Testing Design - Java Broker CPU GC Monitoring

Java Broker CPU/GC Monitoring

When testing the Java broker with the perftest suite of tests one of the problems is that we only gather the result of the tests. If the numbers are better than last time great. However, investigating how the broker handled the load is probably a good thing to do. If CPU usage jumped 100% for only 10% performance benefit we should look at why. Similarly if we spend a lot of time in GC we should check what extra garbage we have started creating.

So as a starter lets enable verbose:gc on the tests runs and monitor the CPU usage additional monitoring can be added to the suite but the first step is to have the ability to gather the data.

Data collection

Current goals are to focus on the Java broker by collecting CPU usage (measured via top -p), and GC data as written by the JVM with verbose gc logging. This same information could be gathered for the client JVM however it would require modification to the existing scripts to log the data.

This data can then be graphed for later review. Eventually some form of automated analysis could be performed from an automated testing platform which could identify problematic commits earlier in our release cycle.

JVM GC Logging

Start the vm with verbose gc being logged to its own file. The file contains log entries in seconds since VM start up, this makes correlating that time with other output, such as CPU, difficult. As The gc file is created very close to the VM startup the file access time can be taken as its creation time. This time can then be used as a start point for the offsets of each log entry. On a linux box the access time can be seen via 'ls -tu' and is shown in minutes or 'stat' which shows the additional seconds field.

CPU Monitoring

Running 'top' in batch mode and fixed on the JVM process with -p will provide a sample of the CPU usage of the process which can then be aligned to the GC data using the same file creation time method. If more data points are required the frequency of top updates can be modified on the command line. An initial period of 0.5 seconds should give good coverage when compared to the GC output.

Test Execution

Integrating with our existing test cases rather than requiring a rewrite of the tests is desirable as we can focus on the collection of data rather than updating the existing test cases.

Broker Monitoring Setup

This section details how we can go about gathering GC and CPU data.

GC Gathering

The JVM has a number of extra GC options which allow us to gather the data very easily.
Setting QPID_OPTS to the following value will create a gc.log file with GC details.

```
export QPID_OPTS="-Xloggc:gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps"
```

CPU Usage Gathering

Gathering CPU usage statistics can be done via 'top' running in batch mode. A simple script can be created to provide the a monitoring rate and a PID to monitor.

```
top -d <CPU_MONITOR_RATE> -Sbcp <PID>  > broker_cpu.log
```

This script will create entries in the broker_cpu.log of the following format:

```
top - 05:16:32 up 24 days,  1:04,  6 users,  load average: 0.08, 0.03, 0.31
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.8% us,  0.3% sy,  0.0% ni, 98.8% id,  0.0% wa,  0.0% hi,  0.1% si
Mem:   4040220k total,  2814272k used,  1225948k free,   194860k buffers
Swap: 16386292k total,        0k used, 16386292k free,  2270140k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
27774 ritchiem  16   0 1254m  64m 8564 S    0  1.6   0:01.17 java -server -DPNAM
```

Log Data Processing

GC

The GC log file has a number of types of entries:

Successful Full GC

```
0.503: [Full GC (System) 0.503: [CMS: 0K->1454K(63872K), 0.0617910 secs] 8321K->1454K(83008K), [CMS Perm :
10933K->10925K(21248K)], 0.0619320 secs]
```

ParNew run

```
9.351: [GC 9.351: [ParNew: 19119K->2112K(19136K), 0.0141410 secs] 50757K->42135K(83008K), 0.0142560 secs]
```

The CMS phases

```
9.366: [GC [1 CMS-initial-mark: 40023K(63872K)] 42272K(83008K), 0.0016310 secs]
9.367: [CMS-concurrent-mark-start]
9.407: [CMS-concurrent-mark: 0.040/0.040 secs]
9.407: [CMS-concurrent-preclean-start]
9.408: [CMS-concurrent-preclean: 0.001/0.001 secs]
9.408: [CMS-concurrent-abortable-preclean-start]
10.495: [CMS-concurrent-abortable-preclean: 0.113/1.088 secs]
10.498: [CMS-concurrent-sweep-start]
10.508: [CMS-concurrent-sweep: 0.010/0.010 secs]
10.509: [CMS-concurrent-reset-start]
10.517: [CMS-concurrent-reset: 0.008/0.008 secs]
```

Failed Full GC

```
357.779: [Full GC 357.779: [CMS357.885: [CMS-concurrent-abortable-preclean: 0.199/0.990 secs]
 (concurrent mode failure): 961425K->13192K(962444K), 0.2641230 secs] 963545K->13192K(981580K), [CMS Perm :
17808K->17777K(29804K)], 0.2649910 secs]
```

YG Rescan

```
10.496: [GC[YG occupancy: 2151 K (19136 K)]10.496: [Rescan (parallel) , 0.0016840 secs]10.497: [weak refs
processing, 0.0007330 secs] [1 CMS-remark: 259108K(260864K)] 261259K(280000K), 0.0025130 secs]
```

These entries can allow us to generate a graph of memory usage as the application runs. At a first approach using the 'ParNew run' entry we can graph the heap usage (50757K->42135K) and the current maximum heap size (83008K).

Timing

As the log entries are timestamped in seconds since VM startup if we want to correlate these values with the CPU or client log then we must convert them to real times. This can be done by looking at the 'Access' time of the gc.log. When retriving the log file (or after starting the JVM) we can gather this information using 'stat':

Output from stat

```
  File: `2009-05-22-1016/broker-results/logging/gc.log'
  Size: 1183063         Blocks: 2320       IO Block: 4096   regular file
Device: fd00h/64768d    Inode: 363245      Links: 1
Access: (0644/-rw-r--r--)  Uid: (  500/ritchiem)   Gid: (  500/ritchiem)
Access: 2009-05-29 11:24:00.000000000 +0100
Modify: 2009-05-22 13:42:22.000000000 +0100
Change: 2009-05-22 13:42:22.000000000 +0100
```

As long as the file as not been opened for reading then the 'Access' time will be the time that the file was created. We can then use this as the base for the offsets in the log file.

CPU

Processing the CPU data can be done in a similar way to the GC log file. As we know how often the top command will run we can use the stat output to give us a base entry for the log and then increment each log entry by the logging interval.

There is a risk here that top does not accurately log at the specified rate. However if logging is performed at sub second intervals the effort in extracting the time data from top and calculating the millisecond value for the log entry is not believed to be worth the effort.

Result Presentation

The results of the GC and CPU can be put through gnuplot and graphed here is an example of what it might look like: