OpenRVDAS Sikuliaq Notes

David Pablo Cohn 2018-04-09

VM Details

CentOS 7 machine, built using CentOS-7-x86_64-DVD-1611, followed by a run of the script `openrvdas/utils/build_openrvdas_centos7.sh`.

Machine name: `openrvdas`
root password: `CentOS`
user: rvdas; password: `rvdas`
mysql root password: `rvdas`

Project base is `/opt/openrvdas`, owned by user `rvdas`.

Code repository: https://github.com/davidpablocohn/openrvdas; there is a sikuliaq branch, containing ship-specific logger definitions and a directory `openrvdas/skq` of sample data and cruise configurations.

Additional documentation: http://tinyurl.com/openrvdas-docs

Quick and Easy Run of the Code

Just to get started, you can mess around with the listen.py script, which you can think of as a specialized form of the 'cat' command: you specify inputs (readers), transformations and outputs (writers):

```
# Read a text file, parse it as NMEA data and write to stdout.
# Note 1: Sikuliaq date/time format differs from openrvdas default,
#         so need to override
# Note 2: A single "-" in place of a filename for --write_file means
#         write to stdout
logger/listener/listen.py \
    --time_format "%Y-%m-%dT%H:%M:%S.%fZ" \
    --file skq/sikuliaq.data \
    --transform_parse_nmea \
    --write_file -

# As before, but now also write to local SQL database
logger/listener/listen.py \
    --time_format "%Y-%m-%dT%H:%M:%S.%fZ" \
    --file skq/sikuliaq.data \
    --transform_parse_nmea \
    --database_password rvdas \
    --write_database rvdas@localhost:data \
    --write_file -
```

Try `logger/listener/listen.py --help`, and look at the documentation at Running OpenRVDAS Loggers for more information.

Different ways of using OpenRVDAS code

There are five(!) obvious ways of using the code, listed below in order of low-level to high-level. You will most likely want to use the latter ones; I'm listing the lower-level ways for pedagogical reasons.

1. Write logger using modules

It's fairly straightforward to manually instantiate and connect components to build a dedicated Python logger:

```
#!/usr/bin/env python3

from logger.readers.network_reader import NetworkReader
from logger.transforms.parse_nmea_transform import ParseNMEATransform
from logger.writers.database_writer import DatabaseWriter
from logger.writers.text_file_writer import TextFileWriter
```

```
network = ':54122'
time_format = '%Y-%m-%dT%H:%M:%S.%fZ'

reader = NetworkReader(network=network)
parser = ParseNMEATransform(time_format=time_format)

# Use defaults from database/settings.py
database_writer = DatabaseWriter()

# With no args, will write to stdout
text_file_writer = TextFileWriter()

while True:
  record = reader.read()
  if record:
    text_file_writer.write(record)

    # Database writer wants parsed records
    parsed_record = parser.transform(record)
    if parsed_record:
      database_writer.write(parsed_record)
```

(Note: if you run the above script, you can feed it data by running listen.py in another window: `logger/listener/listen.py --file skq/sikuliaq.data --write_network :54122`)

2. Use the listen.py script with command line arguments

The `listen.py` script incorporates the most common Readers, Transforms and Writers, providing much of the functionality that one might want in a logger straight from the command line. For example, the invocation:

```
# Writers operate in parallel; the --write_file option with "-" for
# an argument writes to stdout
logger/listener/listen.py \
    --network :54122 \
    --time_format '%Y-%m-%dT%H:%M:%S.%fZ' \
    --transform_parse_nmea \
    --database_password rvdas \
    --write_database rvdas@localhost:data \
    --write_file - \
    -v
```

implements the same dataflow as the previous Python code.

Note that the listen.py script has half a billion command line options and (as documented in `--help`) applies them in order. So, for example

```
logger/listener/listen.py \
    --file skq/sikuliaq.data \
    --transform_slice 2: \
    --transform_timestamp \
    --transform_prefix skq_data \
    --write_file -
```

will strip off the first two fields of each record, then prefix a timestamp to it, then the string '`skq_data`', while the invocation

```
logger/listener/listen.py \
    --file skq/sikuliaq.data \
    --transform_timestamp \
    --transform_prefix skq_data \
    --transform_slice 2: \
    --write_file -
```

will add the timestamp and '`skq_data`', prefix, then strip off those two newly-added columns.

3. Use the listen.py script with a JSON configuration file

The `listen.py` script can also read from a pre-assembled configuration file:

```
logger/listener/listen.py --config_file skq/gyro_1.json
```

where `gyro_1.json` consists of the JSON definition

```
{
    "name": "gyro_1->db",
    "readers": {
        "class": "NetworkReader",
        "kwargs": { "network": ":54122" }
    },
    "transforms": {
        "class": "ParseNMEATransform",
        "kwargs": { "time_format": "%Y-%m-%dT%H:%M:%S.%fZ" }
    },
    "writers": [
        { "class": "TextFileWriter" },
        {
          "class": "DatabaseWriter",
          "kwargs": {
            "user": "rvdas",
            "host": "localhost",
```

```
                "database": "data",
                "password": "rvdas"
            }
        }
    ]
}
```

(Run `listen.py --help` for a full list of the options that the script takes.)

4. Use the run_loggers.py script to run multiple loggers

The gyro_1.json file above encoded the configuration for one logger. We can also define a "cruise configuration" file which encodes configurations for multiple loggers and groups them into "modes": one set of configurations, e.g. when the ship is underway, another when it's in port.

The file skq/skq_cruise.json defines four modes:

- off - no loggers running
- file - all loggers running and saving data to their separate logfiles
- db - all loggers saving data to the SQL database
- file/db - all loggers saving to both logfiles and database

The cruise configuration can be read and used by the run_loggers.py script:

```
logger/utils/run_loggers.py \
    --config skq/skq_cruise.json \
    --mode file/db
```

It will start one subprocess for each logger configuration specified in the "file/db" mode, monitor it for health, and restart it if it dies for any reason.

(If started with the `--interactive` flag, `run_loggers.py` will listen to standard input and, if you enter the name of another mode, will kill off the running loggers and start loggers appropriate to the new mode.)

5. Use the Django web interface to interactively manage loggers

The sample machine has been configured by the `build_openrvdas_centos7.sh` script to run the NGINX web server as a service, with uWSGI as a gateway to Django. Both are relatively lightweight when not in use and have been configured to start on boot.

To make full use of the interface, however, you will need to manually start the OpenRVDAS servers. They have been registered as a service, so you can start them manually by running

```
service openrvdas start
```

as root. You can configure the servers to start on subsequent on boot up by running

```
systemctl enable openrvdas.service
```

This automatic startup can be disabled by, unsurprisingly, running

```
systemctl disable openrvdas.service
```

You'll still need to shut the servers down manually if they're running, with the command

```
service openrvdas stop
```

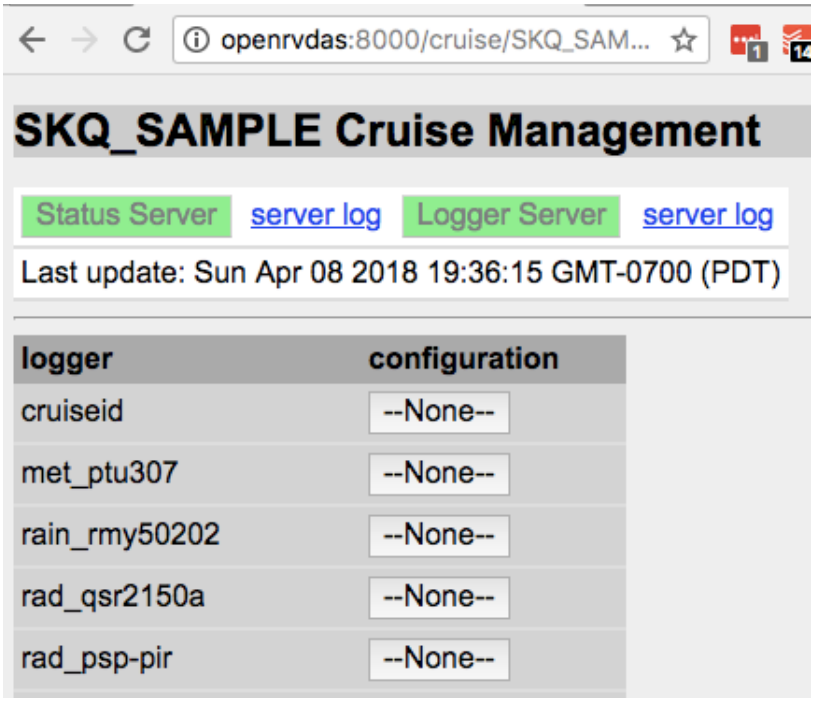
(Note that the servers may be run manually in a terminal window from the project root directory by `running gui/run_servers.py`)

You'll need to register the openrvdas server VM on your network, or at least on your local machine, so that your browser knows how to get to it. Add the line

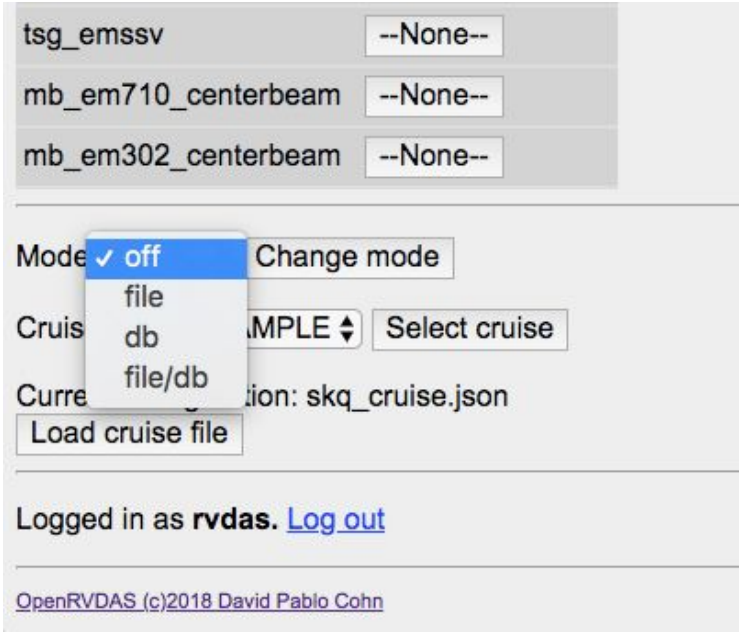
```
XX.XX.XX.XX     openrvdas
```

to your local `/etc/hosts` file, where `XX.XX.XX.XX` is the server's ip address (displayed by running '`ip addr`' or an equivalent command in a terminal on the openrvdas server).

You should now be able to open a browser window to `http://openrvdas:8000` and see the web interface. Log into the interface as user `rvdas` (password `rvdas`), and the screen should look like this:



You can select any of the available configurations for a logger by clicking its configuration button. At the bottom of the page you can also select modes for the current cruise as well as switch between loaded cruise configurations:



The supplied VM has three cruise configurations loaded:

- SKQ_SAMPLE - the configuration described in the previous section

- SKQ_6224 - a version of the previous configuration in which all loggers share UDP port 6224 to simplify testing (each logger configuration in this case includes a RegexFilterTransform to ensure that it only processes and stores UDP records matching the logger's name).[1]
- NBP1700 - minimal setup for a fictitious cruise involving serial port loggers. Running these loggers will require also setting up simulated serial ports, **as described below.**

Running the Sample Cruise: NBP 1700

First, start gui/run_servers.py, as described above in "Use the Django web interface to interactively manage loggers". Then open a browser window to http://openrvdas:8000 and select cruise NBP1700 from the pulldown menu near the bottom of the page and hit "Select cruise".

The NBP1700 loggers are configured to read from serial ports, but for instructional purposes, they are configured to read from *simulated* serial ports (see the cruise configuration file in NBP1700_cruise.json if you'd like to convince yourself what this looks like).

The set of simulated serial ports can be created by a script in `logger/utils`:

```
logger/utils/simulate_serial.py --config test/serial_sim.json
```

It reads a configuration file that tells it which ports to create and from which files to feed them. E.g. the definition

```
"gyr1": {"port": "/tmp/tty_gyr1",
         "logfile": "test/nmea/NBP1700/gyr1/raw/NBP1700_gyr1"
        },
```

tells the script to create /tmp/tty_gyr1 and feed it data from the named log file, using the saved timestamps to simulate the rate at which the data appear.

Once you have `run_servers.py` and `simulate_serial.py` running, you should be able to select a cruise mode on the browser page, and start actual loggers running and reading the simulated ports. You can also select an individual logger's configuration and change/enable/disable it from the resulting dialogue page.

To observe the data being written, open another terminal window on the server. Data written to file can be monitored via

[1] To exercise this configuration, start the loggers running, then in a separate window run `logger/listener/listen.py --file skq/sikuliaq.data --write_network :6224 -v`

```
tail -f /tmp/log/NBP1700/*/raw/*
```

Data written to the database can be observed via

> mysql -u rvdas -p
> Enter password: [rvdas]
>
> MariaDB [(none)]> use data
>
> **Database changed**
> MariaDB [data]> select * from data;

Data written to the network via UDP may be monitored via

```
logger/listener/listen.py --network :6224 --write_file -
```

(all loggers in this configuration write to port 6224)

Displaying Logged Data

Finally, there is a rudimentary display widget that demonstrates the websocket serving infrastructure. The display functionality is fed from data stored in the database and served by the openrvdas servers, so it will only update while the servers are running and data are being written to the database.

Open browser to http://openrvdas:8000/widget/S330Pitch,S330Roll
and you should see a page that looks something like this:

If the s330 logger (from sample cruise NBP1700) is writing to database, the numbers in the widget should be updating. The widget will display any comma-separated list of fields stored in the database. Field definitions for each sensor are in local/sensor/*.json, and you can check the database using the mysql command above to see what's getting written.

The current widget is just meant to be pedagogical in terms of demonstrating how display widgets can request live data from the openrvdas servers.