CCM/Process Communication

Brett Viren + Giovanna Lehmann Miotto

August 6, 2020

Topics

- We want all DAQ configuration information to be well structured, validated and described according to formal **schema**.
- Give user code **type-safe access** to configuration types via code generated from schema.
- Describe the overall DAQ and individual process **life cycle**.
- Provide user code a **uniform-interface** to configuration information so it may run in a variety of necessary contexts.

Schema

a **schema** is a **data structure**

which may be **interpreted**

as **describing** the **structure of data**

(*including that of schema!*)

Categories of schema interpretation

- `translate(schema)` $\rightarrow$ `schema`
- `codegen(schema, template)` $\rightarrow$ `code`
- `validate(schema, data)` $\rightarrow$ *true* | *false*

Describing schema

A schema names and describes **types**.

- Scalar type, based on some fundamental type.
  - number, string, Bool, enum
- Aggregate types.
  - "records" aggregating named fields (aka classes, structs)
  - "sequences" aggregating types (aka arrays, tuples)
- Validation constraints and qualifiers.
  - strings: regex pattern matching
  - numbers: range limits, memory sizes, formats (`float/int`)
  - enum: set of literal string values
  - array: min/max size, all-elements type, per-element types.

Example types for appfwk

Incomplete list of schema relevant for appfmk applications:

ident	a name spelled simply as per given regex
osversion	describe an OS
queuetype	enumerate possible "queue kinds"
queue	ident, capacity, type, kind
daqmodule	ident, module path
host	info about a specific computer host
executable	name a program
applicatoin	a program run on a host
controller	a type of program running other programs
daqprocess	info to initialize a DAQProcess
myprocess	aggregate of daqprocess and DAQModule level info
mymodule	configuration for a specific instance of a DAQModule subclass

Example schema description for appfwk

Schema can be expressed in many languages. Here a snippet in Jsonnet which describes a concrete schema in terms of an abstract schema in a functional manner.

```
function(schema) {

  // ...
  local queue = schema.record("Queue", fields= [
    schema.field("ident", ident, doc="queue name"),
    schema.field("capacity", schema.number(dtype="i4"), 2,
                 doc="Number of entries the queue can hold"),
    schema.field("kind", queuetype,
                 doc="The specific queue implementation to use"),
  ], doc = "Describes a queue connecting DAQ modules"),

  local queuelist = schema.sequence("Queue", queue),
  // ...

  types: [..., queue, queuelist, ...]
}
```

Code from schema

From schema we may generate **perfect** code[1] for:

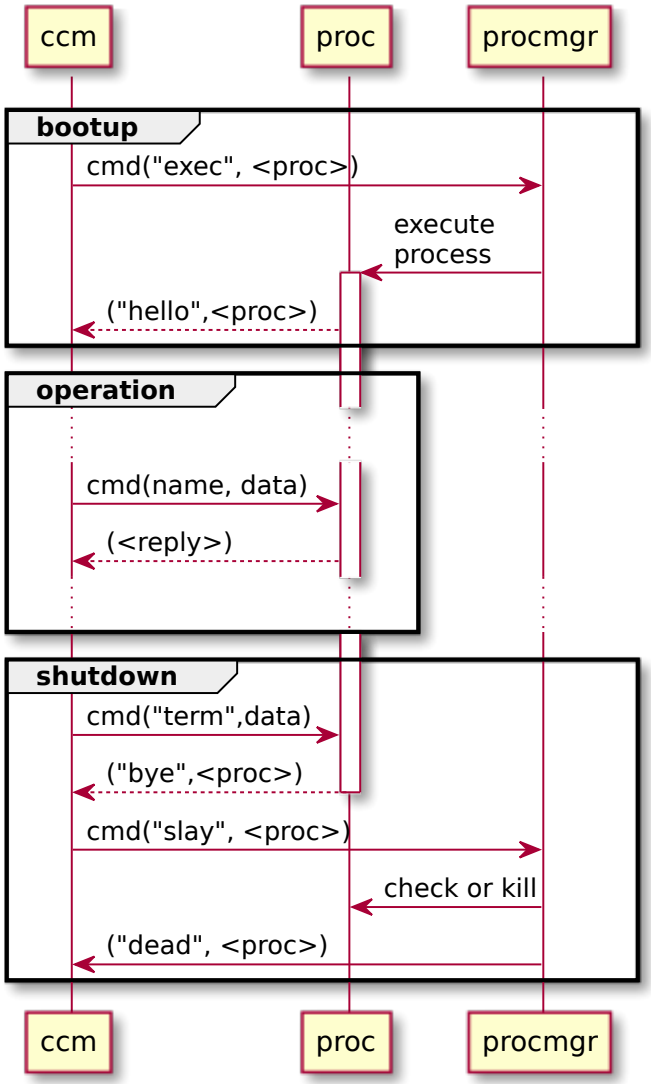
- C++ `struct` representing a configuration object
- Serialization methods between C++ `struct` and others
  - eg, `nlohmann::json` and even raw bytes
- C++ code to reduce boiler plate and simplify user code
- Reference documentation.
- And more....

Code generation commands will be built in to the build system.
Later, some examples how we propose to make use of generated code.

[1] Bugs may still exist of course but they become classes of bugs fixed at a point instead of individual bugs found and fixed in ad-hoc manner.

Approximate, high-level DAQ life cycle

Generally, CCM (RC) issues a command to something and gets a reply.



Process operation phases:

Bootup:

- CCM executes an application
  - (creates a process)
- CCM gets notification of success or failure
- Process idles waiting for more commands

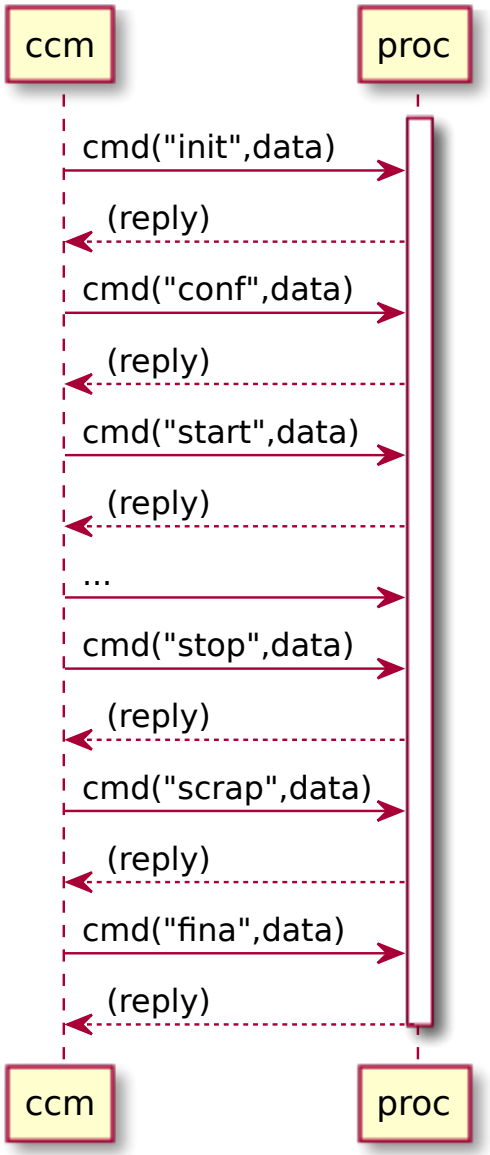
Operation:

- CCM sends commands to process
- Process reacts, replies and waits for more.
- Operational life cycle details next slide.

Shutdown:

- CCM commands process to terminate
- Process acknowledges
- CCM confirms or forces termination

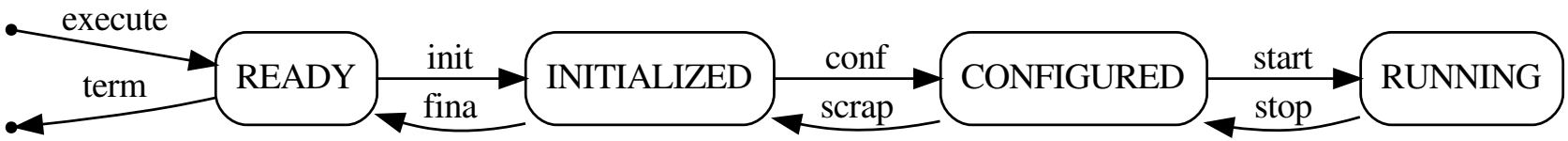
Approximate, DAQ process operational cycle



A process has been executed.

- CCM **pushes** to it a series of **commands**
  - as messages over the LAN.
- A command:
  - has a **type** from a fixed set,
  - carries **data** which follows
  - type- and proc-specific **schema**.
- Process **reacts** and sends a **reply**.
  - each type implies a command with a **clear semantic intent**.
  - command data used to **qualify intent**.

Command intent, ordering and states



`exec` A process is started and in a "blank state" and READY for commands.

`init` Process receives sufficient info to construct its structure (ie, its `Queues` and DAQ`Modules`) and becomes INITIALIZED.

`conf` Process receives and dispatches config info (eg to its constructed DAQ`Modules`) and is CONFIGURED.

`start` Process receives and dispatches run parameters (eg to its DAQ`Modules`) and begins nominal operational RUNNING.

`stop` Process undoes what was done in response to `start` and remains CONFIGURED.

`scrap` Process undoes what was done in response to `conf` and is INITIALIZED.

`fina` Process destroys its structure (ie, `Queues` and DAQ`Modules`) and is READY.

`term` Process exits. RIP.

Configuration methods for a variety of contexts

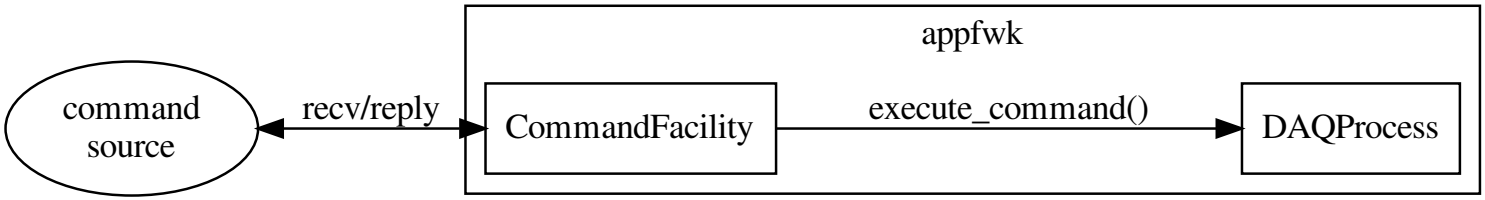
development Provide configuration directly as possible to the code being developed. Frequently change configuration schema and content. Run code in as isolated manner as possible. Apply interactively, deterministically or stochastically.

testing Provide relatively stable configuration at many scales (single application, many applications, many hosts) and across many scenarios (different processes, different parameters, different timing). Bake configuration into unit-, application- and system-tests.

operation Provide full system configuration via real CCM.

Interface for many command sources

appfwk supplies base class CommandFacility (CF) to "inject" commands to DAQProcess. User picks CF based on simple CLI info.



The variety of contexts need a variety of CF implementations, eg:

play file — CF reads commands and their *delay time* from file and forwards to DAQProcess with appropriate pacing.

interactive file — CF reads file with many commands each associated with a *command alias*. CF reads aliases from user, looks up command, calls DAQProcess

mock CCM — CF receives commands on socket and forwards to DAQProcess. A *mock CCM* process reads from file (eg as above) and produces command messages. This "mock" CF could become the interface to the real CCM.

Evolution of appfwk

- Make **initialization** distinct from **configuration**.
- Adjust **command callback function signatures**.
- Enact an internal **command dispatch protocol**.

appfwk evolution - init vs conf

Currently, appfwk will initialize the process by atomically constructing **and** configuring its Queue and DAQModule instances.

To match desired behavior small changes needed are:

- From main() change call of DAQModule::do_init(cfg) to DAQModule::init().
- Existing modules move code needing configuration from init() to their method registered as the *configure* command handler.
  - Provide generated config struct and *mixin* class to simplify this part of user code (example coming)
- Implement fina command handler at DAQProcess level.

appfwk evolution - command handler signatures

Current appfwk command callback signature:

```
void (Child::*f)(const std::vector<std::string>&));
```

Want a signature that:

- can represent all CCM command types including app-specific structure
- reduces ambiguity and minimizes need for interpretation by user-code
- compatible with type-safe schema serialization methods
- supports a command dispatch protocol

The "highest lowest common denominator":

```
void (Child::*f)(const nlohmann::json&));
```

Type-safe configuration `struct`

Discourage ad-hoc interpretation of `nlohmann::json` object in favor of conversion to generated configuration `struct`. Eg:

```
void MyDaqMod::init() {
  register_command("configure", &MyDaqMod::do_configure);
}
void MyDaqMod::do_configure(const nlohmann::json& jcfg) {
  auto cfg = jcfg.get<MyDaqModeCfg>();
  cout << "my parameter: " << cfg.myparameter << endl;
}
```

`MyDaqModCfg` and methods to create it with `json::get<>()` use generated code. That code **perfectly** interprets the input data or throws ERS exceptions.

User code need not worry!

Further DAQModule simplification and type safety

Encourage DAQModule implementation to inherit from a mixin class templated on the configuration `struct` type.
User code now only supplies:

```
void MyDaqMod::configure(MyDaqModeCfg cfg) {
  cout << "my parameter: " << cfg.myparameter << endl;
}
```

The mixin base class provides:

- a method which calls required `register_command()`.
- the command callback method which "retypes" `nlohmann::json` to MyDaqModCfg.
- and then calls the fully typed `configure(MyDaqModCfg)`.

This pattern can be generalized to allow fully-typed callbacks to all CCM commands.

Command dispatch protocol

Currently, DAQProcess::execute_command() sends same command data to each DAQModule. Want to:

- give only an appropriate portion of the command data to each given module's callback method.
- find a solution which avoids treating configuration commands as special cases.

The details t.b.e. with appfwk developers but basic idea is DAQProcess looks into command data.

Possible choice:

- Command data is object with keys holding DAQModule names
- DAQProcess dispatches sub-object values accordingly
- support commands with common data with special key representing "all"

Summary

- The way of organizing and distributing configuration data to the DAQ applications via the run control is taking shape;
- We are collaborating with John/Pengfei to introduce the support for configuration schemas and generated code in the build/release environment;
- The integration of the configuration handling into `appfwk` requires code changes that we will discuss in detail and help implementing for the next release;
- We propose to couple the configuration data distribution with the issuing of commands from the run control and are preparing a mockup to demonstrate the principle in the absence of the final run control;
- Work will continue in parallel on other aspects of the configuration system, such as data storage and archiving (e.g. associated to a run)