Data Queue design

NOAO

<2014-08-01 Fri>

Contents

1 Overview

Software task to process a data-file list as a queue. Focus on resiliency.

NOTE: This file includes some org mode src blocks. Some are for using graphviz to generate figure, some to use SH to run programs described here to get their usage documentation. To get good export, emacs must be configured appropriately.

2 Requirements

2.1 General

1. System shall not loose track of any data pushed into it, even in the case of hardware or software faults that terminate the process. Preserve state if queue process crashes.
2. Required actions shall be performed via a single command-line request.
3. Queue shall reconnect to data source(s) if network goes down and up.

2.2 Functional

- Log all errors and warnings from queue processing to /var/log
- Start data queue process
- Stop data queue process
- Push record from TCP socket onto queue
- Pop record from queue and run action against it
- Configure action to be run upon pop for a specific queue
- List everything in Active Queue and Inactive Queue (the "stash")
- Move selected items from active queue to inactive queue
- Delete selected items from active or inactive queue
- Move selected items in active queue to the tail (next to pop)
- Detect attempt to push a duplicate record onto the queue and ignore duplicate. (do not add to queue or process it, emit warning)

2.3 Commands

1. Disable processing from queue (pop and action)
2. Enable processing from queue (pop and action)
3. Delete <queue> <first-id> <last-id>
4. Deactivate <first-id> <last-id>
5. Activate <first-id> <last-id>
6. List content of queue
7. Advance <first-id> <last-id> (to tail of active queue)

<first-id>, <last-id> Are the checksums (ids) of the start and end of a range of items in a queue.

2.4 Support input format

Read records from TCP that contain **typed** fields:

- full file path [STRING]
- checksum [STRING] (assumed to be a unique identifier for a record)
- size [INTEGER]

2.5 Support these Actions (processing of a record)

- network move
- disk storage
- archive ingest

3 Anti-requirements

These are NOT requirements. We explicitly considered but rejected them.

- priority queue; not necessary and more computationaly expensive. Not supported by Redis. (but python does have built in heapq)

- keep multiple records as distinct if they have same checksum but records are not identical; "never" happens per analysis of postgres DB
- specify records by position in queue; position changes, not supported by Redis
- Use log4j format configuration for logging; native python doesn't support it, developement more pythonic to use native, if its a bother to have Puppet deal with two kinds of configuration files, we will write a converter

4 As-Built Design *<2014-07-25 Fri>*

4.1 Purpose

The purpose of this document is to describe a software component (data-queue) in enough detail to be helpful for a developer who wishes to modify it.

The purpose of the data-queue is to queue data while waiting for another service to process it. The types of processing include:

- transfer over the network to another computer or site
- register data with the mass store
- ingest the data into the archive

4.2 Overview

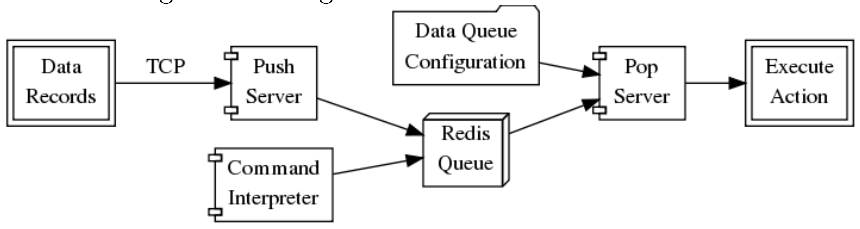
The deployed system contains 4 components:

1. dataq_push_svc.py Read records of TCP socket. Push record onto queue
2. dataq_pop_svc.py Pop records from queue. Run action against record.
3. dataq_cli.py Provides commands for diagnostics and manipulating queue
4. redis: `http://redis.io/` (key-value store)

There is also a test program (test_feeder.py) used to simulate the expect input via TCP.

No init system service has been created yet. Such a script needs to start both push and pop svc programs. Maybe it should start redis also (or that might be started elsewhere).

Here is a diagram showing the data flow.



NB: It would be a good idea to use python asyncio to allow dataq_(push,pop)_svc.py to be combined into a single executable. Essentially, we would use an event loop to allow push and pop to happen "in parallel". Key features are documented but not available until Python 3.4.2 (current available is 3.4.1).

4.3 Components

4.3.1 dataq_push_svc.py

Read text from given (or defaulted) host/port. Push the ID (checksum) from each record onto active queue and store the remaining part of the record (using hmset) using the ID as a key.

We assume that if the ID of two records is the same, the data is too. In that case, we throw one of them away. The same data can still go through the queue multiple times, but the queue cannot contain duplicates. For an action on the same record to be performed twice, would have to be processed once before it shows up again at the head of the queue.

```
.  ~/PYTHON_ENV/bin/activate
~/sandbox/data-q/dataq_push_svc.py --help

usage: dataq_push_svc.py [-h] [--host HOST] [--port PORT] [--cfg CFG]
                         [--loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}]

Read data from socket and push to Data Queue

optional arguments:
  -h, --help            show this help message and exit
  --host HOST           Host to bind to
  --port PORT           Port to bind to
  --cfg CFG             Configuration file
  --loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}
                        Kind of diagnostic output

EXAMPLE: dataq_push_svc.py --host localhost --port 9988
```

4.3.2 dataq_pop_svc.py

Pop a record ID from the active queue (blocking if its empty). Get associated record from redis give ID as key. Syncronously run each action against each record. If an action fails, put the ID back at the top of the queue so it will run later. Only allow N failures. The specific action used is determined by config file when this program is started.

The list "dummy_aq" is used to allow actionP to be toggled while blocked waiting on an empty queue. Without such a mechanism, we can leak one record through the queue after actionP changed to "off".

Failed actions are automatically rerun. The max number of times it will be tried is a configuration parameter ("maximum_errors_per_record"). If an action for a particular id fails more than this, it is moved to the inactive queue. To disable any retry, set the parameter to zero. (the default)

```
~/sandbox/data-q/dataq_pop_svc.py --help

usage: dataq_pop_svc.py [-h] [--host HOST] [--port PORT] [--cfg CFG]
                        [--loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}]

Data Queue service

optional arguments:
  -h, --help            show this help message and exit
  --host HOST           Host to bind to
  --port PORT           Port to bind to
  --cfg CFG             Configuration file
  --loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}
                        Kind of diagnostic output

EXAMPLE: dataq_pop_svc.py --loglevel DEBUG &
```

4.3.3 dataq_cli.py

This program provides commands for diagnostics and manipulating the queue. Its where most of the code is. Multiple switches can be given on the command line, but the order in which their associated functions will be done is fixed internally (switch order doesn't affect it).

I expect we will need more commands here. I'm open to renaming the commands too.

The current set of switches require 0, 1, or 2 additional parameters.

Provides commands to modify queue and queue processing:

1. ☒ Summary of queues to stdout
2. ☒ LIST current contents of queue
3. ☒ ACTION(True|False): un/suspend queue action processing (no pop, no actions)
4. ☒ READ(True|False): un/suspend queue socket read (no push, no socket read)
5. ☒ CLEAR (empty DB without doing any actions)
6. ☒ DUMP a copy of queue into file
7. ☒ LOAD file into queue
8. ☒ ADVANCE range of records (given by start/stop id) to front of line
9. ☒ DEACTIVATE range of records (move ACTIVE to INACTIVE)
10. ☒ ACTIVATE range of records (move INACTIVE to ACTIVE)

```
~/sandbox/data-q/dataq_cli.py --help

usage: dataq_cli.py [-h] [--host HOST] [--port PORT] [--cfg CFG] [--summary]
                    [--info] [--list {active,inactive,records}]
                    [--action {on,off}] [--read {on,off}] [--clear]
                    [--dump DUMP] [--load LOAD] [--advance ADVANCE ADVANCE]
                    [--deactivate DEACTIVATE DEACTIVATE]
                    [--activate ACTIVATE ACTIVATE]
                    [--loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}]

Modify or display the data queue
```

```
optional arguments:
  -h, --help            show this help message and exit
  --host HOST           Host to bind to
  --port PORT           Port to bind to
  --cfg CFG             Configuration file
  --summary, -s         Show summary of queue contents.
  --info, -i            Show info about Redis server.
  --list {active,inactive,records}, -l {active,inactive,records}
                        List queue
  --action {on,off}, -a {on,off}
                        Turn on/off running actions on queue records.
  --read {on,off}, -r {on,off}
                        Turn on/off reading socket and pushing to queue.
  --clear               Delete queue related data from DB
  --dump DUMP           Dump copy of queue into this file
  --load LOAD           File of data records to load into queue
  --advance ADVANCE ADVANCE
                        Move records to end of queue.
  --deactivate DEACTIVATE DEACTIVATE
                        Move records to INACTIVE
  --activate ACTIVATE ACTIVATE
                        Move records to ACTIVE
  --loglevel {CRTICAL,ERROR,WARNING,INFO,DEBUG}
                        Kind of diagnostic output

EXAMPLE: dataq_cli.py --summary
```

4.4 Persistent data structures

This is what we store in Redis. Names on the left are the redis keys.

activeq List of IDs. The active queue. New stuff goes on left, next record to process comes off the right.

activeq_set SET version of activeq so we can tell if an id is already on the active list.

inactiveq List of IDs. The inactive list. Anything from the input stream with unique id that doesn't ultimately get acted upon as it comes off the activeq goes here.

inactiveq_set SET version of inactiveq so we can tell if an id is already on the inactive list.

record_ids Set of IDs used as keys to a record (for both active/inactive queue). For each id there is an hmset(id,rec) to map id to record.

errorcnt errorcnt[id] = cnt; number of Action errors against ID

actionFlag (on|off) Process actions?

readFlag (on|off) Read socket and push to queue?

dummy_aq List used to clear block of AQ when needed on change of actionFlag

5 Assumptions

- "Queue processing is not the bottle-neck"
  - I take this to mean that input records are slow but come from multiple places. Actions take less time than average record arrival.

Attribute	Value
max queue size	on order of 10^4
input record rate	slower than 1/second
time to do an action	less than 10 seconds

6 Use Cases

6.1 Running smoothly

This is the normal use case. If all our software is running well and all data meets format expectations, this is how we think it will run.

1. SensorProxy sends data record to the system (fan in from multiple SensorProxies)
2. System adds record to queue (FIFO order)
3. System pops record off bottom of queue, applies Action to record
   - If Action fails; record pushed to top of queue, failure count incremented

6.2 Systemic Action Failures

If a systemic problem causes a cascade of action failures (e.g. a memory leak in Ingest module makes it run out of RAM), we may get a bunch of failures. When the systemic problem is eliminated, we want to be able to run Action against the records that haven't been processed.

1. Operator is subscribed to error queue notifications.
2. Action is intended to be a Network Move. Action fails.
3. If the Action for the record has failed more than N times; an error is logged and the record is deactivated (moved from active to inactive queue);
4. Otherwise: DQ increments error count against record and pushed record back to the head of the queue. (last to execute)
5. Operator receives notification of error.
6. Operator diagnoses problem and finds that actions are attempting to transfer to a remote machine that is not available.
7. Operator restarts remote machine.
8. Operator re-activates records that produced error.

6.3 Queue gets exceedingly large

6.4 Power failure brings down machine that was running Queue