Python configuration generation simplification

Philip Rodrigues

University of Oxford

October 6, 2021

The problem

- Python configuration generation is too complicated and difficult to use
- Eg, DS developers unable to add TP readout to minidaqapp configuration
- Specifics:
  - Repetition of data/names, that have to be kept in sync. Eg, adding a module requires adding queues, connecting each module endpoint to the queue
  - Lots of non-local editing: eg, add a module, then add it to init, conf, start, stop etc. Taxes working memory; tedious; hard to get right
- Underlying problems (in my opinion):
  - Config is specified as "how the underlying app framework implementation works", rather than "what the user wants to do"
  - Information that could be inferred by code has to be specified by the user (eg, network connections)

Proposed approach

- Minimize the amount of information the user has to provide; have code infer as much as possible
- In this case, user specifies list of modules, their configuration objects (for conf), and connections between them
  - Code infers the queues needed, the necessary start/stop/scrap commands and their order
- Same at the interprocess level: individual applications specified as modules/connections; "top-level" specifies connections between app endpoints, and code infers necessary hostnames/ports/Q2N-N2Q pairs
- Secondary advantage of this approach is that appfwk and other "infrastructure" can be changed without changing all configurations: just change the common utility code that, eg, infers Q2N-N2Q pairs

Scope and implementation

- Change configuration generation without changing anything in appfwk or nanorc; ie, json file schema is unchanged (for now)
- Not addressing any bigger picture things like configuration database, changes to json output, argument explosion, etc. Those are hopefully all orthogonal
- Intention (not really achieved yet): make steps from modules -> application -> system -> json clearly separated, and do validation at each step
- Started by just modifying some configs in the `trigger` package:
  https://github.com/DUNE-DAQ/trigger/tree/philiprodrigues/reduce-confgen-verbosity/python/trigger
- From the top down:
  - A DAQ `System` is built from applications and the connections between them
  - An application (`App` class) is specified by a `ModuleGraph` and the host on which it runs.
  - A `ModuleGraph` is specified by a list of modules, their configs and connections, and external "endpoints" for input and output
- Let's go through from the bottom up:

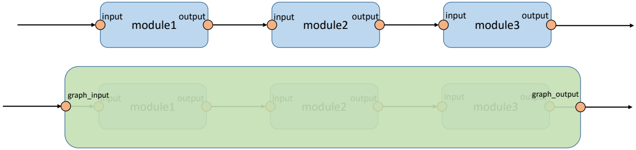
Modules

```python
# Load moo type for configuration
moo.otypes.load_types('trigger/triggerprimitivemaker.jsonnet')
import dunedaq.trigger.triggerprimitivemaker as tpm

import util
from util import Connection as Conn

tpm_module = util.Module(plugin="TriggerPrimitiveMaker",
                         conf=tpm.ConfParams(number_of_loops=-1,
                                             tpset_time_offset=0),
                         connections={"tpset_sink": Conn("ftpchm.tpset_source")})
```

- `connections` specifies that the `tpset_sink` `DAQSink` of this module should be connected to the `tpset_source` `DAQSource` of the module named "`ftpchm`"
- When we generate the full application configuration, tools in `util` will automatically create the necessary queue objects and settings to connect this DAQSink/DAQSource pair.

ModuleGraph 1

- `Modules` are grouped together in a `ModuleGraph`, which holds:
  1. dictionary mapping module names to `Module` objects
  2. dictionary of "endpoints" which are the "public" names for the `ModuleGraph`'s external inputs and outputs
- Endpoint concept allows other applications to make connections to this one without having to know about the internal details of modules and sink/source names
- (Intention is that eventually, can construct a `ModuleGraph` out of other `ModuleGraphs`, but not implemented yet)



ModuleGraph 2

Example:

```
modules = {}

modules["tpm"] = util.Module(plugin="TriggerPrimitiveMaker",
                             conf=tpm.ConfParams(number_of_loops=-1,
                                                 tpset_time_offset=0),
                             connections={"tpset_sink": Conn("ftpchm.tpset_source")})

modules["ftpchm"] = util.Module(plugin="FakeTPCreatorHeartbeatMaker",
                                # No outgoing connections specified here
                                conf=ftpchm.Conf(heartbeat_interval=50000))

the_modulegraph = ModuleGraph(modules)
# Create an outgoing public endpoint named "tpsets_out", which refers to the "tpset_sink" DAQSink in the "ftpchm" module
the_modulegraph.add_endpoint("tpsets_out", "ftpchm.tpset_sink", util.Direction.OUT)
```

Applications

- App class represents an instance of a daq_application running on a particular host
- Consists of a ModuleGraph and a host on which to run
- Collected in dictionary like modules:

```
apps = { "myapp": util.App(modulegraph=the_modulegraph, host="localhost") }

app_connections = {
    "myapp.tpsets_out": util.Publisher(msg_type="dunedaq::trigger::TPSet",
                                       msg_module_name="TPSetNQ",
                                       subscribers=["tpset_consumer1.tpsets_in",
                                                    "tpset_consumer2.tpsets_in"])
}
```

- Should probably make connections work like they do with modules

System

- The `System` class groups applications and their connections together in a single object:
  `the_system = util.System(apps, app_connections)`
- A `util.System` object contains all of the information needed to generate a full set of JSON files that can be read by nanorc

Generating JSON files

- To get from a `System` object to a full set of JSON files involves four steps:
  1. Add networking modules (ie, NetworkToQueue/QueueToNetwork) to applications
  2. For each application, create the python data structures for each DAQ command that the application will respond to
  3. Create the python data structures for each DAQ command that the system will respond to
  4. Convert the python data structures to JSON and dump to the appropriate files
- `make_apps_json(the_system, json_dir, verbose=False)` does all four steps in one go. For debugging/validation, can do each one individually:

```
app_command_datas = dict()

for app_name, app in the_system.apps.items():
    # Step 1
    add_network(app_name, the_system)
    # Step 2
    app_command_datas[app_name] = make_app_command_data(app)

# Step 3
system_command_datas=make_system_command_datas(the_system)

# Step 4
write_json_files(app_command_datas, system_command_datas, json_dir)
```

Next steps

- Look into providing other useful helper functions, eg `register_data_provider()` to indicate that a module provides fragments for a given GeoID
- Come up with a better namespace, and work out how to deploy
- Try to convert minidaqapp config to this scheme (started on this locally)