SAS4236-2020

Delay Analysis in the SAS® Middle Tier

Bob Celestino, SAS Institute Inc.

ABSTRACT

Investigation of performance delays in large distributed software environments like SAS® 9.4 can be quite complex. This paper reviews the aspects of the SAS environment that make it so challenging to analyze. The paper goes on to explain some of the techniques that SAS Technical Support uses to diagnose and solve these issues. For example, the discussion demonstrates how to use log analysis to rule in, or rule out, contributors to the source of the problem. In addition, you will discover how to use high-frequency thread dumps to identify specific problematic components. The discussion also covers how you can use network analysis to measure and identify specific inter-process delays in specific components. Case studies illustrate all the techniques that are presented.

This paper is aimed at system administrators, network engineers, and others who might be involved in the investigation of performance issues on the SAS middle tier.

INTRODUCTION

Distributed software systems such as SAS® 9.4 contain many heterogeneous components that are spread over a variety of hardware. These components are deployed within individual machines, distributed across machines (tiered and clustered), and, frequently, across networks and data centers. Delays that users experience in software systems like these are difficult to pinpoint. The sheer number of components and connections that are in effect for any given user interaction make uncovering the source of a delay nearly intractable. To help you in this endeavor, this paper discusses techniques that you can use to investigate the source of a delay so that it can be understood and remedied.

Consider this typical highly distributed system.

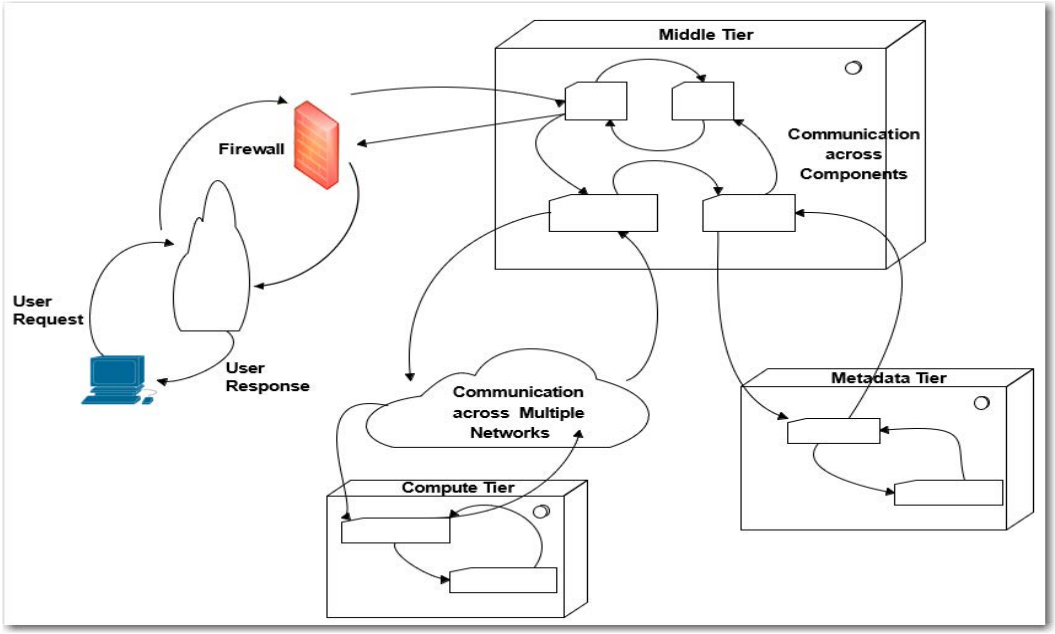


Figure 1. A Heterogeneous Distributed System

In Figure 1, think of the small rectangles as software components. These components can be Java processes, SAS workspace servers, stored-process servers, metadata servers, supporting utilities, databases, and more.

The arrows represent communication among components. Communication occurs via a variety of network protocols, including HTTP, HTTPS, RMI, CORBA, JDBC, and more. The communication occurs within a single machine, as well as across machines and across networks. Most production environments are clustered, which multiplies the components that are shown above by a factor of two or three.

DELAY CONTRIBUTORS

A *delay contributor* is any element that adds time to the overall delay. The goal of *delay analysis* is to identify the elements that contribute most heavily to the overall delay.

For example, perhaps your database is underperforming. Or, one of the SAS web application servers is tuned improperly and it stalls during garbage collection. Or maybe the network between the middle tier and the metadata tier is sluggish. Any *one* of these issues present with the same general symptom: a delay that is experienced by the user.

In Figure 1, each component and each communication path are delay contributors. The total delay that you experience is the sum of all delay contributors in every element that is in effect for the given request. The set of these elements is called the *solution space*. When you start your delay analysis, the solution space is overwhelmingly large.

TRADITIONAL APPROACHES

The approach to performance problems like the ones that are described earlier is often haphazard. The solution space is so large that it might seem easy to fall back on familiar, though less-than-analytic, approaches:

Best guess – Faced with such a large solution space, it might seem easier to make a best guess and see how that works. Everyone has seen cases where making your best guess is the first attempt (for example, increasing the memory, upgrading the operating system (OS), replacing the hard drive, and so on.

This worked before – When the symptom is generic, it is natural to try things that worked in the past. For example, in the past, you might have experienced slow logon times. So you increased the send buffer in the TCP stack on the machine, which resolved the problem. Since this solution worked in the past, you naturally want to try that now.

A wide approach – When users are overwhelmed by possibilities, it is common to try several changes at once in the hopes that one will work. So, you might consider performing several tasks to take care of the problem: updating the OS, increasing the heap settings on each Java virtual machine (JVM), switching from HTTPS to HTTP, and doubling the number of CPUs on the compute tier.

Even though complex environments like the one illustrated in Figure 1 have very large solution spaces, you should take a measured and systematic approach. This paper helps you with this by presenting ways to narrow down the solution space to make the analysis less daunting.

DELAY ANALYSIS VERSUS PERFORMANCE ANALYSIS

Traditional performance analysis looks at individual components and how they perform relative to norms. Traditional performance analysis considers *one delay contributor alone*. In delay analysis, you need to reduce the solution space and focus on only the significant contributors to the delay at hand.

For example, perhaps you suspect a problem with your I/O subsystem or memory management. In UNIX operating environments, you can use **`top`**, **`free`**, **`vmstat`**, and other commands to investigate, as shown in the following output examples.

```
-bash-4.2$ free -mt
              total        used        free      shared  buff/cache   available
Mem:         257784        2527      194770        2242       60486      252056
Swap:          4999           0        4999
Total:       262784        2527      199770
```

Output 1. UNIX Output That Is Generated by the `free -mt` Command

```
-bash-4.2$ vmstat -s
    263971808 K total memory
      2588664 K used memory
     54263568 K active memory
      7198736 K inactive memory
    199444256 K free memory
       245892 K buffer memory
     61693000 K swap cache
      5119996 K total swap
            0 K used swap
      5119996 K free swap
      8914869 non-nice user cpu ticks
         2461 nice user cpu ticks
      4923557 system cpu ticks
   3670354710 idle cpu ticks
       327530 IO-wait cpu ticks
            0 IRQ cpu ticks
        54290 softirq cpu ticks
            0 stolen cpu ticks
     30676361 pages paged in
    348411788 pages paged out
            0 pages swapped in
            0 pages swapped out
    784509341 interrupts
    691544097 CPU context switches
   1576865083 boot time
      9481789 forks
```

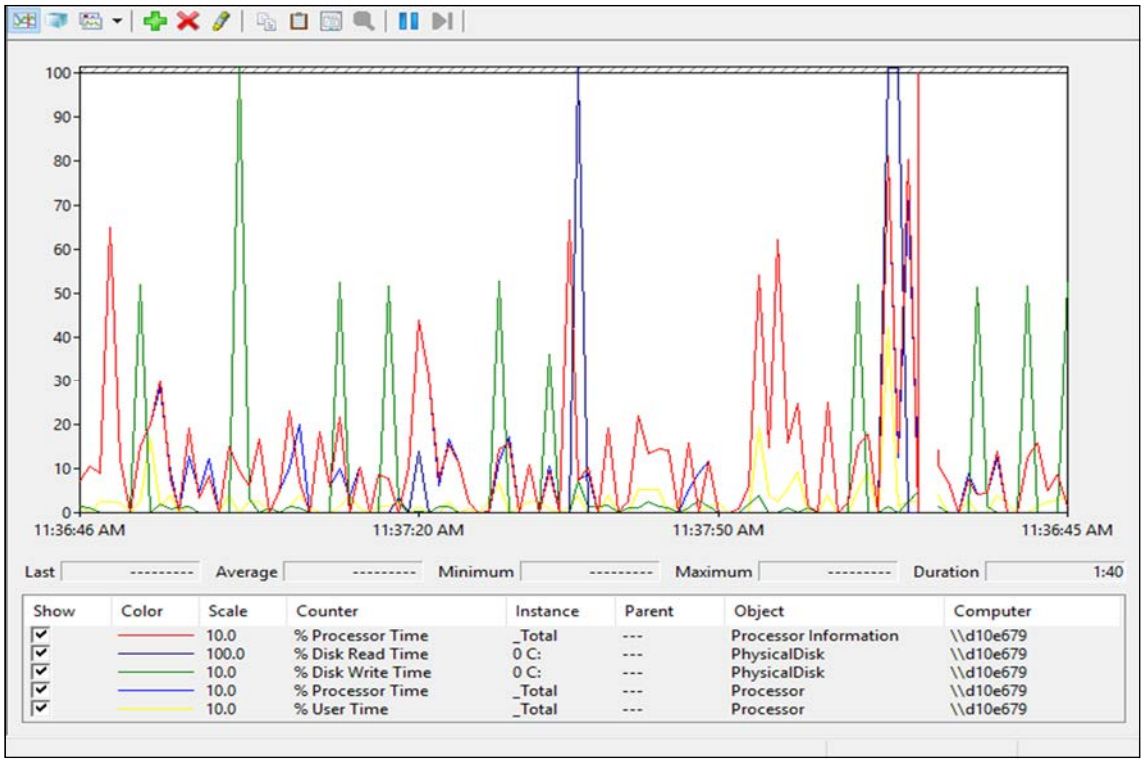
Output 2. UNIX Output That Is Generated by the `vmstat -s` Command

```
top - 21:01:22 up 13 days,  7:56,  2 users,  load average: 0.22, 0.09, 0.06
Tasks: 502 total,   1 running, 501 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.3 us,  0.2 sy,  0.0 ni, 99.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 26397180+total, 19822420+free,  3792496 used, 61955112 buff/cache
KiB Swap:  5119996 total,  5119996 free,        0 used. 25689745+avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
26810 rocele    20   0  143.8g 981220  45388 S   1.0  0.4   0:30.19 java
15077 mysql     20   0 3919000 292044   9612 S   0.7  0.1  82:44.70 mysqld
26194 rocele    20   0  172728   2736   1640 R   0.7  0.0   0:00.49 top
27258 nagios    20   0  444936  25792  10504 S   0.7  0.0   0:00.19 php
    1 root      20   0  193520   6444   2604 S   0.3  0.0  35:25.65 systemd
 7113 root      20   0  118372  59568  58500 S   0.3  0.0  21:31.79 systemd-journal
13942 root      20   0  246408   6756   5248 S   0.3  0.0  19:35.83 vmtoolsd
14021 root      20   0  311516  10732   7504 S   0.3  0.0  20:49.31 sssd_be
14032 root      20   0  245220   4592   3380 S   0.3  0.0  12:11.94 sssd_pam
14061 root      20   0   36936   2156   1524 S   0.3  0.0  10:58.95 systemd-logind
16471 nagios    20   0   91760   3276   1620 S   0.3  0.0   4:28.37 nagios
16740 gdm       20   0  634644  18892   9160 S   0.3  0.0  35:04.01 gsd-color
24487 rocele    20   0  188544   3236   1388 S   0.3  0.0   0:01.73 sshd
26130 rocele    20   0  188412   2896   1384 S   0.3  0.0   0:00.02 sshd
27122 rocele    20   0 7151396 204652  15876 S   0.3  0.1   0:10.99 java
    2 root      20   0       0      0      0 S   0.0  0.0   0:01.24 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:01.22 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    6 root      20   0       0      0      0 S   0.0  0.0   0:00.10 kworker/u64:0
    8 root      rt   0       0      0      0 S   0.0  0.0   0:00.90 migration/0
    9 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
   10 root      20   0       0      0      0 S   0.0  0.0   6:03.48 rcu_sched
   11 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 lru-add-drain
   12 root      rt   0       0      0      0 S   0.0  0.0   0:06.37 watchdog/0
```

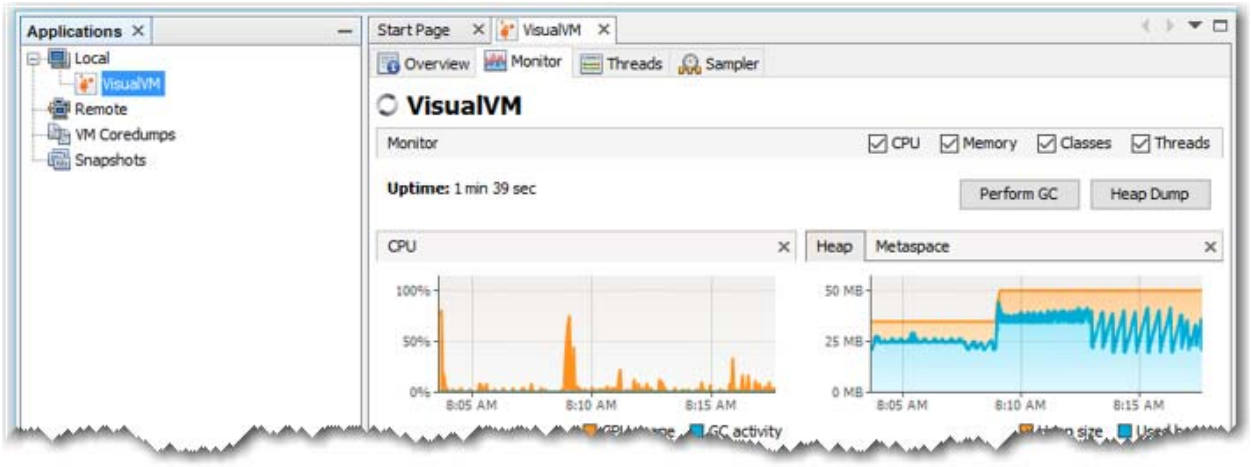
Output 3. UNIX Output That Is Generated by the `top` Command

In Microsoft Windows operating environments, a popular choice for investigating system performance is the Windows Performance Monitor (via the perfmon command).



Output 4. Example Output from the Microsoft Windows Performance Monitor

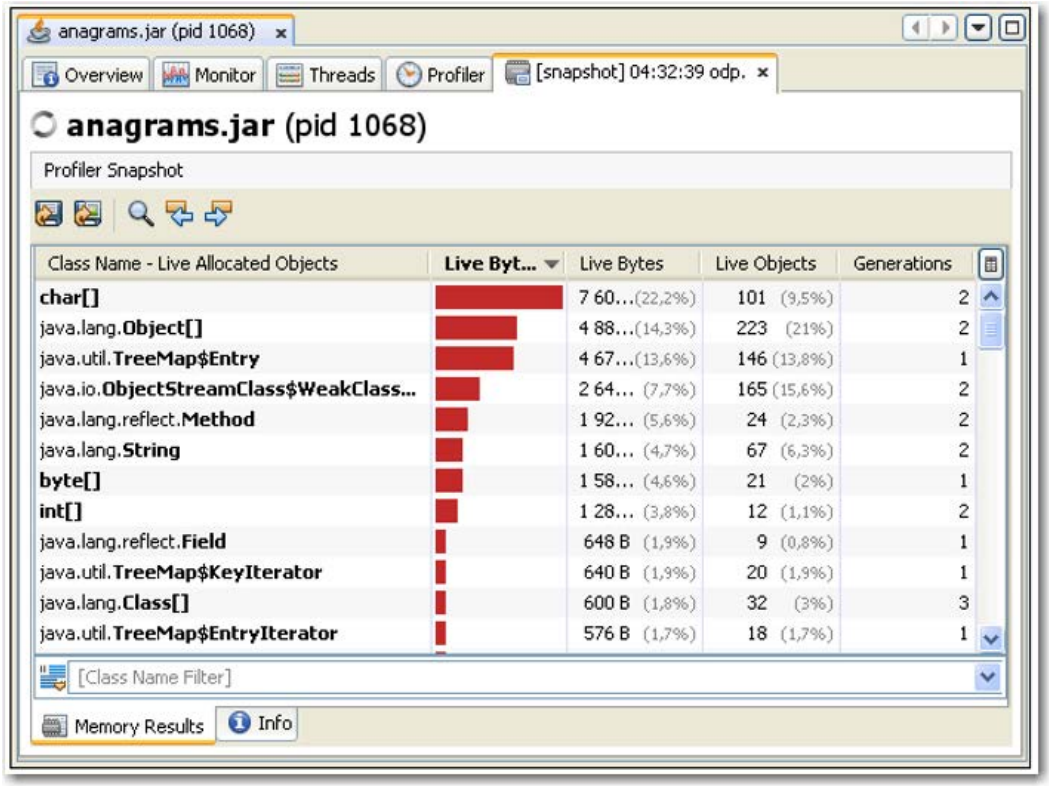
Perhaps you have a performance problem with a Java application that you are developing. In that case, the open-source tool VisualVM provides monitoring and profiling that will help you diagnose the issue.



Display 1. Java Monitoring with Oracle VisualVM

Display 1 shows how you can use VisualVM to monitor Java heap size and garbage-collection duration. Display 2 illustrates the use VisualVM to profile the CPU and memory usages of the application. VisualVM measures and reports performance data down to the

method level This tool is invaluable when you are investigating the performance of Java applications.



Display 2. Java Profiling with VisualVM

LIMITATIONS OF THESE APPROACHES

These traditional performance-analysis techniques are appropriate when you are analyzing a single component. If you happen to know the performance culprit, these tools and techniques are helpful.

The traditional approaches fall short when many components contribute to the delay. In addition, it becomes exponentially more difficult when the components are clustered and distributed across the network. In such cases, you need to use delay analysis where you systematically narrow the problem area and identify the primary contributors to the delay. After you identify the primary contributors to the delay, you can use more traditional performance analysis to examine these individual elements.

PERFORMANCE CULPRIT VERSUS PERFORMANCE VICTIM

Frequently, a high-level component experiences a delay as it waits for resources at a lower level. Often, the higher-level component is more visible and surfaces early in the analysis. However, since that higher-level component is waiting for a subordinate process to complete, it is, in fact, the *victim* of another delay. The visible, higher-level delay is the *proximal cause*. The *root cause* of the delay is the lower level.

Suppose that you are investigating a delay when you open a SAS® Visual Analytics report. After some careful analysis, you eliminate the SAS middle-tier components altogether. You also confirm that the metadata tier is responding well and does not contribute to the delay

at all. Then, you discover that the delay is a result of a very long-running query to the Hadoop data cluster on another machine.

That impressive sleuthing eliminated great swaths of the environment, and you identified a primary delay contributor. But you are not done yet! You need to drill down into the Hadoop data cluster to discover why that query is running so slowly. Perhaps the I/O performance on that machine has degraded, causing slow performance for the queries. The network connection between the middle tier and the Hadoop cluster might be compromised in some way, too, which means that the delay occurs in the network itself. So, the slow query that you identified might be, in fact, a victim of other components that are involved.

LEVERAGING THE SAS® MIDDLE TIER

Generally, the middle tier of any enterprise system is the hub through which all requests and all responses pass. This fact is certainly true of the SAS 9.4 middle tier. This behavior gives you an opportunity to use the middle tier as a performance monitor, where you can measure the delays between requests and responses across the entire environment.

Using the SAS middle tier in this way enables you to eliminate large portions of the solution space. Consider the benefit of being able to quickly reduce the solution space from *everything* to a narrower scope (for example, one of four components on the compute tier).

The next section considers some of the middle-tier components that you can leverage to help in your delay analysis.

SAS® Web Server

You can configure SAS environments with the SAS Web Server as the primary entry point into the environment. In this scenario, the web server acts as a load balancer and handles virtually all traffic to and from the environment. Alternatively, you can configure an external load balancer as the primary entry point. Either way, the logging from this component gives you a high-level view of which requests are significant contributors to the delay.

Note that SAS Web Server does have a logging peculiarity. The web server logs all requests with a timestamp. The timestamp in the entry is the time that the request arrives at the web server. However, the web server writes the log entry after the request has been satisfied. So, if requests take unusually long to complete, you see log entries that appear out of order. However, you can use this logging behavior to your advantage when you start the delay analysis.

Out-of-order entries in the log are the first indication of a significant delay in a request. Consider this excerpt from a fictional log:

```
1.2.3.4 - - [16/Dec/2019:13:22:46 +1100] "POST /cas/v1/tickets/TGT-4830717-K5FWqTeJfIajdbae0Zbc1hqS…. HTTP/1.1" 200 60
1.2.3.4 - - [16/Dec/2019:13:12:42 +1100] "POST /self-service/api/account/verification/residential HTTP/1.1" 400 0
1.2.3.4 - - [16/Dec/2019:13:12:15 +1100] "GET  /self-service/api/accounts/38909 HTTP/1.1" 404 0
1.2.3.4 - - [16/Dec/2019:13:20:46 +1100] "GET  /self-service/oamCallback?authn_try_count=1& ….  HTTP/1.1" 401 70
1.2.3.4 - - [16/Dec/2019:13:22:46 +1100] "GET  /cas/proxyValidate?&ticket=ST-483071 …  HTTP/1.1" 200 704
```

The highlighted requests are suspicious because they are significantly delayed. The first two ran about eight minutes, and the last one ran about two minutes. As a result, you can confidently eliminate most of the environment and focus only on the ***/self-service/*** components.

Web Server Logging Configuration

That logging quirk is helpful and is available for free. However, although this behavior is quite effective at discovering delayed requests, it is still an approximation. A more accurate

measure of the duration of each request is directly available by adding the %D option in the configuration files to display the milliseconds that each request took. For the SAS Web Server, the configuration file (httpd.conf) is available in the ***SAS-configuration-directory*/Lev*N*/Web/WebServer/conf/** directory.

When you open the httpd.conf file, locate the follow line:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referrer}i\" \"%{User-Agent}i\"" /
combined
```

Add %D ms to the line, as shown here:

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referrer}i\" \"%{User-Agent}i\" /
%D ms" combined
```

If you configure your web server for the Secure Sockets Layer (SSL) protocol, the configuration file name is httpd-ssl.conf. This file is also located in **. . ./Web/WebServer/conf/**.

Add %D ms to the following line in httpd-ssl.conf:

```
CustomLog "|/sas94m6/SASHome/SASWebServer/9.4/httpd-2.4/bin/rotatelogs \
-l /sa94m6/config/Lev1/Web/WebServer/logs/ssl_request_%Y-%m-%d-%H.%M.log \
50M" %t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %>s %b %D ms"
```

These log-format changes provide the time it took, in milliseconds, to satisfy the request. If you use other web servers or a hardware-based load balancer, similar configuration changes are available. Whichever of these methods that you choose will be a good starting point in your delay analysis.

HIGH-FREQUENCY THREAD ANALYSIS

The SAS Web Application Servers are a good place to capture a wide variety of interactions across the entire environment. These components are Java based, and you can trigger thread dumps from them. A single thread dump provides a great deal of information, essentially a snapshot of what happened at that *moment in time*.

To gain a better picture of the dynamics of the system, you can trigger a series of thread dumps at a set interval. These high-frequency thread dumps provide dynamic data about the system just as a video gives you more information than a single photo. Analyzing the high-frequency thread dumps provides you with a picture of which tasks are quick and which are slow.

You should choose thread-dump frequency based on the delay that you are investigating. If the interval between thread dumps is too large, you will miss the delay entirely If it is too small, you risk being overwhelmed by the amount of data.

Before moving into the details, you should understand the terms delay resolution and delay uncertainty.

Delay resolution: *Delay resolution* is the smallest delay that is detectable with a 100% confidence. The resolution is a function of the thread-dump interval:

DelayResolution=ThreadDumpInterval * 2

Smaller intervals yield smaller resolutions. But remember, the smaller the interval, the more data that you generate. And that means more data that you eventually have to

investigate. Consider a thread-dump interval of 10 seconds. With this interval, you generate six thread dumps per minute. With a delay resolution of 20 seconds, you are guaranteed to detect any delays of 20 seconds or greater. A thread-dump interval of 1 second generates *60 thread dumps per minute*. At this rate, you are guaranteed to detect a delay of 2 seconds or greater, but you will have far more data to analyze. Considering that a single thread dump can have 1000 threads, this interval results in 60,000 threads per minute! That is a lot of data to analyze, so be sure to choose your interval appropriately.

Delay uncertainty: The thread-dump interval also affects how precisely you can measure the duration of a given delay. You cannot know the exact duration of the delay with this method, but you can compute its upper and lower bounds, as shown below:

ThreadDumpInterval*(n-1) < DelayDuration < ThreadDumpInterval*(n+1)

In this formula, n specifies the number of thread dumps in which the delay is observed.

Consider Table 1 below, which is a representation of a set of high-frequency thread dumps. For easier reading, the table shows only three threads per thread dump. The table also summarizes the thread stack rather than showing the full stack. These threads were triggered with an *interval of 5 seconds*. So, the d*elay resolution is 10 seconds.*

Time	Sample of Threads	Notes
11:27:02	ApplicationThreadPool_thread 73 ... Idle ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	All threads are idle.
11:27:07	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Wait for HTTP response	• A JDBC request was made. • An HTTP request was made.
11:27:12	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	• The JDBC request has not completed. • The HTTP requ **completed**.
11:27:17	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Wait for SOAP response ApplicationThreadPool_thread 75 ... Idle	• The JDBC request has not completed. • A SOAP request was made.
11:27:22	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Wait for SOAP response ApplicationThreadPool_thread 75 ... Idle	• The JDBC request has not completed. • The SOAP request has not completed.
11:27:27	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	• The JDBC request has not completed. • The SOAP request **has completed**.
11:27:32	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Wait for Workspace server response ApplicationThreadPool_thread 75 ... Idle	The JDBC request has not completed.
11:27:37	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	The JDBC request has not completed.

(table continued)

(table continued)

11:27:42	ApplicationThreadPool_thread 73 ... Wait for JDBC response ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	The JDBC request has not completed.
11:27:47	ApplicationThreadPool_thread 73 ... Idle ApplicationThreadPool_thread 74 ... Idle ApplicationThreadPool_thread 75 ... Idle	The JDBC request **has completed**.

Table 1. High-Frequency Thread Dumps

With a 5-second interval:

- The delay resolution is 10 seconds.
- Total delay uncertainty is 10 seconds.

The JDBC request is observed in eight thread dumps. To compute the bounds on the duration of the request, use this formula:

ThreadDumpInterval*(n–1) < DelayDuration < ThreadDumpInterval*(n+1)
5 sec*(8 – 1) < DelayDuration < 5 sec*(8 + 1)
35 sec < DelayDuration < 45 sec

The JDBC request is the largest delay contributor, with a duration of 35-45 seconds. Similarly, the SOAP request is observed in two thread dumps. The duration for that request is 5-15 seconds. And the HTTP request, which appears in only one thread dump, has a duration of less than 10 seconds.

It becomes clear that as you reduce the thread-dump interval, you can improve the resolution and detect smaller delays. You also have the benefit of obtaining a better estimate of the delay duration. Dropping from an interval of 5 seconds to 1 second reduces the uncertainty from 10 seconds to 2 seconds. If necessary, you can even drop the interval can drop to 1/10th of a second.

You need to choose your interval wisely. The intent of high-frequency thread analysis is to identify a delay rather than to measure accurately the duration of a delay. So, choose the largest interval that enables you to detect the delay in question. achieve an appropriate resolution. For example, if you are investigating a 1-minute delay, an interval of 20 seconds or so is reasonable. If you are investigating a delay of 6 seconds, a 2- or 3-second interval is more appropriate.

NETWORK ANALYSIS

When your investigation leads you to components that are not Java components, thread dumps become less effective. In this case, you should use network capture and analysis to examine parts of the system that are delayed.

Again, the middle tier is a good place to start. Because virtually all interactions pass through the middle tier, it is the ideal place to capture network traffic.

Consider the following diagram.

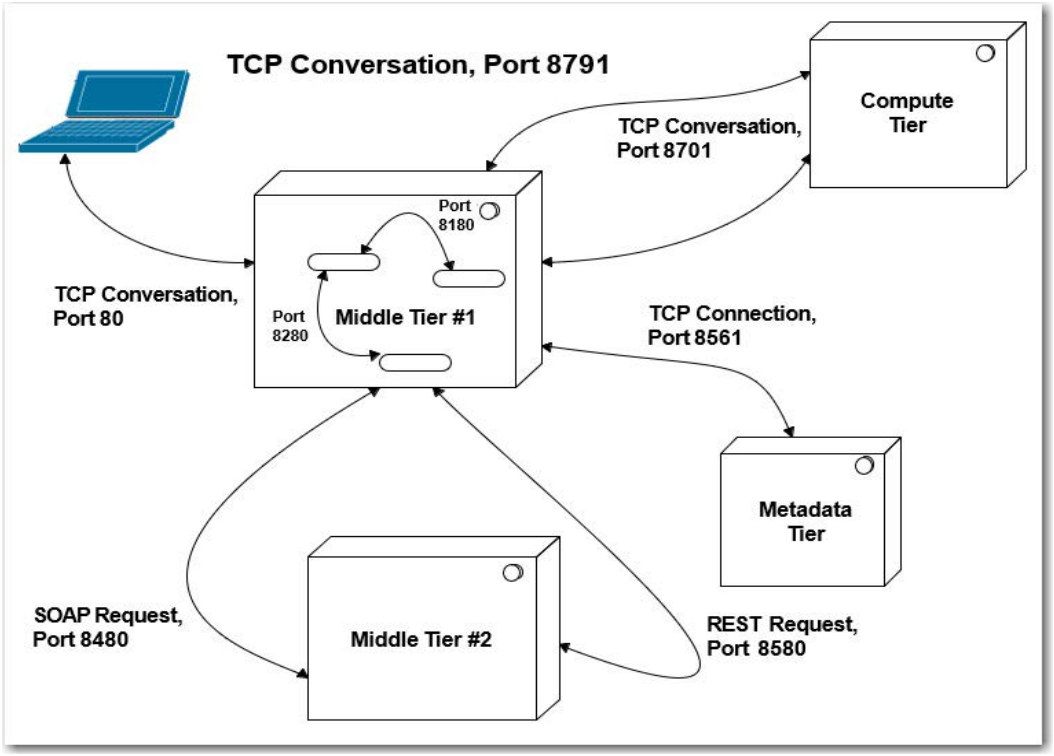


Figure 2. Network Conversations

Each arrow is a network *conversation* between two components. A conversation is defined as all the traffic between two *endpoints*. Endpoints are defined differently based on the context of the communication. For the purpose of delay analysis, you are primarily interested in TCP conversations. In a TCP conversation the endpoint is defined as the IP address and the port of the source or destination.

In Figure 2notice that capturing network activity on **Middle Tier #1** provides access to every conversation in the environment. With that information, you can compute the delay involved in each one.

CAPTURING TRAFFIC

In UNIX operating environments, you can use standard utilities such as *tcpdump* under Linux and AIX, or *snoop* under Solaris, to capture network activity. Generally, these tools are included as part of most common OS distributions. In Windows environments, you can install the open-source Wireshark network protocol analyzer to capture network traffic. For details about how to use these commands and tools, see SAS Note 53780, "Capturing network traffic in order to diagnose problems with your SAS® environment."

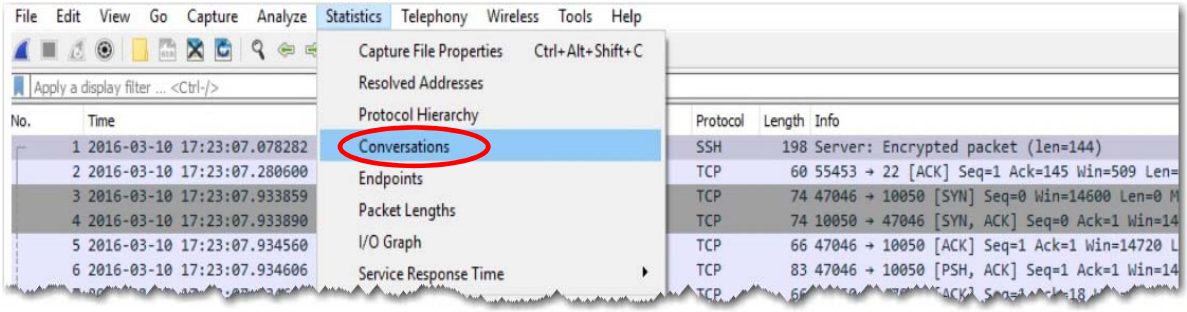
Because network data can grow very large, very quickly, it is a good guideline to limit the total time that you capture to about 10 minutes. The shorter the better, but 10 minutes seems to be a good upper bound. Much longer than that and the data becomes too unwieldy to analyze in a practical matter.

TRAFFIC ANALYSIS

After the data is captured, you need to inspect and analyze it. Network data is captured at the packet level, and you will want to use tools that can assemble these packets into the appropriate communication streams.

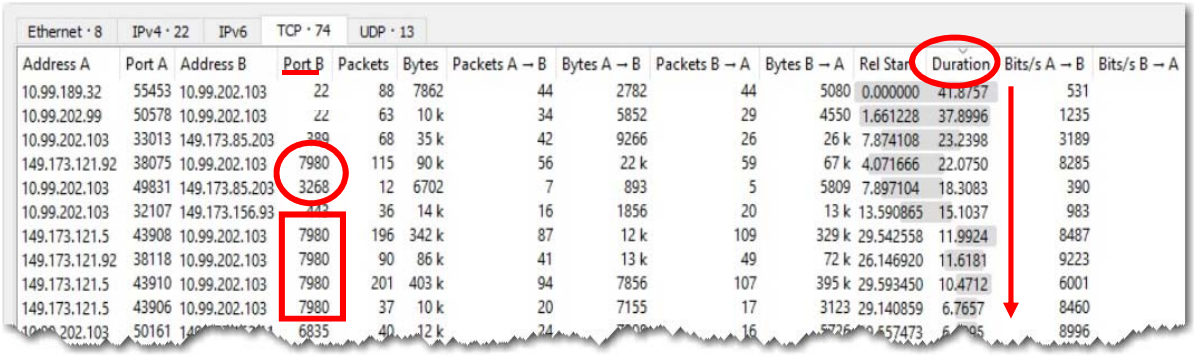
Wireshark, mentioned previously, is the analysis tool of choice. Wireshark has many capabilities. However, this paper focuses on the tool's ability to parse network data into individual conversations.

Start by viewing the list of conversations in the file:



Display 3. Using Wireshark to List Conversations

Conversations are displayed in tables, as shown in Display 4. When you investigate delays, sort conversations by duration:

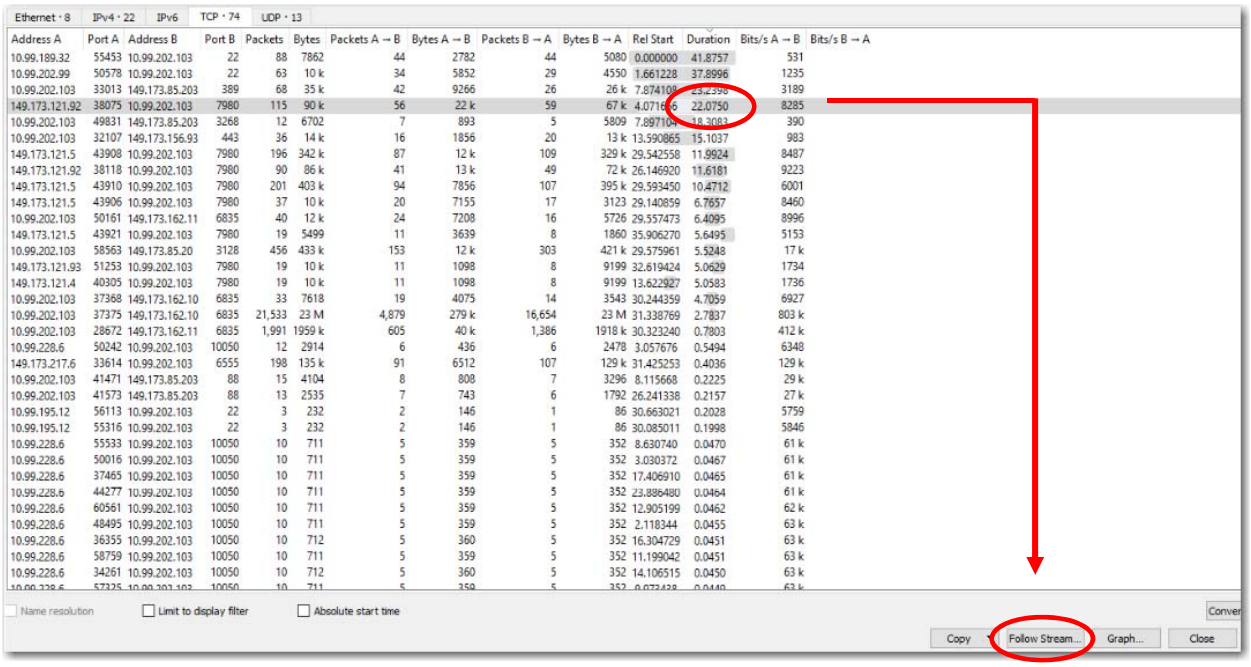


Display 4. Using Wireshark to Find the Longest Conversations

All network captures contain a lot of information--some useful, some not. In Display 4, you can see that there are few very long-duration conversations. You can disregard the first two conversations (which are 41 and 37 seconds in length) because they are on port 22. This port is the secure-shell (SSH) port, and these particular conversations represent a remote session that is used to log on to the server. There are plenty of other conversations that you can disregard as well.

But there are a few interesting conversations in the example that is shown in Display 4. Notice the conversations that are greater than 10 seconds on port 7980. 7980 is the web-server port. So, those conversations are requests to the web server. One of the conversations is just over 22 seconds.

Use the **Follow Stream** option in the UI to drill into that conversation, as shown below:

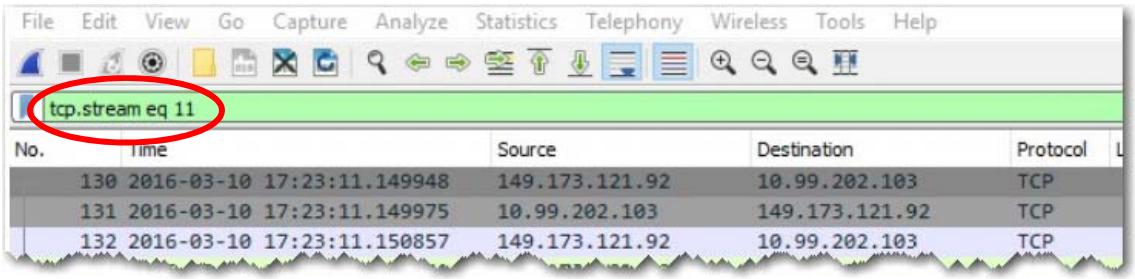


Display 5. Using the Follow Stream Option to Drill into a Conversation

With the output from the follow stream, you can see that this conversation is a request to the SAS portal:

```
GET /SASPortal HTTP/1.1
Host: esrsffdev.ondemand.sas.com
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/48.0.2564.116 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

You can filter on that stream by submitting the command **`tcp.stream eq 11`** in Wireshark, as shown below:



Display 6. Filtering a Stream in Wireshark

The following is a heavily redacted summary of the conversation (requests are in red; responses are in blue). You can use the following summary to follow the progress of the conversation:

```
GET /SASPortal HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:11 GMT
GET /SASPortal/ HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:11 GMT
GET
/SASLogon/login?service=http...%2FSASPortal%2Fj_spring_cas_security_chec
k HTTP/1.1
HTTP/1.1 200 OK
Date: Thu, 10 Mar 2016 22:23:11 GMT
POST
/SASLogon/login?service=https%3…FSASPortal%2Fj_spring_cas_security_check
HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:15 GMT
GET /SASPortal/j_spring_cas_security_check?ticket=ST-999-
lHNa0…jpBwQrDzu-cas HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:15 GMT
GET /SASPortal/ HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:15 GMT
GET /SASPortal/main.do HTTP/1.1
HTTP/1.1 200 OK
Date: Thu, 10 Mar 2016 22:23:15 GMT
GET /SASPortal/logoff.do HTTP/1.
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:20 GMT
GET /SASPortal/Logoff?_locale=en_US HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:20 GMT
GET
/SASLogon/logout?_locale=en_US&_sasapp=Information+Delivery+Portal+4.4
HTTP/1.1
HTTP/1.1 200 OK
Date: Thu, 10 Mar 2016 22:23:20 GMT
GET /SASPortal HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:22 GMT
GET /SASPortal/ HTTP/1.1
HTTP/1.1 302 Found
Date: Thu, 10 Mar 2016 22:23:22 GM
GET
/SASLogon/login?service=https%3A%2F%2Fe….%2FSASPortal%2Fj_spring_cas_sec
urity_check HTTP/1.1
HTTP/1.1 200 OK
Date: Thu, 10 Mar 2016 22:23:22 GMT
```

The full conversation has more details about each request and response. These details include headers, error states, and precise timings. For the purposes of the delay analysis, you should focus on the time that each request/response pair takes. Once you summarize the conversation, you can readily see which specific request is consuming the most time.

CASE STUDIES

This section presents two case studies that SAS Technical Support has encountered.

- Case 1: A sudden increase in the amount of time that is required to log on
- Case 2: Performance degradation (over time) for the amount of time that is required to open a particular report
- Case 3: A periodic slowdown occurs, which recovers without intervention

CASE 1: A SUDDEN INCREASE IN THE AMOUNT OF TIME THAT IS REQUIRED TO LOG ON

Users reported that the time that is required to log on used to be 5-10 seconds. However, that time increased to 1 minute or more. The system administrators for those users restarted their environment a number of times, but the delay persisted. The system administrators and SAS administrators also reported that nothing had changed in their system that might have caused this issue.

Characterize the Delay

SAS Technical Support (specifically, the author of this paper) contacted the customer by phone and discussed the symptoms. Logging on was slow for all users, all the time. Restarting the system did not make any change to the symptom. The customer was asked whether they had restarted the middle tier only or whether they restarted the entire SAS stack. They had tried both solutions over the previous few days, but there was no change.

The customer's hardware environment was quite complex. They have a horizontally and vertically clustered, three-node middle tier, a clustered three-node metadata tier, and a full, grid-based compute tier. The customer expressed dismay that such a robust hardware environment was performing so slowly.

Leverage the Middle Tier

The web-server logs confirmed that the logon process was delayed by approximately one minute. The logon process required a number of request-and-response pairs to the web application server. In addition, it was visually evident that the requests to the web application server were delayed, requiring about one minute to complete. All of the other request/response pairs in the log completed more quickly.

On the middle tier, the logon functionality is deployed exclusively onto SASServer1_1. So, all of the web application servers except SASServer1_1 could be ruled out. This environment had three instances of SASServer1_1, one on each middle-tier node. To make the analysis more manageable, the customer was asked to stop all but one instance of SASServer1_1 for about an hour so that Technical Support could collect data from that server.

Pinpoint the Problem Source: Delay Culprit or Victim?

At this point, I had ruled out the client machine, the network between the client and middle tier, the web server, and all web application servers but one. These components are all part

of the solution space. So, a large portion of that space was eliminated. However, there was more investigation to be done.

I knew that the delay occurred in the request to SASServer1_1. But I was not sure whether SASServer1_1 was the culprit or the victim of some other delay because that server also performs some other tasks (including local processing and making a few external calls to other tiers).

Use Thread-Dump Analysis to Detect the Delay

To help detect the delay, I triggered high-frequency thread dumps from SASServer1_1 during the logon process. With a 30-second trigger interval, I knew the thread dump could detect and identify a 1-minute delay. However, a closer look at the web-server log showed that the one-minute delay resulted from two separate requests to SASServer1_1. One request took 30 seconds; the other took 95 seconds.

So, I used a thread-dump interval of 10 seconds. This interval yielded a delay resolution of 20 seconds, which was enough to confidently identify both delays.

Thread-dump analysis confirmed that SASServer1_1 was waiting for the metadata once to look up the user and then again to authenticate the user. So, SASServer1_1 was indeed a victim. As a result, SASServer1_1 and, in fact, the entire middle tier, could be ruled out as the delay culprit.

Identify the Delay Culprit

I determined that the culprit was the metadata server. Because two separate requests to the metadata server were slow, I suspected that there was a systemic problem affecting those requests. However, it was a possibility that the metadata server itself was a victim of some other delay contributor.

In fact, further investigation of the metadata server suggested that I/O performance was lacking. The customer's systems team performed traditional performance analysis of the file systems on the metadata and discovered that a file system on the primary metadata node was misconfigured. The file system, hosted on a storage area network (SAN), was upgraded a week prior to the problem. During that upgrade, some of the specific tuning for that file was lost, which resulted in degraded I/O performance on the primary metadata node.

CASE 2: PERFORMANCE DEGRADATION (OVER TIME) FOR THE AMOUNT OF TIME THAT IS REQUIRED TO OPEN A PARTICULAR REPORT

A user noted that a report that he runs on a daily basis appears to run slower over time. He observed that restarting the environment (which happens every weekend) restores the performance of the report. Even so, the performance slowly degrades over the week.

Characterize the Delay

I set up a Cisco WebEx meeting with the customer so that I could ask questions and observe the delay. I scheduled the meeting at the end of the week so that the delay would be observable.

The customer noted that their SAS environment is less involved than most. All three tiers are on a single machine, and neither the middle tier nor the metadata tier is clustered. The delay occurred with one report only. This report runs several stored processes and displays the data on one page. The delay grew over time. It ran in about 20 seconds at its best time. However, by the end of the week, the delay degraded to two minutes.

I discovered that the report in question was the only one that is used by the customer that that invokes stored processes. In addition, users run this report continuously throughout the

week. Despite that information, I decided to focus on the report. The delay in question was easily and consistently reproduced, and I knew that whatever I discovered regarding this report likely would apply to the others.

Leverage the Middle Tier

The reported delay that users experienced matched the delay that was visible in the web-server log. That fact meant that we could eliminate the client machine as well as the network between the client and the web server.

Given that, I turned my attention to the web application server. The customer's environment was using only one web application server, which simplified matters. At this point in the analysis, I knew that the delay culprit was either within the SASServer1_1 JVM or it was an external process on which the JVM was waiting.

PinPoint the Source: Thread Dumps or Network Analysis?

I could have used network analysis to check the timing of requests from SASServer1_1 to other processes. However, thread dumps provide insight into both internal and external delays. So, I chose to use high-frequency thread dumps. I decided on a 5-second thread dump interval for a delay resolution of 10 seconds.

A ten-second delay resolution is probably not small enough to capture most internal delays. That resolution was okay because the overall delay was more than 1 minute, and anything smaller than 10 seconds was not significant at that point in the diagnosis.

Delay Culprit

Capturing 5-second thread-dump intervals throughout a 2-minute time frame produced about 24 thread dumps to examine. I asked the customer to reproduce the delay three times and captured 24 dumps each time. Analysis showed that the report made a call to the stored-process server and, in every case, the JVM waited for over a minute for that one call.

To confirm that the stored-process server was at the root of the delay, we restarted that server only and re-ran the test. Performance was instantly restored, which confirmed that the delay culprit was the stored-process server. We investigated further on the server side and found that the particular stored process, custom code that was written by the customer, was not properly cleaning up before exiting the process. Each run of the stored process left more data behind that had to be processed, which slowly degraded performance over the course of the week.

The customer had automated weekend maintenance that restarted the entire SAS environment every Sunday. So, performance was restored each weekend, but it masked the source of the problem.

CASE 3: A PERIODIC SLOWDOWN OCCURS THAT RECOVERS WITHOUT INTERVENTION

A customer reported intermittent performance degradation. This customer's environment is used by people around the world. So, this system experiences activity 24x7. The customer's environment runs in a customer data center on the East Coast of the United States. The SAS administrator noticed that the slowdown seemed to occur every Sunday evening, based on server time as requests came into the data center. The slowdown lasts for about two hours, after which performance returns to normal without any action taken.

Characterize the Delay

Intermittent delays are difficult to solve because you need to collect data when the problem is evident. Such delays are problematic in that you never really know when the problem will occur. However, in complex environments, issues that appear intermittent often (though not always) have an underlying pattern.

I called the customer's SAS administrator to discuss his observations. He mentioned that users worldwide reported the slowdown by using a problem ticketing system, and he saw that the reports of problems appeared to come in at random times, usually on Monday or Tuesday. I asked a few questions about the ticketing system, and we realized that the reports were timestamped with the user's *local* time zone. We converted all the timestamps to the data-center's local time and realized that all the reported episodes clustered around Sunday evening, Eastern Time.

I discovered that the customer has a three-tiered SAS environment that is horizontally and vertically clustered. This environment is wholly contained within the data center on the East Coast. All the SAS components are on the same network, but users access the environment from local computers all over the world. Their access is through an F5 Networks BIC-IP hardware load balancer. The load balancer is on a separate network, but it is in the same data center.

This topology is complex, but it is common for enterprise customers. This customer has a larger-than-usual user base, but it is otherwise unremarkable. The users run rather rudimentary reports that do not perform heavy computation or query exceptionally large data.

Leverage the Middle Tier

Again, I started the diagnosis at the middle tier. Because the middle tier is clustered, I asked the customer to stop the middle services on nodes 2 and 3, leaving only node 1 active. To minimize the impact, we left these nodes offline from 5:00 p.m. through 10:00 p.m. (server time). That time span was enough time to capture the delay.

I wanted to capture data from the middle tier JVMs. So, I enabled garbage-collection logging on each of the JVMs on node 1. Because it was not clear if the middle tier JVMs were the culprit or the victim, I also decided to collect thread dumps and network data.

Capturing Data for an Intermittent Issue

Garbage-collection logging uses few resources and produces small logs that tell me a lot about the health of the JVM. This logging is turned on and collected for the lifetime of the JVM. So, there is no concern over triggering it at the appropriate time.

Thread dumps and network data on the other hand require a bit more finesse. Both artifacts need to cover the period of the delay. However, the size and analysis overhead of these tools preclude the ability to collect data continuously as is done with garbage-collection logging.

Luckily, we were able to narrow down the occurrence to a small window of time. I was confident that we could reliably reproduce the delay on Sunday evening, limiting thread dumps and network data to a minute or two while the report was running.

I used thread-dump interval of one second along with the **`tcpdump`** UNIX utility to capture the network traffic. These tools ran for about two minutes while the report was running.

Proximal Delay Contributor

Before Sunday, I arranged for a practice run with the administrator to ensure that he knew how to run the report and which artifacts to collect. On Sunday ,the data collection went well, giving us all the information we needed.

In reviewing the garbage-collection logging, I first identified a period of extended garbage-collection duration. The JVM appeared to be under considerable stress during the delay period. I corroborated that assumption by reviewing the thread dumps, and I saw that the delays were internal only. All external requests were fast. With this information, I did not need to analyze the network data.

I confirmed that the SASServer1 JVM itself was the *delay contributor*. The JVM slowed down during the Sunday evening period and directly contributed to the delay. But it was not clear why the slowdown occurred. Furthermore, it was not clear how the system recovered without intervention.

Root Cause

Normally, an under-performing JVM is addressed by tuning the JVM. You can adjust several configuration options to alter performance under various conditions. But the analysis showed that the JVM was configured appropriately. Even so, the JVM ran slower each week on Sunday evening.

So, I focused on the middle-tier machine. I discovered that the data center used normal, quiet Sunday evenings for performing maintenance tasks. For a period of about three hours on Sundays, the middle-tier machine runs a virus scans and a full backup. These activities consumed the machine's capacity enough to directly impact the performance of the JVMs that run the SAS web application servers. After these maintenance tasks were complete, performance returned to normal.

CONCLUSION

Delays that are experienced by users of enterprise-class software environments, such as SAS 9.4, are quite difficult to diagnose. This paper has shown you how to narrow down the solution space and identify the root cause of the delay in these ways:

- by leveraging the SAS middle-tier components
- by applying systematic analysis techniques (such as the use of high-frequency thread dumps

By using these techniques, you can likely avoid the performance delays for your users.

RESOURCES

- IBM Corporation. 2019. "IBM Thread and Monitor Dump Analyzer for Java (TMDA)." Available at **www.ibm.com/support/pages/ibm-thread-and-monitor-dump-analyzer-java-tmda**. Accessed on January 30, 2020.
- Sedlacek, Jiri (Oracle Corporation). 2017. "VisualVM: All-in-One Java Troubleshooting Tool." Available at **visualvm.github.io/**. Accessed on January 30, 20201.

(list continued)

- Open source. 2019. "nmon for Linux." Available at **nmon.sourceforge.net/pmwiki.php?n=Main.HomePage**. Accessed on January 30, 2020.
- Wireshark Foundation. 2020. Wireshark. Available at **www.wireshark.org/**. Accessed on January 30, 2020.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Bob Celestino
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Email: **support@sas.com**
Web: **support.sas.com**