ColdFusion Memory Tracking: Real World Performance Example

Posted At : May 24, 2012 6:00 PM | Posted By : Steven Erat
Related Categories: Flex, Java, Quality Assurance, ColdFusion

It is widely known that the built-in ColdFusion Server Monitor can in many cases **cause** a CF server to become entirely unresponsive **if Memory Tracking is enabled**. I've experienced this myself when I previously consulted with customers, and I was able to *save* many clients that engaged me to resolve performance problems by identifying that they had inadvertently enabled Memory Tracking in production. I've written about this before, as have others.

A Test
However, as I am currently working on a Performance Testing project for an enormously large web application, I took this opportunity to observe and measure the impact of enabling Memory Tracking on performance. I was not at all surprised by what happened (the server became entirely unresponsive in very little time), but I was pleased that I was able to document the exact impact in a more empirical manner.

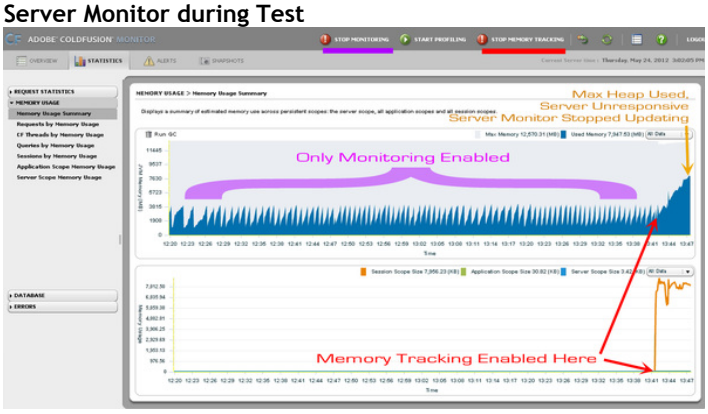
Environment
This experiment took place in a staging environment with 3 machines: One to host ColdFusion, another to host IIS, and a third to host JMeter. A performance test was created in JMeter to moderately exercise the application. It was run as a stress test by applying 100 Virtual Users indefinitely with 0 think time (no delay between a HTTP Response and the next HTTP Request). This means that at any given moment JMeter is making 100 simultaneous Requests to the CF server. The CF server is a virtualized instance with a max heap size set to about 12GB, sitting on Windows 7 with 25 GB RAM and 4 x 2.4GHz processors.

Let'r Rip!
The JMeter Test Plan was started and left to run for several hours, all the while pounding ColdFusion. During this period **Monitoring** was enabled in Server Monitor, but neither Profiling nor Memory Tracking were enabled. The CF Server's throughput was measured in Server Montitor to be steady at about 20 Requests per Second +/- 4 requests (steady range of 16-24). Memory Used by the JVM was a steady 3.9 GB at the peak followed by troughs of about 1.2 GB, with Garbage Collection happening once a minute. This created the typical sawtooth pattern when using the -XX:UseParallelGC JVM GC option. The CPU was typically in the range of 8-12% usage (across the 4 CPUs). The total throughput and memory utilization held steady for the several hours of testing. The app was performing beautifully, with 0 errors logged.

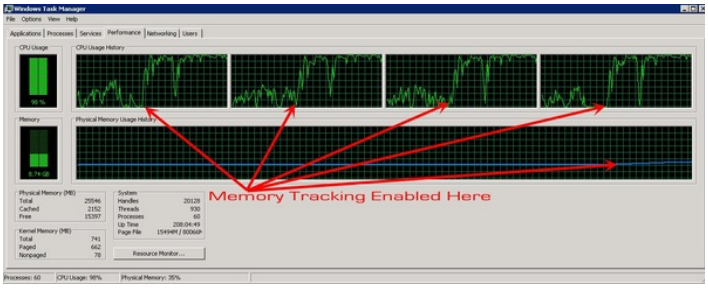
The Death Spiral
Before terminating the JMeter Test Plan execution, I enabled Memory Tracking in the Server Monitor. The JVM memory used began to quickly rise at the rate of about +700 MB per minute. The memory used jumped from a steady 4GB up to 8GB in just 6 minutes before the Server Monitor Interface stopped updating completely. Attempts to disable Memory Tracking were futile as button clicks did not respond. I could only watch the Task Manager on the CF server to continue observing memory and CPU. During the several hours of testing, total System Memory in Task Manager showed about 7.4 GB used, but after Memory Tracking was enabled and Server Monitor became unresponsive, I observed the total System Memory to be 13.7 GB, an increase of about 6 GB. The JVM was at or very close to being at the max heap allowed of 12 GB and was not able to reclaim any memory via GC. At this point I decided to kill the process tree of jrunsvc.exe (which also killed it's child process of jrun.exe, the main server instance). I stopped JMeter and then started ColdFusion again, making sure to disable Memory Tracking before running my next performance test run.

Caveat Emptor
This was a great example to see how Memory Tracking can bring a server to its knees. Often reported and well understood anecdotally, but I thought some actual screenshots of it in action would help illustrate how dangerous this setting can be. Memory Tracking can be used effectively in development and QA for debugging or troubleshooting performance issues, but only when used under small load. I should point out that sometimes Profiling is known to similarly bring down a server, but I was not able to observe any impact of the Profiling setting on this particular application as performance seemed normal when enabled during a stress test.

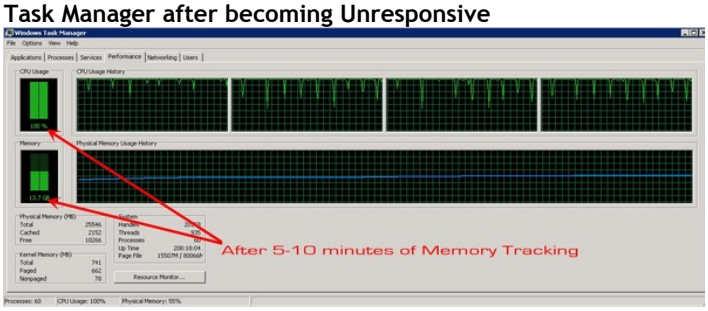
Server Monitor during Test



Task Manager during Test



Task Manager after becoming Unresponsive



Throughput is Kaput

Here's the JRun Metrics taken from just before Memory Tracking was enabled followed by a few entries from after it was turned on. You can see that in the Before case, ColdFusion was processing requests very quickly and at any instant there were only 2 or 3 requests executing, but then suddenly throughput dropped like a rock at at the instant when metrics were logged there were suddenly 25 or 26 requests being executed. More requests shown actively executing at any moment means that fewer are being completed per unit time because CF is slowing down the request processing. Take a look:

```
05/24 13:37:53 metrics Web threads (busy/total): 3/43 Sessions: 17391 Total Memory=13078784 Free=10757407
05/24 13:38:53 metrics Web threads (busy/total): 2/43 Sessions: 17415 Total Memory=13076736 Free=10369752
05/24 13:39:53 metrics Web threads (busy/total): 4/39 Sessions: 17459 Total Memory=13077760 Free=9648191
05/24 13:40:53 metrics Web threads (busy/total): 2/42 Sessions: 17468 Total Memory=13073856 Free=9046854
05/24 13:41:59 metrics Web threads (busy/total): 25/51 Sessions: 17460 Total Memory=13024064 Free=11314420
05/24 13:43:03 metrics Web threads (busy/total): 26/51 Sessions: 17437 Total Memory=12692224 Free=9693140
05/24 13:44:03 metrics Web threads (busy/total): 27/51 Sessions: 17361 Total Memory=12717312 Free=6587471
05/24 13:45:03 metrics Web threads (busy/total): 26/51 Sessions: 17324 Total Memory=12789056 Free=7823622
05/24 13:46:03 metrics Web threads (busy/total): 26/51 Sessions: 17205 Total Memory=12839040 Free=5087221
05/24 13:47:03 metrics Web threads (busy/total): 26/51 Sessions: 17155 Total Memory=12853120 Free=4153407
```

I'll also point out that until then, the CF and JRun logs were clean as a whistle, and then suddenly everything started timing out. Even the requests to the Server Monitor began throwing errors (Flex messaging Errors):

```
05/24 13:45:18 Error [jrpp-54] -
[BlazeDS]Exception when invoking service 'remoting-service': flex.messaging.MessageException: java.util.ConcurrentModificationException
  incomingMessage: Flex Message (flex.messaging.messages.RemotingMessage)
    operation = getCurrentReportsSize
    clientId = 7E604365-BE09-678D-3859-681EE43C4506
    destination = ColdFusion
    messageId = 1ABD7006-AC8D-6953-B48E-802AC5DBDC18
    timestamp = 1337885118569
    timeToLive = 0
    body =
    [
    ]
    hdr(DSRequestTimeout) = 30
    hdr(DSId) = 7E6042CB-EF0E-FC59-6E15-EFB4A6533288
    hdr(DSEndpoint) = my-cfamf
  Exception: flex.messaging.MessageException: java.util.ConcurrentModificationException
                at coldfusion.flash.messaging.ColdFusionAdapter.getFlexError(ColdFusionAdapter.java:457)
                at coldfusion.flash.messaging.ColdFusionAdapter.invoke(ColdFusionAdapter.java:351)
                at flex.messaging.services.RemotingService.serviceMessage(RemotingService.java:183)
                at flex.messaging.MessageBroker.routeMessageToService(MessageBroker.java:1400)
                at flex.messaging.endpoints.AbstractEndpoint.serviceMessage(AbstractEndpoint.java:1023)
                at flex.messaging.endpoints.amf.MessageBrokerFilter.invoke(MessageBrokerFilter.java:103)
                at flex.messaging.endpoints.amf.LegacyFilter.invoke(LegacyFilter.java:158)
                at flex.messaging.endpoints.amf.SessionFilter.invoke(SessionFilter.java:44)
                at flex.messaging.endpoints.amf.BatchProcessFilter.invoke(BatchProcessFilter.java:67)
```

```
at flex.messaging.endpoints.amf.SerializationFilter.invoke(SerializationFilter.java:166)
at flex.messaging.endpoints.BaseHTTPEndpoint.service(BaseHTTPEndpoint.java:291)
at coldfusion.flash.messaging.CFAMFEndPoint.service(CFAMFEndPoint.java:295)
at flex.messaging.MessageBrokerServlet.service(MessageBrokerServlet.java:353)
at coldfusion.flex.ColdFusionMessageBrokerServlet.service (ColdFusionMessageBrokerServlet.java:114)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at coldfusion.bootstrap.BootstrapServlet.service(BootstrapServlet.java:89)
at jrun.servlet.FilterChain.doFilter(FilterChain.java:86)
at coldfusion.filter.FlashRequestControlFilter.doFilter (FlashRequestControlFilter.java:71)
at coldfusion.bootstrap.BootstrapFilter.doFilter(BootstrapFilter.java:46)
at jrun.servlet.FilterChain.doFilter(FilterChain.java:94)
at jrun.servlet.FilterChain.service(FilterChain.java:101)
at jrun.servlet.ServletInvoker.invoke(ServletInvoker.java:106)
at jrun.servlet.JRunInvokerChain.invokeNext(JRunInvokerChain.java:42)
at jrun.servlet.JRunRequestDispatcher.invoke(JRunRequestDispatcher.java:286)
at jrun.servlet.ServletEngineService.dispatch(ServletEngineService.java:543)
at jrun.servlet.jrpp.JRunProxyService.invokeRunnable(JRunProxyService.java:203)
at jrunx.scheduler.ThreadPool$DownstreamMetrics.invokeRunnable(ThreadPool.java:320)
at jrunx.scheduler.ThreadPool$ThreadThrottle.invokeRunnable(ThreadPool.java:428)
at jrunx.scheduler.ThreadPool$UpstreamMetrics.invokeRunnable(ThreadPool.java:266)
at jrunx.scheduler.WorkerThread.run(WorkerThread.java:66)
```