3.1 20191121 (Red Hat 8.3.1-5) (GCC)
```

Example API: client side, sync and async cases

```
using Payload = std::vector<char> ;
using Data = std::tuple<int, Payload> ;

class AsyncHandler {
        Data getData() ; // blocks until Data returned from the server
}

class Sender {
    Sender(const std::string& host) ;

    Data
    SendCommandSync (     const std::string& endpoint,
                          const std::map<std::string, std::string>& parameters,
                          const Payload& data_in) ;

    std::shared_ptr<AsyncHandler>
    SendCommandAsync (    const std::string& endpoint,
                          const std::map<std::string, std::string>& parameters,\
                          const Payload& data_in) ;
}
```

Example API: server side, a user function

```
const uint8_t response[] = "HELLO FROM SOC\n" ;

std::tuple<int, std::vector<uint8_t>>
user_function_cpp (     const std::string& message,
                        const std::map<std::string, std::string> parameters,
                        const std::vector<uint8_t> data_in)
{
std::tuple<int, std::vector<uint8_t>> ret { 0, { std::begin(response), std::end(response)
}} ;

auto seconds = parameters.find("value") ;
if ( message == "sleep" && seconds != parameters.end() ) {
        sleep(std::stoi(seconds->second)) ;
}

return ret ;
}
```

the End