SoC to DAQ communication library
status of the prototype

ATLAS TDAQ phase-II upgrade project

Andrei Kazarov

University of Johannesburg, SA

History of the project

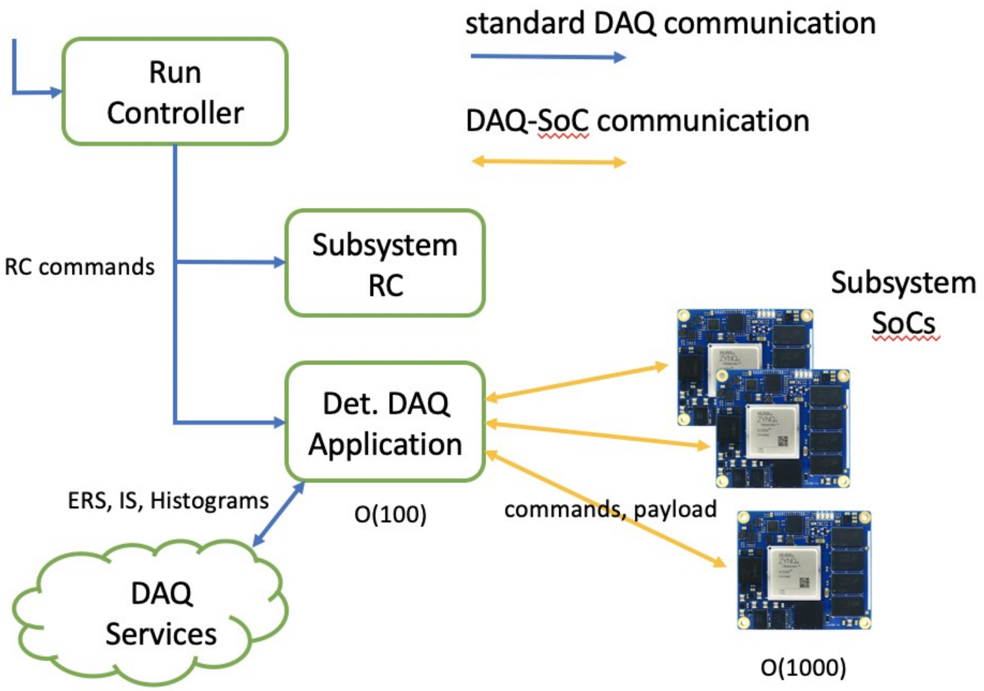
- Sep 2020 - TDAQ week: User Requirement (UR) Document status update
- May 2021 - UR doc presented on TDAQ Phase-II TC meeting
- Jun 2021 - User Requirements presented on 2nd SoC Workshop at CERN
- Oct 2021 - UR document finalized
- Feb 2022 - UR document approved and published in EDMS https://edms.cern.ch/document/2437729/1
- Nov 2022 - The prototype developed and presented at the SoC interest group meeting
- May 2023 - Prototype specification (PS) document released
- Jun 2023 - PS review board
- Aug 2023 - PS doc finalized and published in EDMS https://edms.cern.ch/document/2909134/1

- Now: testing of the prototype on a real SoC

- Next: the Design document: Nov 2023

Use cases overview and high-level design

- An implementation of a protocol to communicate commands (and more generally, to **exchange information**) to a SoC system: DAQ-to-SoC Interface (DAQ2SoC)
- A DAQ application serves as a gateway between DAQ services and SoC eco-system
  - replaces a “monolitic” DAQ RC application on every SoC by one DAQ RC application running on a DAQ node controlling **a number of** SoC systems in (possibly) an isolated network and running (possibly) a non-standard OS



Examples:

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

Prototype idea: HTTP + nginx server

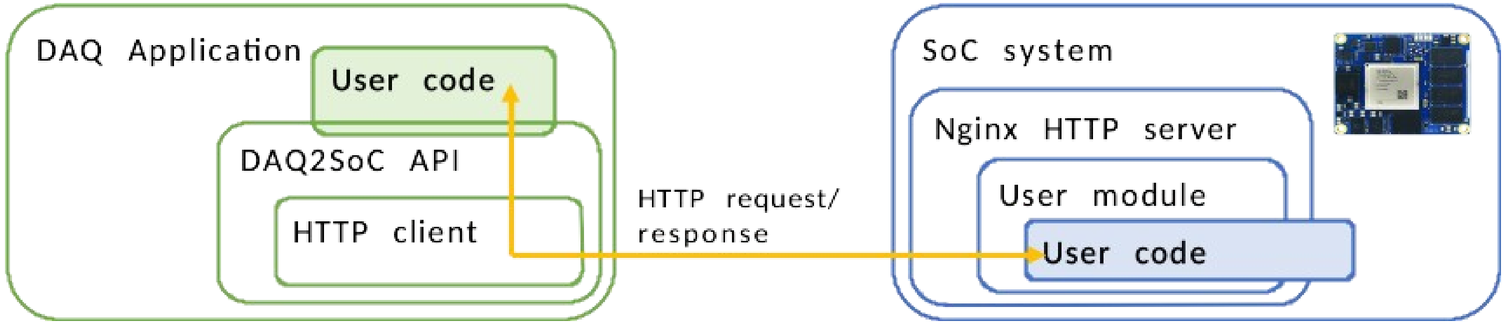
- Goal: provide a simple, portable, open communication layer for developing a DAQ application distributed across x86_64 and aarch64 SoCs: not a “replacement” of the DAQ s/w but an **extension**
- No dependencies on TDAQ/LCG s/w
    - allows to avoid issues related to long-term evolution of SoC s/w, h/w, OS
    - no constraints on DAQ or external s/w coming from ARM
    - allows isolation of SoC systems in a private network
- use **HTTP**(S) as a transport layer, wrapping user requests into standard POST and GET requests
    - any payload can be passed as part of a request and returned back
    - easy to access (standard reverse-proxy) SoC in isolated network
- use of **nginx** http server (pre-built binary) as the server-side application (framework), offloading to it all networking, connection, security, threading functionality
    - a de-facto industry standard, lightweight and performant web server
    - no process management, the nginx “mother” process is started by system and always running, waiting for HTTP requests to spawn worker processes
    - no HTTP details exposed to the user level (user-oriented C++ API)
- client-side: any HTTP client (a helper library available in C++), e.g. JavaScript in web page (payload can be serialized into JSON)

Package content

- Prototype is in gitlab
  - https://gitlab.cern.ch/akazarov/daq2soc
  - https://daq2soc.docs.cern.ch/index.html
- client: C++ API files: header file, implementation, example and a Makefile to compile it
- server:
  - nginx-module: sources and Makefile for compiling a user code into a dynamic daq2soc library loaded by nginx at runtime
  - nginx-server: precompiled binaries for aarch64 and x86_64 nginx http server and dynamic module
    - very few runtime dependencies, runs on a variety of Linux distributions
- common: a single-header file for JSON <-> C++ conversions: used in both client and server for parsing the payload
- quemu: how to run aarch64 CC8/9 linux on x86_64 host and test the binaries

client
common/cpp
doc
quemu
server
.gitlab-ci.yml
README.md

Prototype server implementation: nginx module



- SoC user code is executed in a nginx request handler context (in dynamically loaded module)
  - loaded from a .so library compiled with a simple Makefile: user implements a function of a class with a predefined signature
- Minimal runtime and OS dependencies (libpthread, libcrypto, libpcre, libz)
- The distribution: only 3 binary files and one config file
- User code is a single C++ file with one class and two or three functions to implement
- No build-time dependencies (extra to linux glibc headers)
- payload (can be passed as part of request and returned back): JSON string, handy conversion to and from standard C++ objects and containers (header only)

Server-side API

- User needs to implement UserData::daq_request_function and process the client request
  - you can have some payload passed in and can prepare a payload to return
  - UserData class holds user attributes persisting across requests
- a dedicated thread_function allows to implement code and data structures which run independently on the user requests

```
class UserData {
UserData () // Constructor. Initialize your attributes here.

std::tuple<int, std::vector<uint8_t>> // return value: status code and a payload
daq_request_function(
        const std::string &endpoint, // e.g. 'probe' or 'rc', the part of request URL after /tdaq/
        const std::map< std::string, std::string > parameters, // URL parameters like key=value
        const std::vector< uint8_t > data_in) ) // input payload (e.g. passed as -d from curl)

void
thread_function() // user function to execute a code in separate thread
}
```

Client-side API

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

class AsyncHandler {
public:
    Data getData() ; // blocks until Data is ready
}

class Sender {
public

    Sender(const std::string& host) ; // connect to a server on 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) ;
 }
```

API: C++ to JSON serialization (and back to C++)

- To convert C++ tuples into a JSON string and back, allows passing C++ objects around (JSON is payload in HTTP request)
- Example is a Histogram class (a tuple of a string and array https://daq2soc.docs.cern.ch/_histo_8hpp_source.html)

```
tdaq::soc::Sender mysender { std::string(host) } ;
std::map<std::string, std::string> parameters { {"partition", "ATLAS"} } ;
tdaq::soc::Data res = sender.SendCommandSync(“get_histogram”, parameters) ;
auto const& data = std::get<1>(res) ;
std::string myhist(data.begin(), data.end()) ; // payload to JSON string
Histogram<10> hist = tdaq::daq2soc::json2data<Histogram<10>>(myhist) ; // Histogram from JSON
```

```
example::Histogram<10> hist { "random historgram", {} } ;
// fill with random numbers
…
std::string json = tdaq::daq2soc::data2json(hist) ; // Histogram to JSON
std::vector<uint8_t> ret_data(std::begin(json), std::end(json)) ;
std::get<1>(ret) = ret_data ; // return to the requester
```

Performance tests on a real SoC

- few simple tests (so far) performed on a Xilinx Zync UScale SoC four-core 1.5GHz Cortex A53 ARM processor, nginx running 4 forked worker processes
- Histogram is returned in JSON format

type of test	time (ms)
get single counter, 1000 synchronous requests	275 (0.27ms per request)
get single counter, 200 asynchronous request	260
get histogram of 1024 float values	7
get histogram of 8192 float values	51

Summary

- Prototype is ready
- Review panel passed
- Implementation is being tested on a real SoC
- Design document being prepared for review