Application Activity – formatting and extracting key information

Mark E Taylor
Published on 31/07/2018
1

Application Activity Trace

Application Activity Trace is a mechanism on the MQ Distributed platforms that give a report of all the MQI calls made by a program. Originally configured via a text file, MQ V9 enhanced them by allowing a monitoring application to subscribe to topics that describe the application or channel of interest.

Like all MQ events, they are PCF messages with the need to format them before they can be read by a real person. MQ provides a sample program, **amqsact** to do just that. But that sample program does not make it easy to automate processing of the events. Production-level monitoring tools are expected to take the raw events, format them, and then do something interesting with the contents.

In another post I wrote about the **amqsevt** sample program and extensions that provide JSON output for MQ events. I did not originally intend amqsevt to be used for application activity events. I was more interested in processing the regular events such as “queue full”, and the **amqsact** sample was already available and designed specifically for those events. But it did turn out to work with them anyway. That JSON-capable variant of amqsevt is included in the new MQ V9.1, so you don’t need to download and compile the program yourself. And since it is now part of the product I’ve deleted the original source code from github.

One benefit of using JSON is that there are many tools capable of processing that format. In the earlier article about amqsevt, I showed how events could be passed to Splunk. This new article shows how I can use amqsevt to get a few key fields from application activity trace with a simple filter. That should make any subsequent review – perhaps an application audit – or processing much faster. It ought to be a much lighter-weight mechanism for monitoring what applications do, if you don’t need to preserve the full set of data that activity traces contain.

This can be seen as another piece of a broader strategy of extending MQ’s administrative actions and reports to work with JSON. There is the REST administrative API, error logs via JSON, and even z/OS SMF data can be formatted that way.

The English output from amqsevt

I’ll start off showing the basic English-ish output from amqsevt. For these examples, I’m using the queue-based activity trace information rather than the subscription model, but both work equally well. In a real environment, you may prefer to use the subscription approach to permit multiple independent processors of the events.

Run the command

```
  amqsevt -m QM1 -q SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE
```

and the start of the output will look like this:

```
**** Message #1 (5712 Bytes) on Queue SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE ****
Event Type                      : Activity Trace [209]
Reason                          : None [0]
Event created                   : 2018/07/10 12:38:22.81 GMT
  Queue Mgr Name                : QM1
  Host Name                     : example.hursley.ibm.com
  Start Date                    : 2018-07-10
  Start Time                    : 13:38:22
  Appl Name                     : amqsput
  Channel Type                  : Svrconn
  ...
  ACTIVITY TRACE
    Operation Id                : Connx
    Thread Id                   : 88
    Operation Date              : 2018-07-10
    Operation Time              : 13:38:22
    Connection Id               :
414D514356393030305F4120202020205B2B779523E479A8
    Queue Mgr Name              : QM1
    Qmgr Op Duration            : 1102
    Comp Code                   : Ok [0]
  ...
```

The JSON output from amqsevt

Next, I will do the same thing but adding the new “-o json” option to the command

```
  amqsevt -m QM1 -q SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE -o json
```

Which produces output that starts like this:

```
{
"eventSource" : { "objectName": "SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE",
                  "objectType" : "Queue" },
"eventType" : {
    "name" : "Activity Trace",
    "value" : 209
  },
"eventReason" : {
    "name" : "None",
    "value" : 0
  },
"eventCreation" : {
    "timeStamp"  : "2018-07-10T12:44:26Z",
    "epoch"      : 1531226666
  },
"eventData" : {
  "queueMgrName" : "QM1",
  "hostName" : "example.hursley.ibm.com",
  "startDate" : "2018-07-10",
  "startTime" : "13:44:25",
  "endDate" : "2018-07-10",
  "endTime" : "13:44:26",
  "commandLevel" : 910,
...
```

Any management application that understands JSON could take this and process it directly.

CSV-formatted filtered output from amqsevt

The final demonstration filters the JSON to give single output lines for each MQI verb. These show key fields describing an application's execution. The command to get the messages is identical to before, but there is an extra step after the retrieval.

```
amqsevt -m QM1 -q SYSTEM.ADMIN.TRACE.ACTIVITY.QUEUE -o json | jq -r -f jqFilt
```

I'll show the output and then describe how I created it.

Output

The output looks like this:

```
"amqsput","2018-07-11","08:16:48","Connx",0,"N/A"
"amqsput","2018-07-11","08:16:48","Open",0,"X"
"amqsput","2018-07-
11","08:16:48","Put",0,"X",48,"414D512056393030305F4120202020205B2B779523E4BFBE"
"amqsput","2018-07-11","08:16:48","Close",0,"X"
"amqsput","2018-07-11","08:16:48","Disc",0,"N/A"
"amqsget","2018-07-11","08:16:48","Connx",0,"N/A"
"amqsget","2018-07-11","08:16:48","Open",0,"X"
"amqsget","2018-07-
11","08:16:48","Get",0,"X",38,"414D512056393030305F4120202020205B2B779523E4BFBE"
"amqsget","2018-07-11","08:16:48","Get",2033,"X",250059,0
"amqsget","2018-07-11","08:16:48","Close",0,"X"
"amqsget","2018-07-11","08:16:48","Disc",0,"N/A"
```

This shows the execution of amqsput and amqsget, with a measure of how long in microseconds it took to do the MQPUT and MQGET on queue "X". I've also chosen to show the msgid from the messages which might be valuable for later auditing. You can see how they match up here, demonstrating that the same message was sent and retrieved. You can also see that a second MQGET took 250ms. That is the wait interval after which there's a 2033 (no message available) return code. The rows are in CSV (comma-separated value) style, another common format for direct processing or for import to spreadsheets and databases where they can be analysed.

The jq command

The **jq** command is a commonly-used open source program that manipulates JSON records. It's a bit like **awk** but dealing with JSON structures instead of text fields. There are compiled versions of it for many platforms including Linux and Windows. It is easy to compile on many other platforms too – I often run it on AIX. The **jqFilt** file passed to jq is just a program executed by jq to tell it how to process the input JSON records.

The filter script

Although it might appear complicated at first glance, the program that jq runs is actually rather simple once you understand the principles. JSON fields are referenced by name, separated with a '.' for the nesting of objects within objects. Arrays can be handled too. There is a pipeline within the script, as JSON objects are selected or transformed, and passed to the next stage. There are

functions such as *length* to work with the data. And the script language permits variables. In this program, variables are used to stash the application name which does not appear in the details of every individual MQI operation, but which is useful for each output line in the CSV format.

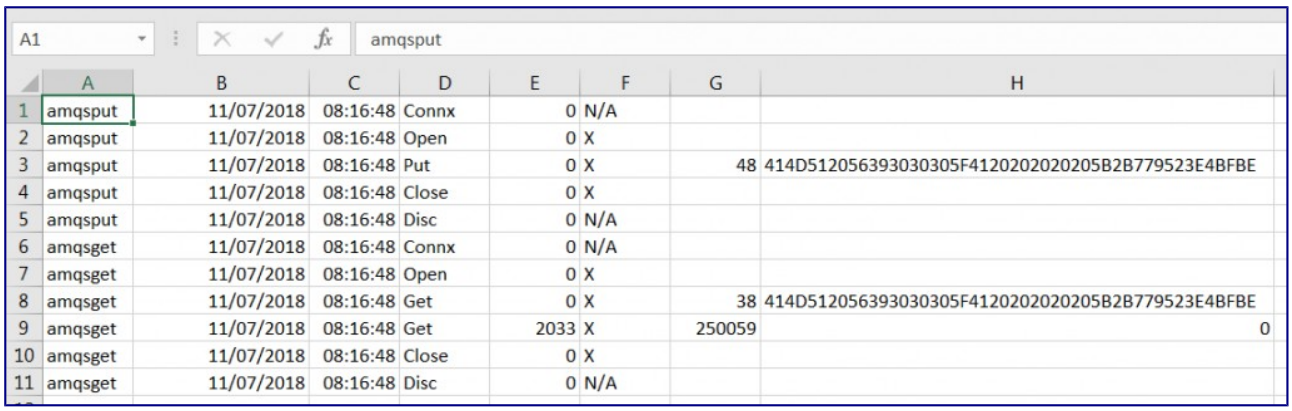
The final step of this script formats the values in a JSON object as CSV. That conversion is a built-in function in jq. It deals with any special escape characters and getting the right levels of quotes.

```
# The -r flag to jq removes one layer of quote characters from the output -
better for CSV import
# Operations are
# 1. Select only activity trace records from stdin.
# 2. Pull out the application name so that the lower entries can use it in the
output.
# 3. If there's an object name, print it; if not, we still print something to
#    keep consistent columns.
# 4. Put out extra fields for the get/put verbs.
# 5. Finally format the whole record as CSV.

select(.eventData.activityTrace != null) |
      .eventData.applName as $applName |
      (.eventData.activityTrace[] |
         [
           $applName,
           .operationDate,
           .operationTime,
           .operationId,
           .reasonCode.value,
           if (.objectName | length) > 0
           then
              .objectName
           else
              "N/A"
           end,
           if .operationId == "Get" or
              .operationId == "Put" or
              .operationId == "Put1"
           then
              .qmgrOpDuration,
              .msgId
           else
              empty
           end
         ]
      ) |
    @csv
```

Imported CSV

And here we can see the data after it has been imported to a spreadsheet.



A1 amqsput

	A	B	C	D	E	F	G	H
1	amqsput	11/07/2018	08:16:48	Connx	0	N/A		
2	amqsput	11/07/2018	08:16:48	Open	0	X		
3	amqsput	11/07/2018	08:16:48	Put	0	X	48	414D512056393030305F4120202020205B2B779523E4BFBE
4	amqsput	11/07/2018	08:16:48	Close	0	X		
5	amqsput	11/07/2018	08:16:48	Disc	0	N/A		
6	amqsget	11/07/2018	08:16:48	Connx	0	N/A		
7	amqsget	11/07/2018	08:16:48	Open	0	X		
8	amqsget	11/07/2018	08:16:48	Get	0	X	38	414D512056393030305F4120202020205B2B779523E4BFBE
9	amqsget	11/07/2018	08:16:48	Get	2033	X	250059	0
10	amqsget	11/07/2018	08:16:48	Close	0	X		
11	amqsget	11/07/2018	08:16:48	Disc	0	N/A		

Conclusion

The full Application Activity Trace output can contain a lot more information than you might want to store long-term, and it can be tricky to process without specialised programs. I hope this article has shown how a couple of simple tools can make the job of selection and analysis much easier.