Troubleshoot Java Stack High CPU Utilization

TAC **Document ID: 119010**

Contributed by Ankit Aggarwal and Tony Pina, Cisco TAC Engineers.
May 28, 2015

Contents

Introduction

This document describes Java Stack (Jstack) and how to use it in order to determine the root cause of high CPU utilization in Cisco Policy Suite (CPS).

Troubleshoot with Jstack

What is Jstack?

Jstack takes a memory dump of a running Java process (in CPS, QNS is a Java process). Jstack has all the details of that Java process, such as threads/applications and the functionality of each thread.

Why do you need Jstack?

Jstack provides the Jstack trace so that engineers and developers can get to know the state of each thread.

The Linux command used to obtain the Jstack trace of the Java process is:

```
# jstack <process id of Java process>
```

The location of the Jstack process in every CPS (previously known as Quantum Policy Suite (QPS)) version is '/usr/java/jdk1.7.0_10/bin/' where 'jdk1.7.0_10' is the version of Java and the version of Java can differ in every system.

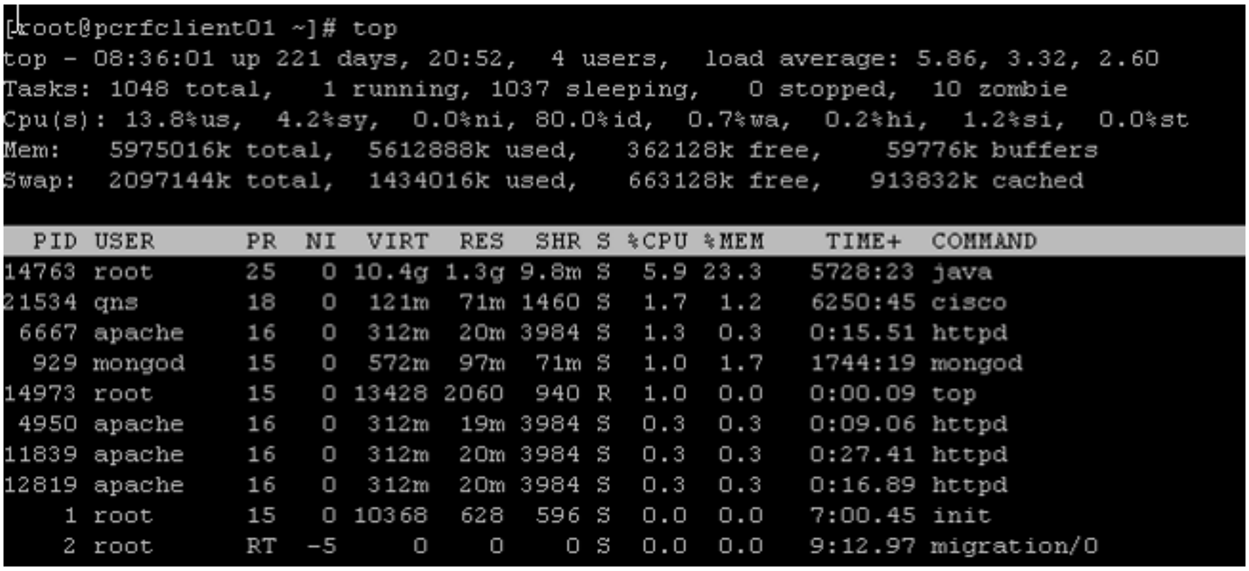
You can also enter a Linux command in order to find the exact path of the Jstack process:

```
# find / −iname jstack
```

Jstack is explained here in order to get you familiar with steps to troubleshoot high CPU utilization issues because of the Java process. In high CPU utilization cases you generally learn that a Java process utilizes the high CPU from the system.

Procedure

Step 1: Enter the ***top*** Linux command in order to determine which process consumes high CPU from the virtual machine (VM).



```
[root@pcrfclient01 ~]# top
top - 08:36:01 up 221 days, 20:52,  4 users,  load average: 5.86, 3.32, 2.60
Tasks: 1048 total,   1 running, 1037 sleeping,   0 stopped,  10 zombie
Cpu(s): 13.8%us,  4.2%sy,  0.0%ni, 80.0%id,  0.7%wa,  0.2%hi,  1.2%si,  0.0%st
Mem:   5975016k total,  5612888k used,   362128k free,    59776k buffers
Swap:  2097144k total,  1434016k used,   663128k free,   913832k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14763 root      25   0 10.4g 1.3g 9.8m S  5.9 23.3   5728:23 java
21534 qns       18   0  121m  71m 1460 S  1.7  1.2   6250:45 cisco
 6667 apache    16   0  312m  20m 3984 S  1.3  0.3   0:15.51 httpd
  929 mongod    15   0  572m  97m  71m S  1.0  1.7   1744:19 mongod
14973 root      15   0 13428 2060  940 R  1.0  0.0   0:00.09 top
 4950 apache    16   0  312m  19m 3984 S  0.3  0.3   0:09.06 httpd
11839 apache    16   0  312m  20m 3984 S  0.3  0.3   0:27.41 httpd
12819 apache    16   0  312m  20m 3984 S  0.3  0.3   0:16.89 httpd
    1 root      15   0 10368  628  596 S  0.0  0.0   7:00.45 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   9:12.97 migration/0
```

From this output, take out the processes which consume more %CPU. Here, Java takes 5.9 % but it can consume more CPU such as more than 40%, 100%, 200%, 300%, 400%, and so on.

Step 2: If a Java process consumes high CPU, enter one of these commands in order to find out which thread consumes how much:

```
# ps –C java –L –o pcpu,cpu,nice,state,cputime,pid,tid | sort
```

OR

```
# ps –C <process ID> –L –o pcpu,cpu,nice,state,cputime,pid,tid | sort
```

As an example, this display shows that the Java process consumes high CPU (+40%) as well as the threads of the Java process responsible for the high utilization.

```
<snip>

0.2 – 0 S 00:17:56 28066 28692
 0.2 – 0 S 00:18:12 28111 28622
 0.4 – 0 S 00:25:02 28174 28641
 0.4 – 0 S 00:25:23 28111 28621
 0.4 – 0 S 00:25:55 28066 28691
 43.9 – 0 R 1–20:24:41 28026 30930
 44.2 – 0 R 1–20:41:12 28026 30927
 44.4 – 0 R 1–20:57:44 28026 30916
 44.7 – 0 R 1–21:14:08 28026 30915
 %CPU CPU NI S TIME     PID  TID
```

What is a thread?

Suppose you have an application (that is, a single running process) inside the system. However, in order to perform many tasks you require many processes to be created and each process creates many threads. Some of the threads could be reader, writer, and different purposes such as Call Detail Record (CDR) creation and so on.

In the previous example, the Java process ID (for example, 28026) has multiple threads which include 30915, 30916, 30927 and many more.

Note: The Thread ID (TID) is in decimal format.

Step 3: Check the functionality of the Java threads which consume the high CPU.

Enter these Linux commands in order to obtain the complete Jstack trace. Process ID is the Java PID, for example 28026 as shown in the previous output.

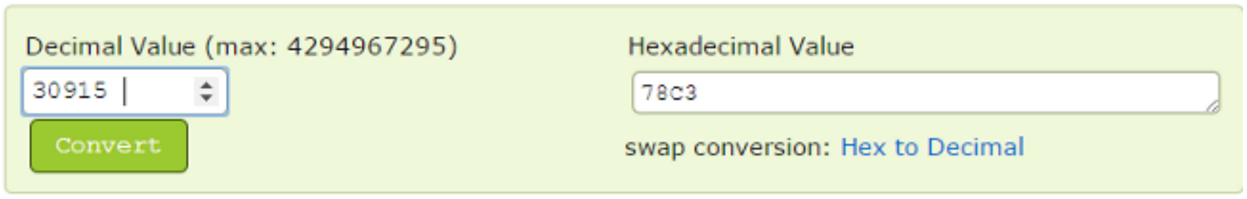
```
# cd  /usr/java/jdk1.7.0_10/bin/

# jstack <process ID>
```

Output of the previous command looks like:

```
2015-02-04 21:12:21
 Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.7-b01 mixed mode):

"Attach Listener" daemon prio=10 tid=0x000000000fb42000 nid=0xc8f waiting on
 condition [0x0000000000000000]
 java.lang.Thread.State: RUNNABLE

"ActiveMQ BrokerService[localhost] Task-4669" daemon prio=10 tid=0x00002aaab41fb800
 nid=0xb24 waiting on condition [0x000000004c9ac000]
 java.lang.Thread.State: TIMED_WAITING (parking)
 at sun.misc.Unsafe.park(Native Method)
 - parking to wait for <0x00000000c2c07298>
 (a java.util.concurrent.SynchronousQueue$TransferStack)
 at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
 at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
 at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
 at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
 at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
 at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
 at java.lang.Thread.run(Thread.java:722)

"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
 nid=0xa0f waiting on condition [0x0000000043a1d000]
 java.lang.Thread.State: TIMED_WAITING (parking)
 at sun.misc.Unsafe.park(Native Method)
 - parking to wait for <0x00000000c2c07298>
 (a java.util.concurrent.SynchronousQueue$TransferStack)
 at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
 at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
 at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
 at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
 at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
 at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
 at java.lang.Thread.run(Thread.java:722)


<snip>


"pool-84-thread-1" prio=10 tid=0x00002aaac45d8000 nid=0x78c3 runnable
 [0x000000004c1a4000]
 java.lang.Thread.State: RUNNABLE
 at sun.nio.ch.IOUtil.drain(Native Method)
 at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:92)
 - locked <0x00000000c53717d0> (a java.lang.Object)
 at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
 - locked <0x00000000c53717c0> (a sun.nio.ch.Util$2)
 - locked <0x00000000c53717b0> (a java.util.Collections$UnmodifiableSet)
 - locked <0x00000000c5371590> (a sun.nio.ch.EPollSelectorImpl)
 at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
 at zmq.Signaler.wait_event(Signaler.java:135)
 at zmq.Mailbox.recv(Mailbox.java:104)
 at zmq.SocketBase.process_commands(SocketBase.java:793)
 at zmq.SocketBase.send(SocketBase.java:635)
 at org.zeromq.ZMQ$Socket.send(ZMQ.java:1205)
 at org.zeromq.ZMQ$Socket.send(ZMQ.java:1196)
 at com.broadhop.utilities.zmq.concurrent.MessageSender.run(MessageSender.java:146)
 at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
 at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
 at java.util.concurrent.FutureTask.run(FutureTask.java:166)
 at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
 at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
 at java.lang.Thread.run(Thread.java:722)
```

Now you need to determine which thread of the Java process is responsible for the high CPU utilization.

As an example, look at TID 30915 as mentioned in Step 2. You need to convert the TID in decimal to hexadecimal format because in Jstack trace, you can only find the hexadecimal form. Use this converter in order to convert the decimal format into the hexadecimal format.



Decimal Value (max: 4294967295): 30915

Hexadecimal Value: 78C3

As you can see in Step 3, the second half of the Jstack trace is the thread which is one of the responsible threads behind the high CPU utilization. When you find the 78C3 (hexadecimal format) in the Jstack trace, then you will only find this thread as 'nid=0x78c3'. Hence, you can find all the threads of that Java process which are responsible for high CPU consumption.

Note: You do not need to focus on the state of the thread for now. As a point of interest, some states of the threads like Runnable, Blocked, Timed_Waiting, and Waiting have been seen.

All the previous information helps CPS and other technology developers assist you to get to the root cause of the high CPU utilization issue in the system/VM. Capture the previously mentioned information at the time the issue appears. Once the CPU utilization is back to normal then the threads that caused the high CPU issue cannot be determined.

CPS logs need to be captured as well. Here is the list of CPS logs from the 'PCRFclient01' VM under the path '/var/log/broadhop':

- ***consolidated−engine***
- ***consolidated−qns***

Also, obtain output of these scripts and commands from the PCRFclient01 VM:

- ***# diagnostics.sh*** (This script might not run on older versions of CPS, such as QNS 5.1 and QNS 5.2.)
- ***# df −kh***
- ***# top***