ddmq Documentation

Martin Dahlö

Sep 20, 2019

Introduction:

ddmq is a file based and serverless messaging queue, aimed at providing a low throughput* messaging queue when you don't want to rely on a server process to handle your requests. It will create a directory for every queue you create and each message is stored as a JSON objects in a file. *ddmq* will keep track of which messages has been consumed and will requeue messages that have not been acknowledged by the consumers after a set timeout. Since there is no server handling the messages, the houskeeping is done by the clients as they interact with the queue.

ddmq is written in Python and should work for both Python 2.7+ and Python 3+, and can also be run as a command-line tool either by specifying the order as options and arguments, or by supplying the operation as a JSON object.

** It could handle ~5000-6000 messages per minute (not via CLI) on a SSD based laptop (~10% of RabbitMQ on the same hardware), but other processes competing for file access will impact performance.*

CHAPTER 1

Key Features

- serverless
- file based
- first in - first out, within the same priority level
- outputs plain text, json or yaml
- input json packaged operations via command-line
- global and queue specific settings
  - custom message expiry time lengths
  - limit the number of times a message will be requeued after exipry
- message specific settings
  - set custom priority of messages (all integers >= 0 are valid, lower number = higher priority)
  - all other message properties can also be changed per message

CHAPTER 2

Installation

```
pip install ddmq
```

CHAPTER 3

Command-Line Usage

```
$ ddmq create -f /tmp/ddmq queue_name
$ ddmq publish /tmp/ddmq queue_name "Hello World!"
$ ddmq consume /tmp/ddmq queue_name
```

CHAPTER 4

Python Module Usage

```
import ddmq
b = ddmq.broker('/tmp/ddmq', create=True)
b.publish(queue='queue_name', msg_text='Hello World!')
msg = b.consume(queue='queue_name')
print(msg.message)
```

CHAPTER 5

Troubleshooting

"ddmq: command not found" when trying to run the command-line tool This is likely because the location where pip installs the *ddmq* executable is not in your PATH. Run the following commands to print out the location where it is installed:

```
import ddmq
ddmq.get_ddmq_bin_path()
```

5.1 Dead Drop Messaging Queue

ddmq is a file based and serverless messaging queue, aimed at providing a low throughput* messaging queue when you don't want to rely on a server process to handle your requests. It will create a directory for every queue you create and each message is stored as a JSON objects in a file. *ddmq* will keep track of which messages has been consumed and will requeue messages that have not been acknowledged by the consumers after a set timeout. Since there is no server handling the messages, the houskeeping is done by the clients as they interact with the queue.

ddmq is written in Python and should work for both Python 2.7+ and Python 3+, and can also be run as a command-line tool either by specifying the order as options and arguments, or by supplying the operation as a JSON object.

5.1.1 Requirements

Python 2.7+ or 3+, should work with both.
Additional modules **required**: pyyaml
Additional modules *recommended*: beautifultable

5.1.2 Installation

```
$ pip install ddmq
```

5.1.3 Key Features

- serverless
- file based
- First in - first out, within the same priority level
- outputs plain text, json or yaml
- input json packaged operations via command-line
- global and queue specific settings
  - custom message expiry time lengths
  - limit the number of times a message will be requeued after exipry
- message specific settings
  - set custom priority of messages (all integers >= 0 are valid, lower number = higher priority)
  - all other message properties can also be changed per message

5.1.4 Command-Line Usage

```
usage: ddmq <command> [<args>]

The available commands are:
view      List queues and number of messages
create    Create a queue
delete    Delete a queue
publish   Publish message to queue
consume   Consume message from queue
ack       Positivly acknowledge a message
nack      Negativly acknowledge a message (possibly requeue)
purge     Purge all messages from queue
clean     Clean out expired messages from queue
json      Run a command packaged as a JSON object

For more info about the commands, run
ddmq <command> -h

Command-line interface to Dead Drop Messaging Queue (ddmq).

positional arguments:
command        Subcommand to run

optional arguments:
-h, --help     show this help message and exit
-v, --version  print version



Examples:

# create a new queue and publish a message to it
$ ddmq create -f /tmp/ddmq queue_name
$ ddmq publish /tmp/ddmq queue_name "Hello World!"
```

(continues on next page)

(continued from previous page)

```
# consume a message from a queue
$ ddmq consume /tmp/ddmq queue_name

# view all queues present in the specified root directory
$ ddmq view /tmp/ddmq

# remove all messages from a queue
$ ddmq purge /tmp/ddmq queue_name

# delete a queue
$ ddmq delete /tmp/ddmq queue_name
```

5.1.5 Python Module Usage

```
# imports both the broker and message module
import ddmq

# create the broker object and specify the path to the root directory
# adding create=True to tell it to create and initiate both the root
# directory and queue directories if they don't already exist
b = ddmq.broker('/tmp/ddmq', create=True)

# publish a message to the specified queue
b.publish(queue='queue_name', msg_text='Hello World!')

# consume a single message from the specified queue
msg = b.consume(queue='queue_name')

# print the message contained
print(msg.message)
```

5.1.6 File Structure

The structure ddmq uses to handle the messages consists of a root directory, with subfolders for each created queue. The messages waiting in a queue are stored in the queue's folder, and messages that have been consumed but not yet acknowledged are stored in the queue's work directory.

```
root/
├── ddmq.yaml
├── queue_one
│   ├── 999.3.ddmqfc24476c6708416caa2a101845dddd9a
│   ├── ddmq.yaml
│   └── work
│       ├── 1538638378.999.1.ddmq39eb64e1913143aa8d28d9158f089006
│       └── 1538638379.999.2.ddmq1ed12af3760e4adfb62a9109f9b61214
└── queue_two
    ├── 999.1.ddmq6d8742dbde404d5ab556bf229151f66b
    ├── 999.2.ddmq15463a6680f942489d54f1ec78a53673
    ├── ddmq.yaml
    └── work
```

In the example above there are two queues created (queue_one, queue_two) and both have messages published to them. In queue_one there are two messages that have been consumed already, but not yet acknowledged (*acked*), so

the messages are stored in the queue_one's work folder. As soon as a message is acked the message will be deleted by default. Messages that are negatively acknowledged (*nacked*) will be requeue by default.

Both the root directory and each queue subfolder will contain config files named *ddmq.yaml* that contains the settings to be used. The root's config file will override the default values, and the queue's config files will override both the default values and the root's config file. If a message is given specific settings when being published/consumed, these settings will override all the ddmq.yaml files.

The message files themselves contain a JSON string with all the properties that make up a message object.

```
{"priority": 999, "queue_number": "1234556789356735", "requeue_counter": 0, "filename
↪": "queue_one/999.2.ddmq1ed12af3760e4adfb62a9109f9b61214", "queue": "queue_one",
↪"requeue_limit": null, "timeout": null, "message": "msg", "requeue": false, "id":
↪"1ed12af3760e4adfb62a9109f9b61214"}
```

5.1.7 ddmq.yaml

The config files in the root and queue directories in YAML format. The parameters that can be changed and their default values are:

```
cleaned: 0              # epoch timestamp when the queue was last cleaned
message_timeout: 600    # the number of seconds after which it will be considered␣
↪expired, after a message is consumed
priority: 999           # the default priority level of published messages. lower␣
↪number = higher priority
requeue: true           # nacked messages are requeued by default, set this to false␣
↪to delete them instead
requeue_prio: 0         # the priority requeued messages will get (0 = highest prio)
```

5.1.8 Use case

Since ddmq handles one file per message it will be much slower than other queues. A quick comparison with RabbitMQ showed that first publishing and then consuming 5000 messages is about 10x slower using ddmq (45s vs 4.5s). The point of ddmq is not performance, but to be used in environments where you can't run a server for some reason.

My own motivation for writing ddmq was to run on a shard HPC cluster where I could not reliably run a server process on the same node all the time. The mounted network storage system was available everywhere and all the time though. The throughput was expected to be really low, maybe <10 messages per day so performance was not the main focus.

Example: parallelization within or beyond nodes with minimal effort

Let's say you have many task to go through, and each task takes more than a couple of seconds. A singel threaded approach to process n files could look like this:

```
program.py:

# go through the file names and process directly
for file in file_names:
    run_task(file)
```

This will take n*seconds_per_task to complete. If you instead submit each task to ddmq, you can start as many consumers as you want to handle the processing, and the time to complete should be around n*seconds_per_task/number_of_consumers

```
program.py:

# init queue
import ddmq
b = ddmq.broker('/tmp/ddmq', create=True)
b.create_queue('tasks')

# go through the file names and submit to queue
for file in file_names:
    b.publish('tasks', msg_text=file)




consumer.py:

# init queue
import ddmq
import time
b = ddmq.broker('/tmp/ddmq', create=True)

while True:
    msg = b.consume('tasks')

    # wait 10s for messages if the queue is empty
    if not msg:
        time.sleep(10)
    else:
        # run the task and acknowledge the message
        run_task(msg.message)
        b.ack(msg)
```

The nice thing about this type of parallelization is that it doesn't matter if you start 8 instances of the consumer script on a single node or if you start 80 instances in total spread over 10 nodes, as long as all of them can read/write to the file system they will work. No need for multithreadded processes or MPI.

5.2 Index

- genindex

5.3 Broker

Defines the broker class which can interact with a ddmq directory. You define a broker by supplying at least a root directory, for example

```
>>> import ddmq
```

```
>>> b = ddmq.broker('../temp/ddmq', create=True)
>>> print(b)
create = True
default_settings = {'priority': 999, 'requeue': True, 'requeue_prio': 0, 'message_
↪timeout': 600, 'cleaned': 0}
```

(continues on next page)

(continued from previous page)

```
global_settings = {'priority': 999, 'requeue': True, 'requeue_prio': 0, 'message_
↪timeout': 600, 'cleaned': 0}
queue_settings = {}
root = ../temp/ddmq
```

```
>>> b.publish('queue_name', "Hello World!")
filename = queue_name/999.1.ddmq89723438b9d0403c91943f4ffaf8ba35
id = 89723438b9d0403c91943f4ffaf8ba35
message = Hello World!
priority = 999
queue = queue_name
queue_number = 123456782356356256566
requeue = False
requeue_counter = 0
requeue_limit = None
timeout = None
```

```
>>> msg = b.consume('queue_name')
filename = 1539702458.999.1.ddmq89723438b9d0403c91943f4ffaf8ba35
id = 89723438b9d0403c91943f4ffaf8ba35
message = Hello World!
priority = 999
queue = queue_name
queue_number = 1234567823561341341356
requeue = False
requeue_counter = 0
requeue_limit = None
timeout = None
```

```
>>> print(msg.message)
Hello World!
```

exception broker.**DdmqError**(*message*, *error*)
Helper class to raise custom errors

class broker.**broker**(*root*, *create=False*, *verbose=False*, *debug=False*)
Class to interact with messaging queues

ack(*queue*, *msg_files=None*, *requeue=None*, *clean=True*)
Positive acknowledgement of message(s)

Parameters

- **queue** – name of the queue the files are in, or the message object to be acked
- **msg_files** – either a single path or a list of paths to message(s) to ack
- **requeue** – True will force message(s) to be requeued, False will force messages to be purged, None (default) will leave it up to the message itself if it should be requeued or not
- **clean** – if True, the client will first clean out any expired messages from the queue's work directory. If False, the client will just ack the message(s) right away and not bother doing any cleaning first (faster).

Returns a list of file names of all messages acknowledged

check_dir(*path*, *only_conf=False*)
Check if the directory contains a ddmq.yaml file to avoid littering non-queue dirs

Parameters

- **path** – path to the directory to check
- **only_conf** – if True, only check if the ddmq.yaml file is present. If False, also check that there is a subdirectory called 'work'

Returns None

clean(*queue*, *force=False*)
Clean out expired message from a specified queue

Parameters **queue** – name of the queue to clean

Returns True if everything goes according to plan, False if no cleaning was done

clean_all()
Clean all the queues in the root director

Parameters **None** –

Returns None

consume(*queue*, *n=1*, *clean=True*)
Consume 1 (or more) messages from a specified queue. The consumed messages will be moved to the queues work folder and have the expiry epoch time prepended to the file name.

Parameters

- **queue** – name of the queue to consume from
- **n** – the number (int) of messages to consume
- **clean** – if True, the client will first clean out any expired messages from the queue's work directory. If False, the client will just consume the message(s) right away and not bother doing any cleaning first (faster).

Returns a single message object if n=1 (default), or a list of the messages that were fetched if n > 1

create_folder(*path*)
Create a folder at a specified path

Parameters **path** – path to the directory to be created

Returns None

create_queue(*queue*)
Create a specified queue

Parameters **queue** – name of the queue to create

Returns True if everything goes according to plan

delete_message(*path*)
Delete a specified message

Parameters **path** – path to the message, or a message object, to be deleted

Returns None

delete_queue(*queue*)
Delete a specified queue

Parameters **queue** – name of the queue to delete

Returns True if everything goes according to plan

get_config_file(*queue=''*)
Get the settings from the config file of a queue or the root dir

Parameters **queue** – if empty, returns the config file from the root folder. If a queue name, will get the config file for that queue

Returns A dict containing all the settings specified in the config file

get_message(*path*)
Get a specified message

Parameters **path** – path to the message to fetch

Returns the requested message

get_message_list(*queue*)
Gets a list of all messages in the specified queue

Parameters **queue** – name of the queue to get messages from

Returns returns 2 lists of file names. The first is the list of all messages still waiting in the queue and the second is a list of all the messages in the queue's work directory

get_queue_number()
Generate the next incremental queue number for a specified queue (epoch time of creation without the decimal punctuation)

Parameters **None** –

Returns a string that is the current timestamp, with the decimal punctuation removed

get_settings(*queue*)
Get the settings for the specified queue. Will try to give a cached version first, and if it is the first time the settings are requested it will read the settings from the config file and store the result

Parameters **queue** – name of the queue to get settings for

Returns None

list_queues()
Generate a list of all valid queues (subdirectories with ddmq.yaml files in them) in the root folder

Parameters **None** –

Returns a list of names of valid queues

nack(*queue*, *msg_files=None*, *requeue=None*, *clean=True*)
Negative acknowledgement of message(s)

Parameters

- **queue** – name of the queue the files are in, or the message object to be nacked
- **msg_files** – either a single path or a list of paths to message(s) to nack
- **requeue** – True will force message(s) to be requeued, False will force messages to be purged, None (default) will leave it up to the message itself if it should be requeued or not
- **clean** – if True, the client will first clean out any expired messages from the queue's work directory. If False, the client will just ack the message(s) right away and not bother doing any cleaning first (faster).

Returns True if everything goes according to plan

publish(*queue*, *msg_text=None*, *priority=None*, *clean=True*, *requeue=False*, *requeue_prio=None*, *timeout=None*, *requeue_counter=0*, *requeue_limit=None*)
Publish a message to a queue

Parameters

- **queue** – name of the queue to publish to
- **msg_text** – the actual message
- **priority** – the priority of the message (default 999). Lower number means higher priority when processing
- **clean** – if True, the client will first clean out any expired messages from the queue's work directory. If False, the client will just publish the message right away and not bother doing any cleaning first (faster).
- **requeue** – if True, the message will be requeud after it expires. If False it will just be deleted.
- **requeue_prio** – if set (int), the message will get this priority when requeued. Default is 0, meaning requeued messages will be put first in the queue.
- **timeout** – if set (int), will override the global and queue specific default setting for how many seconds a message expires after.

Returns a copy of the message published

purge_queue(*queue*)
Purge the specified queue of all messages, but keep the queue folders and config file

Parameters **queue** – name of the queue to purge

Returns a list of 2 numbers; the first is how many messages still waiting in the queue were deleted, and the second how many messages in the queues work directory that was deleted

requeue_message(*path*, *msg=None*)
Requeue a specified message

Parameters **path** – path to the message to requeue

Returns True if everything goes according to plan

update_settings_file(*queue=''*, *package={}*)
Update the settings in a config file for a specified queue or in the root dir

Parameters

- **queue** – if empty, change the config in the root folder. If a queue name, will change the config for that queue
- **package** – a dict containging the changes to the config file

Returns None

version()
Get package version

Parameters **None** –

Returns the package version

5.4 Message

Defines the message class which represents a single message. This class is primarily to be used by the methods in the broker class. You define a message by supplying selected arguments, for example

```
>>> msg = message(queue='queue_name', message='Hello World!')
>>> print(msg)
filename = None
id = None
message = Hello World!
priority = None
queue = queue_name
queue_number = None
requeue = None
timeout = None
```

class message.**message**(*queue=None*, *message=None*, *timeout=None*, *id=None*, *priority=None*, *queue_number=None*, *filename=None*, *requeue=None*, *requeue_counter=None*, *requeue_limit=None*)

Class to represent a single message

classmethod json2msg(*package*)

Converty a JSON object to a message object

msg2json()

Convert a message object to a JSON object

update(*package*)

Update a message object with the parameters supplied by the package (dict)

Python Module Index

b

m

Index