amqpclt Documentation

Release 0.5

Massimo Paladin

November 28, 2013

Contents

amqpclt 0.5 - versatile AMQP client

CHAPTER 1

SYNOPSIS

amqpclt *[OPTIONS]*

DESCRIPTION

amqpclt is a versatile tool to interact with messaging brokers speaking AMQP and/or message queues (see `messaging.queue`) on disk.

It receives messages (see `messaging.message`) from an incoming module, optionally massaging them (i.e. filtering and/or modifying), and sends them to an outgoing module. Depending on which modules are used, the tool can perform different operations.

Here are the supported incoming modules:

- broker: connect to a messaging broker using AMQP, subscribe to one or more queues and receive the messages sent by the broker
- queue: read messages from a message queue on disk (see `messaging.queue`)

Here are the supported outgoing modules:

- broker: connect to a messaging broker using AMQP and send the messages
- queue: store the messages in a message queue on disk (see `messaging.queue`)

Here are some frequently used combinations:

- incoming broker + outgoing queue: drain some destinations, storing the messages on disk
- incoming queue + outgoing broker: (re-)send messages that have been previously stored on disk, optionally with modifications (such as altering the destination)
- incoming broker + outgoing broker: shovel messages from one broker to another

See the "EXAMPLES" sections for concrete examples.

CHAPTER 3

OPTIONS

–callback-code ***CODE*** execute the Python code on each message, see the "CALLBACK" section for more information

–callback-data ***VALUE,...*** pass this data to the user supplied callback code, see the "CALLBACK" section for more information

–callback-path ***PATH*** execute the Python code in the given file on each message, see the "CALLBACK" section for more information

–conf ***PATH*** use the given configuration file, see theconfigURATION FILE section for more information

-c, –count ***INTEGER*** process at most the given number of messages; note: when using an incoming broker, to avoid consuming more messages, it is recommended to enable the –reliable option

–daemon detach **amqpclt** so that it becomes a daemon running in the background

–duration ***SECONDS*** process messages during at most the given number of seconds and then stop

-h, –help print the help page

–incoming-broker-auth ***STRING*** use this authentication string (see `auth.credential`) to authenticate to the incoming broker

–incoming-broker-module ***STRING*** module to use (pikalkombu)

–incoming-broker-type ***STRING*** set the incoming broker type; this can be useful when using features which are broker specific

–incoming-broker-uri ***URI*** use this authentication URI to connect to the incoming broker

–incoming-queue ***KEY=VALUE...*** read incoming messages from the given message queue (see `messaging.queue`)

–lazy initialize the outgoing module only after having received the first message

–log ***STRING*** select logging system, one of: stdout, syslog, file, null

–logfile ***STRING*** select logging file if log system file is selected

–loglevel ***STRING*** select logging level, one of: debug, info, warning and error

–loop when using an incoming message queue, loop over it

–outgoing-broker-auth ***STRING*** use this authentication string (see `auth.credential`) to authenticate to the outgoing broker

–outgoing-broker-module ***STRING*** module to use (pika|kombu)

–outgoing-broker-type ***STRING*** set the outgoing broker type; this can be useful when using features which are broker specific

–outgoing-broker-uri ***URI*** use this authentication URI to connect to the outgoing broker

–outgoing-queue ***KEY=VALUE...*** store outgoing messages into the given message queue (see `messaging.queue`)

–pidfile ***PATH*** use this pid file

–pod print the pod guide

–prefetch ***INTEGER*** set the prefetch value (i.e. the maximum number of messages to received without acknowledging them) on the incoming broker

–quit tell another instance of **amqpclt** (identified by its pid file, as specified by the –pidfile option) to quit

–reliable use AMQP features for more reliable messaging (i.e. client side acknowledgments) at the cost of less performance

–remove when using an incoming message queue, remove the processed messages

–rst print the rst guide

–statistics report statistics at the end of the execution

–status get the status of another instance of **amqpclt** (identified by its pid file, as specified by the –pidfile option); the exit code will be zero if the instance is alive and non-zero otherwise

–subscribe use these options in the AMQP subscription used with the incoming broker; this option can be given multiple times

–timeout-connect ***SECONDS*** use this timeout when connecting to the broker; can be fractional

–timeout-inactivity ***SECONDS*** use this timeout in the incoming module to stop **amqpclt** when no new messages have been received (aka drain mode); can be fractional

–timeout-linger ***SECONDS*** when stopping **amqpclt**, use this timeout to finish interacting with the broker; can be fractional

–version print the program version

–window ***INTEGER*** keep at most the given number of not-yet-acknowledged messages in memory

CHAPTER 4

CONFIGURATION FILE

amqpclt can read its options from a configuration file. For this, the Perl Config::General module is used and the option names are the same as on the command line. For instance:

```
daemon = true
pidfile = /var/run/amqpclt.pid
incoming-queue = path=/var/spool/amqpclt
outgoing-broker-uri = amqp://broker.acme.com:5672/virtual_host
outgoing-broker-auth = "plain name=guest pass=guest"
```

Alternatively, options can be nested:

```
<outgoing-broker>
    uri = amqp://broker.acme.com:5672/virtual_host
    auth = "plain name=guest pass=guest"
</outgoing-broker>
```

Or even:

```
<outgoing>
    <broker>
        uri = amqp://broker.acme.com:5672/virtual_host
        <auth>
            scheme = plain
            name = guest
            pass = guest
        </auth>
    </broker>
</outgoing>
```

The options specified on the command line have precedence over the ones found in the configuration file.

CHAPTER 5

CALLBACK

amqpclt can be given python code to execute on all processed messages. This can be used for different purposes:

- massaging: the code can change any part of the message, including setting or removing header fields
- filtering: the code can decide if the message must be given to the outgoing module or not
- displaying: the code can print any part of the message
- copying: the code can store a copy of the message into files or message queues

To use callbacks, the –callback-path or –callback-code option must be used. The python code must provide functions with the following signature:

- start(self, DATA) (optional) this will be called when the program starts, with the supplied data (see the –callback-data option) as a list reference
- check(self, MESSAGE) (mandatory) this will be called when the program has one message to process; it will be given the message (see messaging.message.Message) and must return either a message (it could be the same one or a new one) or a string describing why the message has been dropped
- idle(self) (optional) this will be called when the program has no message to process
- stop(self) (optional) this will be called when the program stops

The code can be put in a file, on the command line or in the **amqpclt** configuration file, using the “here document” syntax.

Here is an example (to be put in the **amqpclt** configuration file) that prints on stdout a JSON array of messages:

```
callback-code = <<EOF
def start (self):
    self.count = 0
def check(self, msg):
    if self.count:
        sys.stdout.write(", ")
    else:
        sys.stdout.write("[")
    self.count += 1
    sys.stdout.write(msg.serialize())
    return msg
def stop(self):
    if self.count:
        sys.stdout.write("]\n")
    else:
        sys.stdout.write("[]\n")
EOF
```

For simple callback code that only needs the check subroutine, it is enough to supply the “inside code”. If the function definition is missing, the supplied code will be wrapped with:

```
def check(self, msg):
    hdr = msg.header
    ... your code goes here ...
    return msg
```

This allows for instance to remove the message-id header with something like:

```
$ amqpclt ... --callback-code 'del(hdr["foo"])'
```

CHAPTER 6

EXAMPLES

6.1 SENDING

Here is an example of a configuration file for a message sender daemon (from queue to broker), forcing the persistent header to true (something which is highly recommended for reliable messaging) and setting the destination:

```
# define the source message queue
<incoming-queue>
 path = /var/spool/sender
</incoming-queue>
# modify the message header on the fly
callback-code = <<EOF
    hdr["destination"] = "/queue/app1.data"
    hdr["persistent"] = "true"
EOF
# define the destination broker
<outgoing-broker>
    uri = "amqp://broker.acme.com:5672/virtual_host"
</outgoing-broker>
# miscellaneous options
reliable = true
pidfile = /var/run/sender.pid
daemon = true
loop = true
remove = true
```

6.2 SHOVELING

Here is an example of a configuration file for a message shoveler (from broker to broker), clearing some headers on the fly so that messages can be replayed safely:

```
# define the source broker
<incoming-broker>
    uri = "amqp://broker.acme.com:5672/virtual_host"
</incoming-broker>
# define the subscriptions
<subscribe>
```

```
    destination = /queue/app1.data
</subscribe>
<subscribe>
    destination = /queue/app2.data
</subscribe>
# define the destination broker
<outgoing-broker>
    uri = "amqp://dev-broker.acme.com:5672/virtual_host"
</outgoing-broker>
# modify the message destination
callback-code = <<EOF
    hdr["destination"] = "/queue/dest_to_be_replayed"
EOF
```

6.3 RECEIVING

Here is an example of a configuration file for a message receiver (from broker to queue):

```
# define the source broker
<incoming-broker>
    uri = "amqp://broker.acme.com:5672/virtual_host"
    <auth>
        scheme = plain
        name = receiver
        pass = secret
    </auth>
</incoming-broker>
# define the subscriptions
<subscribe>
    destination = /queue/app1.data
</subscribe>
<subscribe>
    destination = /queue/app2.data
</subscribe>
# define the destination message queue
<outgoing-queue>
    path = /var/spool/receiver
</outgoing-queue>
# miscellaneous options
pidfile = /var/run/receiver.pid
```

To run it as a daemon:

```
$ amqpclt --conf test.conf --daemon
```

To use the configuration file above with some options on the command line to drain the queues:

```
$ amqpclt --conf test.conf --timeout-inactivity 10
```

6.4 TAPPING

Callback code can also be used to tap messages, i.e. get a copy of all messages processed by **amqpclt**. Here is some callback code for this purpose that could for instance be merged with the shoveling code above. It also shows how to use the –callback-data option:

```
callback-code = <<EOF
    def start(self, path, qtype="DQS"):
        self.tap_queue = queue.new({"path" : path, "type" : qtype})

    def check(self, msg):
        self.tap_queue.add_message(msg)
        return msg
EOF
```

Callback data must be given to specify which message queue to use:

```
$ amqpclt --conf tap.conf --callback-data "/tmp/tap,DQS"
```

AUTHOR

Massimo Paladin <massimo.paladin@gmail.com> - Copyright (C) 2013 CERN