Jumbled thoughts on configuring multiprocess minidaqapp

Brett Viren

February 4, 2021

Boundary conditions

- Description of desired processes in form of a list of job names and their modules
- Target configuration needed for simple shell command line launch and command but with guesses about factoring for eventual CCM (RC, PM, etc).

Meta Configuration

Given list of desired processes:

- Understand common information
- Factor into “meta parameter set” MPS
- Describe MPS in schema
- Provide object generation code

Object generation:

- Given MPS object, make per-process command sets
- Generate config file for trivial process launcher

Conceptual model of CCM to target

Chicken-and-egg problem right now for CCM/config.

- Can not build the next “meta” levels without knowing what we actually need.
- People providing those levels need to know something about how they are configured.

Break the cycle by develop some initial straw model which both can work toward.

A portion of my mental model of CCM

$\mathcal{S}_R$	Schema to holistically describe configuration for a **type of run** R
$\mathcal{C}_r$	Object to describe configuration for a **specific run** r of type R
$f_{gen}(\mathcal{S}_R) \rightarrow \mathcal{C}_r$	generator of schema-valid configuration object (now scripts, later web gui)
$F_{rc}(\mathcal{C}_r) \rightarrow run$	*Run Control* produces running DAQ as a functional side effect, given configuration. It is composed of parts:
$f_{link}(\mathcal{C}_r) \rightarrow \{\mathcal{L}\}$	produce set of URLs of process's REST *Command Facility* instances
$f_{proc}(\mathcal{C}_r) \rightarrow \{\mathcal{P}\}$	produce set of per-process configuration objects (commands) with info needed to start the processes
$F_{pm}(\mathcal{P}) \rightarrow proc$	A *Process Manager* starts a process
$f_{init}(\mathcal{C}_r) \rightarrow \{c_{init}\}$	A filter function produces a set of **init** command objects for processes.
$F_{rc}(\{\mathcal{L}\}, \{c_{init}\}) \rightarrow S_{initialized}$	RC sends **init** commands bringing DAQ to *initialized* state.
$f_{conf}(\mathcal{C}_r) \rightarrow \{c_{conf}\}$	A filter function produces a set of **conf** command objects for processes.
$F_{rc}(\{\mathcal{L}\}, \{c_{conf}\}) \rightarrow S_{configured}$	RC sends **conf** commands bringing DAQ to *configured* state.
etc	other command types, and possibly some new ones to enact a new command for "light weight" configuration.

Near term development

Trivially mock up RC/PM duties in order to provide straw use case for factoring $\mathcal{S}_R$ and providing $f_{gen}()$.
Mock will look something like this:

- Based on description from minidaqapp experts, define schema for MPS parts of $\mathcal{S}_R$.
- Use `moo.otypes` Python to generate MPS objects.
- Adapter to use MPS object and reuse per-module `moo.otypes` code to produce per process command objects.
- Implement $f_{link}()$ by extracting from MPS.
- Implement $f_{proc}()$ and F_{pm} by generating foreman config file with moo template and $\mathcal{C}_r$ and execute with simple shoreman bash script.
- Implement sending of commands similar to now (`ddrepl` style) but which operate on a set of **init** or set of **conf** instead of one at a time.

Foreman/shoreman

The `shoreman` Bash script literally "just" runs command lines and waits for them to finish, giving useful log capture and clean `Ctrl-C` death. Its config file is in Foreman format:

```
# minidaq.proc file generated by moo
gencfg gencfg.py -o config.json -i meta.jsonnet -s schema.jsonnet
proc1  daq_application --commandFacility rest://hostname:8081
proc2  daq_application --commandFacility rest://hostname:8082
```

Run like:

```
$ shoreman minidaq.proc
$ mockrc.py -c config.json init
$ mockrc.py -c config.json conf
```

`shoreman` super dumb process manager

`gencfg.py` $f_{gen}()$

`mockrc.py` super dumb RC's $f_{cmd}()$