Parsing MQ event messages as Python objects using amqsevt

Matthew Whitehead
Published on 31/10/2018

Python is a popular language for scripting and tooling, and as such it makes a good choice for writing scripts to monitor MQ events.
The biggest complexity is not in reading the messages from MQ event queues but in parsing the PCF that's in them.
Here's where the MQ sample amqsevt is your friend. amqsevt is a sample program that reads event messages from any of the MQ event queues or topics you specify, converts the PCF into text or JSON, and writes them out to stdout.
In this blog I'll show you how you can execute amqsevt from within a Python script, turn the JSON into Python objects, and then process the event messages one at a time. You could trigger your own Python actions based on which messages are generated or based on the values of specific attributes (for example when the putsFailed attribute for a queue is greater than zero).
It might not be an ideal setup for all circumstances. It's not the most performant way of consuming PCF messages, and if you want to guarantee each event message is processed at least once, then this solution isn't suitable for you. But for some scenarios this can be a pragmatic way of consuming and processing PCF messages in Python with minimal complexity.

amqsevt

I won't go into too much detail on how the amqsevt sample program works. Mark Taylor has already written a blog article giving example use cases here.
A short example of how we'll run amqsevt on the command line is as follows:

```
/opt/mqm/samp/bin/amqsevt -w 10 -m QM1 -o json -q
SYSTEM.ADMIN.STATISTICS.QUEUE -q SYSTEM.ADMIN.ACCOUNTING.QUEUE
```

This command will read all accounting and statistics messages that are produced in 10 seconds, and write them to stdout as JSON. You can try running it yourself, just make sure that you have turned on at least some accounting or statistics messages (see below for the settings I used). You will also want to start a sample program such as amqsput, in order to put some messages to a queue like the SYSTEM.DEFAULT.LOCAL.QUEUE. This will cause accounting messages to be generated when amqsput terminates, and will give you some example numbers in the put statistics of the messages.

Queue manager setup

To test out the example code below I set up my queue manager with the following runmqsc commands:

```
ALTER QMGR ACCTINT(60) ACCTMQI(ON) ACCTQ(ON)
4 : ALTER QMGR ACCTINT(60) ACCTMQI(ON) ACCTQ(ON)
AMQ8005I: IBM MQ queue manager changed.
```

This turns on accounting MQI and queue events and sets the interval between new event messages to 60 seconds (unless an application connects and disconnects in less than 30 seconds, in which cases an event message will be published when the disconnect happens.
ALTER QMGR STATINT(30) STATMQI(ON) STATQ(ON)
5 : ALTER QMGR STATINT(30) STATMQI(ON) STATQ(ON)
AMQ8005I: IBM MQ queue manager changed.
This turns on statistics MQI and queue events and sets the interval between new messages being published to 30 seconds.

Example Python script

Now let's execute amqsevt periodically from a python script. Here's a short Python script that calls amqsevt with a 10 second get-with-wait; waits for amqsevt to return; consumes the output from it as JSON; turns the JSON into Python objects for operating on; then repeats. I've published the full script here on github.com, but here are some snippets from it:
Create a function to display 4 of the different types of event message, for example:

```
def display_accounting_queue_message():
    try:
        print "Application: " + str(x.eventData.applName)
        print "Conn ID: " + str(x.eventData.connectionId)
        for i in range(0, x.eventData.objectCount):
            print " - " + x.eventData.queueAccountingData[i].queueName
            print "   Opens : " +
str(x.eventData.queueAccountingData[i].opens)
            print "   Closes: " +
str(x.eventData.queueAccountingData[i].closes)
            print "   Puts  : " +
str(x.eventData.queueAccountingData[i].puts[1]) + " (failed = " +
str(x.eventData.queueAccountingData[i].putsFailed) + ")"
            print "   Gets  : " +
str(x.eventData.queueAccountingData[i].gets[1]) + " (failed = " +
str(x.eventData.queueAccountingData[i].getsFailed) + ")"
    except AttributeError:
        print "Error getting attribute"
```

Loop indefinitely calling amqsevt with a 10 second wait:

```
while 1:

    ( rc, jsonOutput ) = commands.getstatusoutput(
"/opt/mqm/samp/bin/amqsevt -w 10 -m QM1 -o json -q
SYSTEM.ADMIN.STATISTICS.QUEUE -q SYSTEM.ADMIN.ACCOUNTING.QUEUE" )

    if( rc == 0 ):
    ...
```

Parse each returned block of JSON into a Python object:

```
            for x in jsonOutput.split('\n\n'):
                # Parse JSON into a Python object
                try:
                    x = json.loads(x, object_hook=lambda d:
namedtuple('X', d.keys())(*d.values()))
                except ValueError:
                    print "Error parsing amqsevt output as JSON -> " +
str(x)
                    exit

                print "************************************"
```

```
print "Source: " + x.eventSource.objectName
print "Type: " + x.eventType.name
...
```

Call the relevant formatting function to output certain attributes from each event message type:

```
if (x.eventType.name == "Accounting MQI"):
    display_accounting_mqi_message()
elif (x.eventType.name == "Accounting Queue"):
    display_accounting_queue_message()
elif (x.eventType.name == "Statistics MQI"):
    display_statistics_mqi_message()
elif (x.eventType.name == "Statistics Queue"):
    display_statistics_queue_message()
else:
    print "Unknown message type: " + x.eventType.name
```

Example output

Running the above script for a minute or two, running amqsput and/or amqsget in other terminal windows at the same time, leads to output that looks like this:

```
**********************************

Source: SYSTEM.ADMIN.ACCOUNTING.QUEUE

Type: Accounting MQI

Application: runmqsc

Conn ID: 414D5143514D434F4E462020202020203CD9D65B01F3F722

Total Opens : 0

Total Closes: 0

Total Puts : 0 (failed = 0)

Total Gets : 0 (failed = 0)

**********************************

Source: SYSTEM.ADMIN.ACCOUNTING.QUEUE

Type: Accounting MQI

Application: runmqsc

Conn ID: 414D5143514D434F4E462020202020203CD9D65B01F3F722

Total Opens : 1
```

```
Total Closes: 1

Total Puts : 0 (failed = 0)

Total Gets : 0 (failed = 0)

************************************

Source: SYSTEM.ADMIN.STATISTICS.QUEUE

Type: Statistics MQI

Total Opens for QM : 10

Total Closes for QM: 4

Total Puts for QM : 0

Total Gets for QM : 0

************************************

Source: SYSTEM.ADMIN.STATISTICS.QUEUE

Type: Statistics Queue

- SYSTEM.ADMIN.QMGR.EVENT

Puts : 0 (failed = 0)

Gets : 0 (failed = 0)

- SYSTEM.CLUSTER.COMMAND.QUEUE

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)

- SYSTEM.INTER.QMGR.PUBS

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)

- SYSTEM.INTER.QMGR.FANREQ

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)
```

```
- SYSTEM.BROKER.DEFAULT.STREAM

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)

- SYSTEM.BROKER.ADMIN.STREAM

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)

- SYSTEM.BROKER.INTER.BROKER.COMMUNICATIONS

Puts : 0 (failed = 0)

Gets : 0 (failed = 1)

- SYSTEM.ADMIN.PERFM.EVENT

Puts : 0 (failed = 0)

Gets : 0 (failed = 0)

- SYSTEM.ADMIN.ACTIVITY.QUEUE

Puts : 0 (failed = 0)

Gets : 0 (failed = 0)

**********************************

Source: SYSTEM.ADMIN.ACCOUNTING.QUEUE

Type: Accounting MQI

Application: amqsput

Conn ID: 414D5143514D434F4E462020202020203CD9D65B05F3F722

Total Opens : 2

Total Closes: 2

Total Puts : 0 (failed = 0)

Total Gets : 0 (failed = 0)

**********************************
```

```
Source: SYSTEM.ADMIN.ACCOUNTING.QUEUE

Type: Accounting Queue

Application: amqsput

Conn ID: 414D5143514D434F4E462020202020203CD9D65B05F3F722

- Q2

Opens : 1

Closes: 1

Puts : 0 (failed = 0)

Gets : 0 (failed = 0)
```

The attributes I've decided to output are a selection of those that are available. You'll notice from looking at the Python code that the different message types have different JSON structures. The "Accounting Queue" and "Statistics Queue" messages contain an array of the objects that have been used, either by a specific application (in the case of accounting queue messages) or across the whole queue manager (in the case of statistics queue messages), and you pull out specific MQI call stats for each entry in the array. The "Accounting MQI" and "Statistics MQI" messages contains a single set of attributes that relate either to a specific application (in the case of accounting queue messages) or to the whole queue manager (in the case of statistics messages).
You can edit the script to output the complete JSON payload that I've selected attributes from to see which other attributes are available.
You can see that for the accounting messages the connection ID of the application doing the puts and gets is included in the output. You can use this in combination with commands like DISPLAY CONN(<conn-id>) to view more information about the application responsible for the workload.

Using the script for other event types

I've shown how you can consume accounting and statistics messages, but you can use the same code to consume other types of event message. For example you could specify additional queues to the amqsevt executable such as SYSTEM.ADMIN.PERF.EVENT, SYSTEM.ADMIN.COMMAND.EVENT, SYSTEM.ADMIN.CONFIG.EVENT, SYSTEM.ADMIN.QMGR.EVENT and so on. You can also give amqsevt a system topic to consume messages from, such as "-t \$SYS/MQ/INFO/QMGR/QM1/Monitor/STATQ/Q2/PUT" to subscribe to put statistics for Q2. Because each message has its own structure I haven't given examples of parsing each event type, but you can create a script that works for the events you are interested in.

Caveats

I’m certainly not an expert at Python so my way of executing amqsevt and consuming its output may not be the optimum way of doing it. Please feel free to submit improvements as pull requests to the github.com repo.

The settings I’ve used are just examples used to demonstrate the concept. You probably wouldn’t want to turn on all event messages all of the time, especially if you’re not running something permanently to consume them. You can turn each of the message types I’ve used on a per-queue basis. Simply look for the equivalent settings on queue objects.