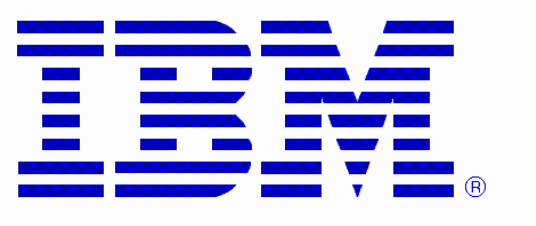


WebSphere Application Server Troubleshooting and Performance Lab on Docker

Authors

- Kevin Grigorenko (kevin.grigorenko@us.ibm.com)

Version History

- V2 (May 20th, 2019): Convert to Docker and modernize.
- V1 (December 14th, 2016): First version on VMWare.

Contents

1 Introduction

WebSphere Application Server[1] (WAS) is a platform for serving Java-based applications. WAS comes in two major product forms:

1. Traditional WAS[2] (colloquially: tWAS or WAS Classic): Released in 1998 and still fully supported.
2. WAS Liberty[3] (colloquially: Liberty or WebSphere Liberty): Released in 2012 and designed for fast startup, composability, and the cloud. The commercial WAS Liberty product is built on top of the open source core called OpenLiberty[4].

The two products share some source code but differ in significant ways[5].

Both Traditional WAS and WAS Liberty come in different flavors including *Base* and *Network Deployment (ND)* in which ND layers additional features such as advanced high availability on top of Base.

1.1 Lab

This lab assumes the installation and use of Docker to run the lab. For example, install Docker Desktop for Windows or Mac hosts:

- Windows ("Requires Microsoft Windows 10 Professional or Enterprise 64-bit.")
  - Download: https://hub.docker.com/editions/community/docker-ce-desktop-windows
  - For details, see https://docs.docker.com/docker-for-windows/install/
- Mac ("Requires Apple Mac OS Sierra 10.12 or above")
  - Download: https://hub.docker.com/editions/community/docker-ce-desktop-mac
  - For details, see https://docs.docker.com/docker-for-mac/install/
- For a Linux host, simply install and start Docker (sudo systemctl start docker):
  - For an example, see https://docs.docker.com/install/linux/docker-ce/fedora/

This lab covers the major tools and techniques for troubleshooting and performance tuning for both Traditional WAS and WAS Liberty, in addition to specific tools for each. There is significant overlap because a lot of troubleshooting and tuning occurs at the operating system and Java levels, largely independent of WAS.

The lab Docker images come with Traditional WAS and WAS Liberty pre-installed so installation and configuration steps are largely skipped.

Note that the way we are using Docker in these lab Docker images[6,7,8] is to run multiple services in the same container (e.g. Remote Desktop, VNC, Traditional WAS, WAS Liberty, a full GUI server, etc.) and although this approach is valid and supported[9], it is not generally recommended for production Docker usage. In this case, Docker is used primarily for easy distribution and building of this lab.

[1] https://www.ibm.com/cloud/websphere-application-platform
[2] https://www.ibm.com/support/knowledgecenter/en/SSAW57/mapfiles/product_welcome_wasnd.html
[3] https://www.ibm.com/support/knowledgecenter/en/SSAW57_liberty/as_ditamaps/was900_welcome_liberty_ndmp.html
[4] https://github.com/OpenLiberty/open-liberty
[5] http://public.dhe.ibm.com/ibmdl/export/pub/software/websphere/wasdev/documentation/ChoosingTraditionalWASorLiberty-16.0.0.4.pdf
[6] https://github.com/kgibm/dockerdebug/blob/master/fedorawasdebug/Dockerfile
[7] https://github.com/kgibm/dockerdebug/blob/master/fedorajavadebug/Dockerfile
[8] https://github.com/kgibm/dockerdebug/blob/master/fedoradebug/Dockerfile
[9] https://docs.docker.com/config/containers/multi-service_container/

1.2 Operating System

This lab is built on top of Linux (specifically, Fedora Linux, which is the open source foundation of RHEL/CentOS). The concepts and techniques apply generally to other supported operating systems although details of other operating systems vary significantly and are covered elsewhere[10].

1.3 Java

Traditional WAS ships with a packaged IBM Java 8 on Linux, AIX, Windows, z/OS, and IBM i.

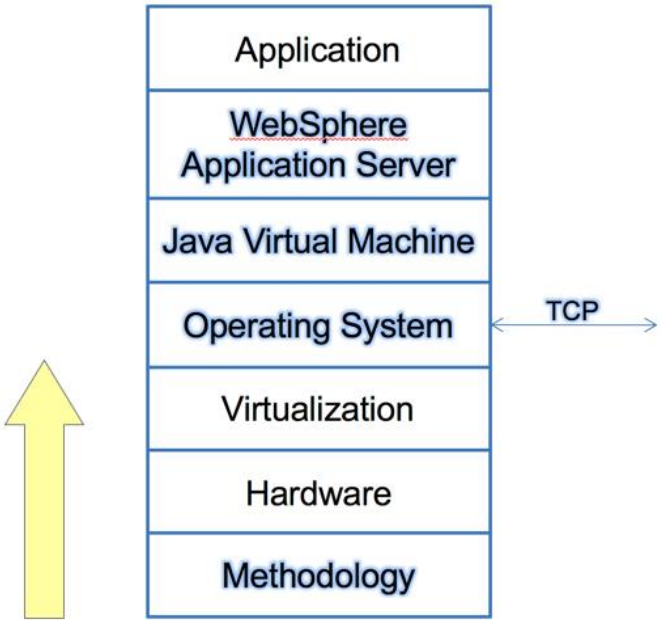
WAS Liberty supports any Java 8 or Java 11 compliant Java (with some minimum requirements[11]).

This lab uses IBM Java 8 for both Traditional WAS and WAS Liberty. The concepts and techniques apply generally to other supported Java runtimes although details of other Java runtimes (e.g. HotSpot) vary significantly and are covered elsewhere[12].

The IBM Java virtual machine (named J9) has become largely open sourced into the OpenJ9 project[13]. OpenJ9 ships with OpenJDK through the AdoptOpenJDK project[14]. OpenJDK is somewhat different than the JDK that IBM Java uses. WAS Liberty >= 19.0.0.1 supports running with OpenJDK+OpenJ9 >= 11.0.2, although some tooling such as HealthCenter is not yet available in OpenJ9, so the focus of this lab continues to be IBM Java 8.

2 Core Concepts

Problem determination and performance tuning are best done with all layers of the stack in mind. This lab will focus on the layers in bold below:

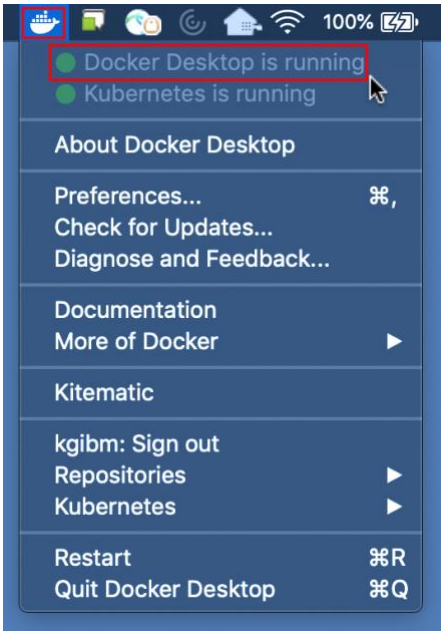


[10] https://publib.boulder.ibm.com/httpserv/cookbook/Operating_Systems.html
[11] https://www.ibm.com/support/knowledgecenter/SSAW57_liberty/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/rwlp_restrict.html?view=kc#rwlp_restrict__rest13
[12] https://publib.boulder.ibm.com/httpserv/cookbook/Java.html
[13] https://github.com/eclipse/openj9
[14] https://adoptopenjdk.net/

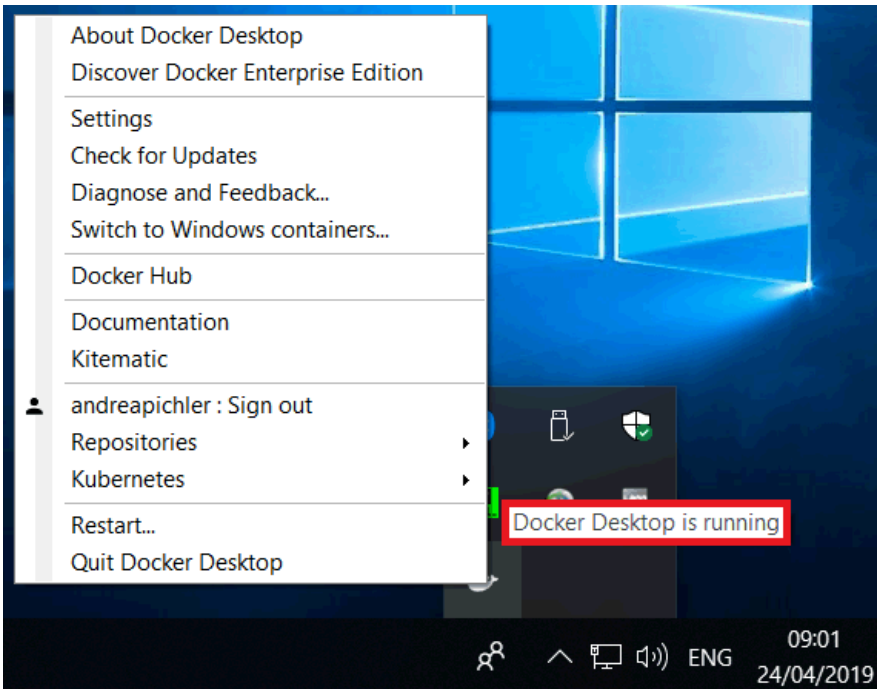
3 Starting the Docker Container

1. Ensure that Docker is started. For example, start Docker Desktop and ensure it is running:

   macOS:



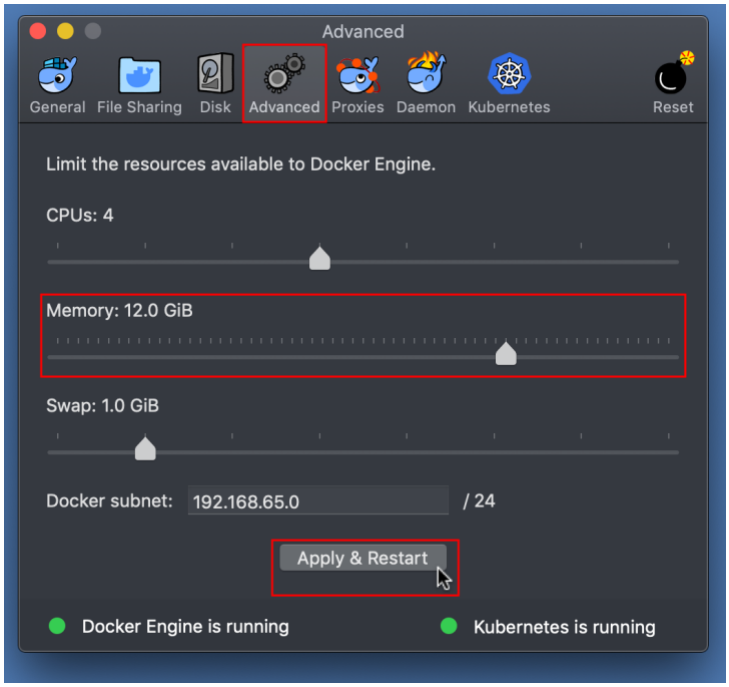
   Windows:



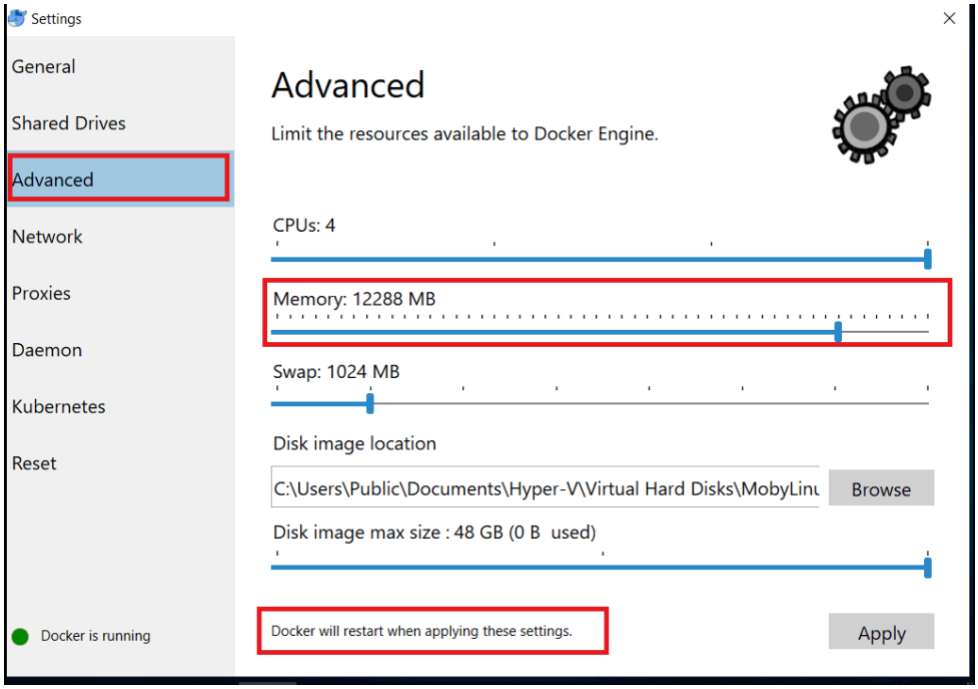
2. Ensure that Docker receives sufficient resources, particularly memory:
   a. Click the Docker Desktop icon and select **Preferences…** (on macOS) or **Settings** (on Windows)
   b. Select the **Advanced** tab.
   c. Increase **Memory**, ideally to at least 8GB.

d. Click **Apply**

macOS:

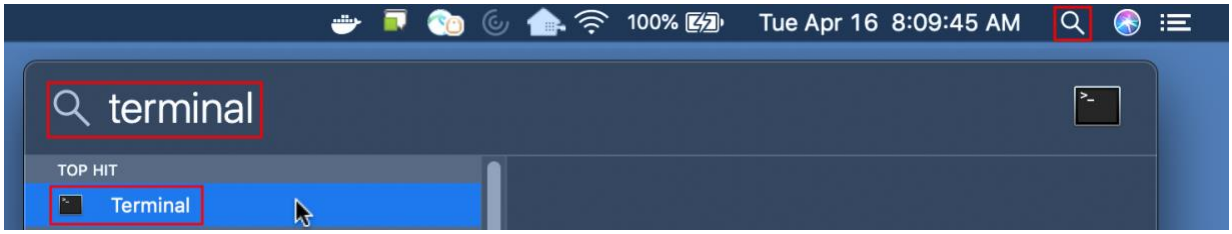


Windows:

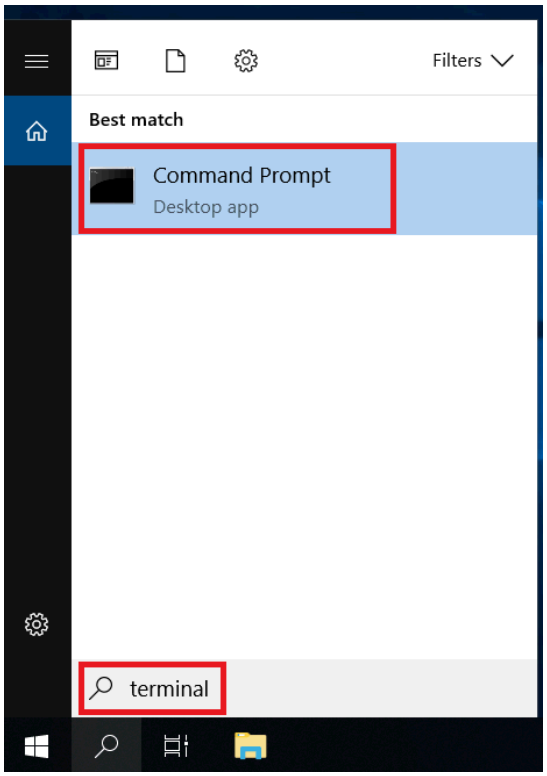


3. Open a terminal or command prompt:

   macOS:



   Windows:



4. Download the images:

```
docker pull kgibm/fedorawasdebug
```

   a. Note that these images are about 20GB. If you plan to run this in a classroom setting, consider performing all the steps up to and including this item before arriving at the classroom.

5. Start the lab Docker container:

```
docker run --cap-add SYS_PTRACE --ulimit core=-1 --rm -p 9080:9080 -p 9443:9443 -p
9043:9043 -p 9081:9081 -p 9444:9444 -p 5901:5901 -p 5902:5902 -p 3390:3389 -p 22:22
-p 9082:9082 -p 9445:9445 -it kgibm/fedorawasdebug
```

6. After about 30 seconds, VNC or Remote Desktop into the container:

   a. macOS built-in VNC client:

      i. Open another tab in the terminal and run:

         1. open vnc://localhost:5902

2. Password: **websphere**

b. Linux VNC client:

i. Open another tab in the terminal and run:

1. vncviewer localhost:5902
2. Password: **websphere**

c. Windows 3rd party VNC client:

i. If you are able to install and use a 3rd party VNC client, then connect to **localhost** on port **5902** with password **websphere**.

d. Windows Remote Desktop client:

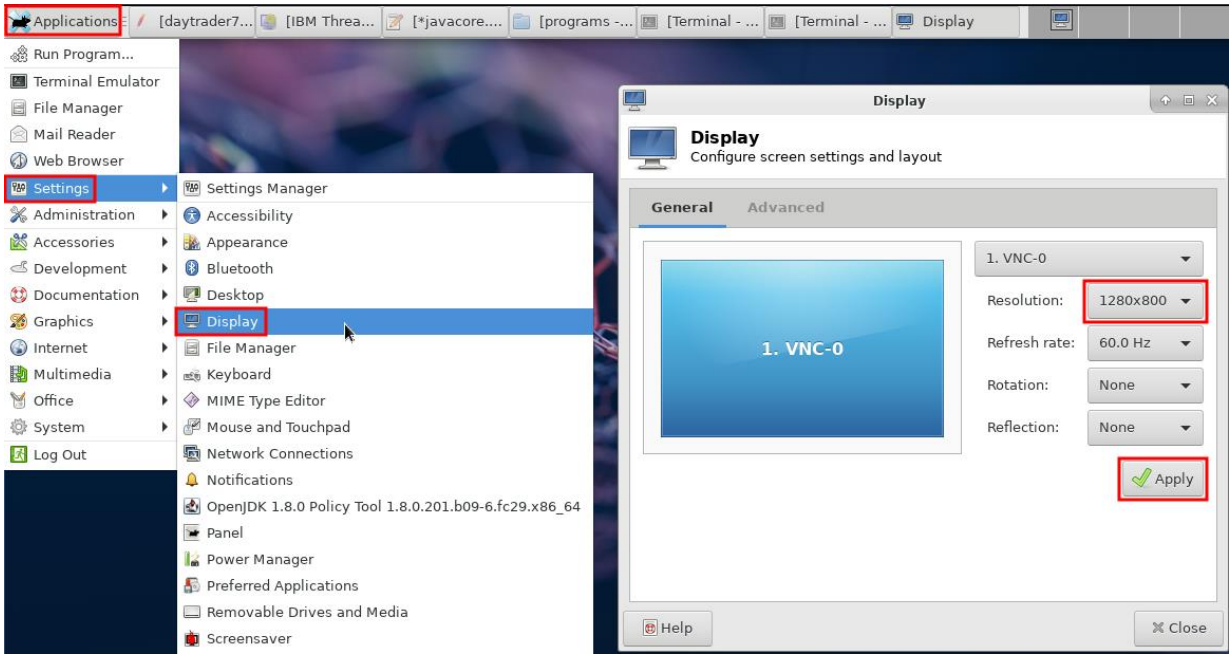
i. Windows requires a few steps to make Remote Desktop work with a Docker container. See Appendix: Windows Remote Desktop Client for instructions.

e. SSH:

i. If you want to simulate a production-like environment, you can SSH into the container (e.g. using terminal ssh or PuTTY) although you'll need one of the GUI methods above to run most of this lab:

1. ssh was@localhost
2. Password: **websphere**

7. When using VNC, you may change the display resolution from within the container and the VNC client will automatically adapt. For example:



8. Test WAS Liberty by going to http://localhost:9080/ in your host browser or the remote desktop/VNC browser.

9. Test Traditional WAS by going to http://localhost:9081/swat/ in your host browser or in the remote desktop/VNC browser.

a. Note that Traditional WAS takes a few minutes to start up after the container has started.

b. Test the Traditional WAS Administrative Console by going to https://localhost:9043/ibm/console in your client browser or in the remote desktop/VNC browser.

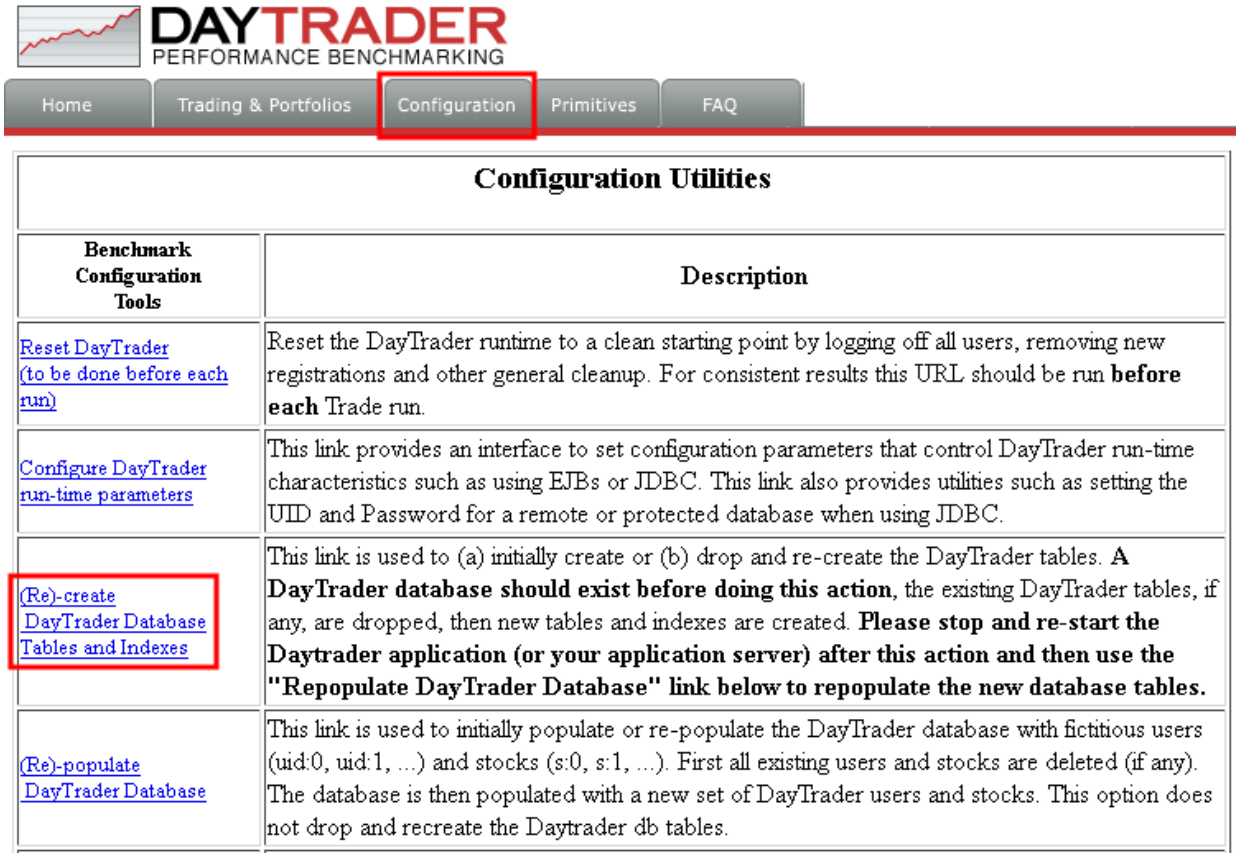
   i. User: **wsadmin**

   ii. Password: **websphere**

3.1 The DayTrader Sample Application

The DayTrader7 sample application[15] that we'll be using is pre-installed in the lab image but requires some initial preparation:

1. Open http://localhost:9080/daytrader/
2. Click the **Configuration** tab, and click **(Re)-create DayTrader Database Tables and Indexes**:



[15] https://github.com/WASdev/sample.daytrader7

3. This will open a new tab and the database should be created with the following output:

TradeBuildDB: **** Database Product detected: Apache Derby ****

TradeBuildDB: **** The DDL file at path */dbscripts/derby/Table.ddl* will be used ****

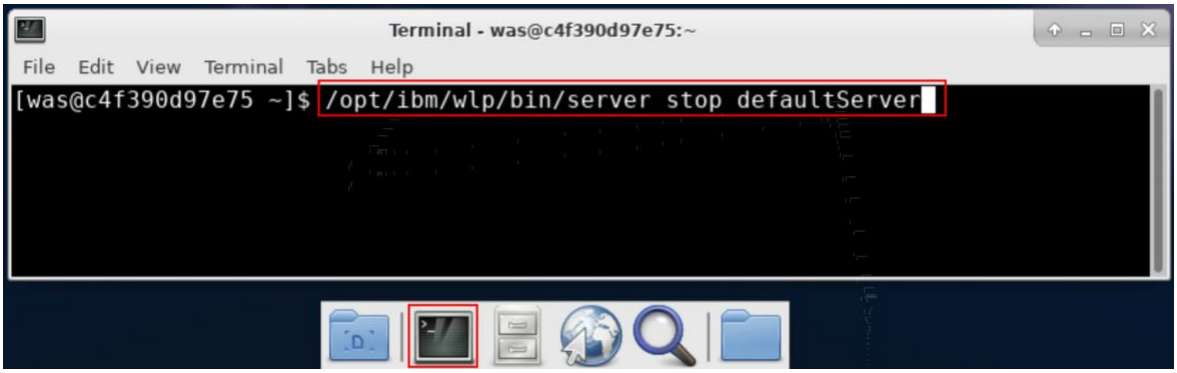
TradeBuildDB: Building DayTrader Database...
This operation will take several minutes. Please wait...
TradeBuildDB: **** Dropping and Recreating the DayTrader tables... ****

TradeBuildDB: **** DayTrader tables successfully created! ****

Please Stop and Re-start your Daytrader application (or your application server) and then use the "Repopulate Daytrader Database" link to populate your database.

4. The DayTrader application requires that the application is restarted after creating the database tables, so we will simply restart Liberty.
5. In the remote desktop or VNC viewer, open a terminal, type the following command to stop the Liberty server and press Enter:

```
/opt/ibm/wlp/bin/server stop defaultServer
```

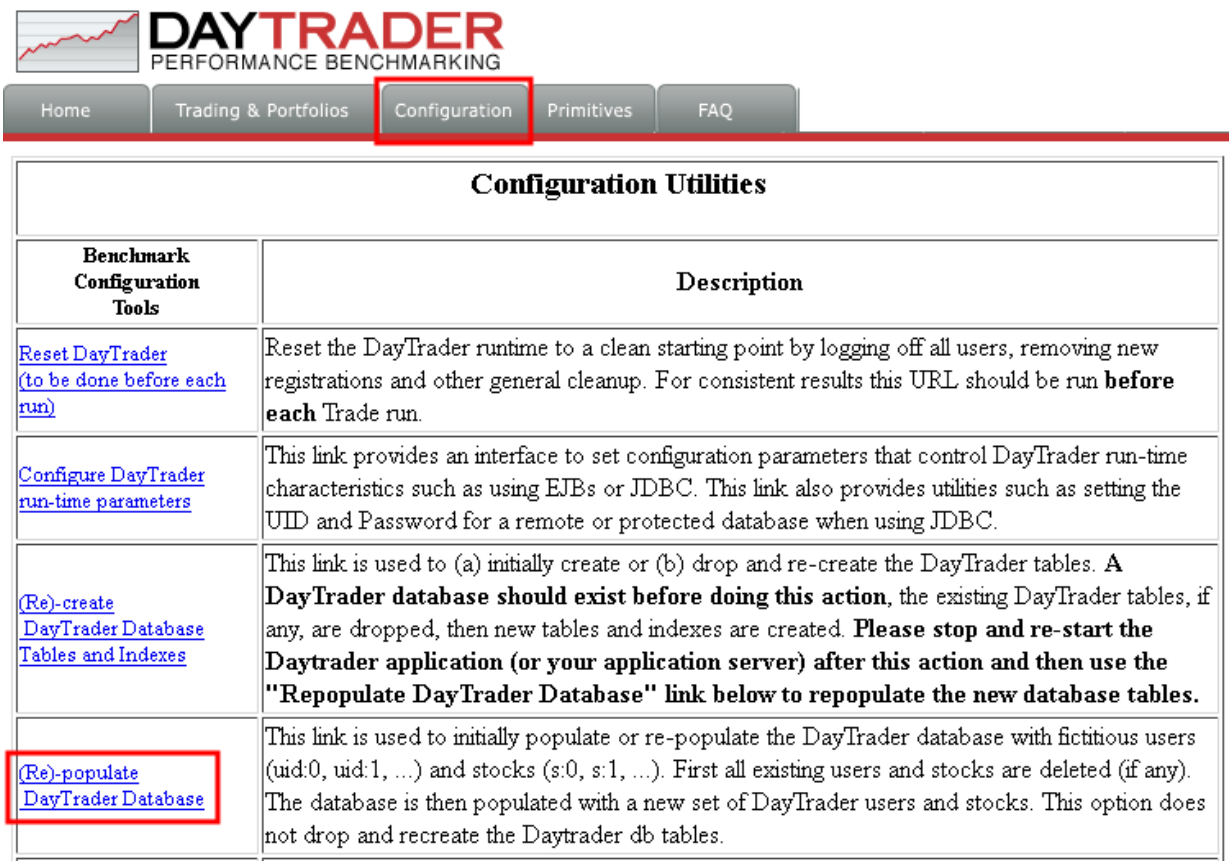


6. After the previous command completes, type the following command to start the Liberty server and press Enter:

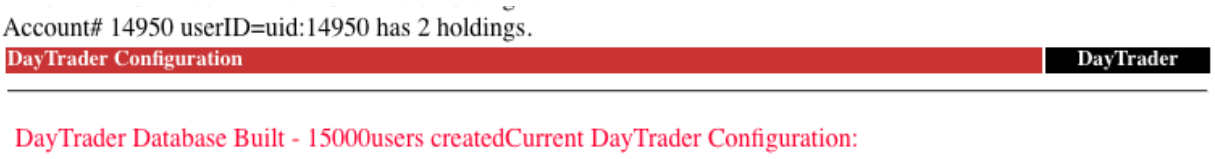
```
/opt/ibm/wlp/bin/server start defaultServer
```

7. Once the start command completes, refresh the DayTrader website at http://localhost:9080/daytrader/

8. Click **Configuration** and click **(Re)-populate DayTrader Database**:



9. This will take about 5 minutes depending on your computer and disk speeds. Keep scrolling to the bottom of the browser and wait until the bottom of the browser output shows "DayTrader Database Built":

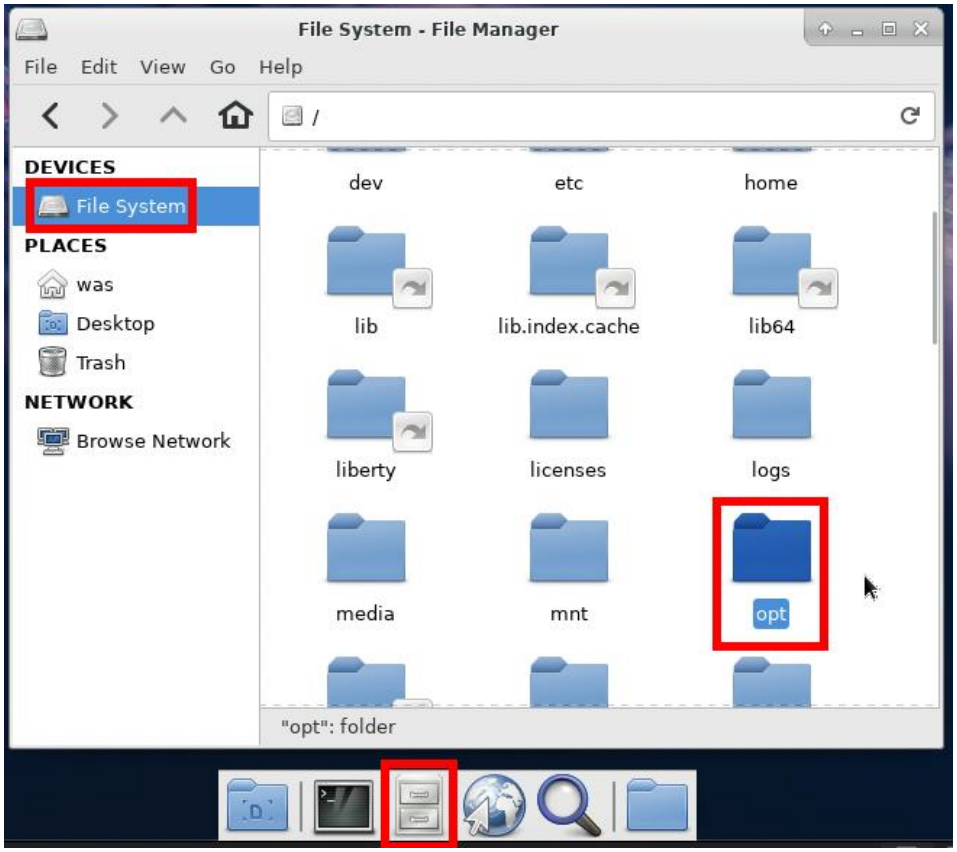


3.2 Apache Jmeter

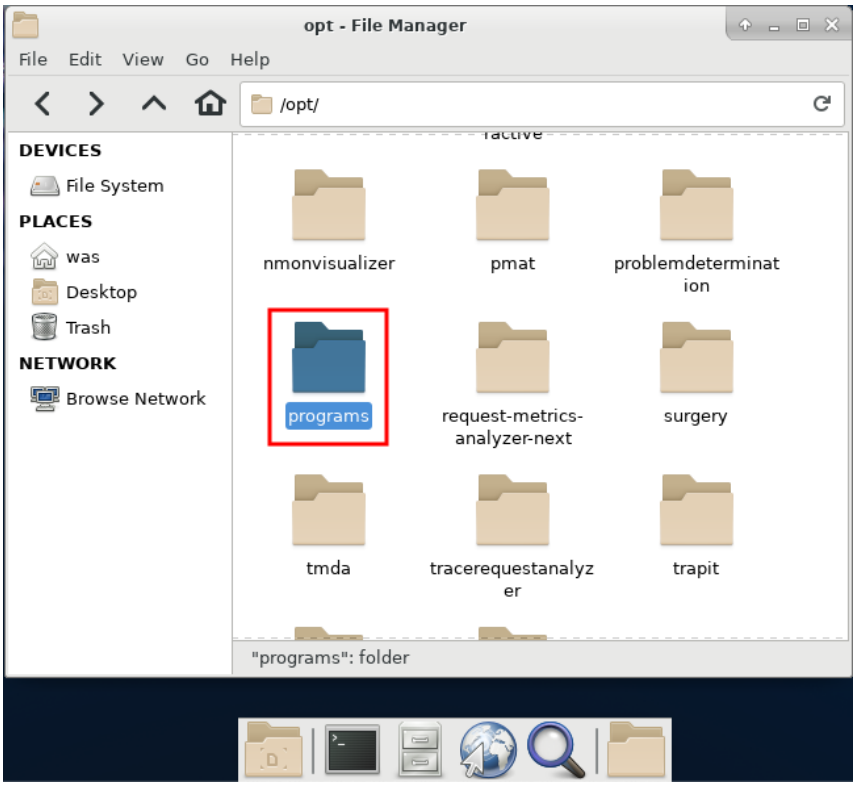
Apache JMeter[16] is a free tool that drives artificial, concurrent user load on a website. The tool is pre-installed in the lab image and we'll be using it to simulate website traffic to the DayTrader application.

[16] https://jmeter.apache.org/

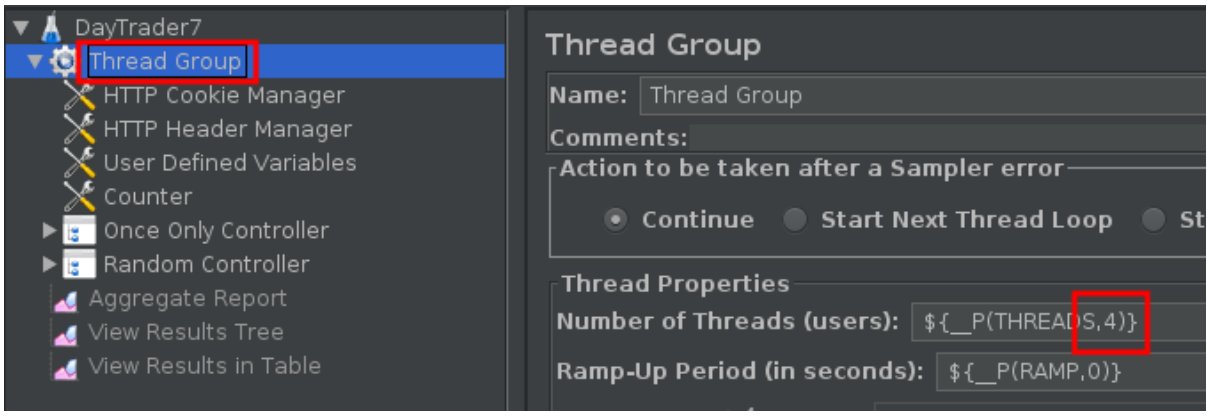
1. Open File Manager and navigate to /opt:



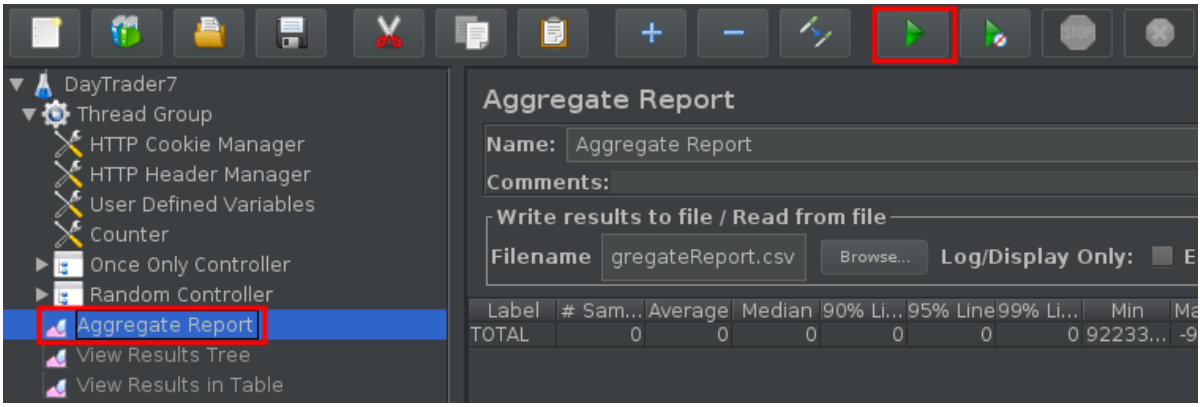
2. Navigate to **/opt/programs/**:



3. Double click on **JMeter**
4. Click **File → Open** and select **/opt/daytrader7/jmeter_files/daytrader7.jmx**
5. By default, the script will execute 4 concurrent users. You may change this if you want (e.g. based on the number of CPUs available):

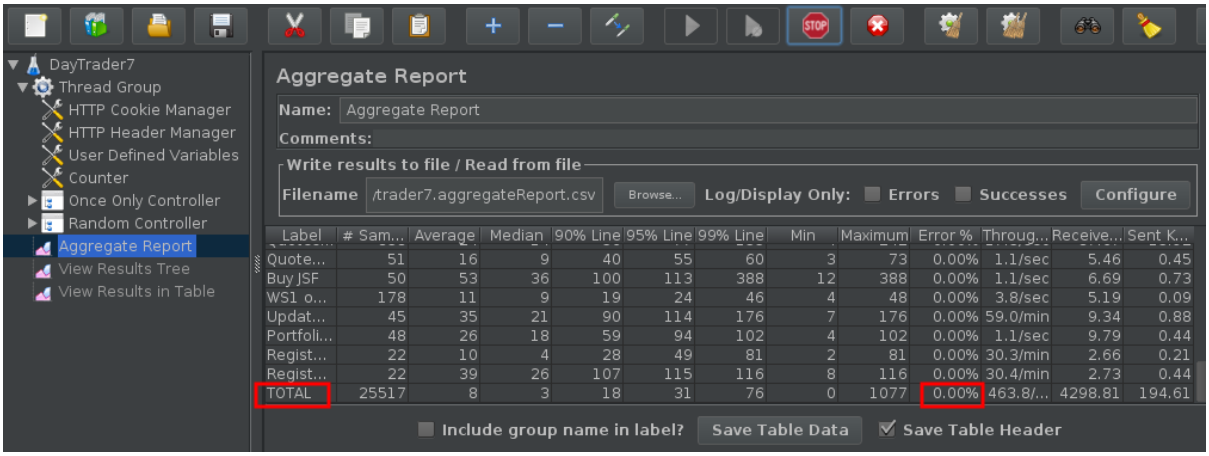


6. Click the green run button to start the stress test and click the **Aggregate Report** item to see the real-time results.



7. It will take some time for the responses to start coming back and for all of the pages to be exercised.

8. Ensure that the **Error %** value for the **TOTAL** row is always 0%. If there are any errors, review the WAS logs.



4 Basics

First, we'll start with the three basics that should be checked for most problems and performance issues:

1. Operating system CPU and memory usage
2. Thread dumps
3. Garbage Collection

5 WebSphere Linux Performance and Hang MustGather

IBM WebSphere Support provides a script called **linperf.sh** as part of the document, "MustGather: Performance, hang, or high CPU issues with WebSphere Application Server on Linux"[17] (similar scripts exist for other operating systems). This script should be pre-installed on all machines where you run WAS and it should be run when you have performance or hang issues and the resulting files should be uploaded when you open such a support case with IBM.

The linperf.sh script is pre-installed in the lab image at **/opt/linperf/linperf.sh**. In this exercise, you will run this script and analyze the output. The script demonstrates key Linux performance tools that are generally useful whether you decide to run this tool or use the commands individually.

5.1 linperf.sh Theory

First, let's discuss what this script does at a high level:

1. The script is executed with a set of process IDs (PIDs) of the suspect WAS processes.
2. The script gathers the output of the **netstat** command. This produces a snapshot of all active TCP and UDP network sockets.
3. The script gathers the output of the **top** command for the duration of the script (default 4 minutes). This produces periodic snapshots of a summary of system resources (CPU, memory, etc.) and the CPU usage details of the top *processes* using CPU.
4. The script gathers the output of the **top -H** command for each specified PID for the duration of the script. This produces periodic snapshots of a summary of system resources and the CPU usage details of the top *threads* using CPU in each PID.
5. The script gathers the output of the **vmstat** command for the duration of the script. This produces periodic snapshots of a summary of system resources. This is similar to the top command.
6. The script periodically requests a thread dump for each specified PID (default every 30 seconds). This produces detailed information on the Java process such as the threads and what they're doing.
7. The script gathers the output of the **ps** command for each specified PID on the same interval as the thread dumps. This produces detailed information on the command line of each PID and other resource utilization details. This is similar to the top command.

5.2 linperf.sh Lab

Now, let's run the script:

1. Make sure that the JMeter test is running.
2. Open a terminal on the lab image.
3. First, we'll need to find the PID(s) of WAS. There are a few ways to do this, and you only need to choose one method:
    a. Show all processes (**ps -elf**), search for the process using something unique in its command line (**grep defaultServer**), exclude the search command itself (**grep -v grep**), and then select the fourth column (in bold below):

[17] https://www-01.ibm.com/support/docview.wss?uid=swg21115785

```
$ ps -elf | grep defaultServer | grep -v grep
4 S was       1567     1 99  80   0 - 802601 -     19:26 pts/1    00:03:35 java -
javaagent:/opt/ibm/wlp/bin/tools/ws-javaagent.jar -Djava.awt.headless=true -
Xshareclasses:name=liberty,nonfatal,cacheDir=/output/.classCache/ -jar
/opt/ibm/wlp/bin/tools/ws-server.jar defaultServer
```

b. Search for the process using something unique in its command line using **pgrep -f**:

```
$ pgrep -f defaultServer
1567
```

4. Execute the **linperf.sh** command and pass the PID gathered above (replace 1567 with your PID):

```
$ /opt/linperf/linperf.sh 1567
Tue Apr 23 19:29:26 UTC 2019 MustGather>> linperf.sh script starting [...]
```

5. Wait for 4 minutes for the script to finish:

```
[...]
Tue Apr 23 19:33:33 UTC 2019 MustGather>> linperf.sh script complete.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Output files are contained within ----->
linperf_RESULTS.tar.gz.   <----
Tue Apr 23 19:33:33 UTC 2019 MustGather>> The javacores that were created are NOT
included in the linperf_RESULTS.tar.gz.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Check the <profile_root> for the
javacores.
Tue Apr 23 19:33:33 UTC 2019 MustGather>> Be sure to submit linperf_RESULTS.tar.gz,
the javacores, and the server logs as noted in the MustGather.
```

6. As mentioned at the end of the script output above, the resulting **linperf_RESULTS.tar.gz** does not include the thread dumps from WAS. Move them over to the current directory:

```
mv /opt/ibm/wlp/output/defaultServer/javacore.* .
```

At this point, if you were creating a support case, you would upload **linperf_RESULTS.tar.gz**, **javacore***, and all the WAS logs; however, instead, we will analyze the results to learn about these basic Linux performance tools:

1. Extract **linperf_RESULTS.tar.gz**:

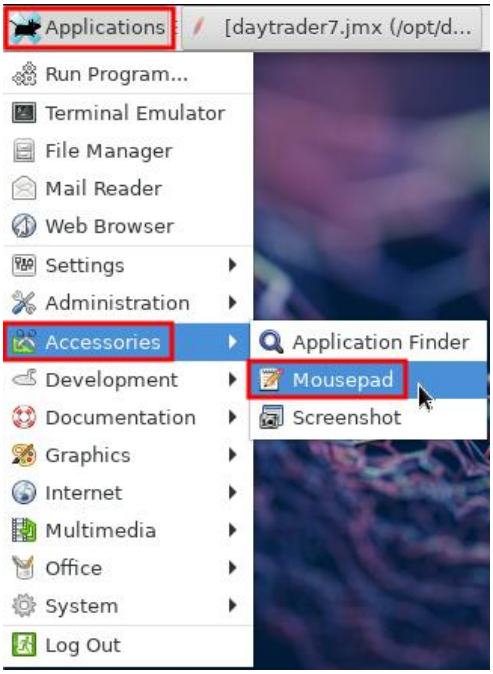
```
tar xzf linperf_RESULTS.tar.gz
```

2. This will produce various ***.out** files from the various Linux utilities.

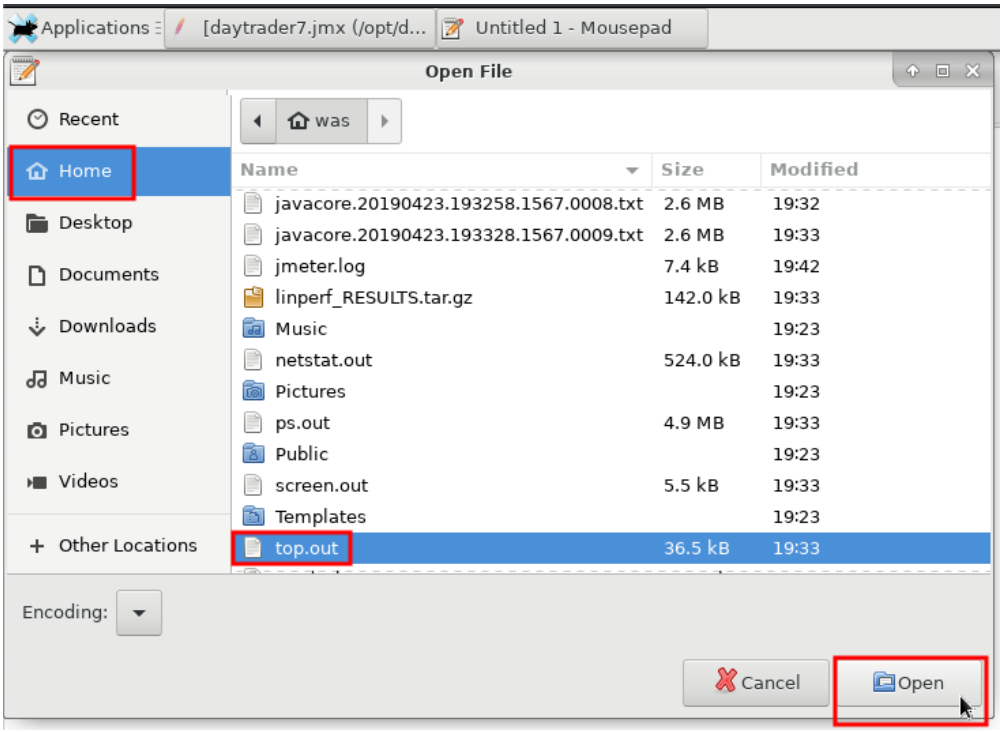
6 Linux top

top is one of the most basic Linux performance tools. Open **top.out** to review the output.

If you would like to open text files in the Linux container using a GUI tool, you may use a program such as **mousepad**:



Then click File > Open, and find the file where you ran **linperf.sh** such as in the Home directory:



There will be multiple sections of output, each prefixed with a timestamp which represents the previous interval (**linperf.sh** uses a default interval of 60 seconds). In the following example, the data represents CPU usage between 19:28:27 - 19:29:27. Review all intervals to understand CPU usage over time. For example, here is one interval:

```
Tue Apr 23 19:29:27 UTC 2019
top - 19:29:27 up  2:49,  1 user,  load average: 5.59, 2.41, 1.16
```

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.7 us, 23.9 sy,  0.0 ni, 20.9 id,  1.5 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,    395.9 free,   1777.5 used,   9820.0 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1567 was       20   0 3216340 374372  36356 S 181.2   3.0   5:34.40 java -jav+
 1854 was       20   0 3701404 417256  24580 S  37.5   3.4   1:21.23 /usr/bin/+
  414 was       20   0  187948  19652  14296 S   6.2   0.2   0:00.22 xfsetting+
 2631 was       20   0   10676   4380   3820 R   6.2   0.0   0:00.02 top -bc -+
 2640 was       20   0   10676   4316   3756 S   6.2   0.0   0:00.01 top -bH -+
    1 root      20   0    3784   2956   2696 S   0.0   0.0   0:00.05 /bin/sh /+
    9 root      20   0   23916  21112   7532 S   0.0   0.2   0:00.31 /usr/bin/+
   13 root      20   0  151676   4188   3684 S   0.0   0.0   0:00.01 /usr/sbin+
   14 root      20   0    9264   5852   5184 S   0.0   0.0   0:00.01 /usr/sbin+
   15 root      20   0    6960   3636   3148 S   0.0   0.0   0:00.00 /usr/sbin+
```

One place to start is to check the server's RAM:

```
MiB Mem :  11993.4 total,    395.9 free,   1777.5 used,   9820.0 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem
```

The values may be in bytes, KB, MB, or other formats depending on various settings.

The two values in bold are the important values:

1. The first bold value on the first line shows the total amount of RAM; in this example, about 11.9GB.
2. The second bold value on the second line shows the approximate amount of RAM that is available for applications if they need it (including readily reclaimable page cache and memory slabs); in this example, about 9.8GB. Notice that the actual amount of free RAM (first line, second column, in *italics*) is only about 395MB. Linux, like most other modern operating systems, is aggressive in using RAM for various caches, primarily the file cache, to improve disk I/O speeds; however, most of this memory is reclaimable if applications demand it. Note that Linux is particularly aggressive with its default **swappiness**[18] value and in some cases it will prefer to page out application pages instead of reclaiming file cache pages. Consider setting vm.swappiness=0 for production workloads that perform little file I/O and require most of the RAM.

Next, review the server's overall CPU usage:

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.7 us, 23.9 sy,  0.0 ni, 20.9 id,  1.5 wa,  0.0 hi,  0.0 si,  0.0 st
```

The value in bold is the important value. **id** represents the percent of time during the interval that all CPUs were idle. It is better to look at **idle%** instead of **user%**, **system%**, etc. because this ensures that you quickly capture all potential users of CPU (including I/O wait, **nice**d processes[19], and hypervisor stealing[20]). Subtract the **id** number from 100 to get the approximate total CPU usage; in this example, (100 - 20.9) ~= 79.1%.

[18] https://publib.boulder.ibm.com/httpserv/cookbook/Operating_Systems-Linux.html#Swappiness
[19] **nice** and **renice** are commands to change the relative scheduling priorities of processes. Nice% reflects non-default, positively niced processes' CPU utilization.
[20] In a virtualized environment, the percent of time this host wanted CPU but waited for the hypervisor. This may mean CPU overcommit and should be reviewed.

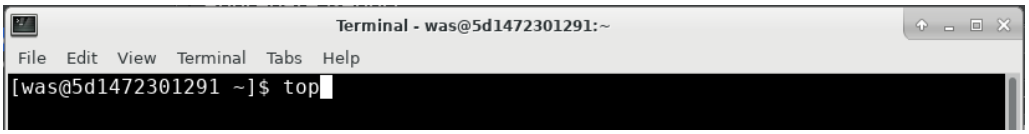
Next, top prints a sorted list of the highest CPU-using processes:

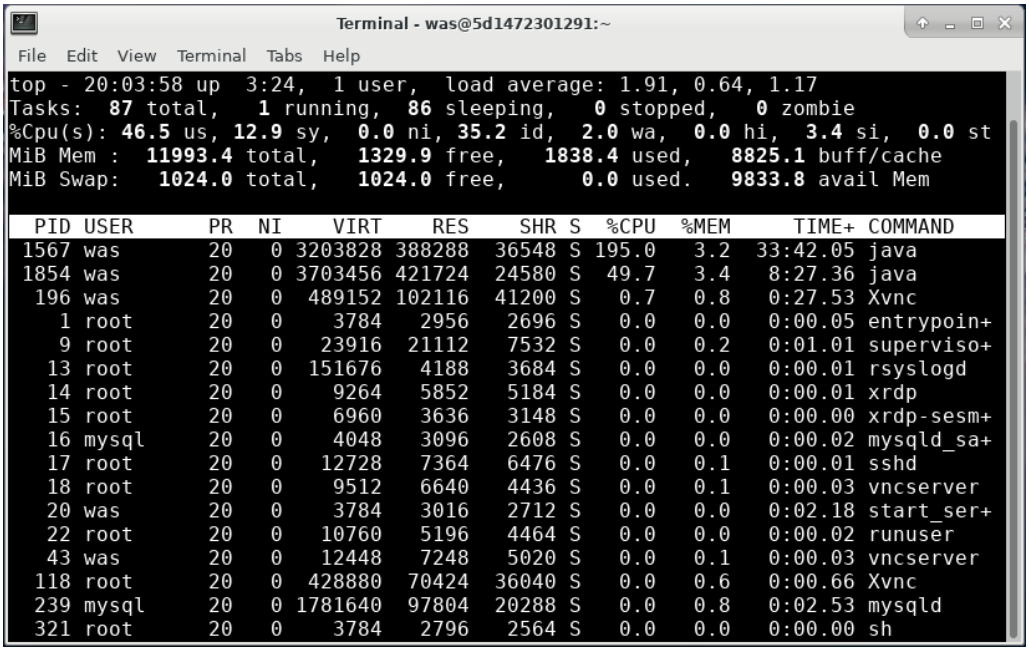
```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1567 was       20   0 3216340 374372  36356 S 181.2   3.0   5:34.40 java -jav+
 1854 was       20   0 3701404 417256  24580 S  37.5   3.4   1:21.23 /usr/bin/+
  414 was       20   0  187948  19652  14296 S   6.2   0.2   0:00.22 xfsetting+ …
```

The two columns in bold are the important values:

1. The first bold column is the PID of each process which is useful for running more detailed commands.
2. The second bold column is the percent of CPU used by that PID for the interval as a percentage of one CPU. For example, PID 1567 consumed about 181.2% of one CPU which means that approximately the equivalent of 1.8 CPU threads were used. In this example, the container had 4 CPU threads available (see **/proc/cpuinfo** on your system), so PID 1567 consumed about (1.812 / 4) * 100 ~= 45.3% of total CPU.

The **top** command may be run in interactive mode by simply running the **top** command. This is a useful place to start when you begin investigating a system. The command will dynamically update every few seconds (this interval may be specified with the **-d S** options where **S** is in fractional seconds). Press **q** to quit top.





7 Linux top -H

top -H is similar to top except that the **-H** flag shows the top CPU usage by thread instead of by PID. Open **topdashH*.out** to review the output. Again, this file shows multiple intervals, so it's important to review all intervals to understand CPU usage over time. Here is an example interval:

```
Tue Apr 23 19:29:27 UTC 2019

Collected against PID 1567.
```

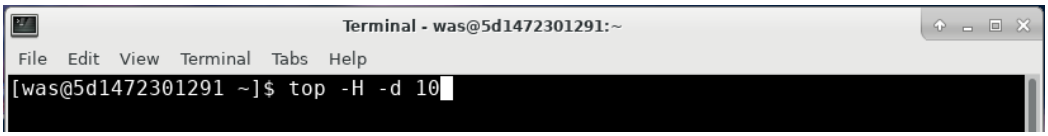
```
top - 19:29:27 up  2:49,  1 user,  load average: 5.59, 2.41, 1.16
Threads:  88 total,  12 running,  76 sleeping,   0 stopped,   0 zombie
%Cpu(s): 54.8 us, 19.4 sy,  0.0 ni, 24.2 id,  1.6 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,    395.8 free,   1777.5 used,   9820.1 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 1571 was       20   0 3216340 374372  36356 S  12.5   3.0   0:00.02 Signal Re+
 1638 was       20   0 3216340 374372  36356 S  12.5   3.0   0:08.09 Inbound R+
 2347 was       20   0 3216340 374372  36356 S  12.5   3.0   0:06.54 Default E+
 2386 was       20   0 3216340 374372  36356 S  12.5   3.0   0:04.94 Default E+
 2406 was       20   0 3216340 374372  36356 S  12.5   3.0   0:04.59 Default E+
 2439 was       20   0 3216340 374372  36356 S  12.5   3.0   0:03.52 Default E+
 2514 was       20   0 3216340 374372  36356 S  12.5   3.0   0:01.58 Default E+
 2539 was       20   0 3216340 374372  36356 S  12.5   3.0   0:00.80 Default E+ …
```

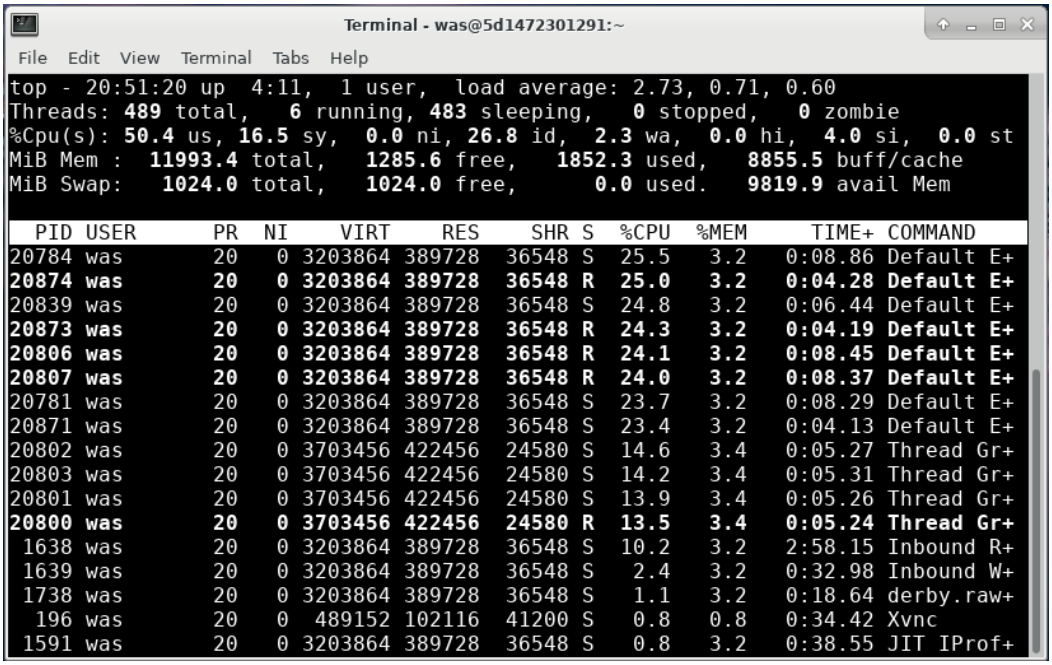
The three columns in bold are the important values:

1. The first bold column is the thread ID (TID) of each thread (the column is still called "PID" because Linux treats threads as "lightweight processes") which is useful for running more detailed commands. This value may be converted to hexadecimal and searched for in a matching thread dump.
2. The second bold column is the percent of CPU used by that TID for the interval as a percentage of one CPU (similar to the previous top output, except it's for the TID instead of the PID).
3. On recent versions of Linux, the third bold column is the name of the thread. This is incredibly useful to get a quick understanding of what threads in the Java process are consuming most of the CPU. For example:
    a. **Default Executor** threads are generally application threads processing HTTP and other user work on WAS Liberty,
    b. **WebContainer** threads are application threads processing HTTP work on Traditional WAS,
    c. **Inbound…** threads are WAS Liberty threads processing new inbound user requests,
    d. **GC Slave** threads are JVM threads processing garbage collection,
    e. **JIT Comp…** threads are JVM threads processing Just-in-Time (JIT) compilation,
    f. etc.

In the above example, the top threads are mostly **Default Executor** threads, each using about (0.125 / 4) * 100 ~= 3.125% of total CPU which means that most of the CPU usage is application threads handling user work, spread about evenly across threads.

As in the case of top, the **top -H** command may be run in interactive mode and could be considered an even better place to start when you begin investigating a system; however, note that **top -H** is much more expensive than top (especially if you don't provide a particular PID with **-p**) because it must traverse the data for all PIDs and all TIDs. Therefore, if you want to use **top -H** in interactive mode, consider using a large interval such as 10 seconds or more:





8 IBM Java and OpenJ9 Thread Dumps

Thread dumps are snapshots of process activity, including the thread stacks that show what each thread is doing. Thread dumps are one of the best places to start to investigate problems. If a lot of threads are in similar stacks, then that behavior might be an issue or a symptom of an issue.

For IBM Java or OpenJ9, a thread dump is also called a javacore or javadump. HotSpot-based thread dumps are covered elsewhere[21].

This exercise will demonstrate how to review thread dumps in the free IBM Thread and Monitor Dump Analyzer (TMDA) tool[22].

8.1 Thread Dumps Theory

An IBM Java or OpenJ9 thread dump is generated in a **javacore*.txt** in the working directory of the process with a snapshot of process activity, including:

- Each Java thread and its stack.
- A list of all Java synchronization monitors, which thread owns each monitor, and which threads are waiting for the lock on a monitor.

[21] https://publib.boulder.ibm.com/httpserv/cookbook/Troubleshooting-Troubleshooting_Java-Troubleshooting_Oracle_Java.html#Troubleshooting-Troubleshooting_Oracle_Java-Thread_Dump
[22] https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=2245aa39-fa5c-4475-b891-14c205f7333c

- Environment information, including Java command line arguments and operating system ulimits.
- Java heap usage and information about the last few garbage collections.
- Detailed native memory and classloader information.

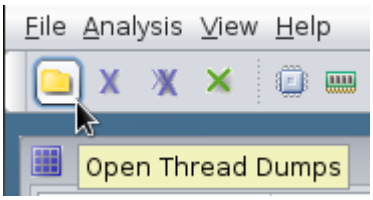
Thread dumps generally do not contain sensitive information about user requests, but they may contain sensitive information about the application or environment, so they should be treated sensitively.

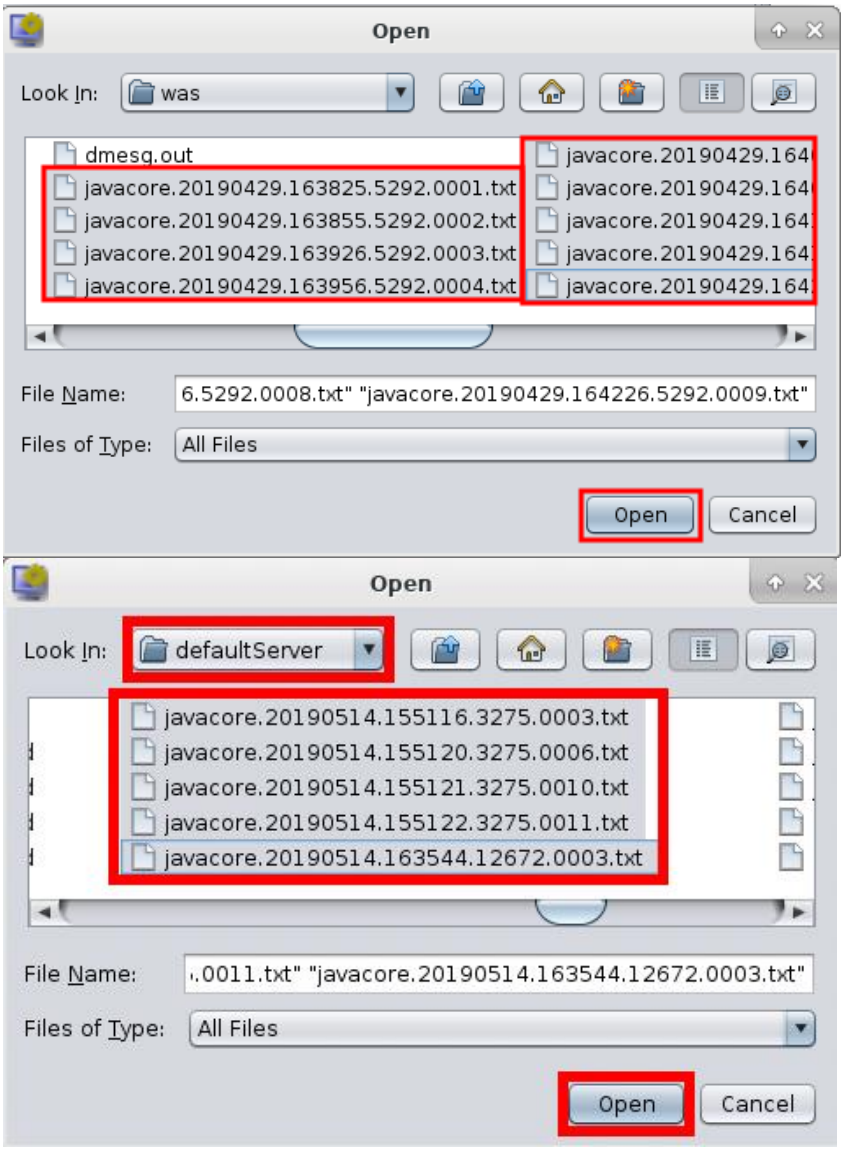
8.2 Thread Dumps Lab

We will review the thread dumps gathered by linperf.sh above:

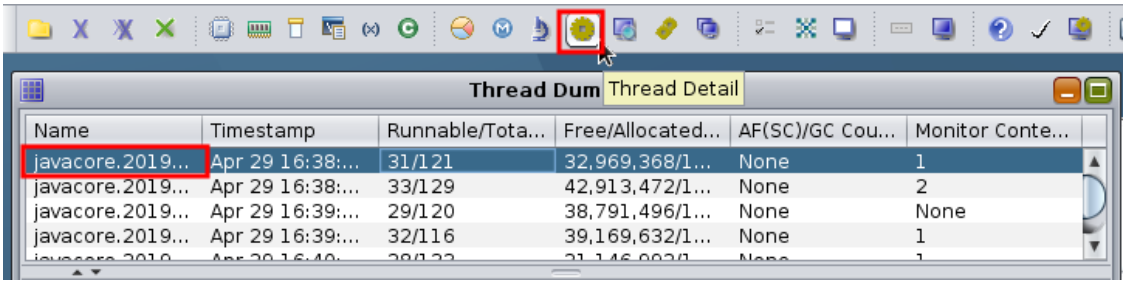
1. Open **/opt/programs/** in the file browser and double click on **TMDA**:

2. Click Open Thread Dumps and select all of the **javacore*.txt** files using the Shift key. These may be in your home directory (/home/was) if you moved them above; otherwise, they're in the default working directory of **/opt/ibm/wlp/output/defaultServer**:



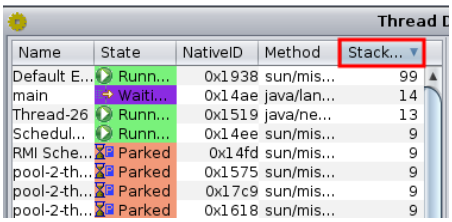


3. Select a thread dump and click the **Thread Detail** button:

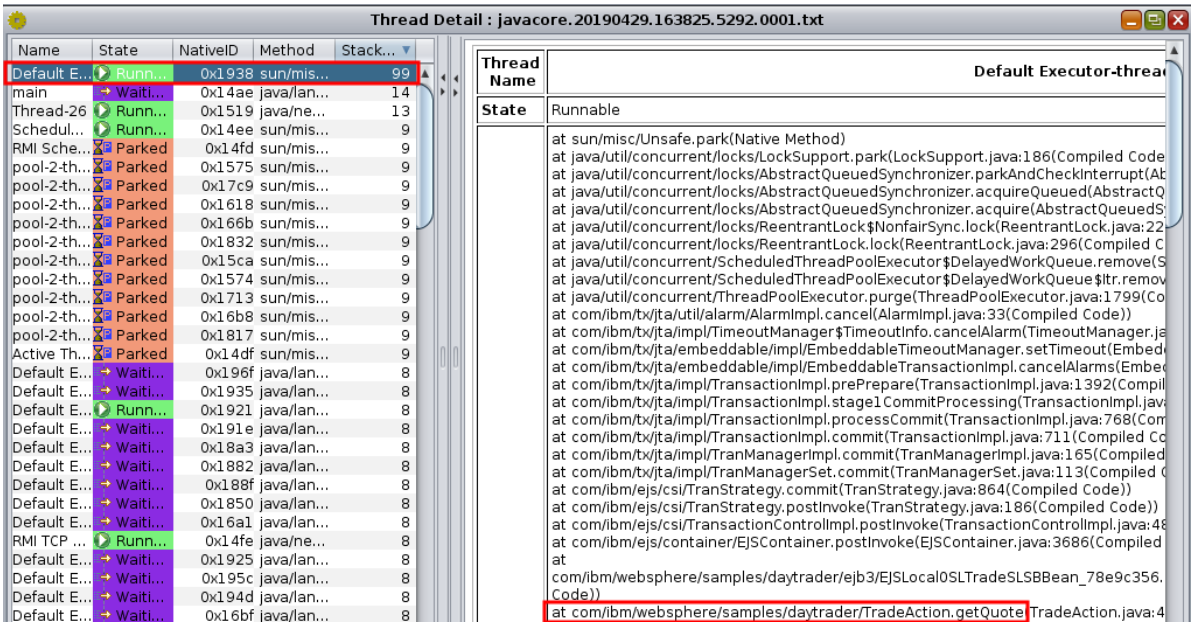


4. Click on the **Stack Depth** column to sort by thread stack depth in ascending order.

5. Click on the **Stack Depth** column again to sort again in descending order:

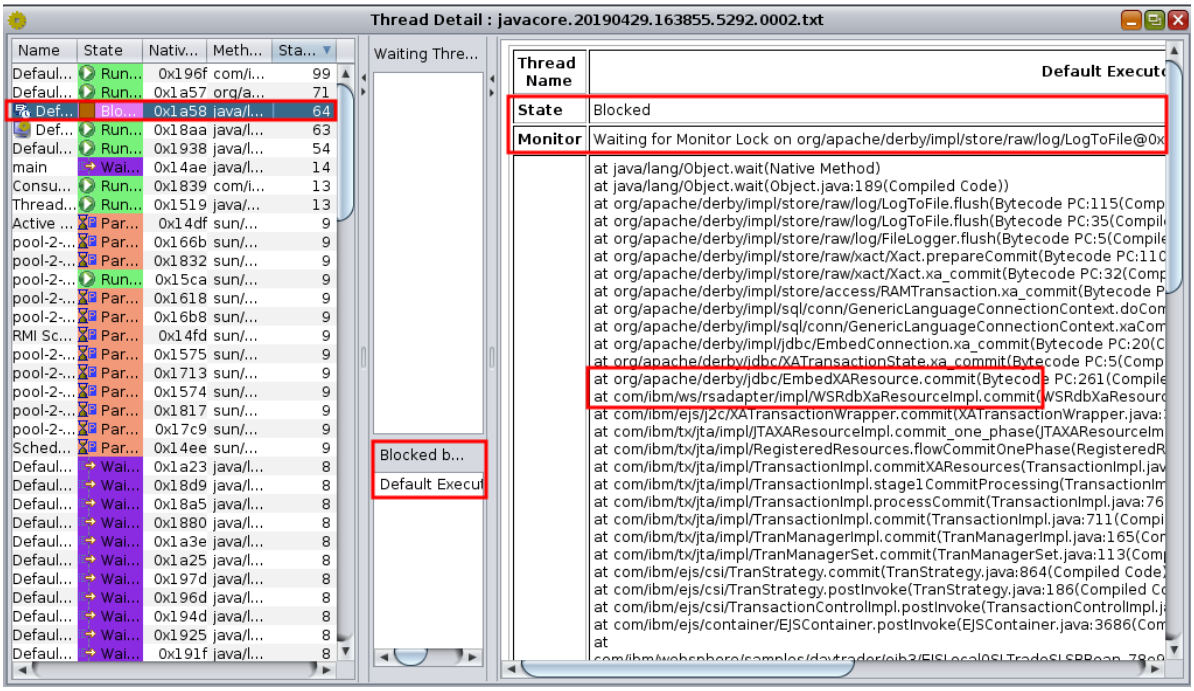


6. Generally, the threads of interest are those with stack depths greater than ~20. Select any such rows and review the stack on the right (if you don't see any, then close this thread dump and select another from the list):



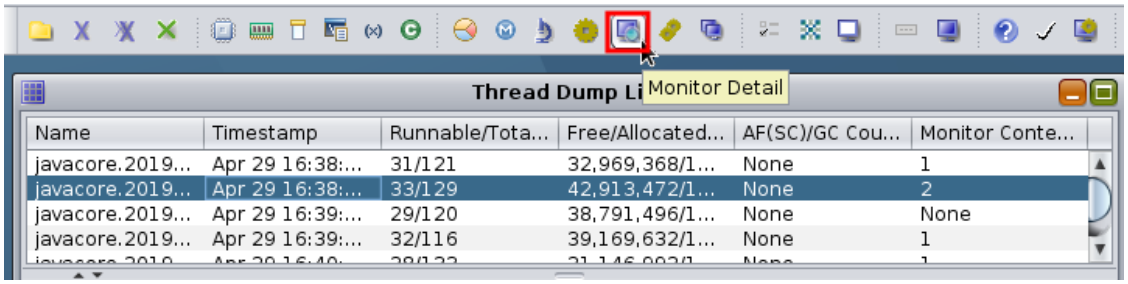
1. Generally, to understand which code is driving the thread, skip any non-application stack frames. In the above example, the first application stack frame is TradeAction.getQuote.
2. Thread dumps are simply snapshots of activity, so just because you capture threads in some stack does not mean there is necessarily a problem. However, if you have a large number of thread dumps, and an application stack frame appears with high frequency, then this may be a problem or an area of optimization. You may send the stack to the developer of that component for further research.

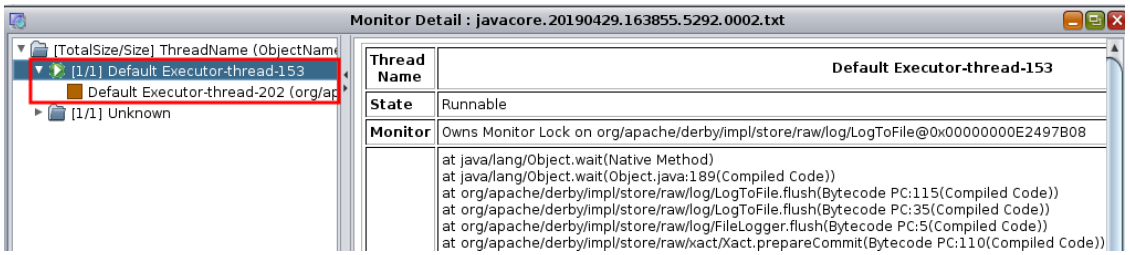
7. In some cases, you may see that one thread is blocked on another thread. For example:



1. The **Monitor** line shows which monitor this thread is waiting for, and the stack shows the path to the request for the monitor. In this example, the application is trying to commit a database transaction. This lab uses the Apache Derby database engine which is not a very scalable database. In this example, optimizing this bottleneck may not be easy and may require deep Apache Derby expertise.
2. You may click on the thread name in the **Blocked by** view to quickly see the thread stack of the other thread that owns the monitor.
3. Lock contention is a common cause of performance issues and may manifest with poor performance and low CPU usage.

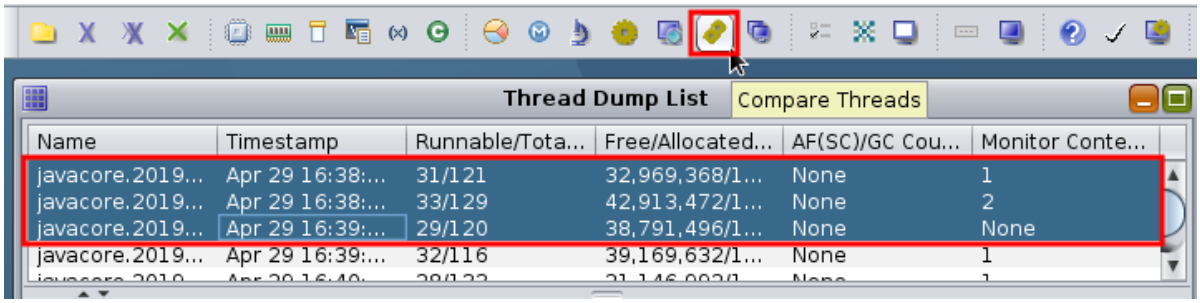
8. An alternative way to review lock contention is by selecting a thread dump and clicking **Monitor Detail**:

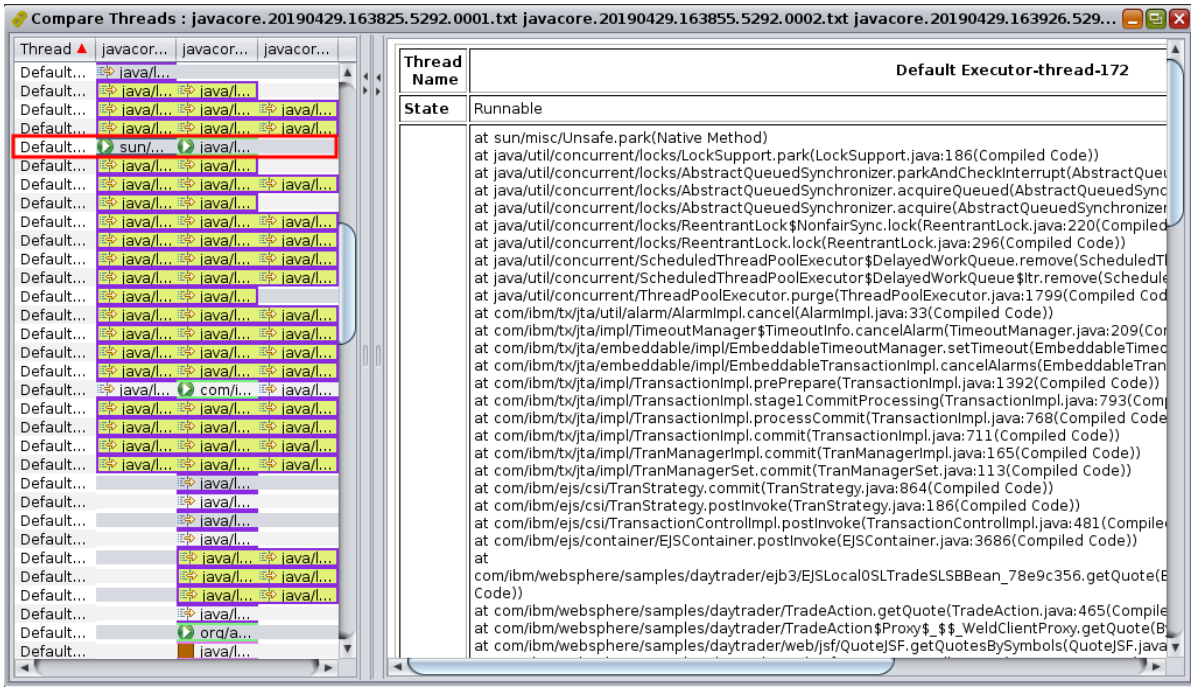




1. This shows a tree view of the monitor contention which makes it easier to explore the relationships and number of threads contending on monitors. In the above example, **Default Executor-thread-153** owns the monitor and **Default Executor-thread-202** is waiting for the monitor.

9. You may also select multiple thread dumps and click the **Compare Threads** button to see thread movement over time:

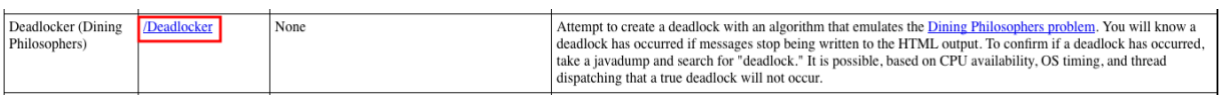




1. Each column is a thread dump and shows the state of each thread (if available) over time. Generally, you're interested in thread that are runnable (Green Arrow) or otherwise in the same concerning top stack frame. Click on each cell in that row and review the thread dump on the right. If the thread dump is always in the same stack, this is a potential issue. If the thread stack is changing a lot, then this is usually normal behavior.
2. In general, focus on the main application thread pools such as DefaultExecutor, WebContainer, etc.

Next, let's simulate a hung thread situation and analyze the problem with thread dumps:

1. Open a browser to http://localhost:9080/swat/
2. Scroll down and click on http://localhost:9080/swat/Deadlocker:



3. After about 30 seconds, gather a thread dump of the Liberty process by sending it the **SIGQUIT (3)** signal. Although the name of the signal includes the word "QUIT", the signal is

captured by the JVM, the JVM pauses for a few hundred milliseconds to produce the thread dump, and then the JVM continues. This same command is performed by **linperf.sh**. It is a quick and cheap way to quickly understand what your JVM is doing:

```
kill -3 $(pgrep -f defaultServer)
```

a. Note that here we are using a sub-shell to send the output of the pgreg command (which finds the PID of defaultServer) as the argument for the kill command.

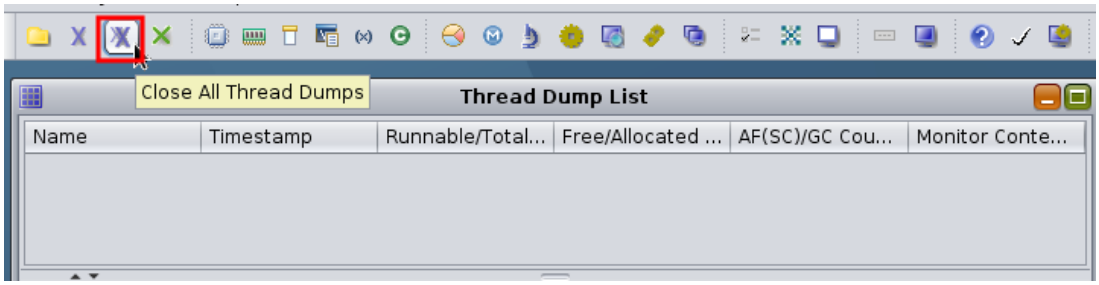
b. This can be simplified even further with the **pkill** command which combines **pgrep** functionality:

```
pkill -3 -f defaultServer
```

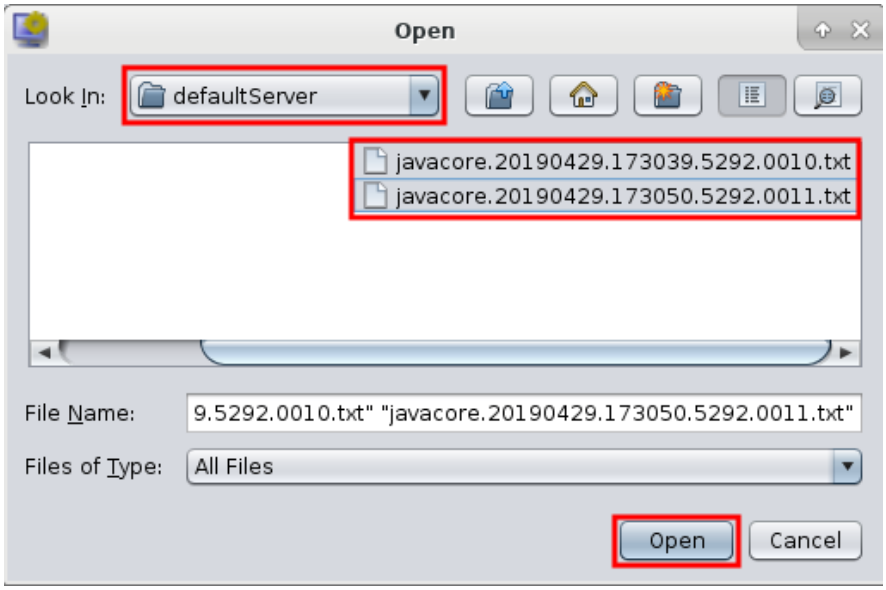
4. Wait about 30 seconds and perform the same command:

```
pkill -3 -f defaultServer
```

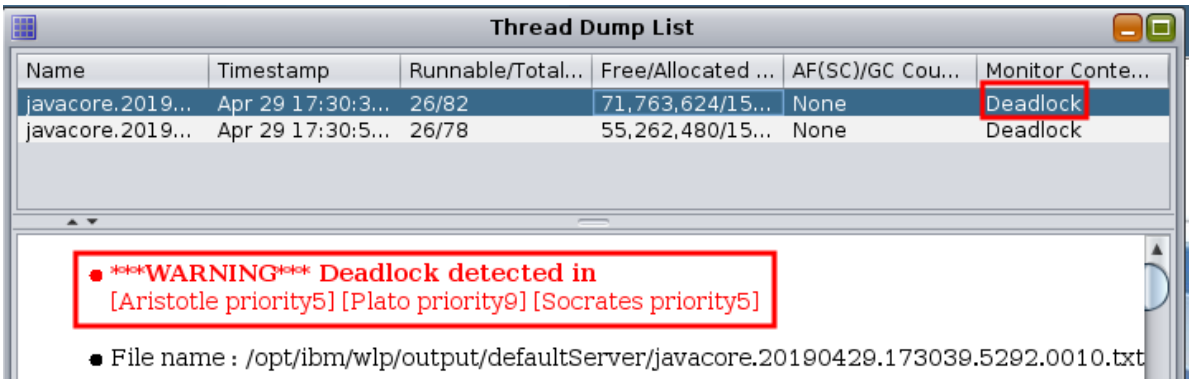
5. In the TMDA tool, clear the previous list of thread dumps:



6. Click **File** > **Open Thread Dumps** and navigate to **/opt/ibm/wlp/output/defaultServer** and select both thread dumps and click **Open**:

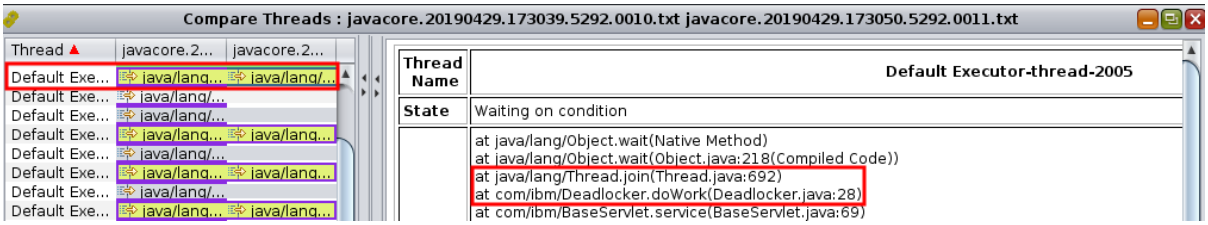


7. When you select the first thread dump, TMDA will warn you that a deadlock has been detected:

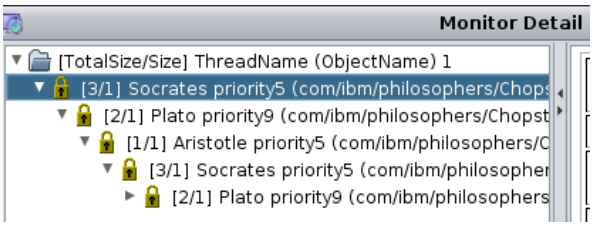


   a. Deadlocks are not common and mean that there is a bug in the application or product.

8. Use the same procedure as above to review the **Monitor Details** and **Compare Threads** to find the thread that is stuck. In this example, the **DefaultExecutor** application thread actually spawns threads and waits for them to finish, so the application thread is just in a Thread.join:

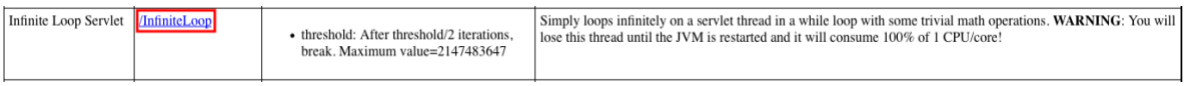


9. The actual spawned threads are named differently and show the blocking:



Next, let's combine what we've learned about the **top -H** command and thread dumps to simulate a thread that is using a lot of CPU:

1. Go to http://localhost:9080/swat/
2. Scroll down and click on http://localhost:9080/swat/InfiniteLoop:

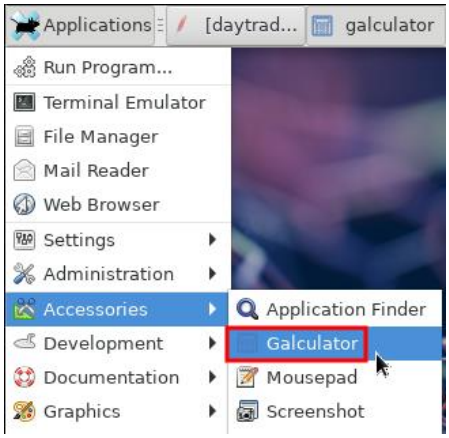


3. Go to the container terminal and start **top -H** with a 10 second interval:

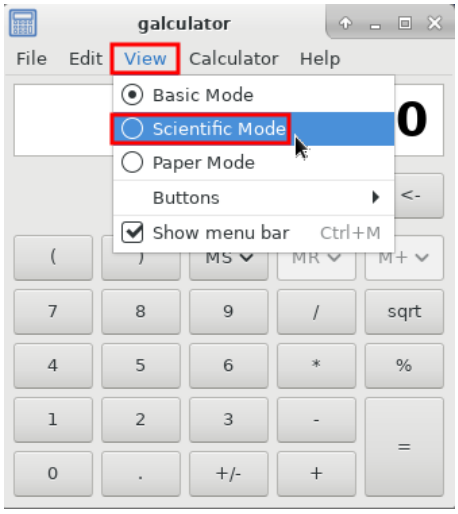
```
top -H -d 10
```

```
top - 17:48:13 up  2:34,  0 users,  load average: 0.92, 0.42, 0.28
Threads: 485 total,   2 running, 483 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.6 us,  0.3 sy,  0.0 ni, 74.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  11993.4 total,   1454.5 free,   1774.4 used,   8764.5 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   9896.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
22129 was       20   0 3183240 243412  34792 R  99.9   2.0   1:41.74 Default E+
22007 was       20   0 3183240 243412  34792 S   0.6   2.0   0:00.64 JIT Sampl+
  161 was       20   0  494588 106308  40600 S   0.3   0.9   0:34.47 Xvnc
```

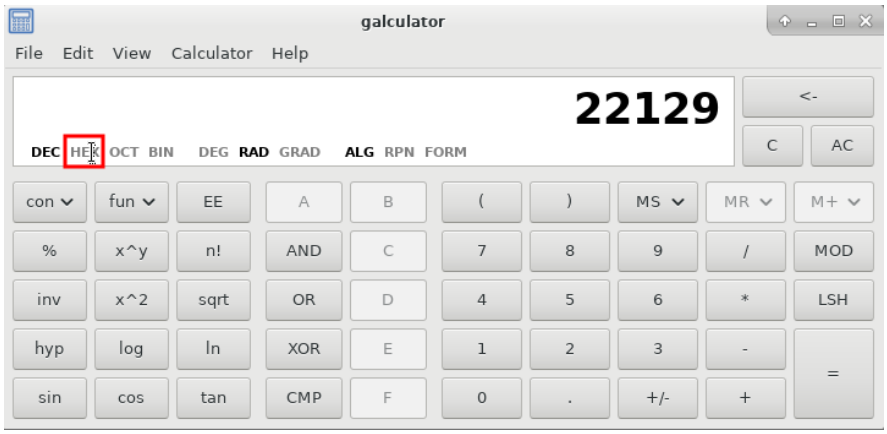
4. Notice that a single thread is consistently consuming ~100% of a single CPU thread.
5. Convert the PID to hexadecimal. In the example above, 22129 = 0x5671.
    a. In the container, open Galculator:



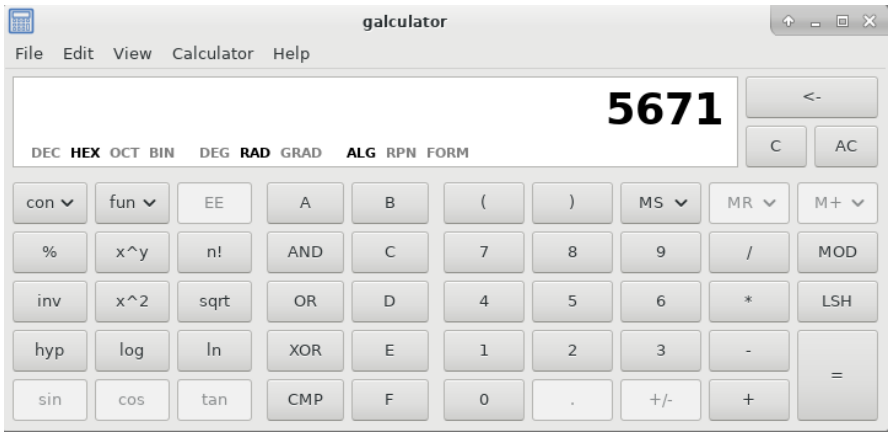
    b. Click View > Scientific Mode:



c. Enter the decimal number (in this example, 22129), and then click on **HEX**:



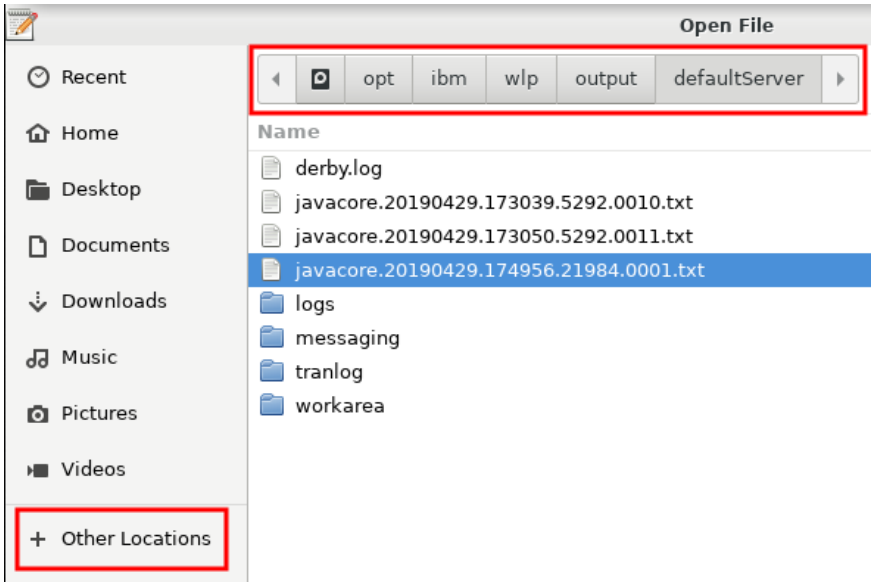
d. The result is 0x5671:



6. Take a thread dump of the parent process:

```
pkill -3 -f defaultServer
```

7. Open the most recent thread dump from **/opt/ibm/wlp/output/defaultServer/** in a text editor such as **mousepad**:



8. Search for the native thread ID in hex (in this example, 0x5671) to find the stack trace consuming the CPU (if captured during the thread dump):

```
3XMTHREADINFO      "Default Executor-thread-16" J9VMThread:0x0000000001D87000, omrthread_t:0x00007F8518037A60, java/lang/Thread:
3XMJAVALTHREAD            (java/lang/Thread getId:0x5C, isDaemon:true)
3XMTHREADINFO1            (native thread ID:0x5671, native priority:0x5, native policy:UNKNOWN, vmstate:CW, vm thread flags:0x00
3XMTHREADINFO2            (native stack address range from:0x00007F858C73B000, to:0x00007F858C77B000, size:0x40000)
3XMCPUTIME                CPU usage total: 205.198890739 secs, current category="Application"
3XMHEAPALLOC              Heap bytes allocated since last GC cycle=0 (0x0)
3XMTHREADINFO3            Java callstack:
4XESTACKTRACE                at com/ibm/InfiniteLoop.doWork(InfiniteLoop.java:27(Compiled Code))
4XESTACKTRACE                at com/ibm/BaseServlet.service(BaseServlet.java:69)
4XESTACKTRACE                at javax/servlet/http/HttpServlet.service(HttpServlet.java:790)
4XESTACKTRACE                at com/ibm/ws/webcontainer/servlet/ServletWrapper.service(ServletWrapper.java:1255)
```

9 Garbage Collection

Garbage collection (GC) automatically frees unused objects. Healthy garbage collection is one of the most important aspects of Java programs. The proportion of time spent in garbage collection versus application time should be less than 10% and ideally less than 1%[23].

This lab will demonstrate how to enable verbose garbage collection in WAS for the sample DayTrader application, exercise the application using Apache JMeter, and review verbose garbage collection data in the free IBM Garbage Collection and Memory Visualizer (GCMV) tool[24].

9.1 Garbage Collection Theory

All major Java Virtual Machines (JVMs) are designed to work with a maximum Java heap size. When the Java heap is full (or various sub-heaps), an allocation failure occurs and the garbage collector will run to try to find space. Verbose garbage collection (verbosegc) prints detailed information about each one of these allocation failures.

Always enable verbose garbage collection, including in production (benchmarks show an overhead of ~0.13% for IBM Java[25]), using the options to rotate the verbosegc logs. For IBM Java[26] - 5 historical files of roughly 20MB each:

```
-Xverbosegclog:verbosegc.%seq.log,5,50000
```

9.2 Garbage Collection Lab

Add the verbosegc option to the jvm.options file:

1. Stop the JMeter test.
2. Stop the Liberty server.

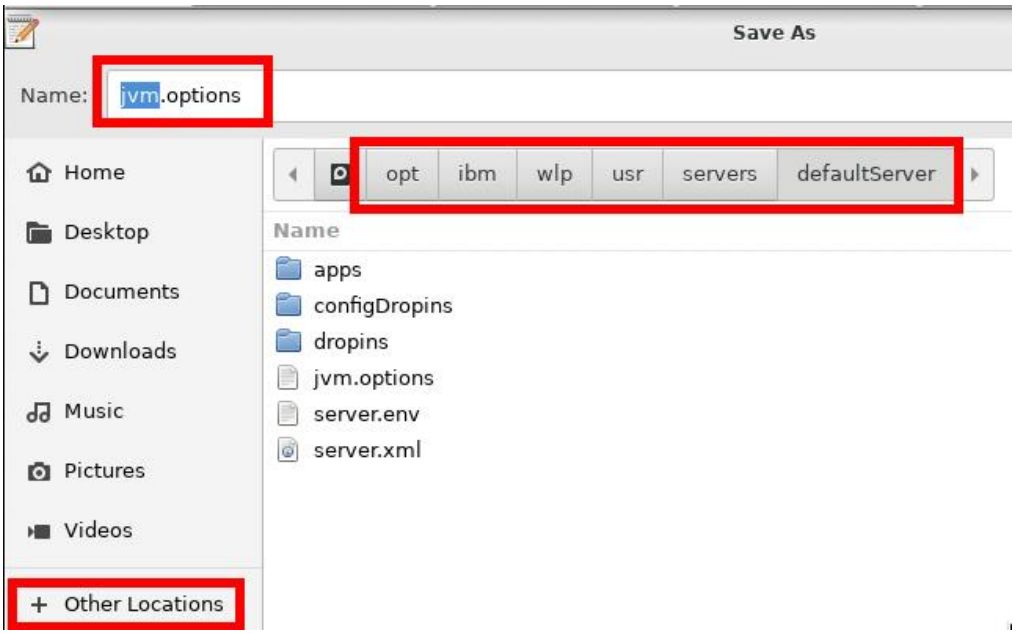
```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Open a text editor such as mousepad and add the following line to it:

```
-Xverbosegclog:logs/verbosegc.%seq.log,5,50000
```

[23] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-Garbage_Collection_and_Memory_Visualizer_GCMV.html#Major_Tools-Garbage_Collection_and_Memory_Visualizer_GCMV-Analysis
[24] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-Garbage_Collection_and_Memory_Visualizer_GCMV.html
[25] https://publib.boulder.ibm.com/httpserv/cookbook/Java-IBM_Java_Runtime_Environment.html#Java-IBM_Java_Runtime_Environment-Garbage_Collection-Verbose_garbage_collection_verbosegc
[26] http://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.lnx.80.doc/diag/appendixes/cmdline/xverbosegclog.html

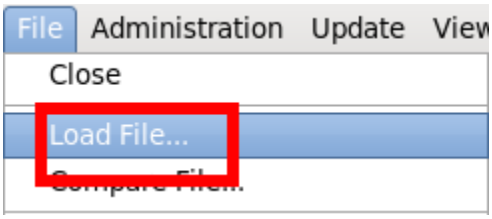
4. Save the file to **/opt/ibm/wlp/usr/servers/defaultServer/jvm.options**



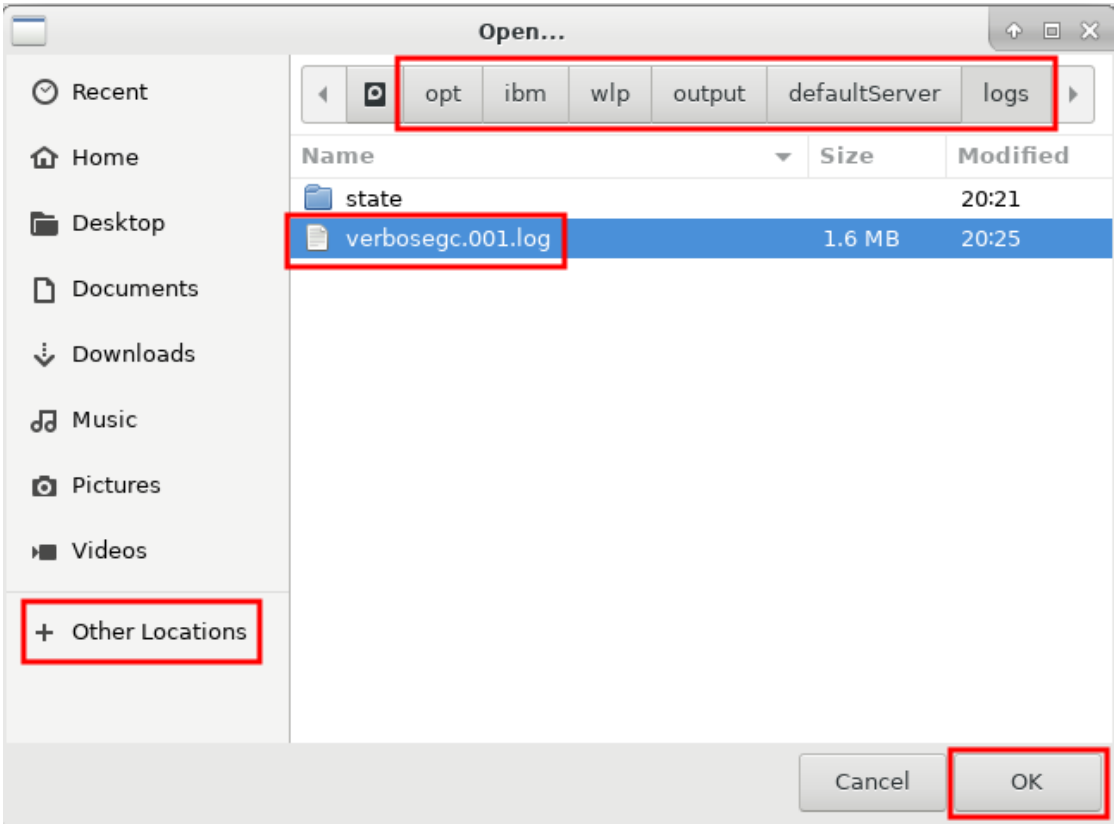
5. Start the Liberty server

```
/opt/ibm/wlp/bin/server start defaultServer
```

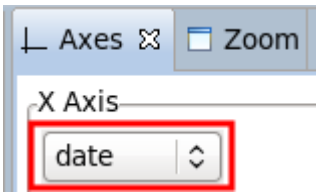
6. Start the JMeter test
7. Run the test for about 5 minutes
8. Stop the JMeter test
9. Open **/opt/programs/** in the file browser and double click on **GCMV**:
10. Click **File** > **Load File...** and select the **verbosegc.001.log** file. For example:



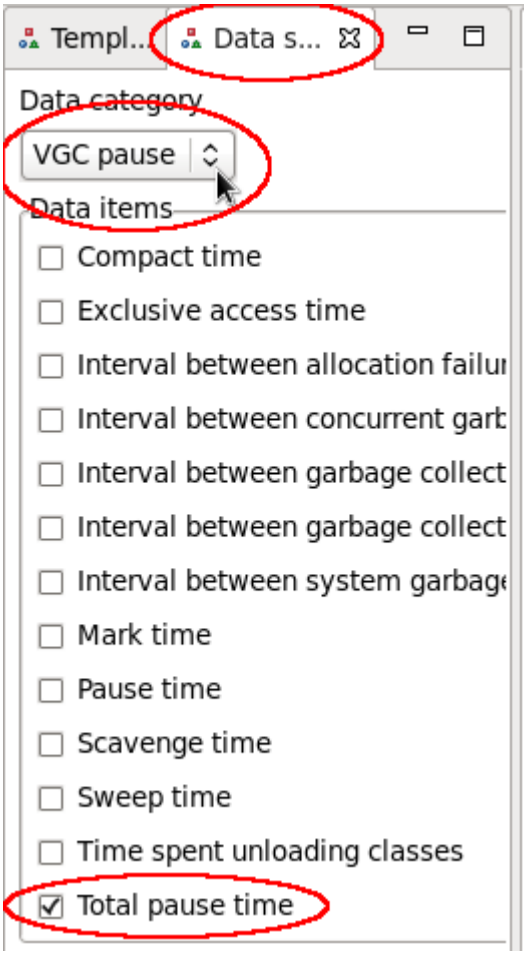
11. Select **/opt/ibm/wlp/output/defaultServer/logs/verbosegc.001.log**



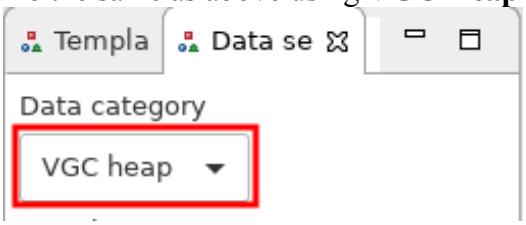
12. Once the file is loaded, you will see the default line plot view. It is common to change the **X-axis** to **date** to see absolute timestamps:

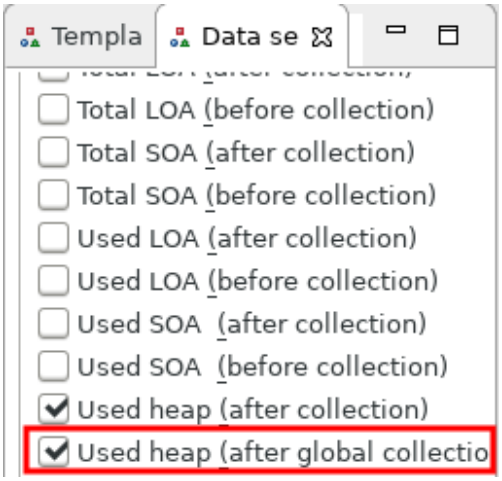


13. Click the **Data Selector** tab in the top left, choose **VGC Pause** and check **Total pause time** to add the total garbage collection pause time plot to the graph:

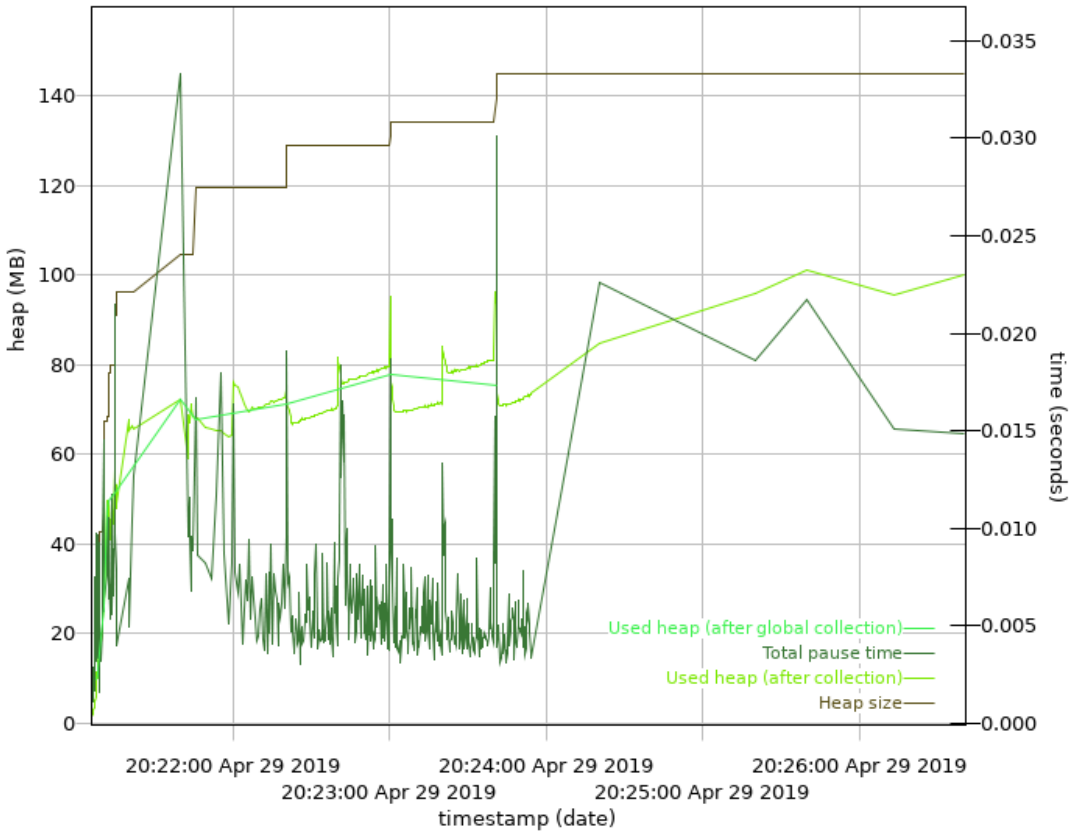


14. Do the same as above using **VGC Heap** and check **Used heap (after global collection)**:





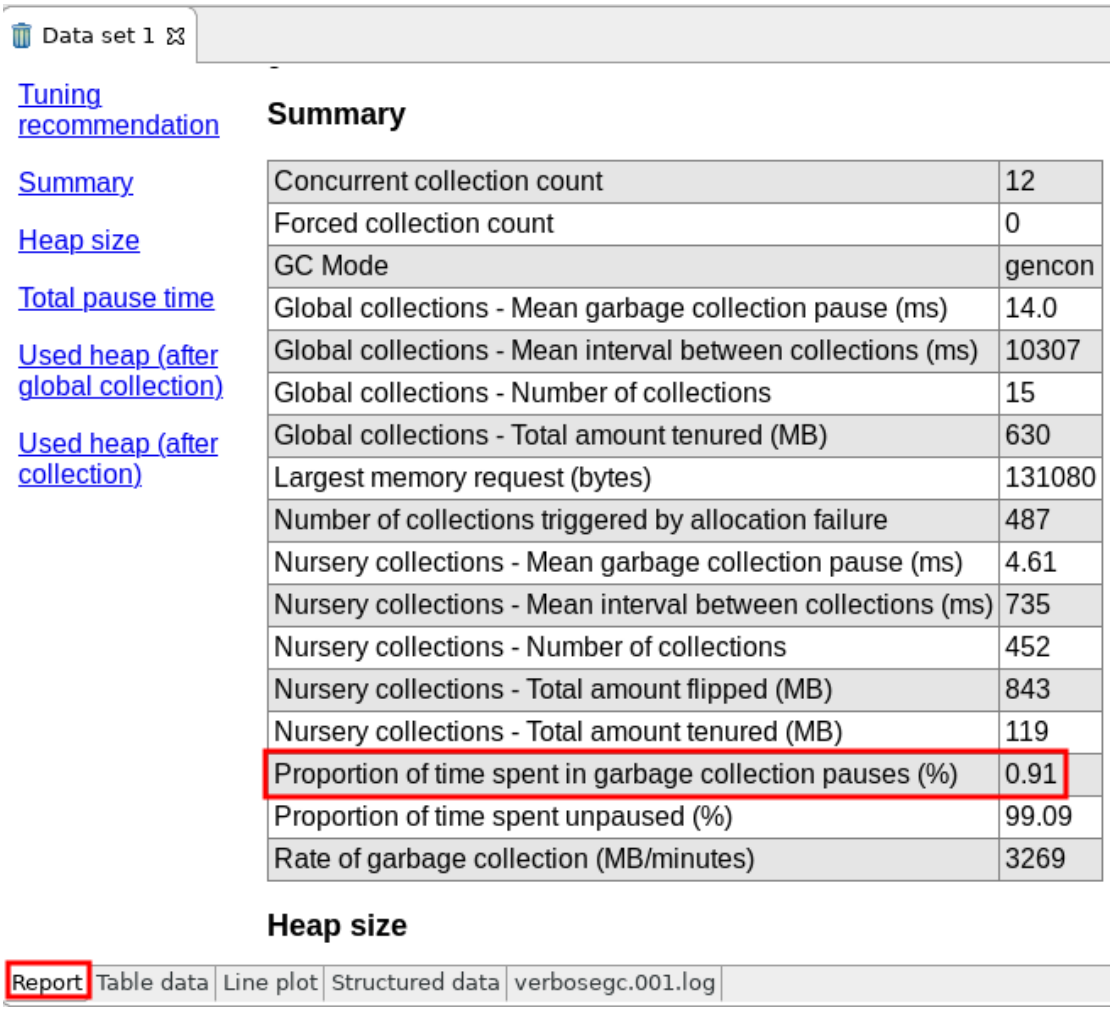
15. Observe the heap usage and pause time magnitude and frequency over time. For example:



1. This shows that the heap size reaches 145MB and the heap usage (after global collection)

reached ~80MB.

16. More importantly, we want to know the proportion of time spent in GC. Click the **Report** tab and review the **Proportion of time spent in garbage collection pauses (%)**:



1. If this number is less than 1%, then this is very healthy. If it's less than 5% then it's okay. If it's less than 10%, then there is significant room for improvement. If it's greater than 10%, then this is concerning.

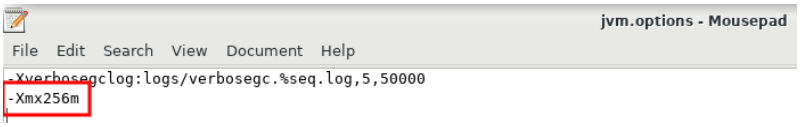
Next, let's simulate a memory issue.

1. Stop the JMeter test.
2. Stop Liberty:

```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Edit **/opt/ibm/wlp/usr/servers/defaultServer/jvm.options**, add an explicit maximum heap size of 256MB on a new line and save the file:

```
-Xmx256m
```

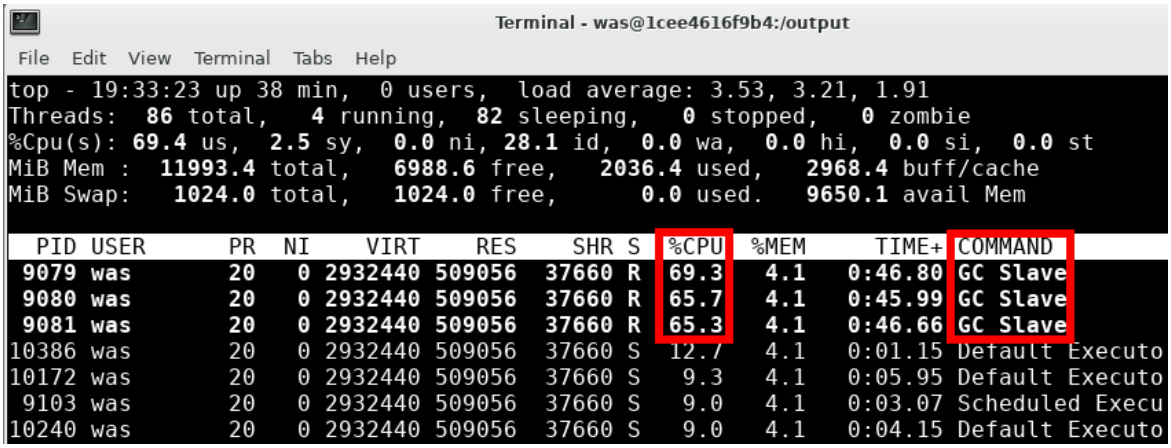


4. Start Liberty

```
/opt/ibm/wlp/bin/server start defaultServer
```

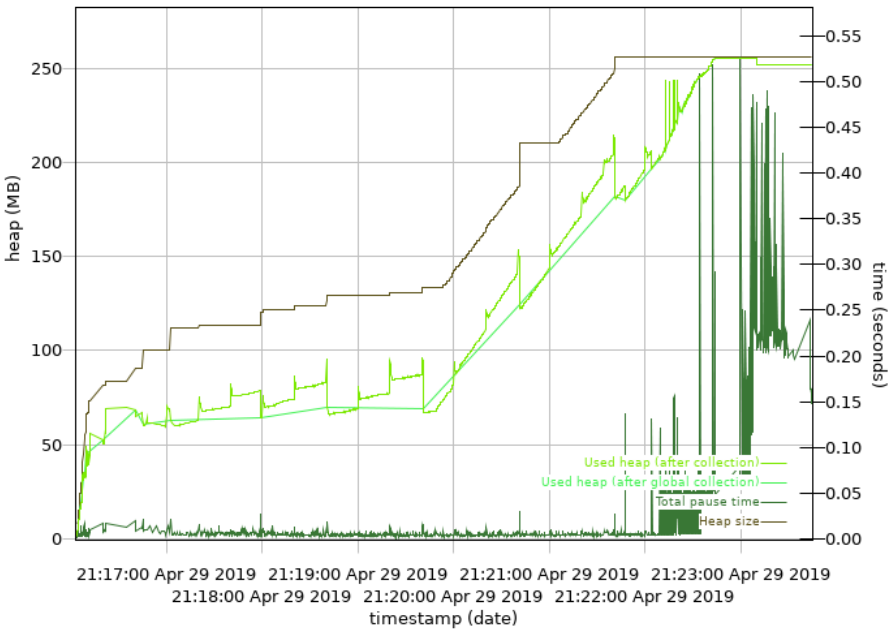
5. Start the JMeter test.
6. Let the JMeter test run for about 5 minutes.
7. Stop the JMeter test.
8. Open your browser to http://localhost:9080/swat/AllocateObject?size=1048576&iterations=300&waittime=1000&retainData=true
    1. This will allocate three hundred 1MB objects with a delay of 1 second between each allocation, and hold on to all of them to simulate a leak.
    2. This will take about 5 minutes to run and you can watch your browser output for progress.
    3. You can run **top -H** while this is running. As memory pressure builds, you'll start to see **GC Slave** threads consuming most of the CPUs instead of application threads (garbage collection also happens on the thread where the allocation failure occurs, so you may also see a single application thread consuming a similar amount of CPU as the GC Slave threads):

```
top -H -p $(pgrep -f defaultServer) -d 5
```



4. At some point, browser output will stop because the JVM has thrown an OutOfMemoryError.

9. Close and re-open the **verbosegc*log** file in GCMV:



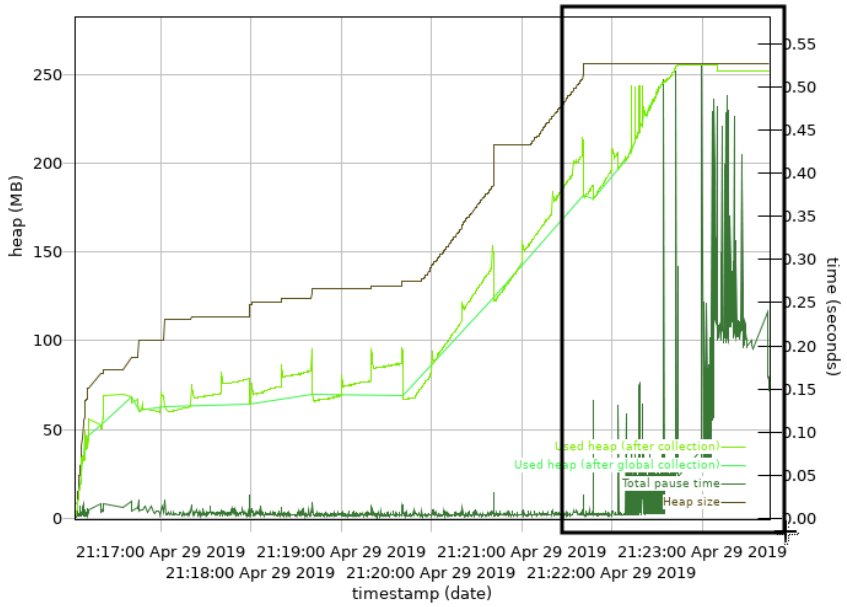
   1. We can quickly see how the heap usage reaches 256MB and the pause time magnitude and durations increase significantly.

10. Click on the **Report** tab and review the **Proportion of time spent in garbage collection pauses (%)**:

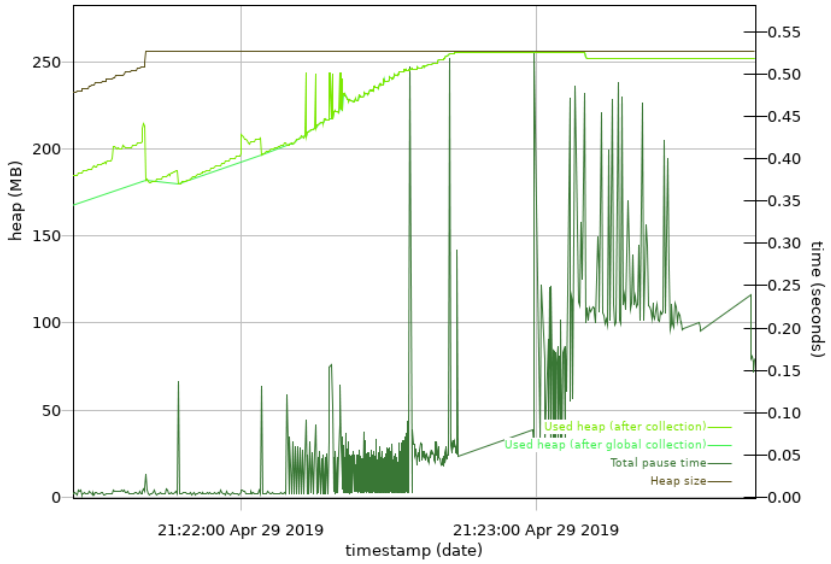
Proportion of time spent in garbage collection pauses (%)	11.29

11. At first, 11% might not seem too bad and doesn't line up with what we know about what happened. This is because, by default, the GCMV Report tab shows statistics for the entire duration of the verbosegc log file. Since we had run the JMeter test for 5 minutes and it was healthy, the average proportion of time in GC is lower for the whole duration.

12. Click on the **Line plot** tab and zoom in to the area of high pause times by using your mouse button to draw a box around those times:

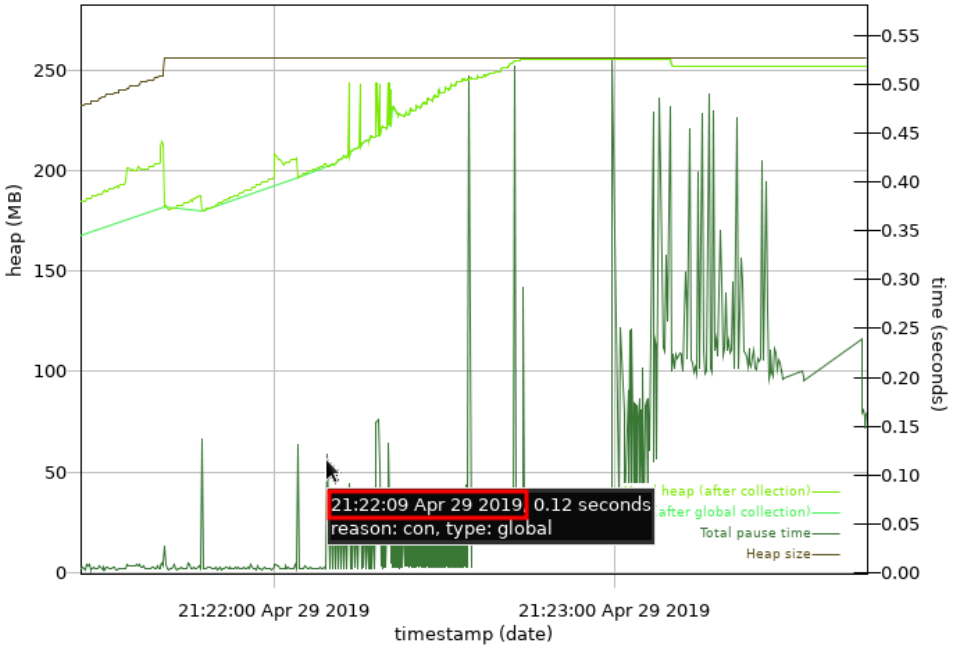


13. This will zoom the view to that bounding box:

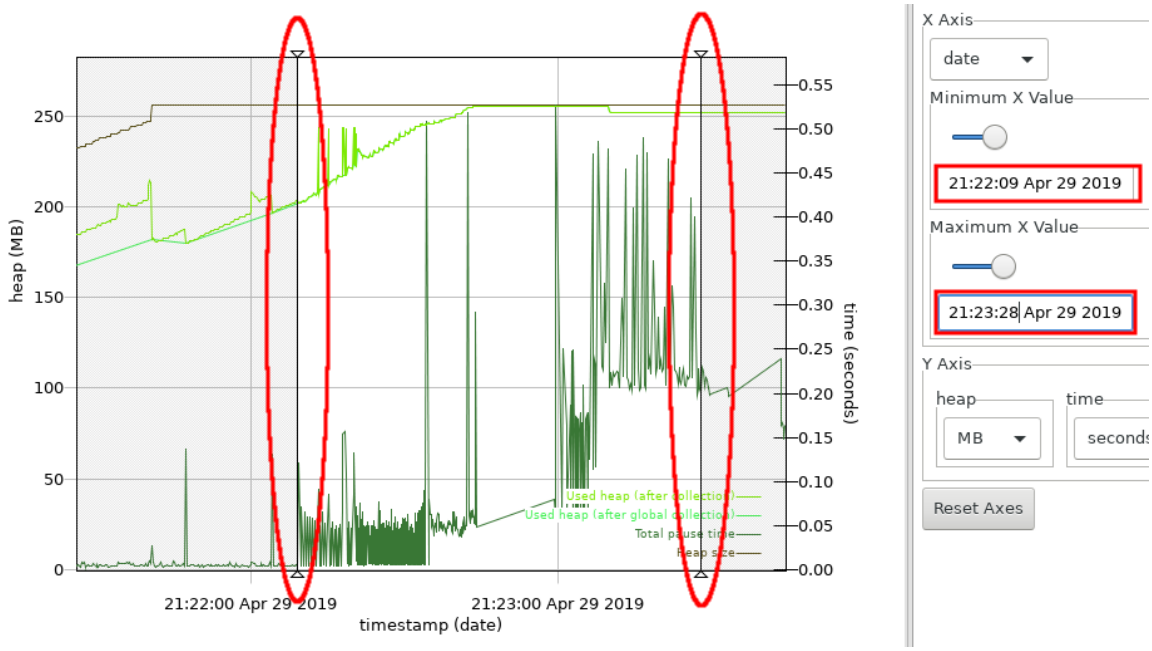


14. However, zooming in is just a visual aid. To change the report statistics, we need to match the X-axis to the period of interest.

15. Hover your mouse over the approximate start and end points of the section of concern (frequent pause time spikes) and note the times of those points (in terms of your selected X Axis type):



16. Enter each of the values in the minimum and maximum input boxes and press **Enter** on your keyboard in each one to apply the value. The tool will show vertical lines with triangles showing the area of the graph that you've cropped to.



17. Click on the **Report** tab at the bottom and observe the proportion of time spent in garbage collection for this period is very high (in this example, ~50%).

Proportion of time spent in garbage collection pauses (%)	50.7

18. This means that the application is doing very little work and is very unhealthy. In general, there are a few, non-exclusive ways to resolve this problem:
    1. Increase the maximum heap size.
    2. Decrease the object allocation rate of the application.
    3. Resolving memory leaks through heapdump analysis.
    4. Decrease the maximum thread pool size.

10 Other Topics

The above three sections – operating system CPU and memory, thread dumps, and garbage collection – are the three key elements that should be reviewed for all problems and performance issues. The rest of the lab will review other problem types and performance tuning and other types of tools.

11 Methodology

First, let's review some general tips about problem determination and performance methodology:

11.1 The Scientific Method

Troubleshooting is the act of understanding problems and then changing systems to resolve those problems. The best approach to troubleshooting is the scientific method which is basically as follows:

1. Observe and measure evidence of the problem. For example: "Users are receiving HTTP 500 errors when visiting the website."
2. Create prioritized hypotheses about the causes of the problem. For example: "I found exceptions in the logs. I hypothesize that the exceptions are creating the HTTP 500 errors."
3. Research ways to test the hypotheses using experiments. For example: "I searched the documentation and previous problem reports and the exceptions may be caused by a default setting configuration. I predict that changing this setting will resolve the problem if this hypothesis is true."
4. Run experiments to test hypotheses. For example: "Please change this setting and see if the user errors are resolved."
5. Observe and measure experimental evidence. If the problem is not resolved, repeat the steps above; otherwise, create a theory about the cause of the problem.

11.2 Organizing an Investigation

Keep track of a summary of the situation, a list of problems, hypotheses, and experiments/tests. Use numbered items so that people can easily reference things in phone calls or emails. The summary should be restricted to a single sentence for problems, resolution criteria, statuses, and next steps. Any details are in the subsequent tables. The summary is a difficult skill to learn, so try to constrain yourself to a single (short!) sentence. For example:

Summary

1. Problems: 1) Average website response time of 5000ms and 2) website error rate > 10%.
2. Resolution criteria: 1) Average response time of 300ms and 2) error rate of <= 1%.
3. Statuses: 1) Reduced average response time to 2000ms and 2) error rate to 5%.
4. Next steps: 1) Investigate database response times and 2) gather diagnostic trace.

Problems

#	Problem	Case #	Status	Next Steps
1	Average response time greater than 300ms	TS001234567	Reduced average response time to 2000ms by increasing heap size	Investigate database response times
2	Website error rate greater than 1%	TS001234568	Reduced website error rate to 5% by fixing an application bug	Run diagnostic trace for remaining errors

Hypotheses for Problem #1

#	Hypothesis	Evidence	Status
1	High proportion of time in garbage collection leading to reduced performance	• Verbosegc showed proportion of time in GC of 20% • Increased Java maximum heap size to -Xmx1g and proportion of time in GC went down to 5%	• Further fine-tuning can be done, but at this point 5% is a reasonable number
2	Slow database response times	• Thread stacks showed many threads waiting on the database	• Gather database response times

Hypotheses for Problem #2

#	Hypothesis	Evidence	Status
1	NullPointerException in com.application.foo is causing errors	• NullPointerExceptions in the logs correlate with HTTP 500 response codes	• Application fixed the NullPointerException and error rates were halved
2	ConcurrentModificationException in com.websphere.bar is causing errors	• ConcurrentModificationExceptions correlate with HTTP 500 response codes	• Gather WAS diagnostic trace capturing some exceptions

Experiments/Tests

#	Experiment/Test	Date/Time	Environment	Changes	Results
1	Baseline	2019-01-01 09:00:00 UTC to 2019-01-01 17:00:00 UTC	Production server1	None	• Average response time 5000ms • Website error rate 10%

2	Reproduce in a test environment	2019-01-02 11:00:00 UTC to 2019-01-01 12:00:00 UTC	Test server1	None	• Average response time 8000ms • Website error rate 15%
3	Test problem #1 - hypothesis #1	2019-01-03 12:30:00 UTC to 2019-01-01 14:00:00 UTC	Test server1	Increase Java heap size to 1g	• Average response time 4000ms • Website error rate 15%
4	Test problem #1 - hypothesis #1	2019-01-04 09:00:00 UTC to 2019-01-01 17:00:00 UTC	Production server1	Increase Java heap size to 1g	• Average response time 2000ms • Website error rate 10%
5	Test problem #2 - hypothesis #1	2019-01-05	Production server1	Application bugfix	• Average response time 2000ms • Website error rate 5%
6	Test problem #1 - hypothesis #2	TBD	Test server1	Gather WAS JDBC PMI	• TBD
7	Test problem #2 - hypothesis #2	TBD	Test server1	Enable WAS diagnostic trace com.ibm.foo=all	• TBD

11.3 Performance Tuning Tips

1. Performance tuning is usually about focusing on a few key variables. We will highlight the most common tuning knobs that can often improve the speed of the average application by 200% or more relative to the default configuration. The first step, however, should be to use and be guided by the tools and methodologies. Gather data, analyze it and create hypotheses: then test your hypotheses. Rinse and repeat. As Donald Knuth says: "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time […]. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail." (Donald Knuth, Structured Programming with go to Statements, Stanford University, 1974, Association for Computing Machinery)
2. There is a seemingly daunting number of tuning knobs. Unless you are trying to squeeze out every last drop of performance, we do not recommend a close study of every tuning option.
3. In general, we advocate a bottom-up approach. For example, with a typical WebSphere Application Server application, start with the operating system, then Java, then WAS, then the application, etc. Ideally, investigate these at the same time. The main goal of a performance tuning exercise is to iteratively determine the bottleneck restricting response times and throughput. For example, investigate operating system CPU and memory usage, followed by

Java garbage collection usage and/or thread dumps/sampling profilers, followed by WAS PMI, etc.

4. One of the most difficult aspects of performance tuning is understanding whether or not the architecture of the system, or even the test itself, is valid and/or optimal.
5. Meticulously describe and track the problem, each test and its results.
6. Use basic statistics (minimums, maximums, averages, medians, and standard deviations) instead of spot observations.
7. When benchmarking, use a repeatable test that accurately models production behavior, and avoid short term benchmarks which may not have time to warm up.
8. Take the time to automate as much as possible: not just the testing itself, but also data gathering and analysis. This will help you iterate and test more hypotheses.
9. Make sure you are using the latest version of every product because there are often performance or tooling improvements available.
10. When researching problems, you can either analyze or isolate them. Analyzing means taking particular symptoms and generating hypotheses on how to change those symptoms. Isolating means eliminating issues singly until you've discovered important facts. In general, we have found through experience that analysis is preferable to isolation.
11. Review the full end-to-end architecture. Certain internal or external products, devices, content delivery networks, etc. may artificially limit throughput (e.g. Denial of Service protection), periodically mark services down (e.g. network load balancers, WAS plugin, etc.), or become saturated themselves (e.g. CPU on load balancers, etc.).

12 Heap Dumps

Heap dumps are snapshots of Java objects in a process. On IBM Java, the two heapdump formats are Portable Heapdump (PHD) and System Dump (core), with a core including memory contents and thread stacks.

This lab will demonstrate how to exercise the sample DayTrader application using Apache JMeter, request a heap dump of WebSphere Application Server, and review the heapdump file in the free IBM Memory Analyzer Tool (MAT)[27].

12.1 Heap Dump Theory

Heap dumps are used for investigating the causes of OutOfMemoryErrors, sizing applications, and reviewing memory contents under various conditions. Starting with WAS 8.0.0.2, the first OutOfMemoryError produces both a PHD and system core file.

System dumps are a superset of PHD files and are generally recommended, although they may contain sensitive customer information. The general recommendation is to always use system cores, and if security is a concern, extract a PHD file from the core using jextract for normal usage, and save the core file in case it is needed for advanced analysis.

A few key definitions:

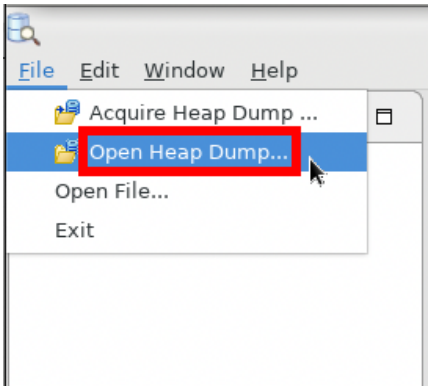
- The dominator tree is a transformation of the graph which creates a spanning tree, removes cycles, and models the keep-alive dependencies.
- The retained set of X is the set of objects which would be removed by the garbage collector when X is garbage collected.

[27] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Memory_Analyzer_Tool.html#Major_Tools-IBM_Memory_Analyzer_Tool_MAT-Standalone_Installation

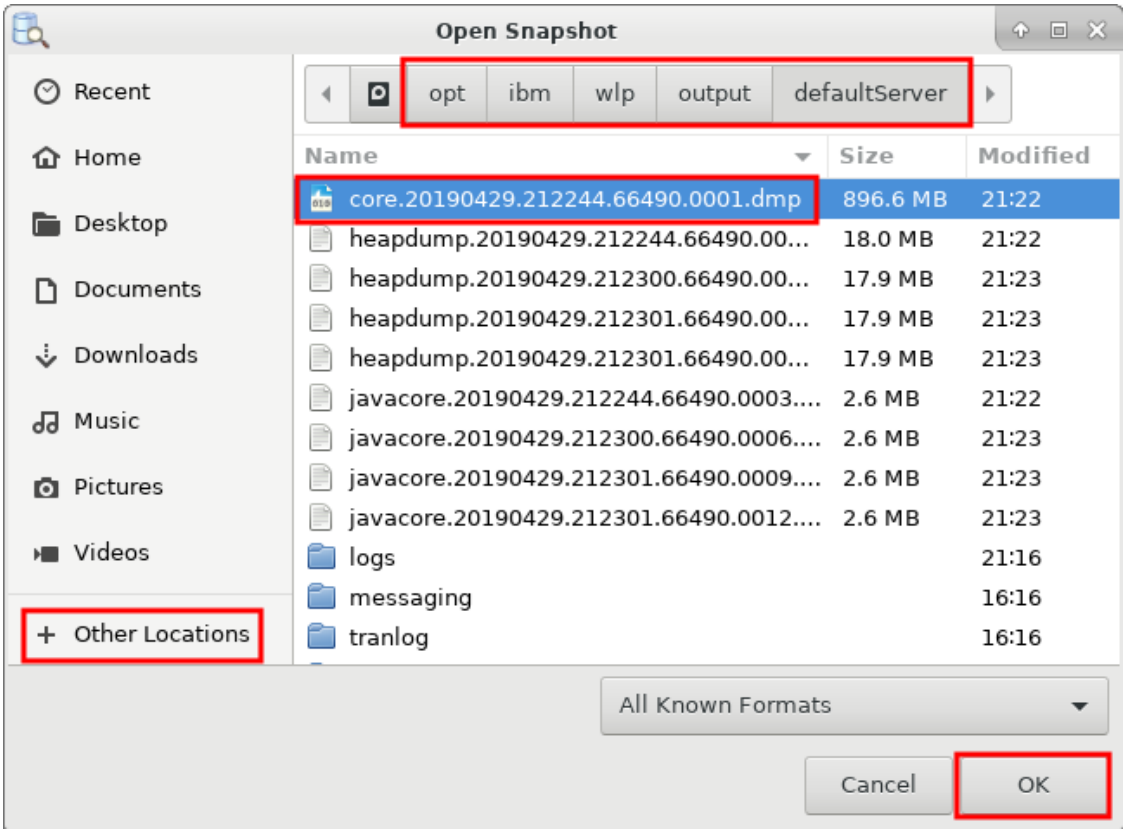
A significant difference between PHDs and core dumps is that PHDs only have the object relationships so they do not contain sensitive user information (although they do contain class names which may be considered sensitive by the application team), whereas core dumps contain all the memory at the time of the dump which may include user information. Core dumps should be treated with very high sensitivity and encrypted if necessary using a tool such as gpg[28].

12.2 Heap Dump Lab

1. Open **/opt/programs/** in the file browser and double click on **MAT**.
2. Click **File** > **Open Heap Dump...**

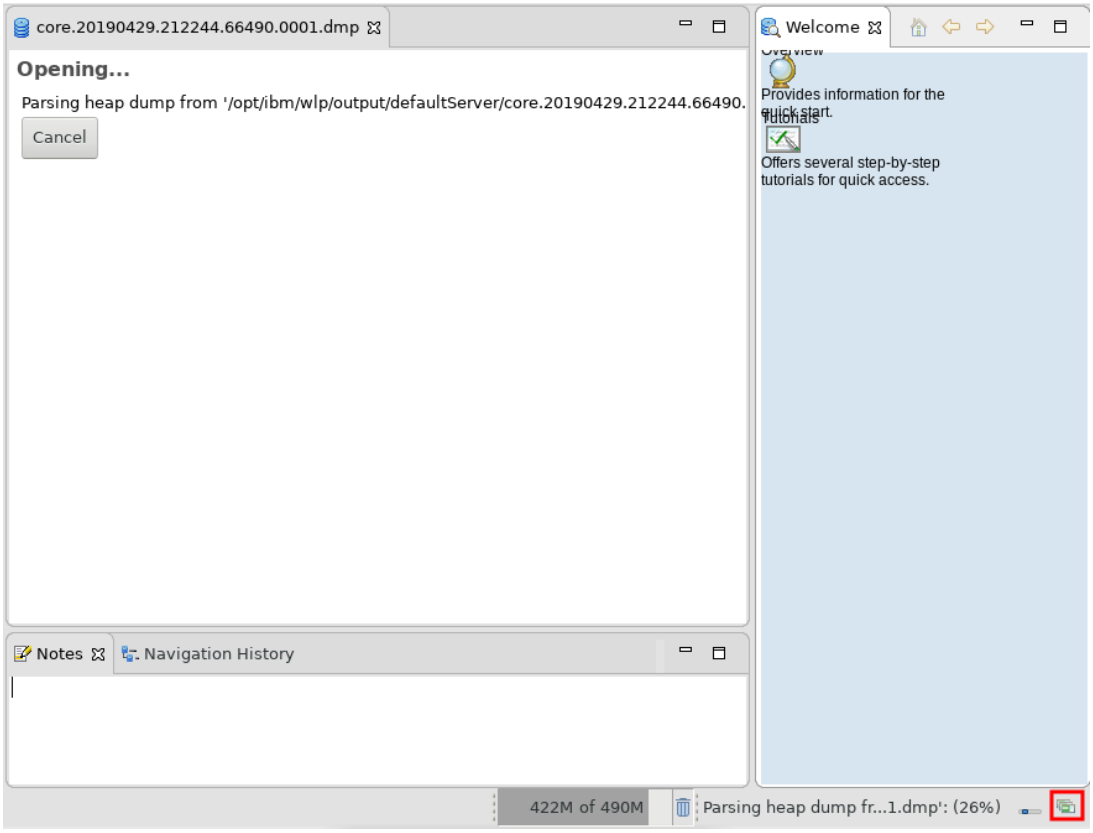


3. Select the **core.*.dmp** file produced in the previous garbage collection lab:

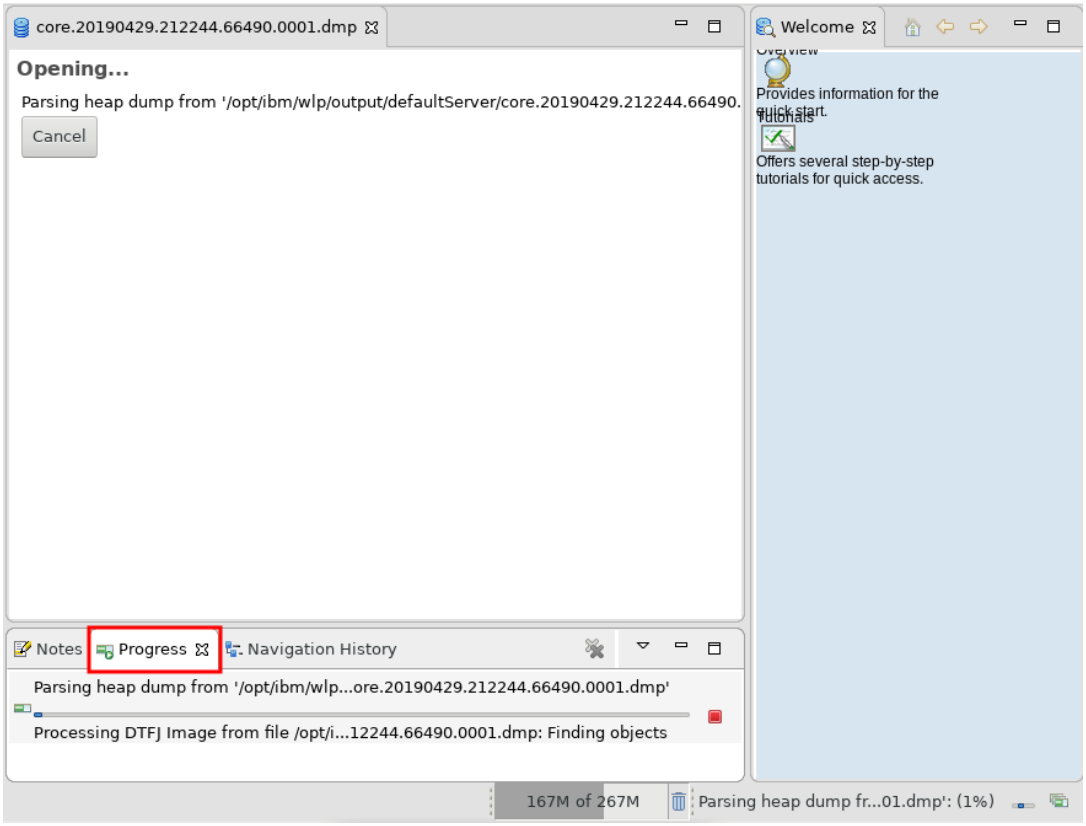


[28] https://publib.boulder.ibm.com/httpserv/cookbook/Appendix-POSIX.html#Resources-POSIX-gpg

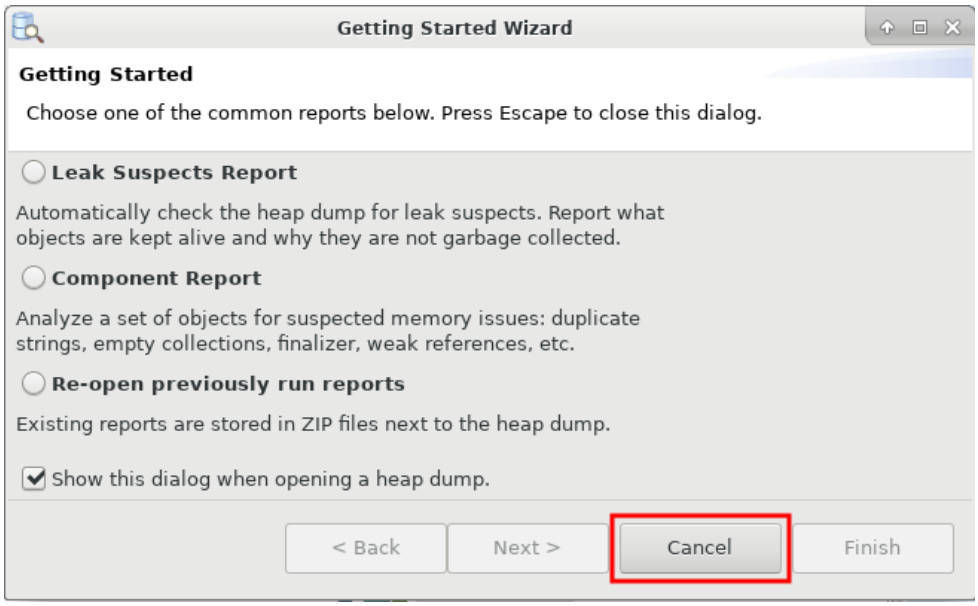
4. Click on the progress icon in the bottom right corner to get a detailed view of the progress:



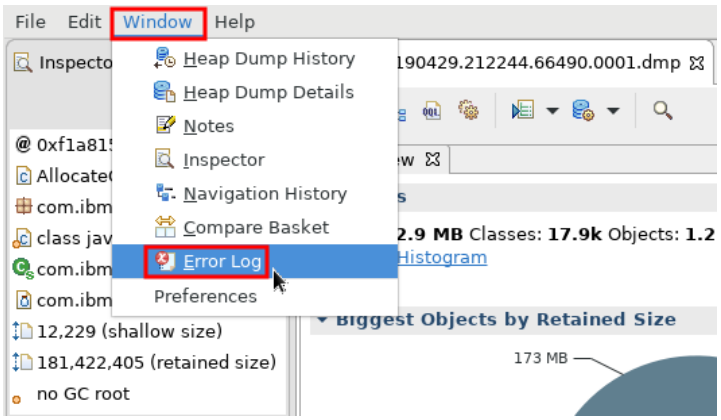
5. Now the **Progress** view is opened:



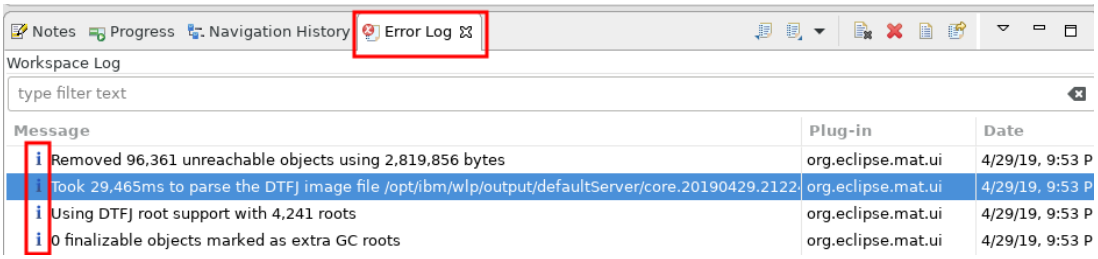
6. After the dump finishes loading, a pop-up will appear with suggested actions such as running the leak suspect report. For now, just click **Cancel**:



7. The first thing to check is to see whether there were any errors processing the dump. Click **Window** > **Error Log**:

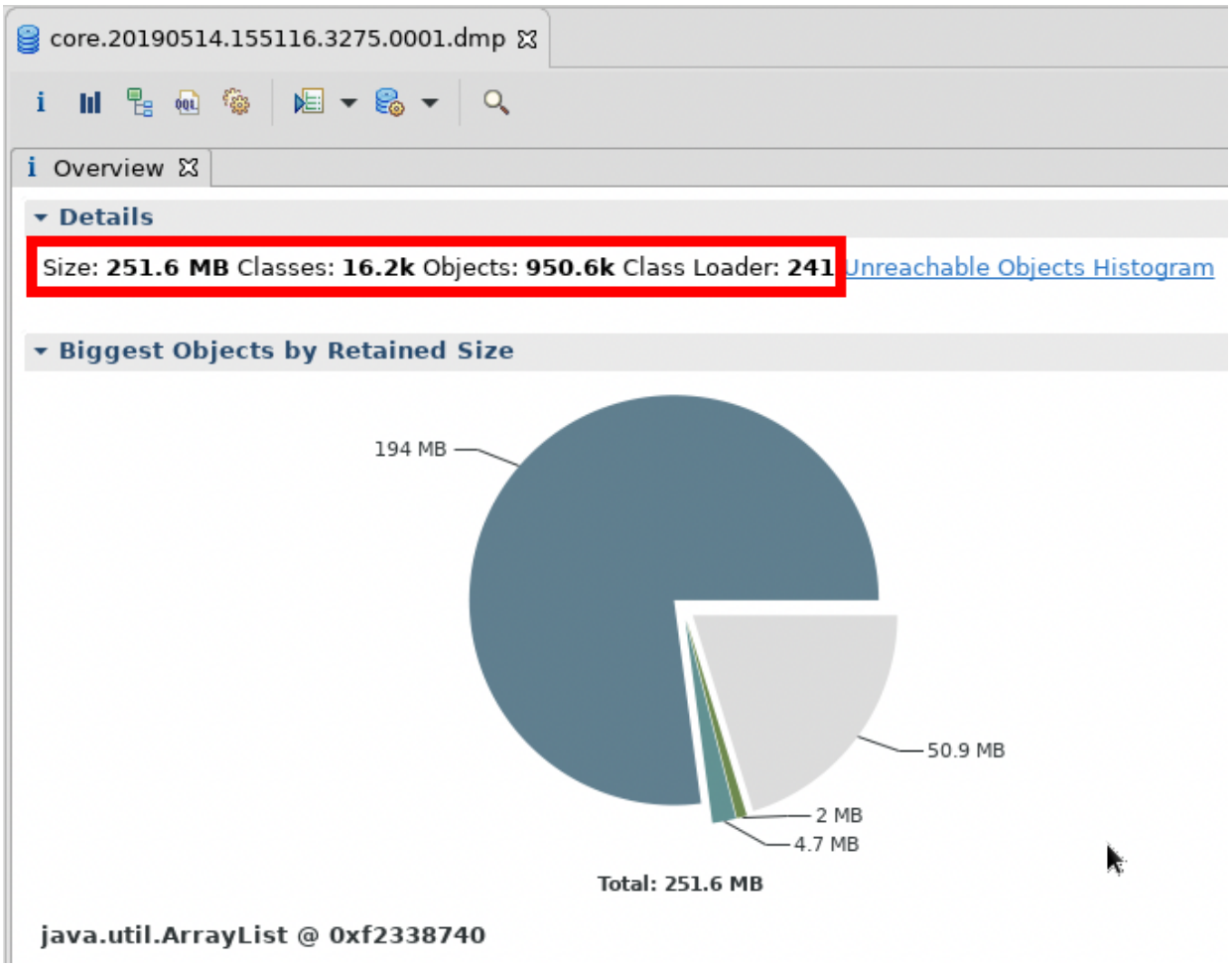


8. Review the list and check for any warnings or errors:



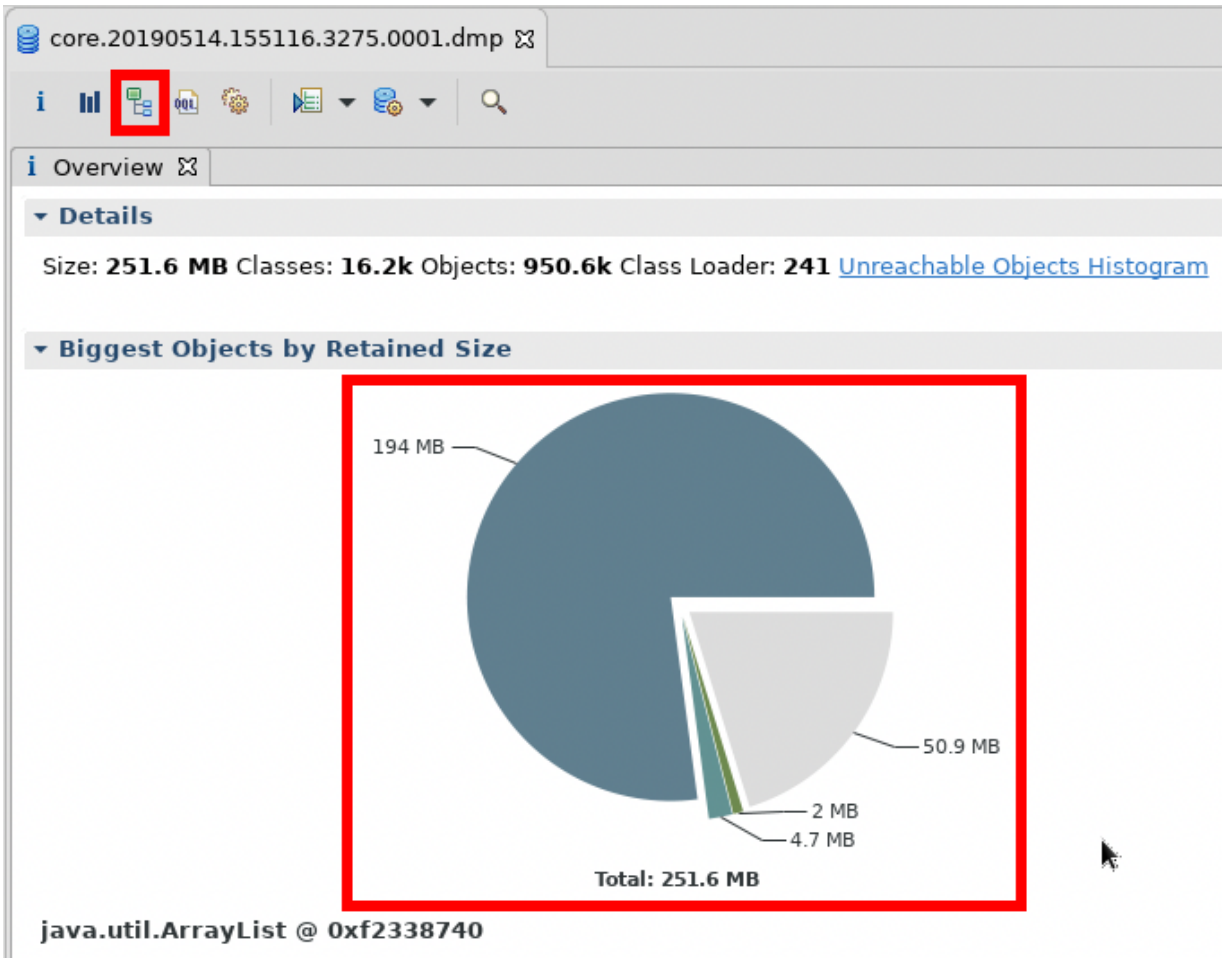
    1. It is possible to have a few warnings without too many problems. If you believe the warnings are limiting your analysis, consider opening an IBM Support case to investigate the issue with the IBM Java support team.

9. The overview tab shows the total live Java heap usage and the number of live classes, classloaders, and objects:

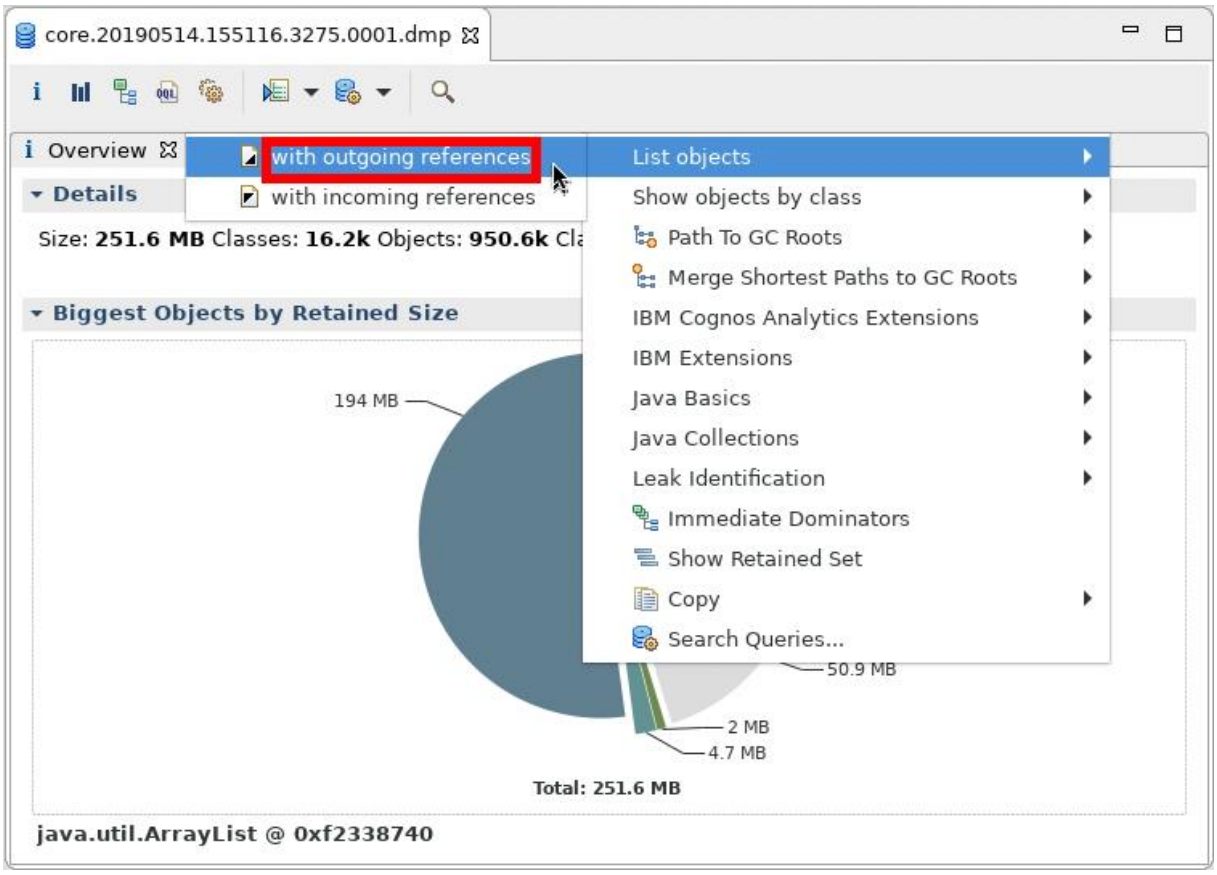


1. By default, MAT performs a full "garbage collection" when it loads the dump so everything you see is only pertaining to live Java objects. You can click on the **Unreachable Objects Histogram** link to see a histogram of any objects that are trash.

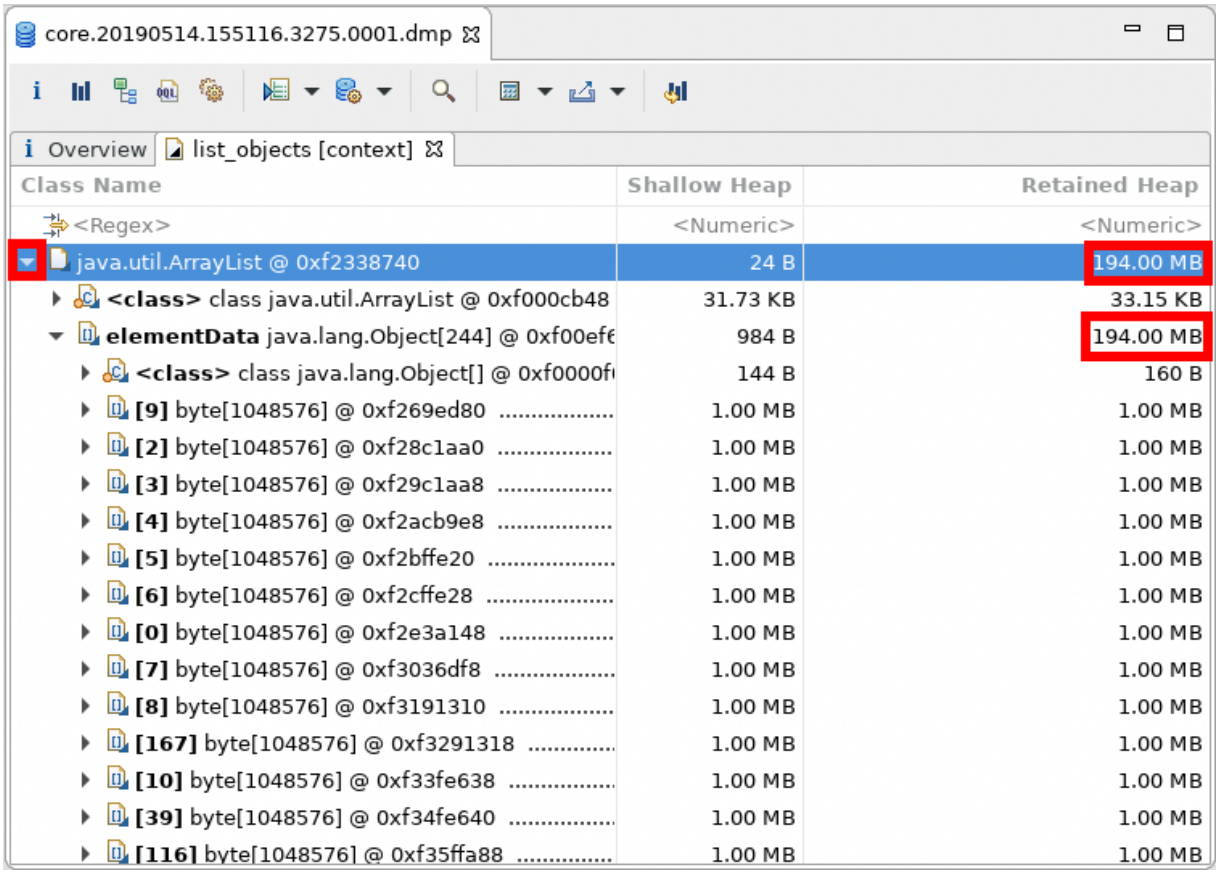
10. The pie chart on the **Overview** tab shows the largest dominator objects so it's a subset of the **Dominator Tree** button:



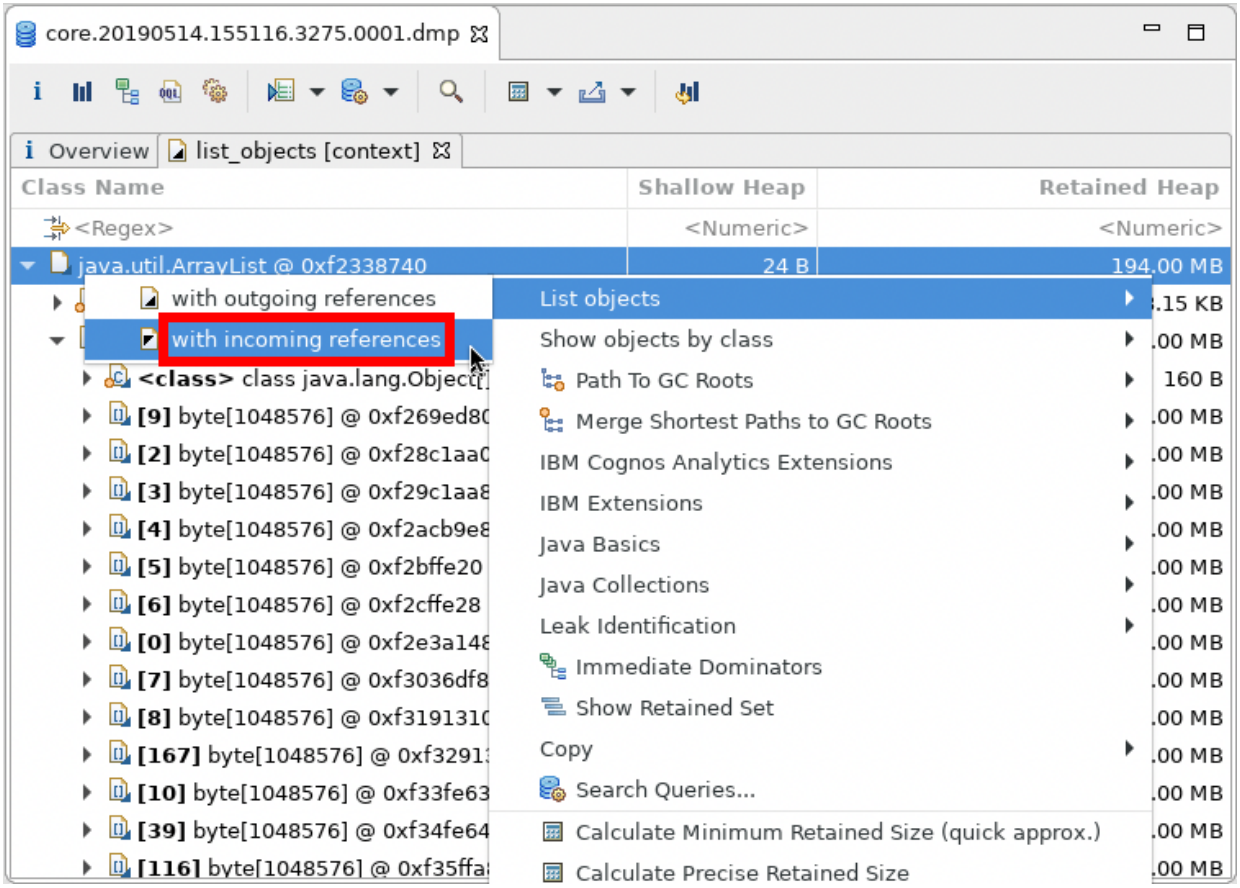
11. You may left click on a pie slice and select **List objects** > **with outgoing references** to review the object graph of the large dominator:



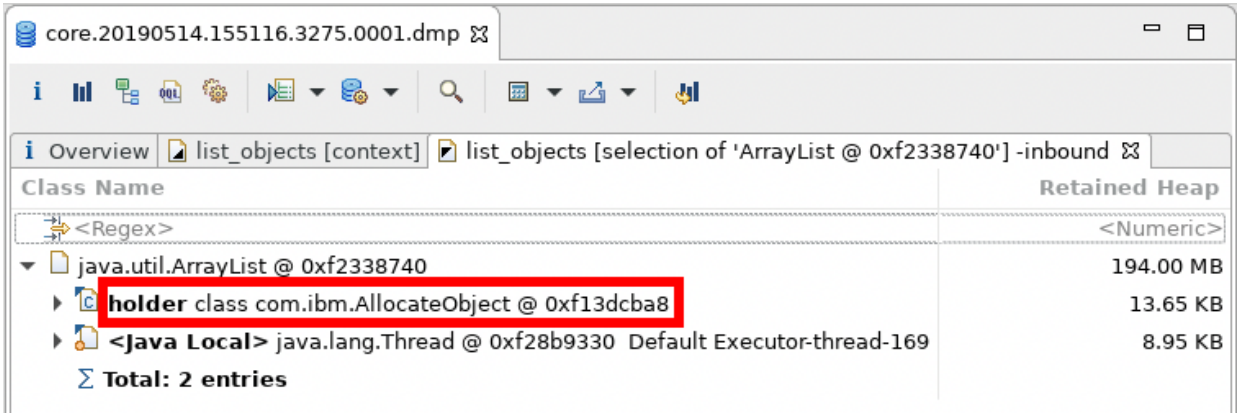
12. Expand the outgoing references tree and walk down the path with the largest **Retained Heap** values; in this example, there is an ArrayList that retains 194MB. Continue walking down the tree and you will find an Object array with hundreds of elements, each of about 1MB, which matches what we executed to create the OutOfMemoryError:



13. In this case, we want to find out what references this ArrayList, so right click on it and select **List objects** > **with incoming references**:



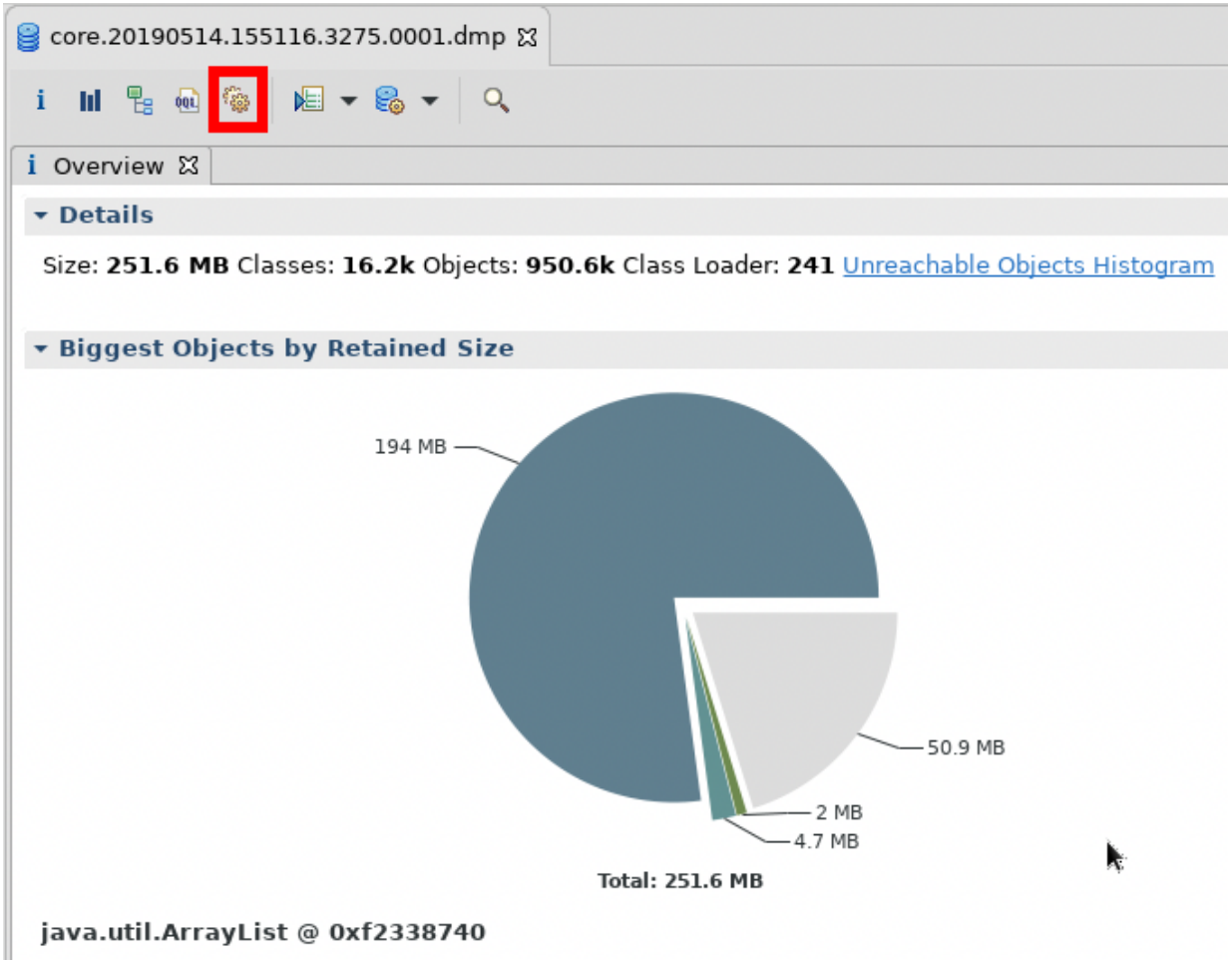
14. This results in the following view:



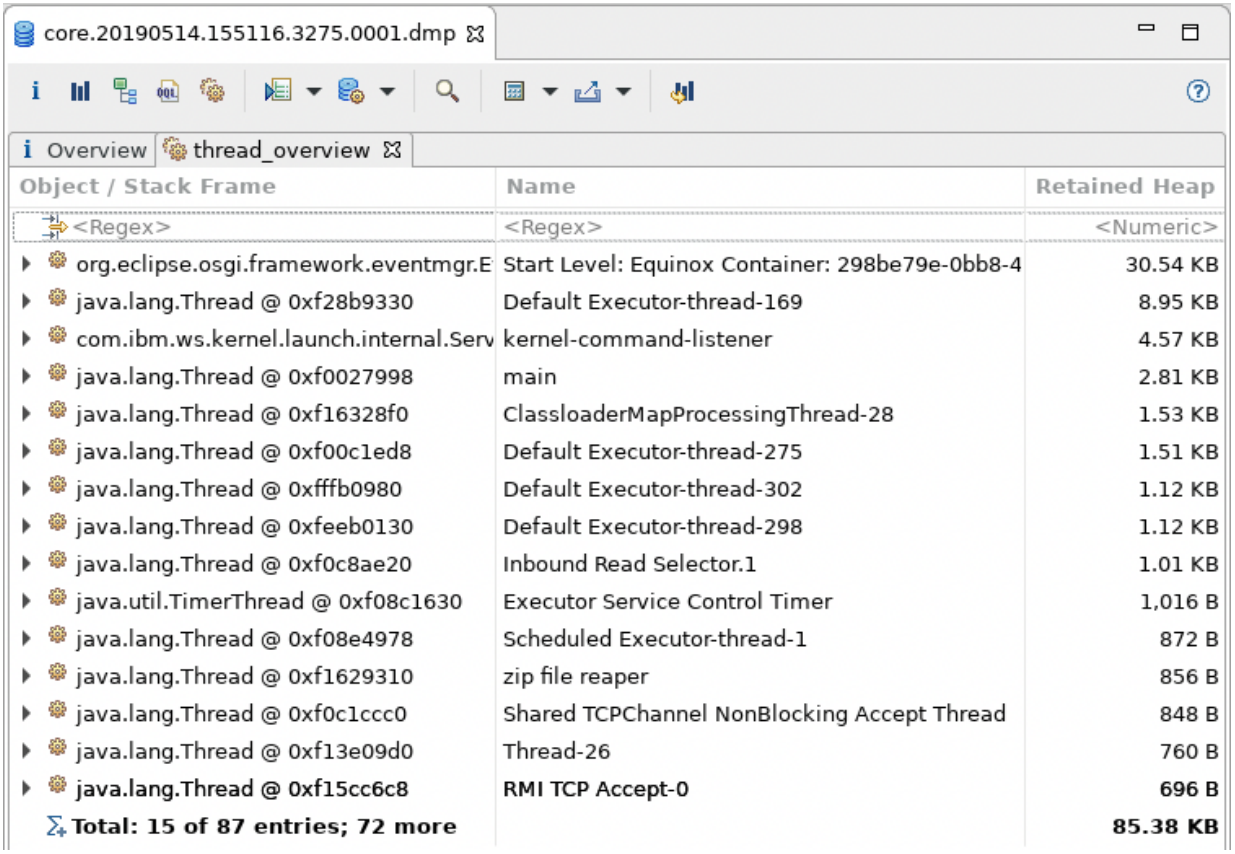
1. In this example, there are two references to the ArrayList. The first is that the class com.ibm.AllocateObject has a static field called holder which references the ArrayList. We know it is static because of the word **class** in front of the class name. The second is the thread **Default Executor-thread-169**.

15. From the above analysis, we know there is what appears to be a leak into a static ArrayList and

there is a thread that has a reference to it, so naturally we want to see what that thread is doing. Open the **Thread Overview** query:

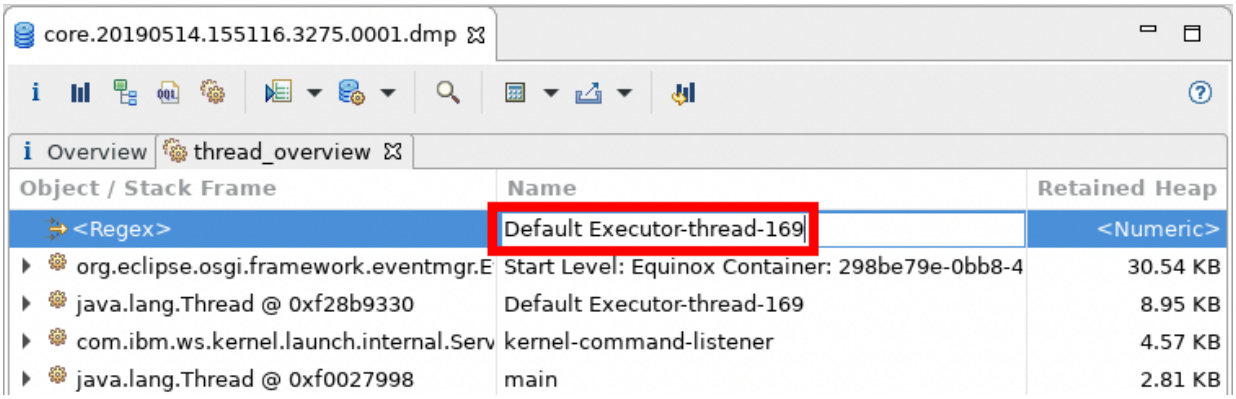


16. This will list every thread, the thread name, the retained heap of the thread, other thread details, and the stack frame along with stack frame locals:

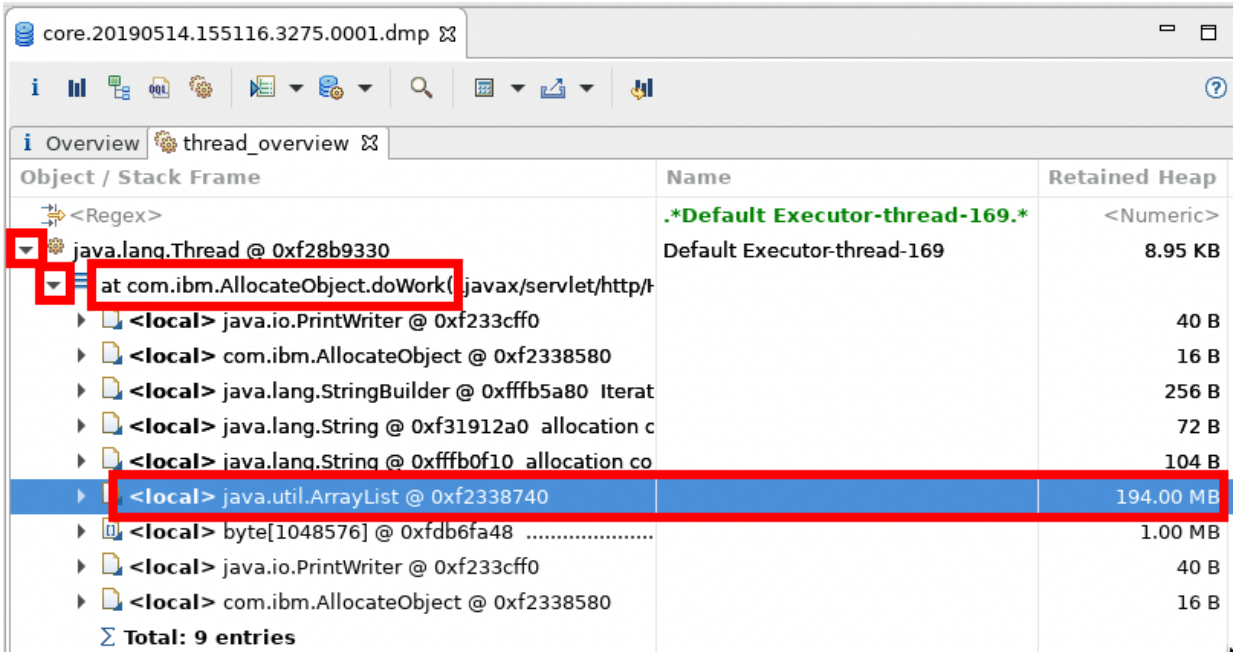


Object / Stack Frame	Name	Retained Heap
<Regex>	<Regex>	<Numeric>
org.eclipse.osgi.framework.eventmgr.E	Start Level: Equinox Container: 298be79e-0bb8-4	30.54 KB
java.lang.Thread @ 0xf28b9330	Default Executor-thread-169	8.95 KB
com.ibm.ws.kernel.launch.internal.Serv	kernel-command-listener	4.57 KB
java.lang.Thread @ 0xf0027998	main	2.81 KB
java.lang.Thread @ 0xf16328f0	ClassloaderMapProcessingThread-28	1.53 KB
java.lang.Thread @ 0xf00c1ed8	Default Executor-thread-275	1.51 KB
java.lang.Thread @ 0xfffb0980	Default Executor-thread-302	1.12 KB
java.lang.Thread @ 0xfeeb0130	Default Executor-thread-298	1.12 KB
java.lang.Thread @ 0xf0c8ae20	Inbound Read Selector.1	1.01 KB
java.util.TimerThread @ 0xf08c1630	Executor Service Control Timer	1,016 B
java.lang.Thread @ 0xf08e4978	Scheduled Executor-thread-1	872 B
java.lang.Thread @ 0xf1629310	zip file reaper	856 B
java.lang.Thread @ 0xf0c1ccc0	Shared TCPChannel NonBlocking Accept Thread	848 B
java.lang.Thread @ 0xf13e09d0	Thread-26	760 B
java.lang.Thread @ 0xf15cc6c8	**RMI TCP Accept-0**	**696 B**
Total: 15 of 87 entries; 72 more		**85.38 KB**

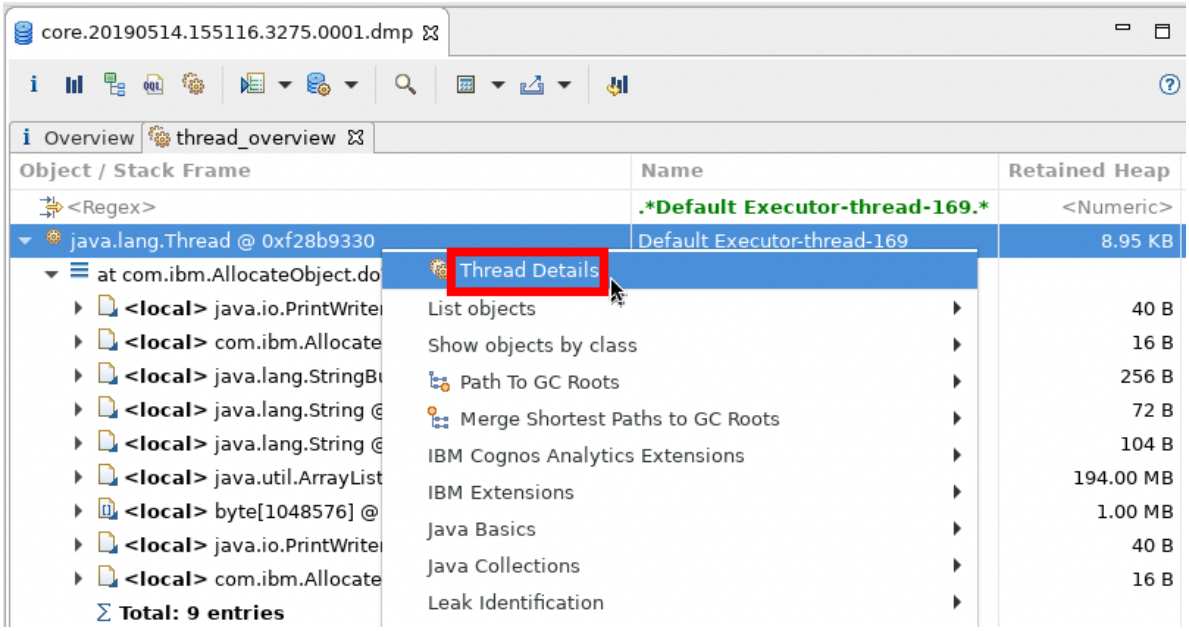
17. We know from above that the thread that references the ArrayList is named **Default Executor-thread-169**. In your case, the thread may be named differently. You may enter this thread name into the **Name** column's **<Regex>** input:



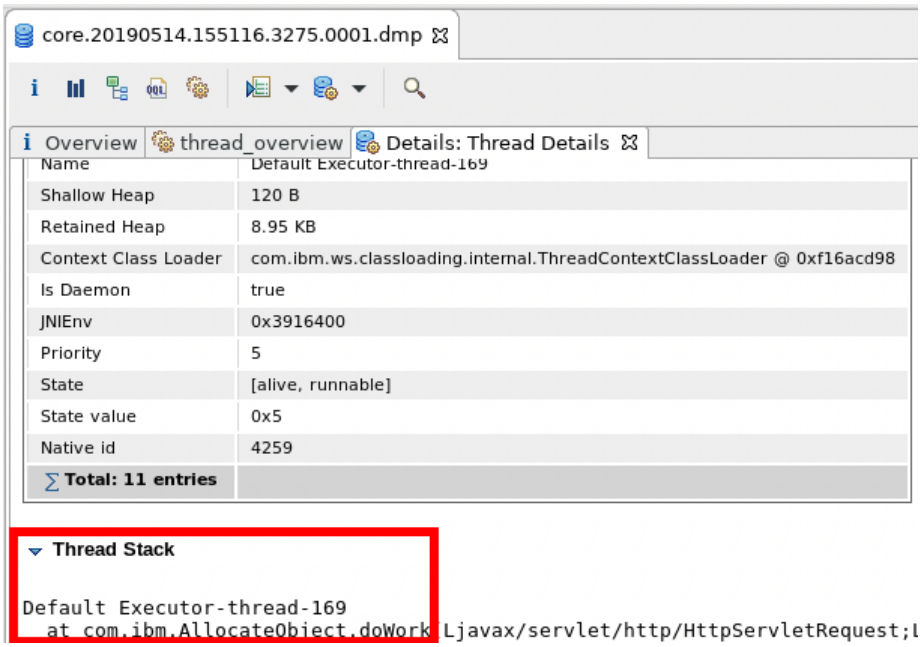
18. Press Enter to filter the results, expand the thread stack and find the servlet that caused the leak:



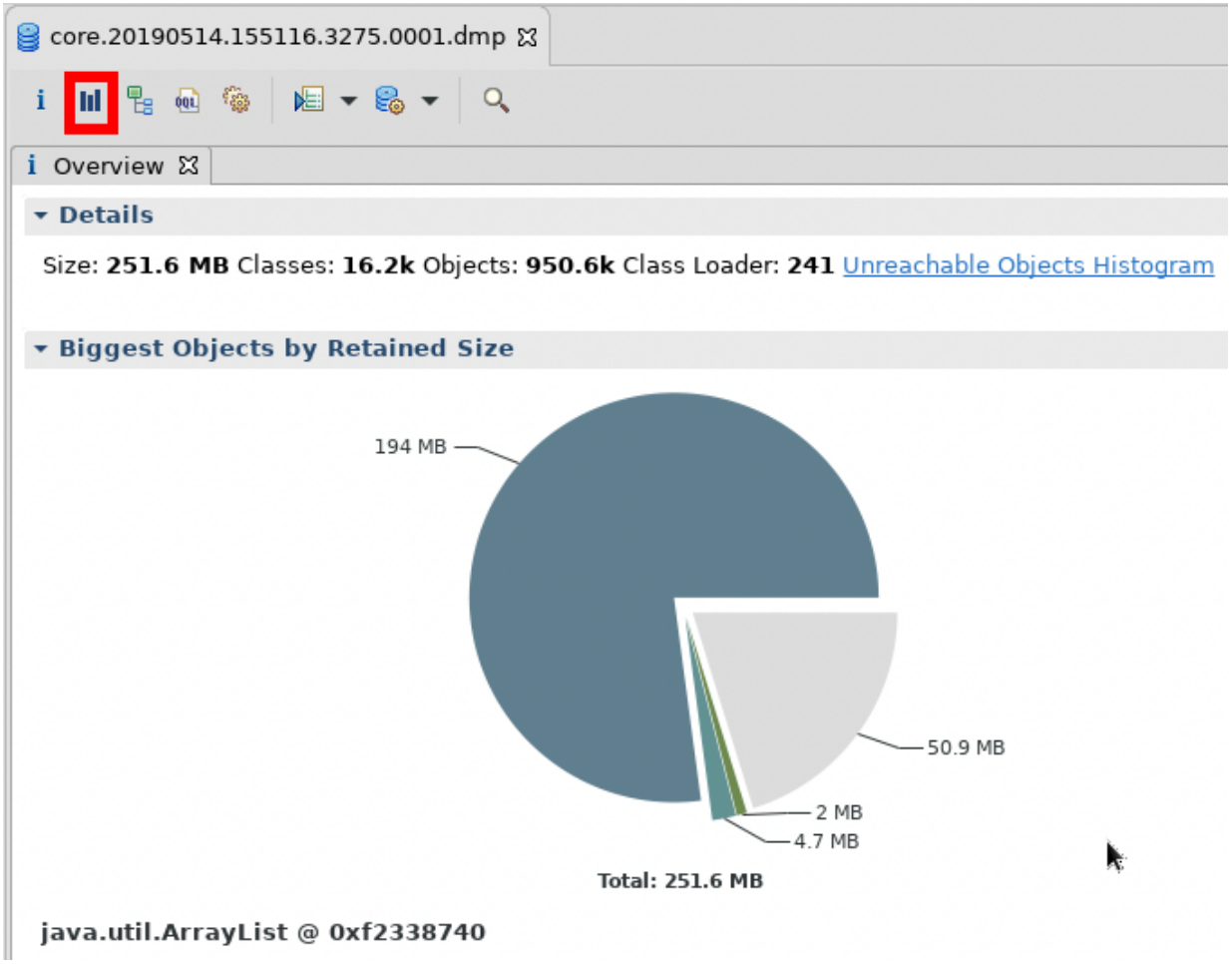
1. Note that you can see the actual objects on each stack frame. In this case, we can clearly see the servlet has a reference to the AllocateObject class and the ArrayList which is retaining most of the heap. This stack usually makes it much easier for the application developer to understand what happened. Right click on the thread and select **Thread Details** to get a full thread stack that may be copy-and-pasted:



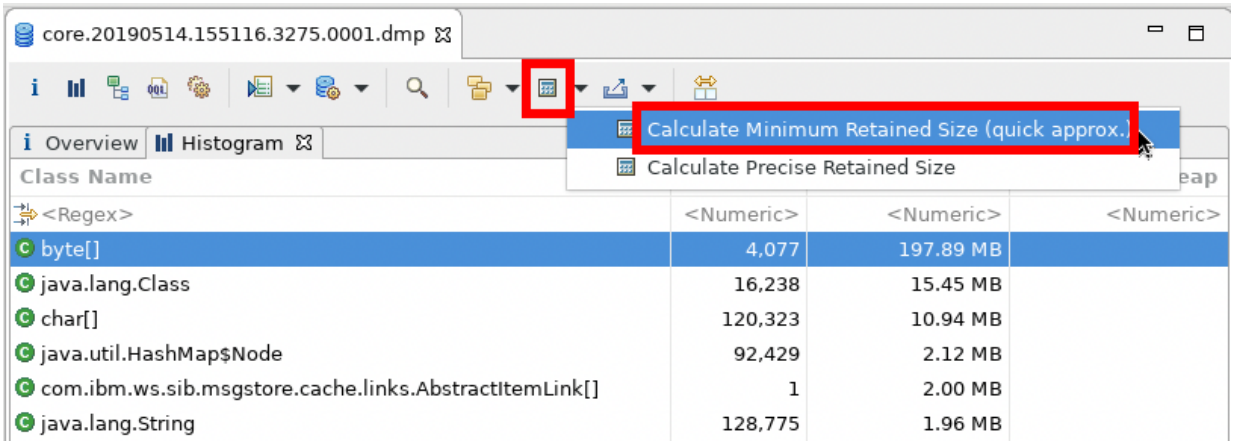
2. Scroll down to see the full stack:



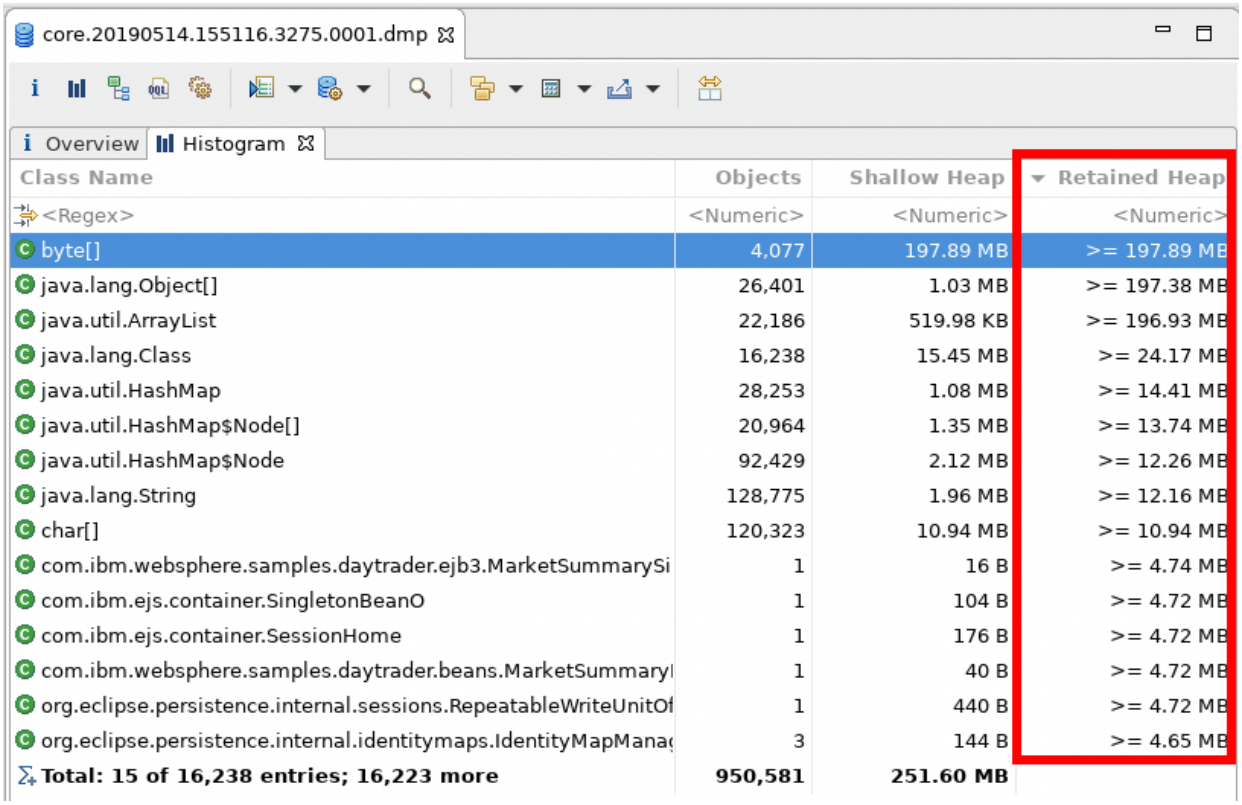
19. Another common view to explore is the **Histogram**:



20. Click on the calculator button and select **Calculate Minimum Retained Size (quick approx.)** to populate the **Retained Heap** column for each class:

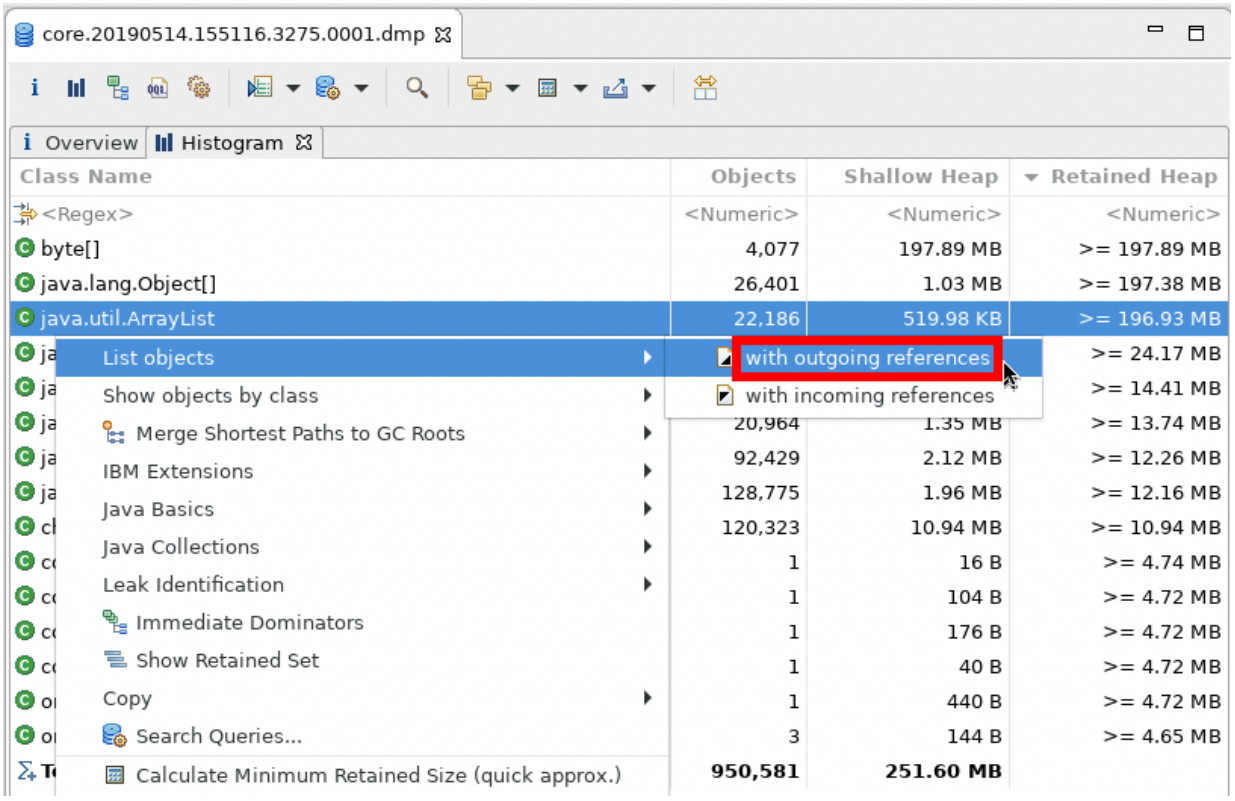


21. This fills in the retained heap column which then you can click to sort descending:

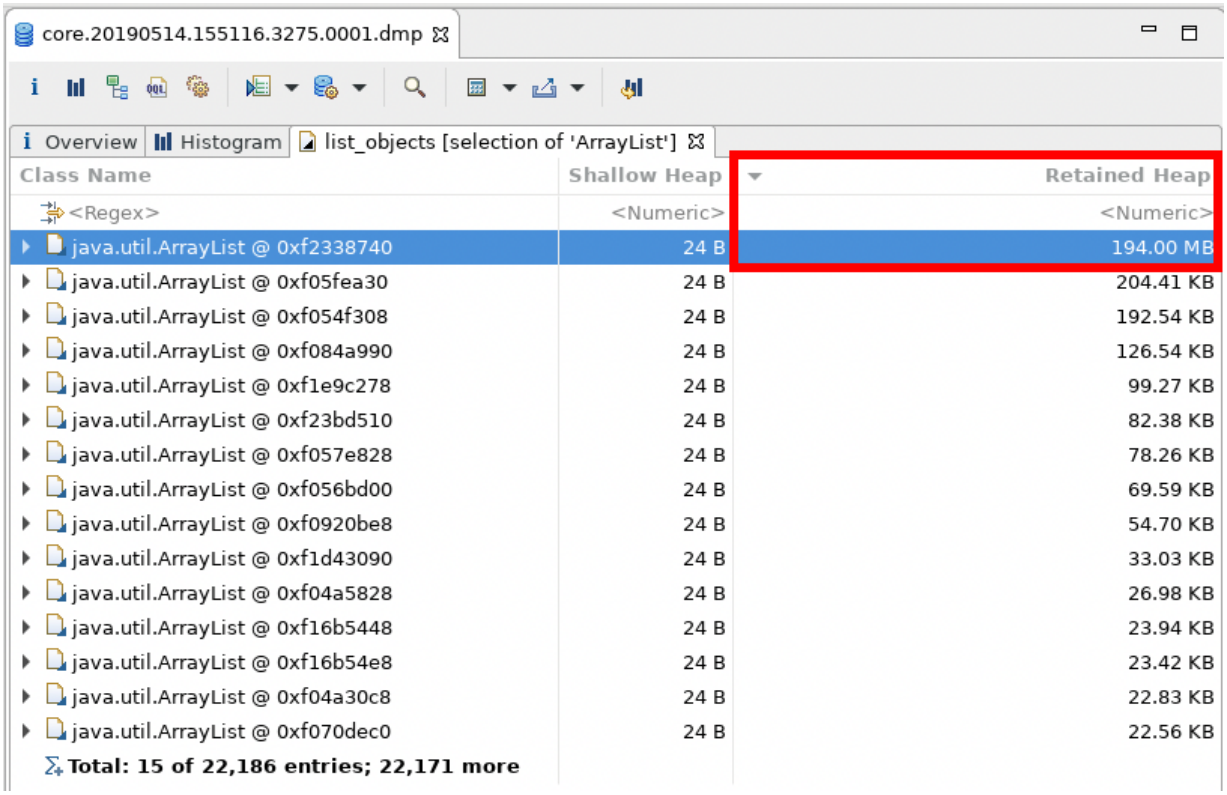


Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	4,077	197.89 MB	>= 197.89 MB
java.lang.Object[]	26,401	1.03 MB	>= 197.38 MB
java.util.ArrayList	22,186	519.98 KB	>= 196.93 MB
java.lang.Class	16,238	15.45 MB	>= 24.17 MB
java.util.HashMap	28,253	1.08 MB	>= 14.41 MB
java.util.HashMap$Node[]	20,964	1.35 MB	>= 13.74 MB
java.util.HashMap$Node	92,429	2.12 MB	>= 12.26 MB
java.lang.String	128,775	1.96 MB	>= 12.16 MB
char[]	120,323	10.94 MB	>= 10.94 MB
com.ibm.websphere.samples.daytrader.ejb3.MarketSummarySi	1	16 B	>= 4.74 MB
com.ibm.ejs.container.SingletonBeanO	1	104 B	>= 4.72 MB
com.ibm.ejs.container.SessionHome	1	176 B	>= 4.72 MB
com.ibm.websphere.samples.daytrader.beans.MarketSummaryI	1	40 B	>= 4.72 MB
org.eclipse.persistence.internal.sessions.RepeatableWriteUnitOf	1	440 B	>= 4.72 MB
org.eclipse.persistence.internal.identitymaps.IdentityMapManag	3	144 B	>= 4.65 MB
Total: 15 of 16,238 entries; 16,223 more	**950,581**	**251.60 MB**	

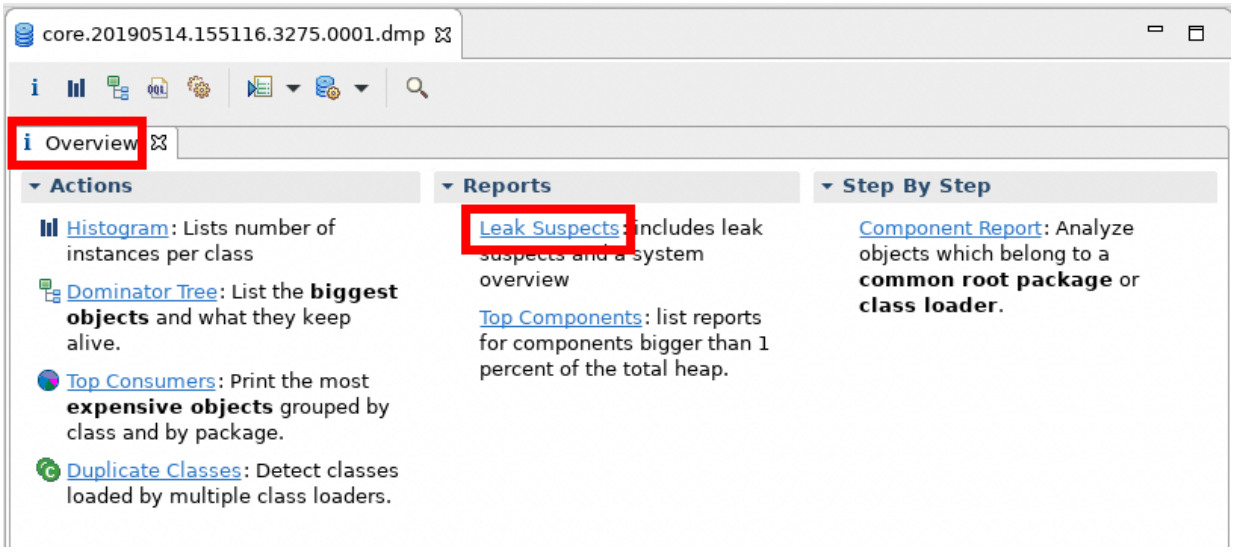
22. You may click on a row with a large retained heap size, right click and select outgoing references. For example:



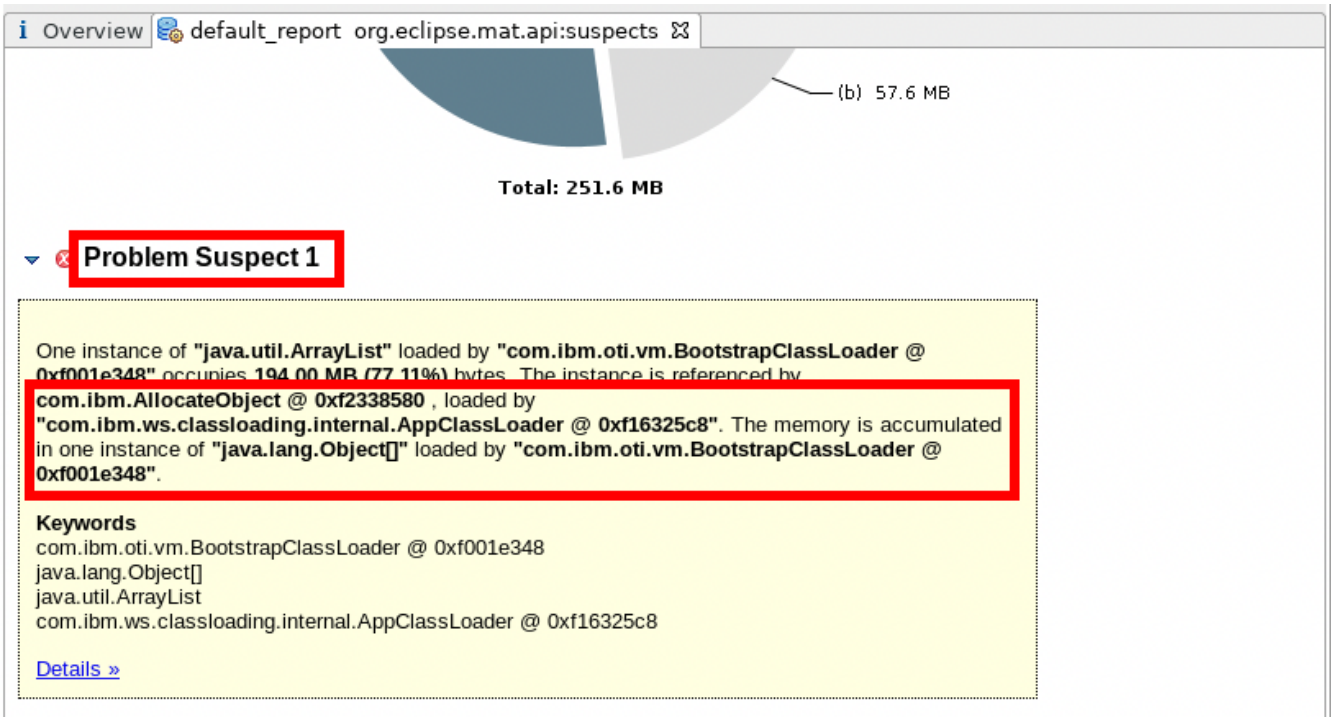
23. Then sort by **Retained Heap** and again you will find the large object:



24. The next common view to explore is the **Leak Suspects** view. On the **Overview** tab, scroll down and click on **Leak Suspects**:

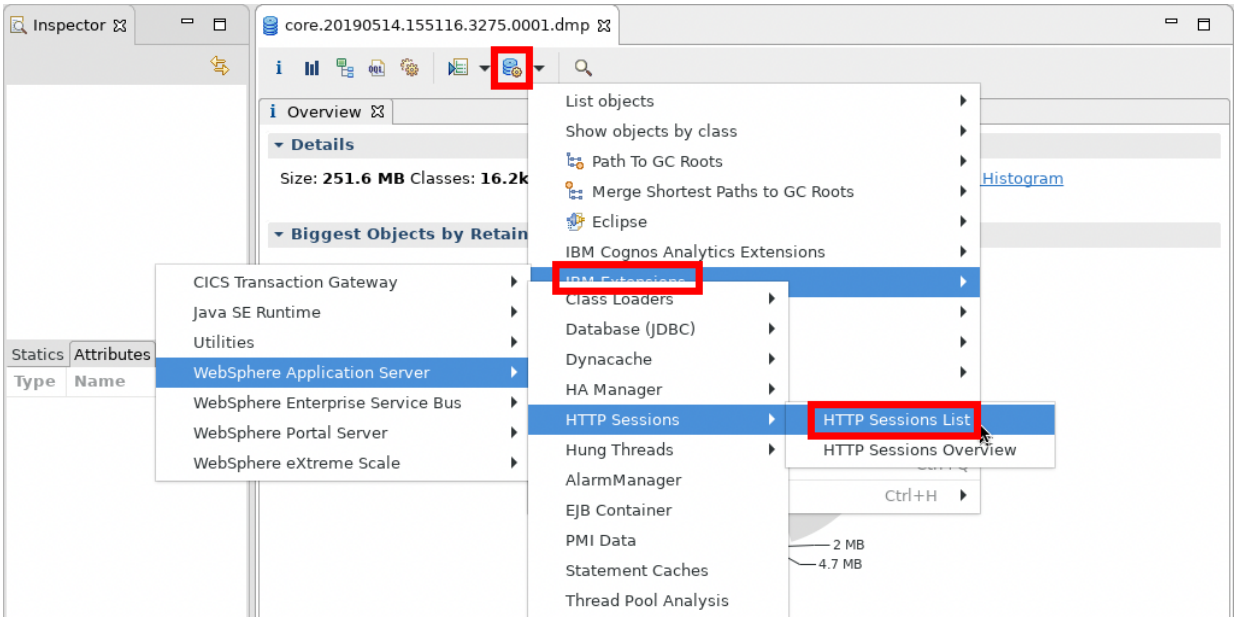


25. The report will list leak suspects in the order of their size. The following example shows the same leaking Object[] inside the ArrayList:



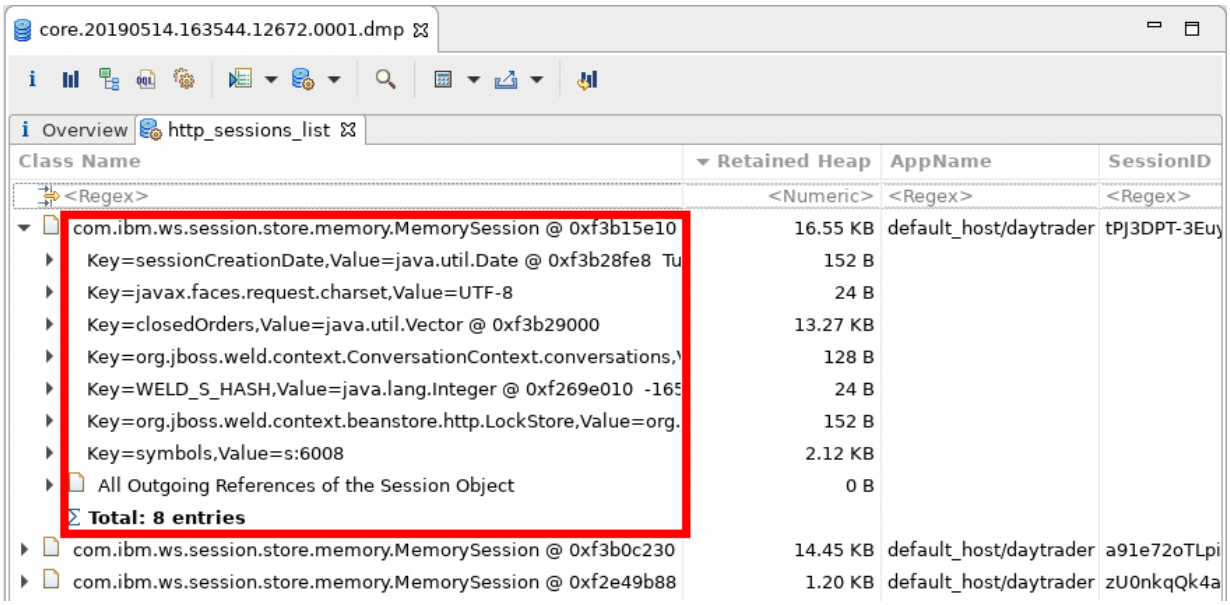
The IBM Extensions for Memory Analyzer (IEMA)[29] provide additional extensions on top of MAT with WAS, Java, and other related queries.

1. As one example, you can see a list of all HTTP sessions and their attributes with: **Open Query Browser > IBM Extensions > WebSphere Application Server > HTTP Sessions > HTTP Sessions List**:



[29] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Memory_Analyzer_Tool.html#Major_Tools-IBM_Memory_Analyzer_Tool_MAT-Installation

2. Each HTTP session is listed, as well as how much Java heap it retains, which application it's associated with, and other details, including all of the attribute names and values:



3. You may explore the other extensions under IBM Extensions. Some only apply to Traditional WAS, some only to WAS Liberty, and some to both. Unlike MAT, IEMA is not officially supported but we try to fix and enhance it as time permits.

13 Health Center

IBM Monitoring and Diagnostics for Java - Health Center[30] is free and shipped with IBM Java. Among other things, Health Center includes a statistical CPU profiler that samples Java stacks that are using CPU at a very high rate to determine what Java methods are using CPU. Health Center generally has an overhead of less than 1% and is suitable for production use. In recent versions, it may also be enabled dynamically without restarting the JVM.

This lab will demonstrate how to enable Java Health Center, exercise the sample DayTrader application using Apache JMeter, and review the Health Center file in the IBM Java Health Center Client Tool.

13.1 Health Center Theory

The Health Center agent gathers sampled CPU profiling data, along with other information:

- Classes: Information about classes being loaded
- Environment: Details of the configuration and system of the monitored application
- Garbage collection: Information about the Java heap and pause times
- I/O: Information about I/O activities that take place.
- Locking: Information about contention on inflated locks
- Memory: Information about the native memory usage

[30] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Java_Health_Center.html

- Profiling: Provides a sampling profile of Java methods including call paths

The Health Center agent can be enabled in two ways:

1. At startup by adding -Xhealthcenter:level=headless to the JVM arguments
2. At runtime, by running ${IBM_JAVA}/bin/java -jar ${IBM_JAVA}/jre/lib/ext/healthcenter.jar ID=${PID} level=headless

Note: For both items, you may add the following arguments to limit and roll the total file usage of Health Center data:

-Dcom.ibm.java.diagnostics.healthcenter.headless.files.max.size=BYTES

-Dcom.ibm.java.diagnostics.healthcenter.headless.files.to.keep=N (N=0 for unlimited)

The key to produce the final Health Center HCD file is that the JVM should be gracefully stopped (there are alternatives to this by packaging the temporary files but this isn't generally recommended).

Consider always enabling HealthCenter in headless mode[31] for post-mortem debugging of issues.

13.2 Health Center Lab

1. Stop the Apache JMeter test.
2. Stop the Liberty server.

```
/opt/ibm/wlp/bin/server stop defaultServer
```

3. Add the following line to **/opt/ibm/wlp/usr/servers/defaultServer/jvm.options**:

```
-Xhealthcenter:level=headless
```

4. Start the Liberty server

```
/opt/ibm/wlp/bin/server start defaultServer
```

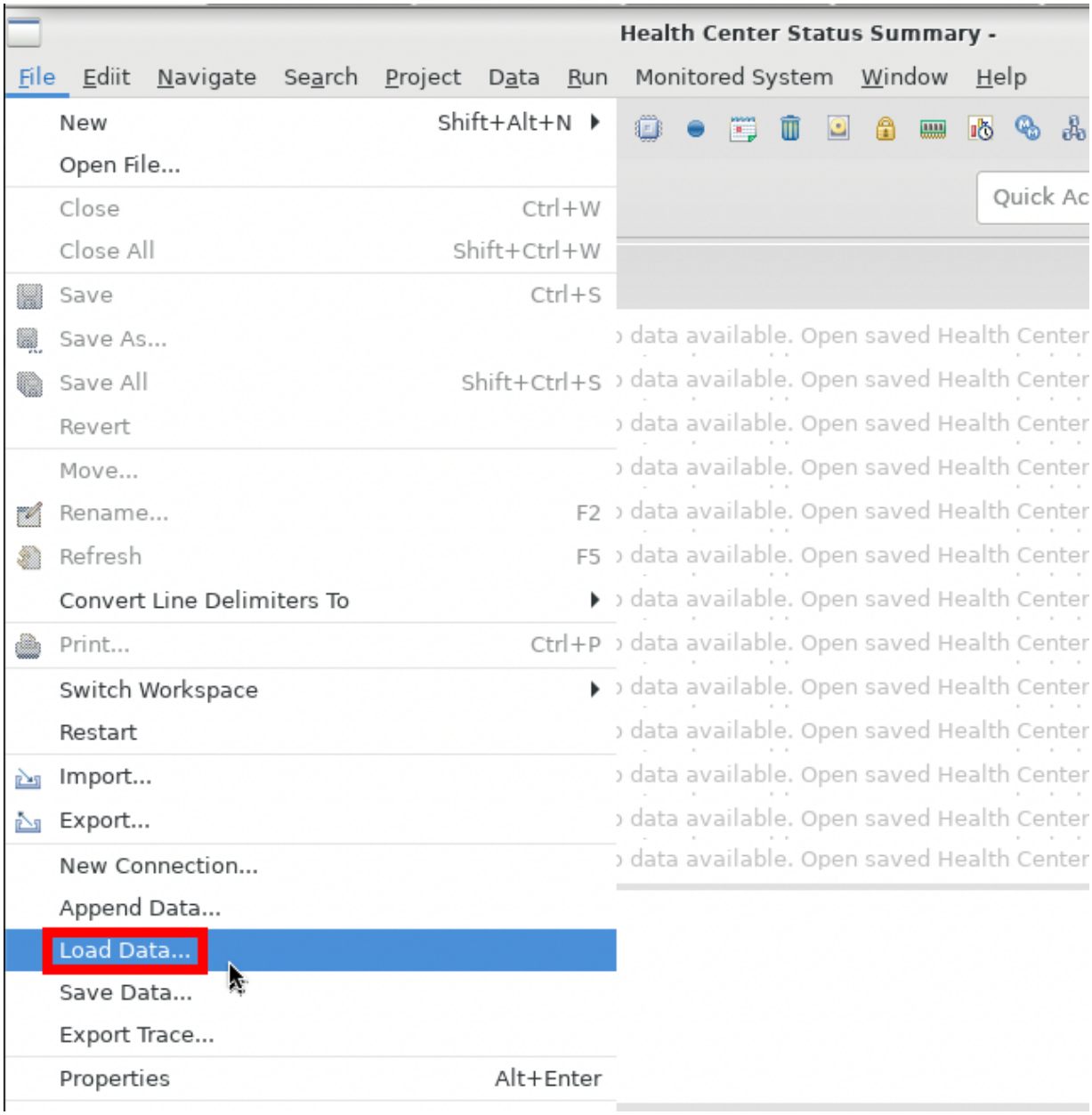
5. Start the Apache JMeter test and run it for 5 minutes.
6. Stop the JMeter test.
7. Stop the Liberty server

```
/opt/ibm/wlp/bin/server stop defaultServer
```

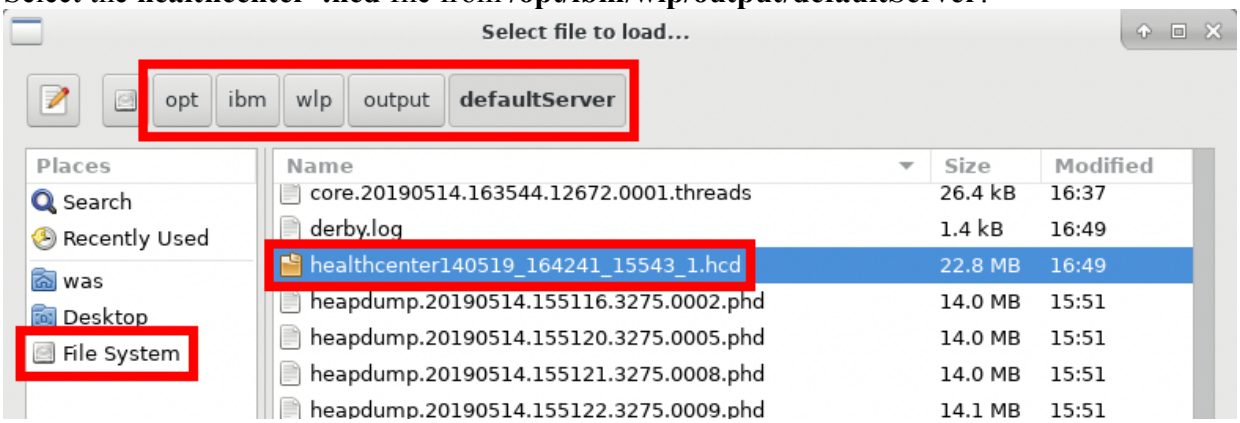
8. Open **/opt/programs/** in the file browser and double click on **Health Center**.

[31] https://publib.boulder.ibm.com/httpserv/cookbook/Major_Tools-IBM_Java_Health_Center.html#Major_Tools-IBM_Java_Health_Center-Gathering_Data

9. Click **File > Load Data...** (note that it's towards the bottom of the **File** menu; **Open File** does not work):

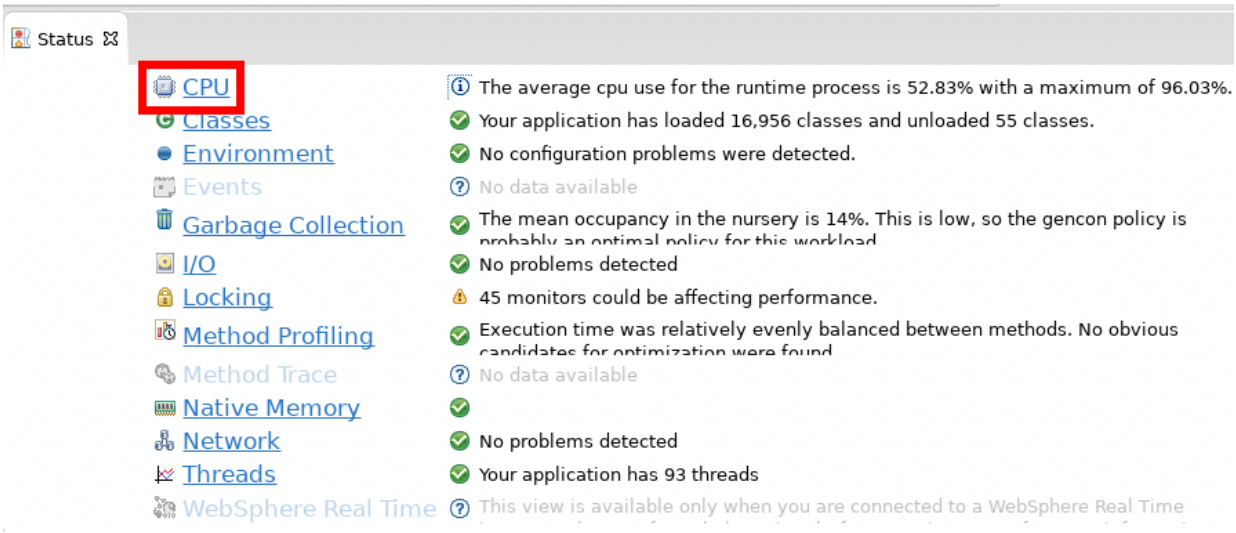


10. Select the **healthcenter*.hcd** file from **/opt/ibm/wlp/output/defaultServer**:

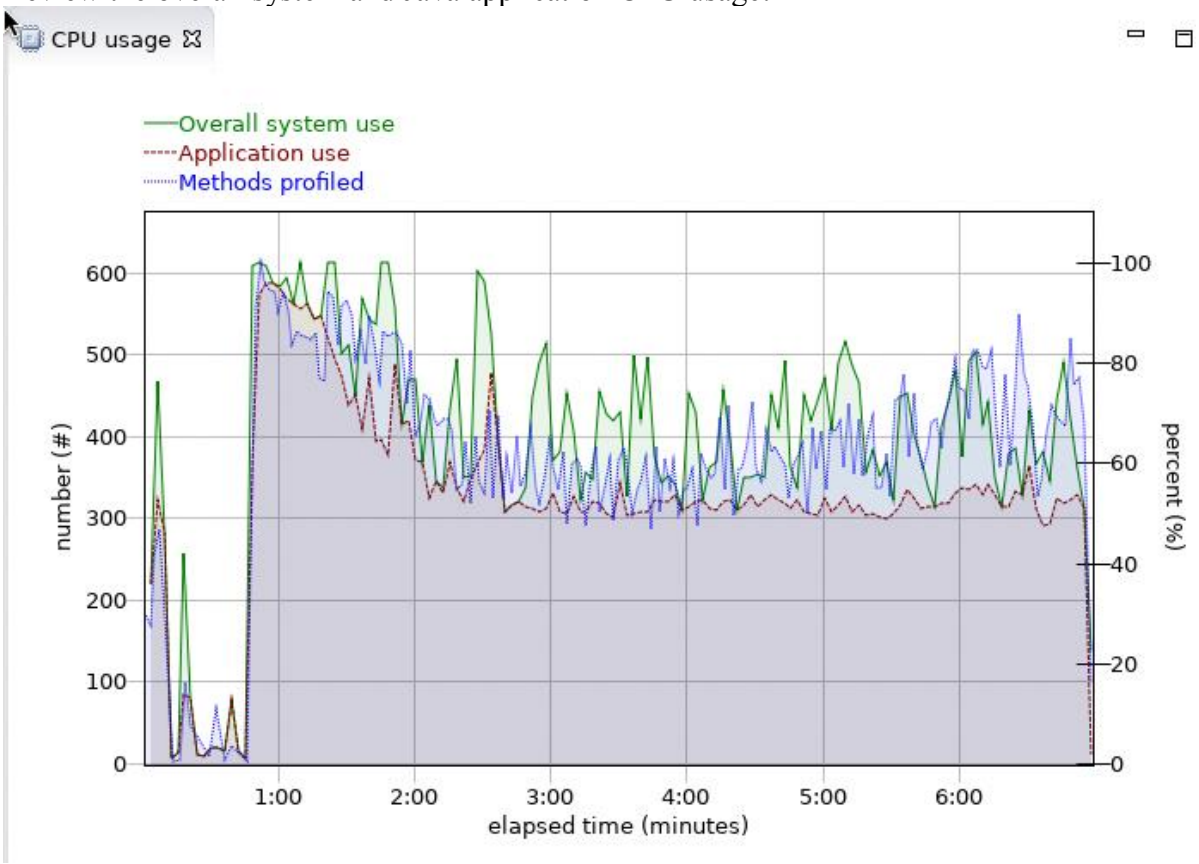


11. Wait for the data to complete loading:

12. Click on CPU:



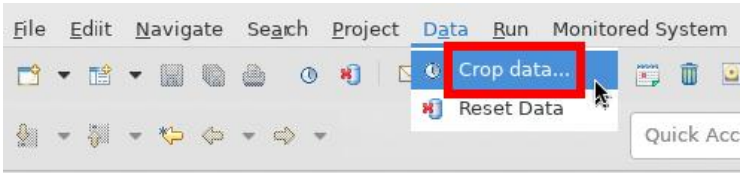
13. Review the overall system and Java application CPU usage:



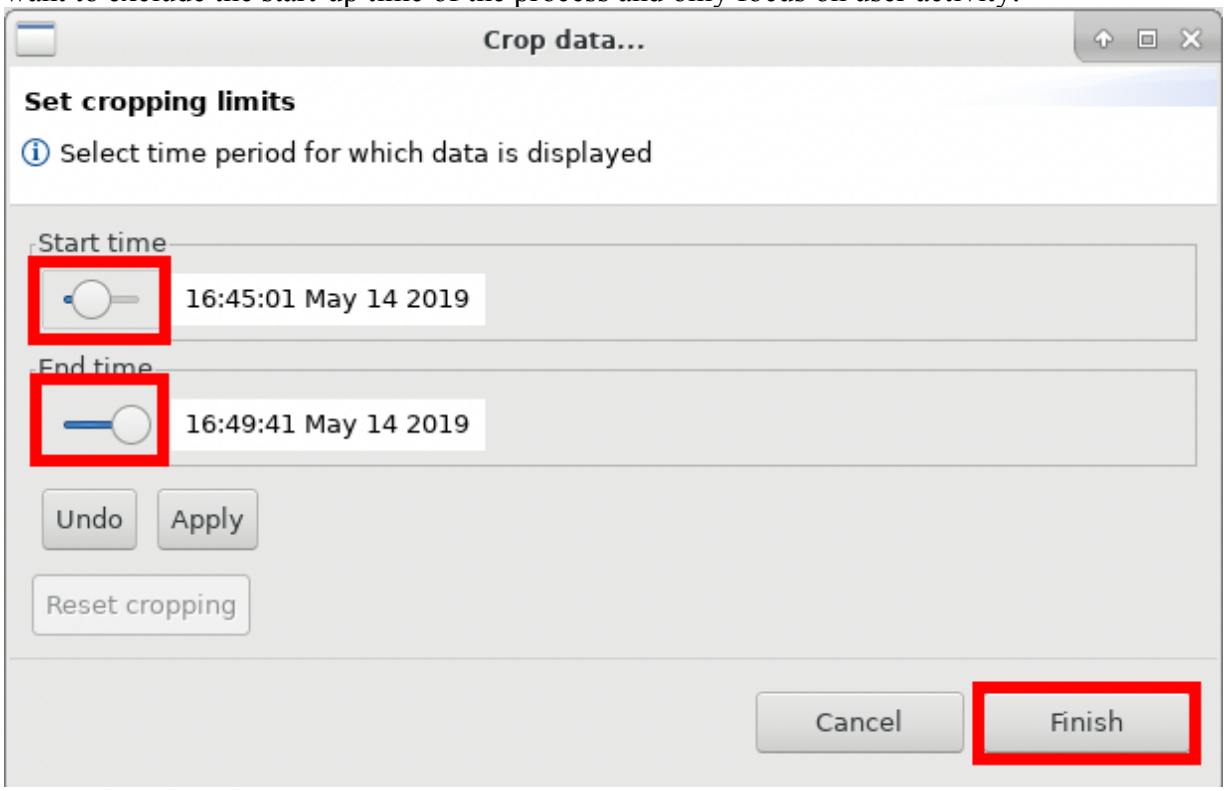
14. Right click anywhere in the graph and change the **X-axis** to **date** (which changes all other views to **date** as well):

1. For large Health Center captures, this may take significant time to change and there is no obvious indication when it's complete. The best way to know is when the CPU usage of Health Center drops to a low amount.

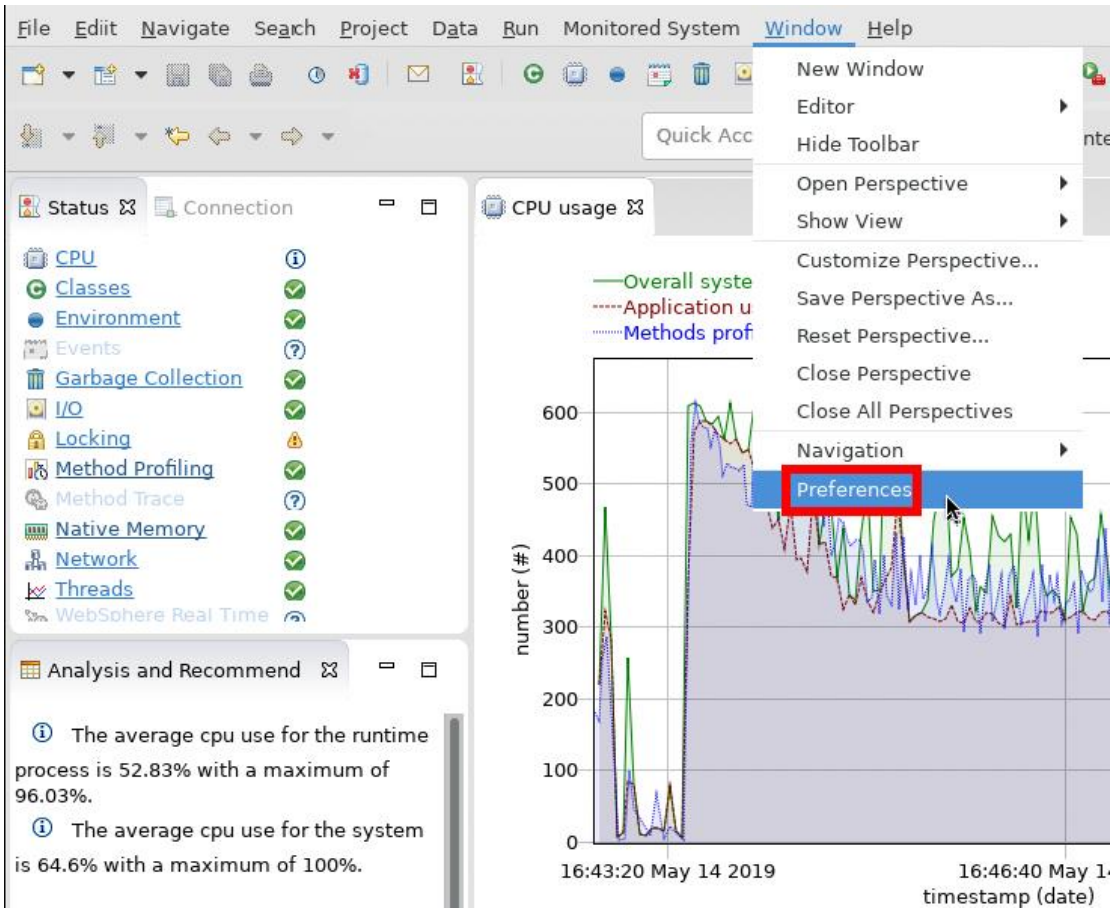
15. Click **Data > Crop data...**



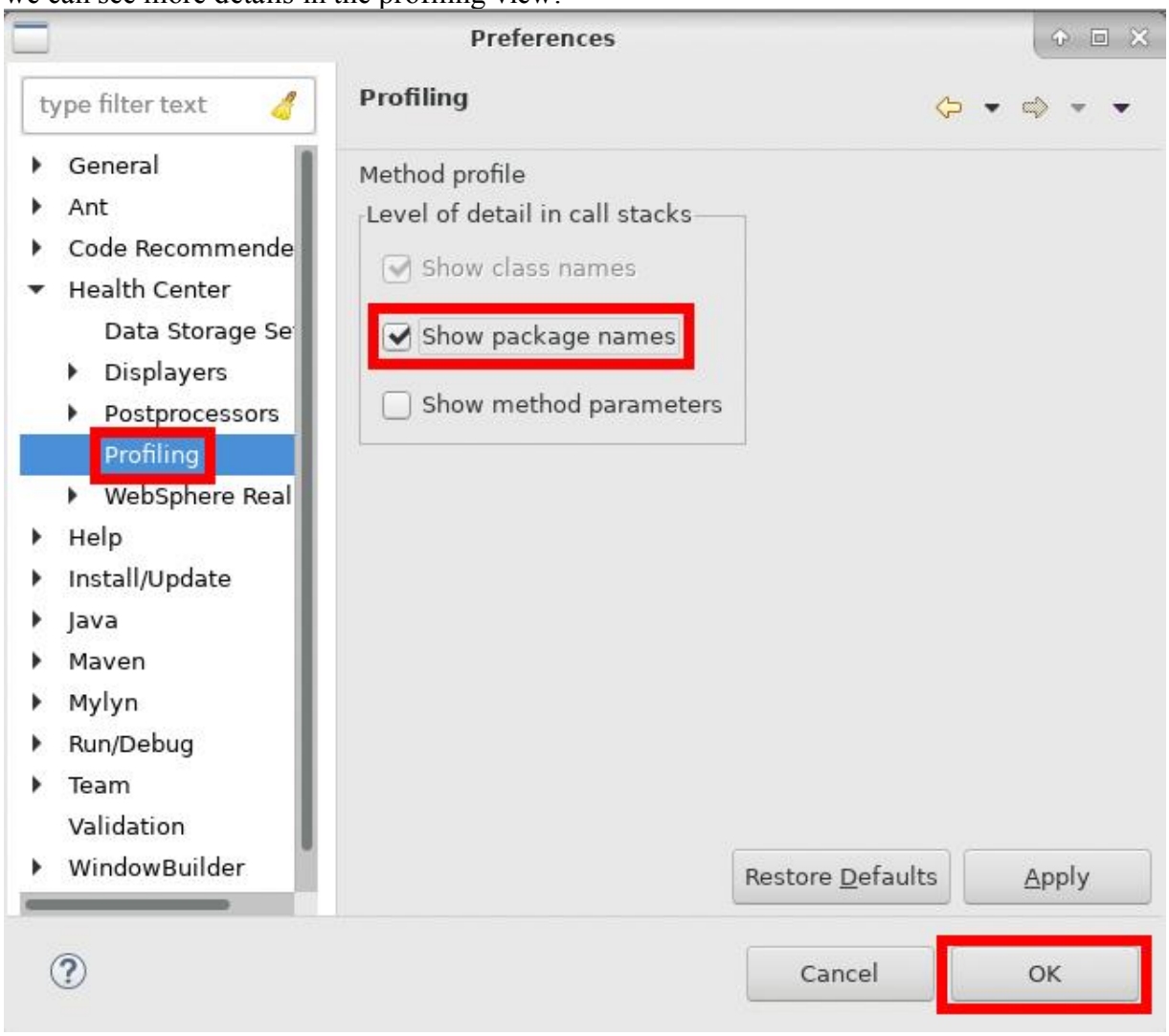
16. Change the **Start time** and **End time** to match the period of interest. For example, usually you want to exclude the start-up time of the process and only focus on user activity:



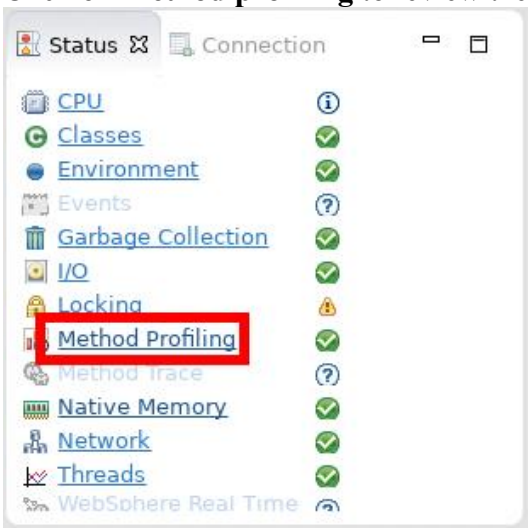
17. Click **Window > Preferences**:



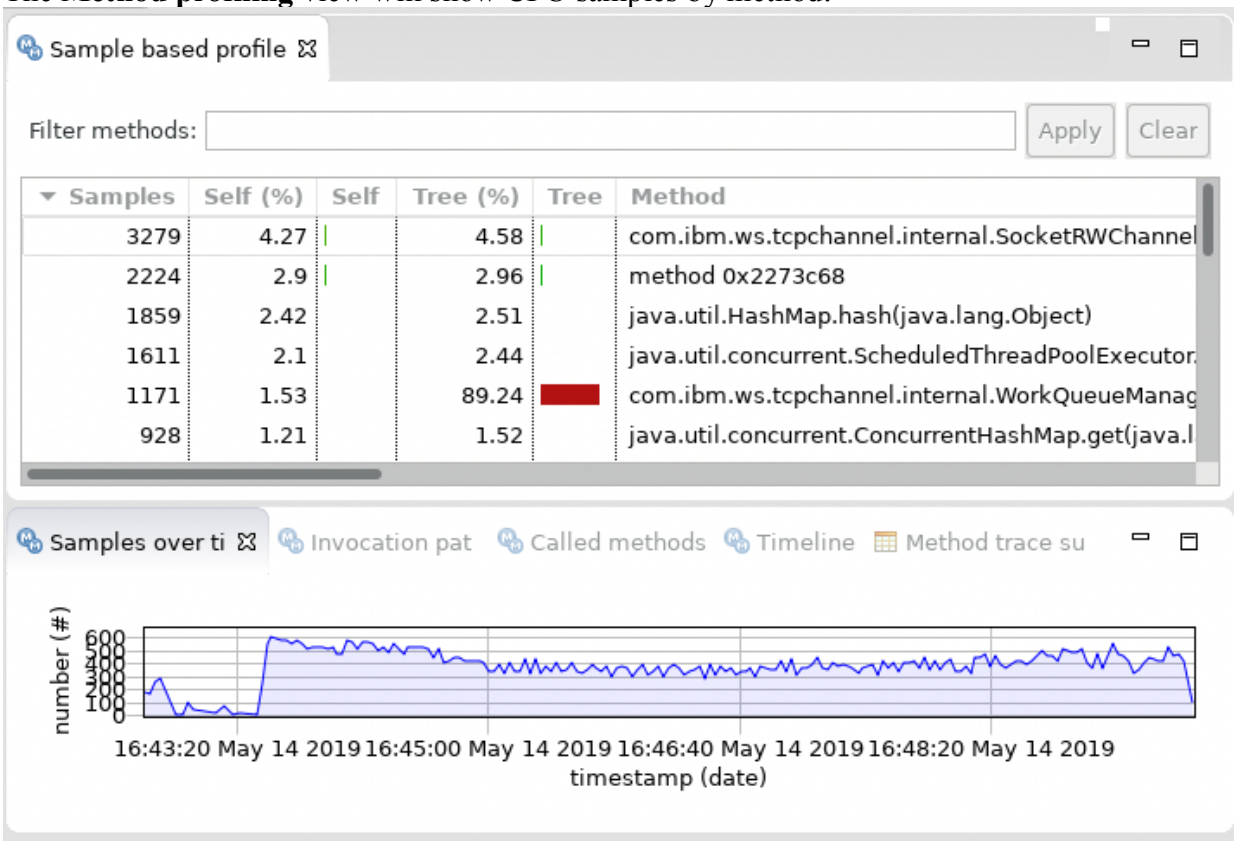
18. Check the **Show package names** box under **Health Center > Profiling** and press **OK** so that we can see more details in the profiling view:



19. Click on **Method profiling** to review the CPU sampling data:

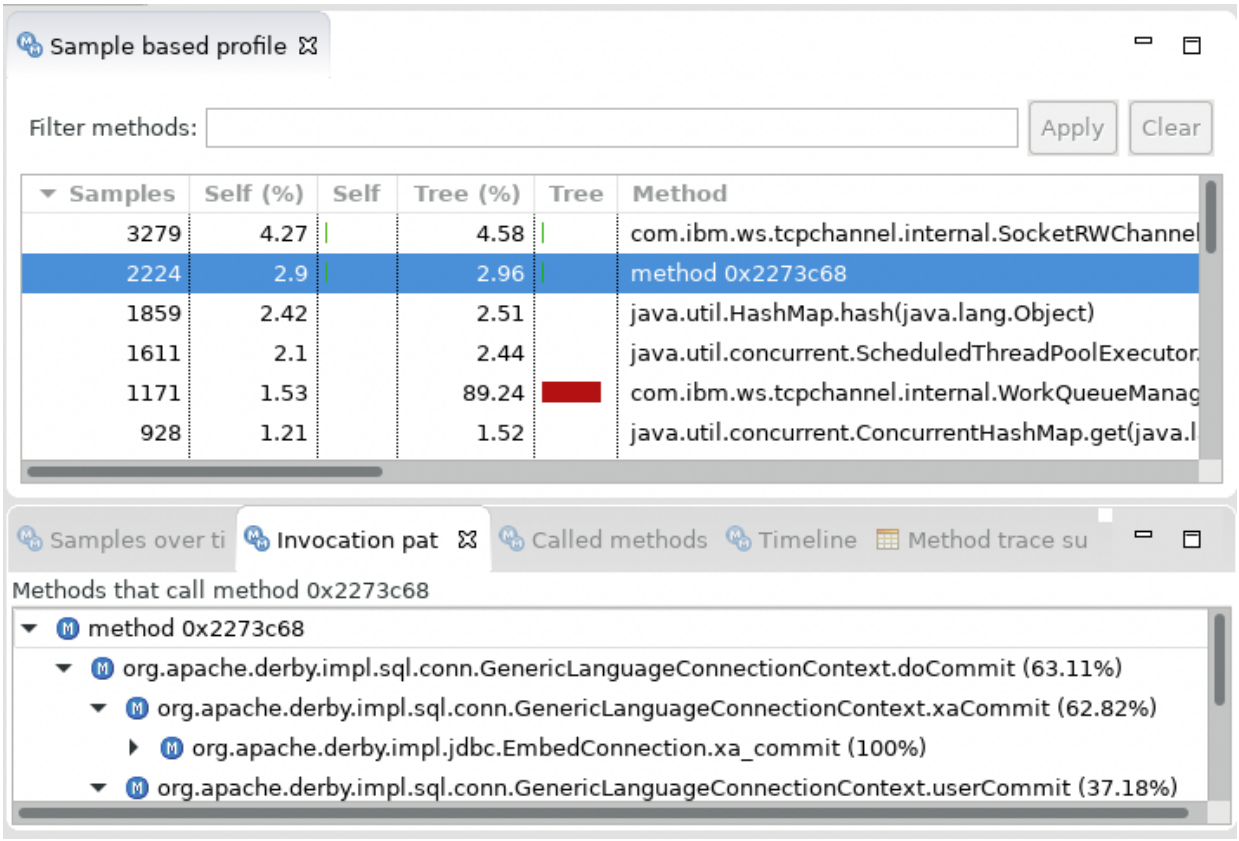


20. The **Method profiling** view will show CPU samples by method:



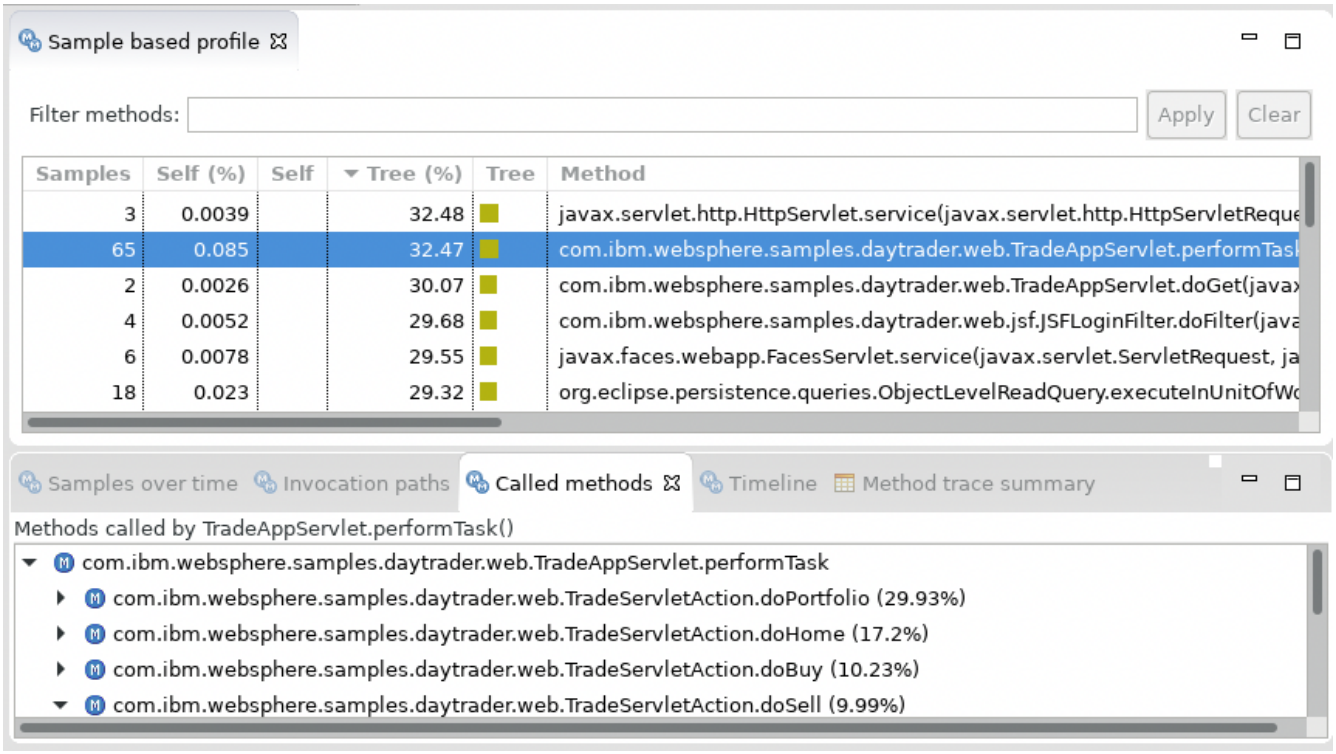
21. The **Self (%)** column reports the percent of samples where a method was at the top of the stack. The **Tree (%)** column reports the percent of samples where a method was somewhere else in the stack. Make sure to check that the **Samples** column is at least in the hundreds or thousands; otherwise, the CPU usage is likely not that high or a problem did not occur. The **Self** and **Tree** percentages are a percent of samples, not of total CPU.

22. Any methods over ~1% are worthy of considering how to optimize or to avoid. For example, ~2% of samples were in method 0x2946f10 (for various reasons, some methods may not resolve but you can usually figure things out from the invocation paths). Selecting that row and switching to the **Invocation Paths** view shows the percent of samples leading to those calls:



1. In the above example, 63.11% of samples (i.e. of 2.9% of total samples) were invoked by org.apache.derby.impl.sql.conn.GenericLanguageConnectionContext.doCommit.

23. If you sort by **Tree %**, skip the framework methods from Java and WAS, and find the first application method. In this example, about 32% of total samples was consumed by com.ibm.websphere.samples.daytrader.web.TradeAppServlet.performTask and all of the methods it called. The **Called Methods** view may be further reviewed to investigate the details of this usage; in this example, doPortfolio drove most of the CPU samples.



14 Crashes

Crashes are operating system events that destroy the Java process. By default, IBM Java and OpenJ9 capture most crash signals from the operating system and produce helpful diagnostics before the process terminates.

14.1 Crashes Theory

Unlike Java exceptions, crashes are events that, in general, cannot be recovered. Crashes occur at a native code level either inside the JVM itself, inside JNI libraries, or inside operating system libraries.

14.2 Crash Lab

1. Open a browser to http://localhost:9082/jni_web_hello_world/jniwrapper?str=test
2. This will cause a crash.
3. The first place to look for a potential crash is in the stderr of the process. For this Liberty server, the stdout and stderr are written to **/opt/ibm/wlp/usr/servers/test/logs/console.log**. Open this file in **Mousepad** or the terminal and find the output starting with **Unhandled exception**. For example:

```
Unhandled exception
Type=Segmentation error vmState=0x00040000
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000
```

```
Signal_Code=00000001
Handler1=00007F1731B16B00 Handler2=00007F17313FCFB0
InaccessibleAddress=0000000000000000
RDI=00007F16A8079848 RSI=00007F172762B023 RAX=0000000000000000 RBX=0000000000000018
RCX=00000000FFD9FFF0 RDX=0000000000000000 R8=0000000000000000 R9=00007F1715552700
R10=00007F173344D170 R11=00007F1733171E40 R12=0000000000000000 R13=00007F1731BE39CC
R14=00007F171554F810 R15=0000000000000000
RIP=00007F172762A1CD GS=0000 FS=0000 RSP=00007F171554F510
EFlags=0000000000010246 CS=0033 RBP=00007F171554F540 ERR=0000000000000004
TRAPNO=000000000000000E OLDMASK=0000000000000000 CR2=0000000000000000
xmm0 ffffffff00000000 (f: 0.000000, d: -nan)
xmm1 7257650074736574 (f: 1953719680.000000, d: 6.239805e+242)
xmm2 ff00000000ff0000 (f: 16711680.000000, d: -5.486124e+303)
xmm3 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm4 00000000000000ff (f: 255.000000, d: 1.259867e-321)
xmm5 bcca000000000000 (f: 0.000000, d: -7.216450e-16)
xmm6 bc1c000000000000 (f: 0.000000, d: -3.794708e-19)
xmm7 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm8 0074736574000a64 (f: 1946159744.000000, d: 1.820179e-306)
xmm9 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm10 3fd4618bc21c5ec2 (f: 3256639232.000000, d: 3.184537e-01)
xmm11 4120a4d2906fa32a (f: 2423235328.000000, d: 5.453853e+05)
xmm12 000000003e23f24e (f: 1042543168.000000, d: 5.150848e-315)
xmm13 00000000467332ce (f: 1181954816.000000, d: 5.839632e-315)
xmm14 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm15 3fc8a1142284508a (f: 579096704.000000, d: 1.924157e-01)
Module=/opt/jni_web_hello_world/target/libNativeWrapper.so
Module_base_address=00007F1727629000
Symbol=Java_com_example_NativeWrapper_testNativeMethod
Symbol_address=00007F172762A139
Target=2_90_20190306_411656 (Linux 4.9.125-linuxkit)
CPU=amd64 (4 logical CPUs) (0x2ed95f000 RAM)
----------- Stack Backtrace -----------
Java_com_example_NativeWrapper_testNativeMethod+0x94 (0x00007F172762A1CD
[libNativeWrapper.so+0x11cd])
(0x00007F1731BB62C4 [libj9vm29.so+0x13e2c4])
(0x00007F1731BB3A51 [libj9vm29.so+0x13ba51])
(0x00007F1731AA439E [libj9vm29.so+0x2c39e])
(0x00007F1731A91090 [libj9vm29.so+0x19090])
(0x00007F1731B50DA2 [libj9vm29.so+0xd8da2])
---------------------------------------
JVMDUMP039I Processing dump event "gpf", detail "" at 2019/05/15 16:25:37 - please
wait.
JVMDUMP032I JVM requested System dump using
'/opt/ibm/wlp/usr/servers/test/core.20190515.162537.202.0001.dmp' in response to an
event
JVMDUMP010I System dump written to
/opt/ibm/wlp/usr/servers/test/core.20190515.162537.202.0001.dmp
JVMDUMP032I JVM requested Java dump using
'/opt/ibm/wlp/usr/servers/test/javacore.20190515.162537.202.0002.txt' in response
to an event
JVMDUMP010I Java dump written to
/opt/ibm/wlp/usr/servers/test/javacore.20190515.162537.202.0002.txt
JVMDUMP032I JVM requested Snap dump using
'/opt/ibm/wlp/usr/servers/test/Snap.20190515.162537.202.0003.trc' in response to an
event
JVMDUMP010I Snap dump written to
/opt/ibm/wlp/usr/servers/test/Snap.20190515.162537.202.0003.trc
JVMDUMP007I JVM Requesting JIT dump using
```

```
'/opt/ibm/wlp/usr/servers/test/jitdump.20190515.162537.202.0004.dmp'
JVMDUMP010I JIT dump written to
/opt/ibm/wlp/usr/servers/test/jitdump.20190515.162537.202.0004.dmp
JVMDUMP013I Processed dump event "gpf", detail "".
```

4. There are a few things to point out in the above output:
    a. **Type=Segmentation error**: This shows the human readable cause of the crash.
    b. **Signal_Number=0000000b**: This shows the crash signal (which is what the human readable cause comes from). From the Linux terminal, run the "kill -l" command to list all signals and convert **Signal_Number** from hexadecimal to decimal; in this example, 0xb = 11 = SIGSEGV:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

    c. **Module=/opt/jni_web_hello_world/target/libNativeWrapper.so**: This tells you the failing shared library. This quickly shows the main suspect. In this case, we can see the crash is in a third party native library[32] and not the JVM.
    d. **Symbol=Java_com_example_NativeWrapper_testNativeMethod**: If the crash occurred in JNI code, this shows the native JNI method the crash occurred in. This can be helpful for the developer of the module or for quick internet searches.
    e. **JVMDUMP039I Processing dump event "gpf", detail "" at 2019/05/15 16:25:37 - please wait.**: This message shows that the JVM captured the crash (gpf = general protection fault; in other words, segmentation fault) and created various diagnostics.
5. The first thing to do is to open the javacore*txt file that the crash produced:
    a. **1TISIGINFO Dump Event "gpf" (00002000) received**: This shows that a crash was handled.
    b. **1XHEXCPCODE Signal_Number: 0000000B**: This shows the hexadecimal signal number.
    c. **1XHEXCPMODULE Module: /opt/jni_web_hello_world/target/libNativeWrapper.so**: This shows the crashing module.
    d. **1XHEXCPMODULE Symbol: Java_com_example_NativeWrapper_testNativeMethod**: This shows the crashing JNI method.
    e. Search the file for **Current thread** to find the crashing thread stack:

```
1XMCURTHDINFO  Current thread
```

[32] https://github.com/kgibm/jni_web_hello_world

```
3XMTHREADINFO      "Default Executor-thread-56" J9VMThread:0x0000000001B07B00, omrthread_t:0x00007F169C00C5E8,
java/lang/Thread:0x00000000FF7804E0, state:R, prio=5
3XMJAVALTHREAD            (java/lang/Thread getId:0x6C, isDaemon:true)
3XMTHREADINFO1            (native thread ID:0x54D, native priority:0x5, native policy:UNKNOWN, vmstate:R, vm
thread flags:0x00000020)
3XMTHREADINFO2            (native stack address range from:0x00007F1715513000, to:0x00007F1715553000,
size:0x40000)
3XMCPUTIME               CPU usage total: 0.268345952 secs, current category="Application"
3XMHEAPALLOC             Heap bytes allocated since last GC cycle=6634184 (0x653AC8)
3XMTHREADINFO3           Java callstack:
4XESTACKTRACE                at com/example/NativeWrapper.testNativeMethod(Native Method)
4XESTACKTRACE                at com/example/JNIWrapper.service(JNIWrapper.java:33)
4XESTACKTRACE                at javax/servlet/http/HttpServlet.service(HttpServlet.java:791)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/servlet/ServletWrapper.service(ServletWrapper.java:1255)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/servlet/ServletWrapper.handleRequest(ServletWrapper.java:743)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/servlet/ServletWrapper.handleRequest(ServletWrapper.java:440)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/WebAppFilterChain.invokeTarget(WebAppFilterChain.java:182)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/WebAppFilterChain.doFilter(WebAppFilterChain.java:93)
4XESTACKTRACE                at
com/ibm/ws/security/jaspi/JaspiServletFilter.doFilter(JaspiServletFilter.java:56)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/FilterInstanceWrapper.doFilter(FilterInstanceWrapper.java:201)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/WebAppFilterChain.doFilter(WebAppFilterChain.java:90)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/WebAppFilterManager.doFilter(WebAppFilterManager.java:996)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/filter/WebAppFilterManager.invokeFilters(WebAppFilterManager.java:1134)
4XESTACKTRACE                at com/ibm/ws/webcontainer/webapp/WebApp.handleRequest(WebApp.java:4968)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/osgi/DynamicVirtualHost$2.handleRequest(DynamicVirtualHost.java:314)
4XESTACKTRACE                at com/ibm/ws/webcontainer/WebContainer.handleRequest(WebContainer.java:992)
4XESTACKTRACE                at
com/ibm/ws/webcontainer/osgi/DynamicVirtualHost$2.run(DynamicVirtualHost.java:279)
4XESTACKTRACE                at
com/ibm/ws/http/dispatcher/internal/channel/HttpDispatcherLink$TaskWrapper.run(HttpDispatcherLink.java:1061)
4XESTACKTRACE                at
com/ibm/ws/http/dispatcher/internal/channel/HttpDispatcherLink.wrapHandlerAndExecute(HttpDispatcherLink.java:4
17)
4XESTACKTRACE                at
com/ibm/ws/http/dispatcher/internal/channel/HttpDispatcherLink.ready(HttpDispatcherLink.java:376)
4XESTACKTRACE                at
com/ibm/ws/http/channel/internal/inbound/HttpInboundLink.handleDiscrimination(HttpInboundLink.java:532)
4XESTACKTRACE                at
com/ibm/ws/http/channel/internal/inbound/HttpInboundLink.handleNewRequest(HttpInboundLink.java:466)
4XESTACKTRACE                at
com/ibm/ws/http/channel/internal/inbound/HttpInboundLink.processRequest(HttpInboundLink.java:331)
4XESTACKTRACE                at
com/ibm/ws/http/channel/internal/inbound/HttpInboundLink.ready(HttpInboundLink.java:302)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/NewConnectionInitialReadCallback.sendToDiscriminators(NewConnectionInitialReadC
allback.java:165)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/NewConnectionInitialReadCallback.complete(NewConnectionInitialReadCallback.java
:74)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/WorkQueueManager.requestComplete(WorkQueueManager.java:503)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/WorkQueueManager.attemptIO(WorkQueueManager.java:573)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/WorkQueueManager.workerRun(WorkQueueManager.java:954)
4XESTACKTRACE                at
com/ibm/ws/tcpchannel/internal/WorkQueueManager$Worker.run(WorkQueueManager.java:1043)
4XESTACKTRACE                at
com/ibm/ws/threading/internal/ExecutorServiceImpl$RunnableWrapper.run(ExecutorServiceImpl.java:239)
4XESTACKTRACE                at
java/util/concurrent/ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1160(Compiled Code))
4XESTACKTRACE                at
java/util/concurrent/ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:635)
4XESTACKTRACE                at java/lang/Thread.run(Thread.java:812)
3XMTHREADINFO3           Native callstack:
4XENATIVESTACK               (0x00007F173142A702 [libj9prt29.so+0x4e702])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F173142A77E [libj9prt29.so+0x4e77e])
```

```
4XENATIVESTACK               (0x00007F173142A874 [libj9prt29.so+0x4e874])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F173142A5DB [libj9prt29.so+0x4e5db])
4XENATIVESTACK               (0x00007F1731426C62 [libj9prt29.so+0x4ac62])
4XENATIVESTACK               (0x00007F1731427A04 [libj9prt29.so+0x4ba04])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F1730CBE406 [libj9dmp29.so+0x1a406])
4XENATIVESTACK               (0x00007F1730CBE59D [libj9dmp29.so+0x1a59d])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F1730CBACED [libj9dmp29.so+0x16ced])
4XENATIVESTACK               (0x00007F1730CB627D [libj9dmp29.so+0x1227d])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F1730CB75E0 [libj9dmp29.so+0x135e0])
4XENATIVESTACK               (0x00007F1730CC093C [libj9dmp29.so+0x1c93c])
4XENATIVESTACK               (0x00007F1730CA8D4D [libj9dmp29.so+0x4d4d])
4XENATIVESTACK               (0x00007F1730CA8365 [libj9dmp29.so+0x4365])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F1730CAB96B [libj9dmp29.so+0x796b])
4XENATIVESTACK               (0x00007F1730CABAEC [libj9dmp29.so+0x7aec])
4XENATIVESTACK               (0x00007F1730CC255B [libj9dmp29.so+0x1e55b])
4XENATIVESTACK               (0x00007F1731B166F2 [libj9vm29.so+0x9e6f2])
4XENATIVESTACK               (0x00007F17313FDEC8 [libj9prt29.so+0x21ec8])
4XENATIVESTACK               (0x00007F1731B168E6 [libj9vm29.so+0x9e8e6])
4XENATIVESTACK               (0x00007F17313FD14F [libj9prt29.so+0x2114f])
4XENATIVESTACK               (0x00007F173340B070 [libpthread.so.0+0x13070])
4XENATIVESTACK               Java_com_example_NativeWrapper_testNativeMethod+0x94 (0x00007F172762A1CD
[libNativeWrapper.so+0x11cd])
4XENATIVESTACK               (0x00007F1731BB62C4 [libj9vm29.so+0x13e2c4])
4XENATIVESTACK               (0x00007F1731BB3A51 [libj9vm29.so+0x13ba51])
4XENATIVESTACK               (0x00007F1731AA439E [libj9vm29.so+0x2c39e])
4XENATIVESTACK               (0x00007F1731A91090 [libj9vm29.so+0x19090])
```

i. This shows both the Java stack and the native stack which is very useful to understand what's driving the crash.

6. Next, we'll want to look at the core dump in the Java dump viewer.

   a. Execute **jdmpview**, passing the core dump from the messages above. For example:

```
$ jdmpview -core /opt/ibm/wlp/usr/servers/test/core.20190515.162537.202.0001.dmp
DTFJView version 4.29.5, using DTFJ version 1.12.29003
Loading image from DTFJ...

For a list of commands, type "help"; for how to use "help", type "help help"
Available contexts (* = currently selected context) :

Source : file:///opt/ibm/wlp/usr/servers/test/core.20190515.162537.202.0001.dmp
      *0 : PID: 1576 : JRE 1.8.0 Linux amd64-64 (build 8.0.5.31 - pxa6480sr5fp31-
20190311_03(SR5 FP31))
```

   b. Next, if you know it's a crash, run the **!gpinfo** command to get similar information to what we say in **stderr**:

```
> !gpinfo
Failing Thread: !j9vmthread 0x1b07b00
Failing Thread ID: 0x54d (1357)
gpInfo:
J9Generic_Signal_Number=00000004 Signal_Number=0000000b Error_Value=00000000
Signal_Code=00000001
Handler1=00007F1731B16B00 Handler2=00007F17313FCFB0
InaccessibleAddress=0000000000000000
RDI=00007F16A8079848 RSI=00007F172762B023 RAX=0000000000000000 RBX=0000000000000018
RCX=00000000FFD9FFF0 RDX=0000000000000000 R8=0000000000000000 R9=00007F1715552700
R10=00007F173344D170 R11=00007F1733171E40 R12=0000000000000000 R13=00007F1731BE39CC
R14=00007F171554F810 R15=0000000000000000
RIP=00007F172762A1CD GS=0000 FS=0000 RSP=00007F171554F510
EFlags=0000000000010246 CS=0033 RBP=00007F171554F540 ERR=0000000000000004
TRAPNO=000000000000000E OLDMASK=0000000000000000 CR2=0000000000000000
```

```
xmm0 ffffffff00000000 (f: 0.000000, d: -nan)
xmm1 7257650074736574 (f: 1953719680.000000, d: 6.239805e+242)
xmm2 ff00000000ff0000 (f: 16711680.000000, d: -5.486124e+303)
xmm3 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm4 00000000000000ff (f: 255.000000, d: 1.259867e-321)
xmm5 bcca000000000000 (f: 0.000000, d: -7.216450e-16)
xmm6 bc1c000000000000 (f: 0.000000, d: -3.794708e-19)
xmm7 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm8 0074736574000a64 (f: 1946159744.000000, d: 1.820179e-306)
xmm9 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm10 3fd4618bc21c5ec2 (f: 3256639232.000000, d: 3.184537e-01)
xmm11 4120a4d2906fa32a (f: 2423235328.000000, d: 5.453853e+05)
xmm12 000000003e23f24e (f: 1042543168.000000, d: 5.150848e-315)
xmm13 00000000467332ce (f: 1181954816.000000, d: 5.839632e-315)
xmm14 0000000000000000 (f: 0.000000, d: 0.000000e+00)
xmm15 3fc8a1142284508a (f: 579096704.000000, d: 1.924157e-01)
Module=/opt/jni_web_hello_world/target/libNativeWrapper.so
Module_base_address=00007F1727629000
Symbol=Java_com_example_NativeWrapper_testNativeMethod
Symbol_address=00007F172762A139
```

c. Finally, run **info thread** to see the crashing thread's Java stack and related stack frame locals:

```
> info thread
 process id: 202

  thread id: 1357
   registers:
   native stack sections:
   native stack frames:
   properties:
   associated Java thread:
    name:          Default Executor-thread-56
    Thread object: java/lang/Thread @ 0xff7804e0
    Daemon:        true
    ID:            108 (0x6c)
    Priority:      5
    Thread.State:  RUNNABLE
    JVMTI state:   ALIVE RUNNABLE
    Java stack frames:
     bp: 0x00000000024b5b40  method: String
com/example/NativeWrapper.testNativeMethod(String)  (Native Method)
      objects: 0xfe125fb0
     bp: 0x00000000024b5b88  method: void
com/example/JNIWrapper.service(javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)  source: JNIWrapper.java:33
      objects: 0xfe1128e0 0xfe112cc8 [...]
```

7. Next, we'll want to look at the core dump in the operating system debugger. All we usually need is the native stack trace details which are provided to the module owner to review, although sometimes they also need the full core dump.

   a. Load the Linux debugger, passing the executable that crashed and the path to the core dump from the messages above. For example:

```
$ gdb /opt/ibm/java/bin/java
/opt/ibm/wlp/usr/servers/test/core.20190515.162537.202.0001.dmp
[...]
```

```
Program terminated with signal SIGSEGV, Segmentation fault.
#0  __pthread_kill (threadid=<optimized out>, signo=11)
    at ../sysdeps/unix/sysv/linux/pthread_kill.c:56
56      return (INTERNAL_SYSCALL_ERROR_P (val, err)
Missing separate debuginfos, use: dnf debuginfo-install libgcc-8.3.1-2.fc29.x86_64
sssd-client-2.0.0-5.fc29.x86_64
```

b. Next, type the **bt** command and prese enter, and continue to press enter until the full stack is printed:

```
(gdb) bt
#0  __pthread_kill (threadid=<optimized out>, signo=11)
    at ../sysdeps/unix/sysv/linux/pthread_kill.c:56
#1  0x00007f173142aeed in omrdump_create ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9prt29.so
#2  0x00007f1730cac4e2 in doSystemDump ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9dmp29.so
#3  0x00007f1730ca8365 in protectedDumpFunction ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9dmp29.so
#4  0x00007f17313fdec8 in omrsig_protect ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9prt29.so
#5  0x00007f1730cab96b in runDumpFunction ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9dmp29.so
#6  0x00007f1730cabaec in runDumpAgent ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9dmp29.so
#7  0x00007f1730cc255b in triggerDumpAgents ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9dmp29.so
#8  0x00007f1731b166f2 in generateDiagnosticFiles ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#9  0x00007f17313fdec8 in omrsig_protect ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9prt29.so
#10 0x00007f1731b168e6 in vmSignalHandler ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#11 0x00007f17313fd14f in masterSynchSignalHandler ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9prt29.so
#12 <signal handler called>
#13 0x00007f172762a1cd in Java_com_example_NativeWrapper_testNativeMethod
(env=0x1b07b00,
    c=0x19d7230, s=0x24b5b40) at com_example_NativeWrapper.c:12
#14 0x00007f1731bb62c4 in ffi_call_unix64 ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#15 0x00007f1731bb3a51 in ffi_call () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#16 0x00007f1731aa439e in VM_BytecodeInterpreter::run(J9VMThread*) ()
   from /opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
--Type <RET> for more, q to quit, c to continue without paging--c
#17 0x00007f1731a91090 in bytecodeLoop () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#18 0x00007f1731b50da2 in c_cInterpreter () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#19 0x00007f1731b0055a in runJavaThread () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#20 0x00007f1731b5072f in javaProtectedThreadProc () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#21 0x00007f17313fdec8 in omrsig_protect () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9prt29.so
#22 0x00007f1731b4cb8a in javaThreadProc () from
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9vm29.so
#23 0x00007f173186b2c6 in thread_wrapper () from
```

```
/opt/ibm/java/jre/lib/amd64/compressedrefs/libj9thr29.so
#24 0x00007f173340058e in start_thread (arg=<optimized out>) at
pthread_create.c:486
#25 0x00007f1733112683 in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
```

c. The above stack should be sent to the developer of the module for them to investigate the crash.

i. If you're curious, you can further investigate the crash if you can guess where to look. In the above example, we know the code is crashing in our JNI library and we can see the top method of that library is in frame #13, so switch to that frame:

```
(gdb) frame 13
#13 0x00007f172762a1cd in Java_com_example_NativeWrapper_testNativeMethod
(env=0x1b07b00,
    c=0x19d7230, s=0x24b5b40) at com_example_NativeWrapper.c:12
12        printf("Printing nonsense value: %d", *p);
```

ii. If the module is compiled with symbols (as it always should be on most operating systems[33]), then you'll see the actual code that crashed.

iii. In this example, we can further display the value of the pointer that's likely causing the crash:

```
(gdb) print p
$1 = (int *) 0x0
```

iv. This shows that the code tried to dereference a NULL pointer, thus causing the SIGSEGV.

15 Native Memory Leaks

Native memory leaks and native OutOfMemoryErrors (NOOMs) are one of the more complicated problem determination topics. This lab will simulate a native memory leak and show how to diagnose it.

15.1 Native Memory Theory

A Java process is at heart a native operating system process. The operating system provides each process a virtual address space depending on the processor architecture. For most 32-bit processes and CPUs, this is 0 – 4GB, and for most 64-bit processes and CPUs, this is 0 – 16EB (practically, 0 – 256TB). As a program runs, process virtual memory usage is converted to physical memory addresses in RAM.

Out of this virtual address space, Java carves out a chunk for the Java heap with a maximum size specified by -Xmx. However, Java also has various other native data structures outside of the Java heap that support the Java program, the JIT compiler, etc. In addition, any third party native libraries or OS libraries may consume additional native memory. In particular, each class and classloader has a corresponding native structure in the Java process virtual address space that is outside the Java heap.

By default, 64-bit Java uses a performance optimization called compressed references. This requires that all classloader, thread, and monitor native backing data structures are allocated in the 0-4GB virtual address space range. Therefore, if there is a leak of classes, classloaders, threads, and/or

[33] https://publib.boulder.ibm.com/httpserv/cookbook/Troubleshooting-Troubleshooting_Operating_Systems-Troubleshooting_Linux.html#Troubleshooting-Troubleshooting_Linux-Debug_Symbols

monitors, if there is no available space in this range (e.g. due to the volume of those structures or other native libraries allocating into that space [e.g. DirectByteBuffers]), then a native OutOfMemoryError will be thrown. It is possible to disable compressed references (often at a large performance cost); however, the NOOM is caused by a leak, then ultimately this will not resolve the issue because at some point the address space usage will exhaust physical RAM and paging will cause a similar problem as the NOOM.

15.2 Native Memory Leak Lab

This lab will leak classloaders which use native memory outside the Java heap and this will cause a NOOM. This will show to diagnose and analyze the native leak.

1. Ensure that the Liberty **test** server is started.

```
$ /opt/ibm/wlp/bin/server status test
Server test is running [...]
```

2. If the Liberty **test** server is not started, then start it:

```
$ /opt/ibm/wlp/bin/server start defaultServer
```

3. Open a browser to http://localhost:9082/jni_web_hello_world/jniwrapper?str=nativemem
    a. This simply consumes a large chunk of memory below 4GB to simulate the issue faster[34].
4. Use the **ab** program (Apache Bench; a utility that's part of the httpd package) to execute multiple calls to a servlet running in Liberty which will leak a classloader/thread each time:

```
$ ab -n 1000000 -c 4 http://localhost:9082/swat/ClassloaderLeak
```

5. In a separate tab, you can run the following command to watch the native memory of the process increasing:

```
$ watch ps -o vsz,rss,command -p $(pgrep -f test)
```

6. After about 5 minutes, the virtual address space below 4GB will become exhausted and a native OutOfMemoryError is thrown. You will see this in **/opt/ibm/wlp/usr/servers/test/logs/console.log**. For example:

```
JVMDUMP039I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError"
at 2019/05/20 20:29:03 - please wait.
JVMDUMP010I System dump written to
/opt/ibm/wlp/usr/servers/test/core.20190520.202903.172.0001.dmp
JVMDUMP010I Java dump written to
/opt/ibm/wlp/usr/servers/test/javacore.20190520.202903.172.0003.txt
```

7. After the javacore has been written, you can dump some final information and then kill the JVM:

```
$ cat /proc/$(pgrep -f test)/smaps > smaps_$(hostname)_$(date
+"%Y%m%d_%H%M%S_%N").txt
$ pkill -9 -f test
```

8. The first step is to open the **javacore*txt** file:
    a. Review the **1TISIGINFO** line. Note that, unlike the previous Java OutOfMemoryError

[34] https://github.com/kgibm/jni_web_hello_world/blob/7b1de65a9669e6edaedace3f5ad886f3a922b790/src/main/c/com_example_NativeWrapper.c#L18

exercise above, this time the detail of the OOM shows "**native memory exhausted**":

```
1TISIGINFO     Dump Event "systhrow" (00040000) Detail "java/lang/OutOfMemoryError"
"native memory exhausted" received
```

b. Review the "Object Memory" section which shows where the parts of the Java heap are placed in virtual memory:

```
1STHEAPTYPE    Object Memory
NULL           id                 start              end                size               space/region
1STHEAPSPACE   0x00007FC5800B9DF0         --                 --                 --         Generational
1STHEAPREGION  0x00007FC5800BA630 0x0000000080000000 0x00000000EDA00000 0x000000006DA00000 Generational/Tenured Region
1STHEAPREGION  0x00007FC5800BA280 0x00000000EDA00000 0x00000000FCF10000 0x000000000F510000 Generational/Nursery Region
1STHEAPREGION  0x00007FC5800B9ED0 0x00000000FCF10000 0x0000000100000000 0x00000000030F0000 Generational/Nursery Region
```

i. In this case, the Java heap (2GB) is completely within the 0-4GB virtual address space so it will compete for the 0-4GB space with class/thread/monitor native memory allocations. Place the Java heap below 4GB is a performance optimization for small Java heaps. If necessary, you may place the Java heap above 4GB with the option **-Xgc:preferredHeapBase=0x100000000** which places the Java heap to start at 4GB, although this will reduce the performance of the JVM by a few %. If the Java heap is large, the JVM automatically places it above 4GB.

c. Review the NATIVEMEMINFO section which lists the native memory allocations that the JVM is aware of (this does not include all native memory allocations in the process). The most common drivers of NOOMs are highlighted:

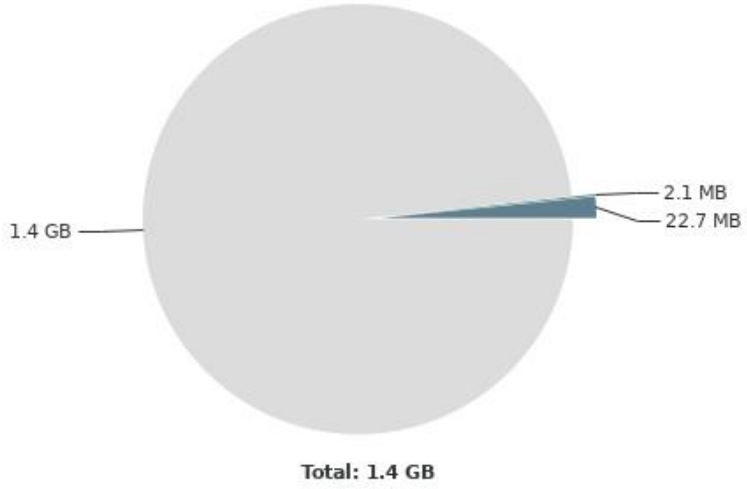
```
0SECTION       NATIVEMEMINFO subcomponent dump routine
NULL           ================================
0MEMUSER
1MEMUSER       JRE: 4,317,991,208 bytes / 4120179 allocations
1MEMUSER       |
2MEMUSER       +--VM: 3,539,467,856 bytes / 2092694 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Classes: 1,315,543,360 bytes / 2088812 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Shared Class Cache: 16,777,312 bytes / 2 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 1,298,766,048 bytes / 2088810 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Memory Manager (GC): 2,193,055,024 bytes / 886 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Heap: 2,147,545,088 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 45,509,936 bytes / 885 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Threads: 20,283,360 bytes / 350 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Stack: 911,232 bytes / 47 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Native Stack: 18,743,296 bytes / 48 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 628,832 bytes / 255 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Trace: 614,744 bytes / 388 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JVMTI: 54,776 bytes / 47 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--JVMTI Allocate(): 176 bytes / 1 allocation
```

```
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 54,600 bytes / 46 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JNI: 495,432 bytes / 1365 allocations
3MEMUSER       |  +--Port Library: 7,967,616 bytes / 195 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Unused <32bit allocation regions: 7,945,960 bytes / 37
allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 21,656 bytes / 158 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Other: 1,453,544 bytes / 651 allocations
1MEMUSER       |
2MEMUSER       +--JIT: 368,481,472 bytes / 1702 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JIT Code Cache: 268,435,456 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--JIT Data Cache: 6,291,648 bytes / 3 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Other: 93,754,368 bytes / 1698 allocations
1MEMUSER       |
2MEMUSER       +--Class Libraries: 10,283,480 bytes / 66183 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Harmony Class Libraries: 2,000 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--VM Class Libraries: 10,281,480 bytes / 66182 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--sun.misc.Unsafe: 615,232 bytes / 41 allocations
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Direct Byte Buffers: 99,808 bytes / 27 allocations
4MEMUSER       |  |  |  |
5MEMUSER       |  |  |  +--Other: 515,424 bytes / 14 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 9,666,248 bytes / 66141 allocations
1MEMUSER       |
2MEMUSER       +--Unknown: 399,758,400 bytes / 1959600 allocations
```

d. In this example, about 1.2GB of native memory (outside the Java heap) is consumed by classes and classloaders. This is the primary suspect for this NOOM.

9. Given that the primary suspects are classes/classloaders, next we'll analyze the heap dump. Open the Memory Analyzer Tool at **/opt/programs/MAT** and load the **core*dmp** file. This may take 20-30 minutes so while it's loading you can review the additional details in the next steps and return once the loading is complete.

    a. When you first load the core dump, you'll see that there is no large dominator so there are no large pie pieces:

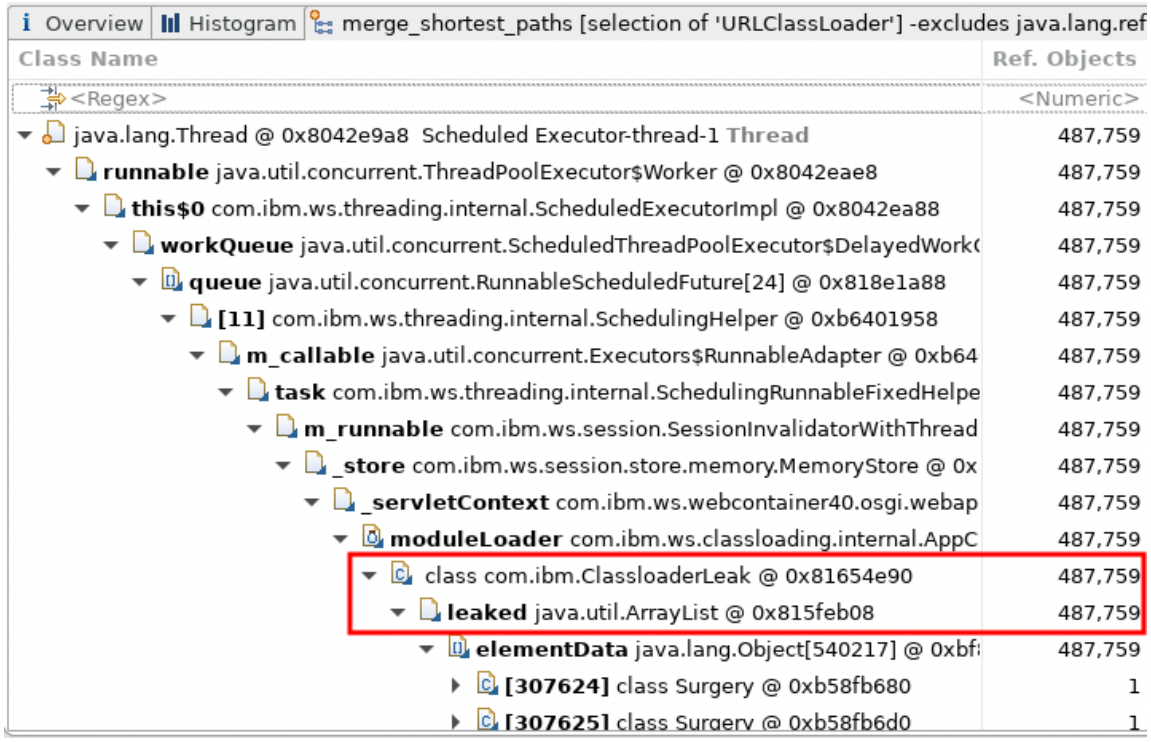


b. As in the Java OOM exercise, open the **Histogram**, click on the **Calculator**, select **Calculate minimum retained size (quick approx.)** and sort by **Retained Heap** descending. The top few items show a large number of **URLClassLoaders** retaining about 1GB (of Java heap):

Overview | Histogram

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.net.URLClassLoader	487,759	48.38 MB	>= 971.35 MB
sun.misc.URLClassPath	487,761	22.33 MB	>= 617.80 MB
sun.misc.URLClassPath$JarLoader	487,787	26.05 MB	>= 368.47 MB
java.util.HashMap	1,484,954	56.65 MB	>= 301.79 MB
java.util.jar.JarFile	487,809	37.22 MB	>= 264.27 MB
java.util.HashMap$Node[]	1,482,746	101.56 MB	>= 245.36 MB

c. Right click on URLClassLoader and select **Merge Shortest Paths to GC Roots** > **excluding all phantom/weak/soft etc. references**. Expand the path all the way down until you find the place where the classloaders are leaked. This shows that the **com.ibm.ClassloaderLeak** class has a static **leaked** ArrayList which is leaking the classloaders.

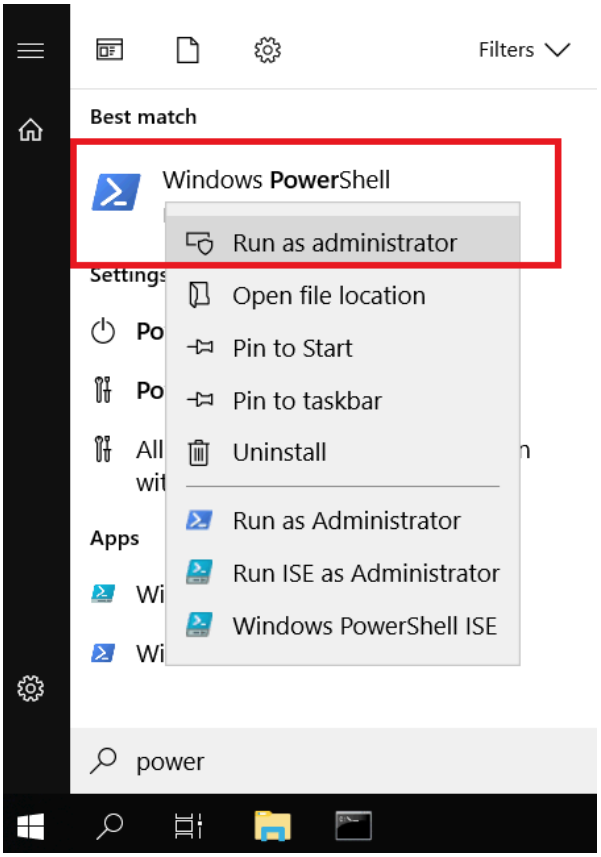


16 Appendix

16.1 Windows Remote Desktop Client

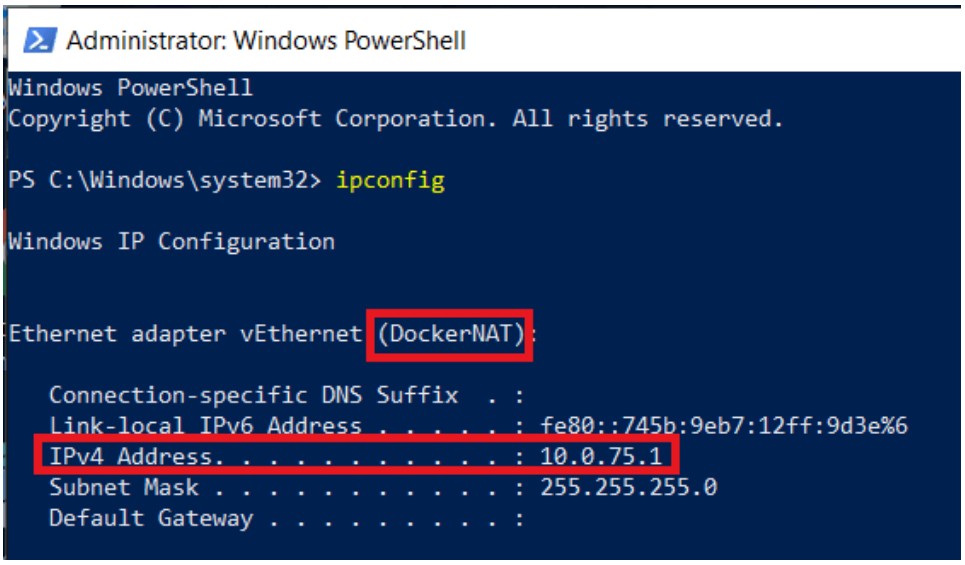
Windows requires extra steps to configure remote desktop to connect to a container[35]:

1. Open **PowerShell** as Administrator:



[35] https://social.msdn.microsoft.com/Forums/en-US/872129e4-07a5-48c3-86f7-996854e7a920/how-to-connect-via-rdp-to-container?forum=windowscontainers

2. Run **ipconfig** and copy the **IPv4** address of the **DockerNAT** adapter. For example:



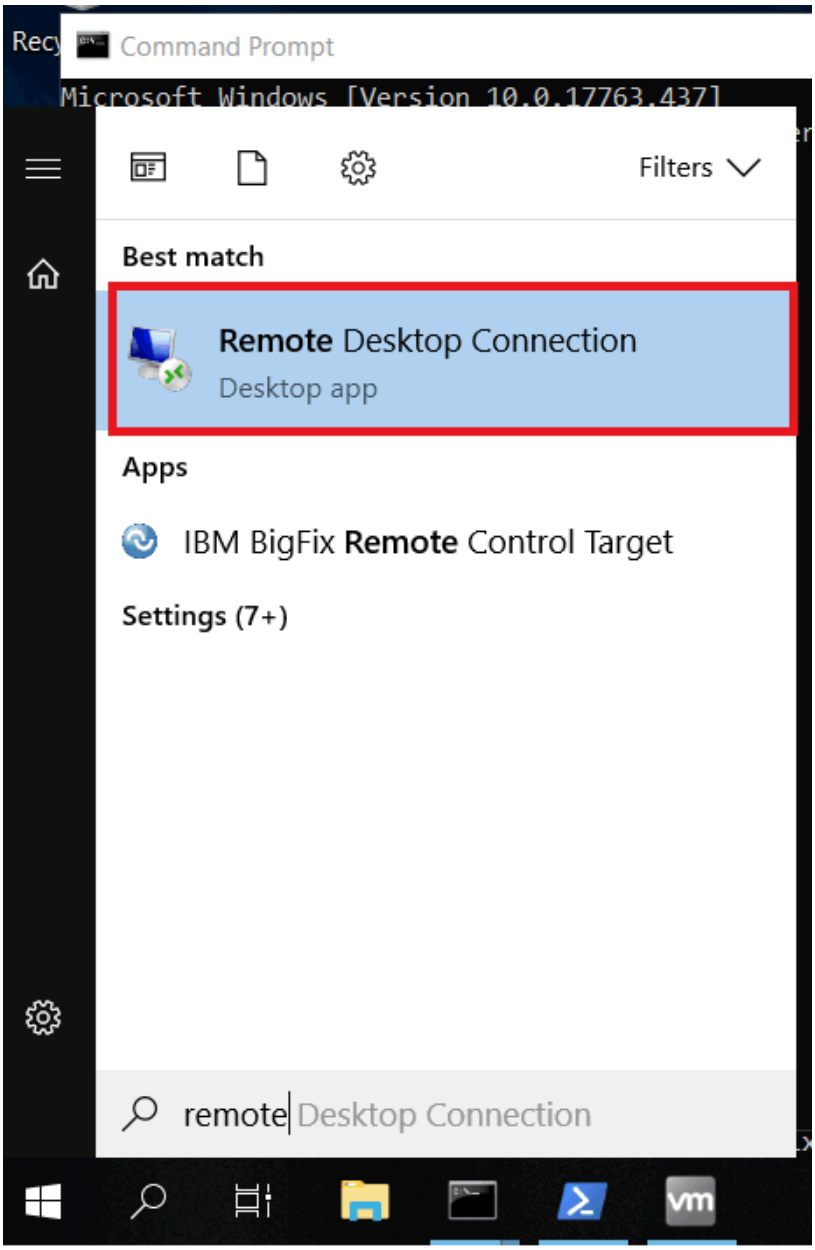
3. Run the following command in **PowerShell**:

```
New-NetFirewallRule -Name "myRDP" -DisplayName "Remote Desktop Protocol" -Protocol
TCP -LocalPort @(3389) -Action Allow
```

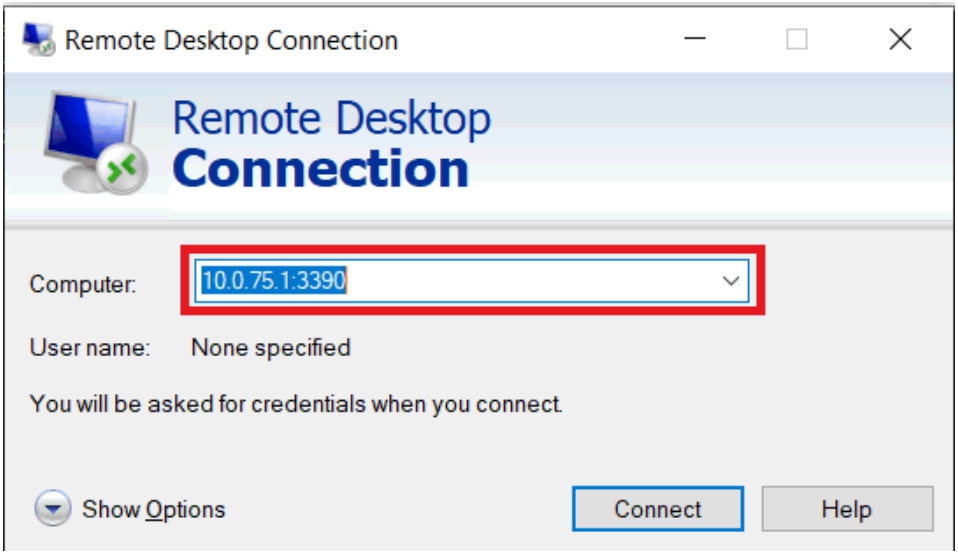
4. Run the following command in **PowerShell**:

```
New-NetFirewallRule -Name "myContainerRDP" -DisplayName "RDP Port for connecting to
Container" -Protocol TCP -LocalPort @(3390) -Action Allow
```

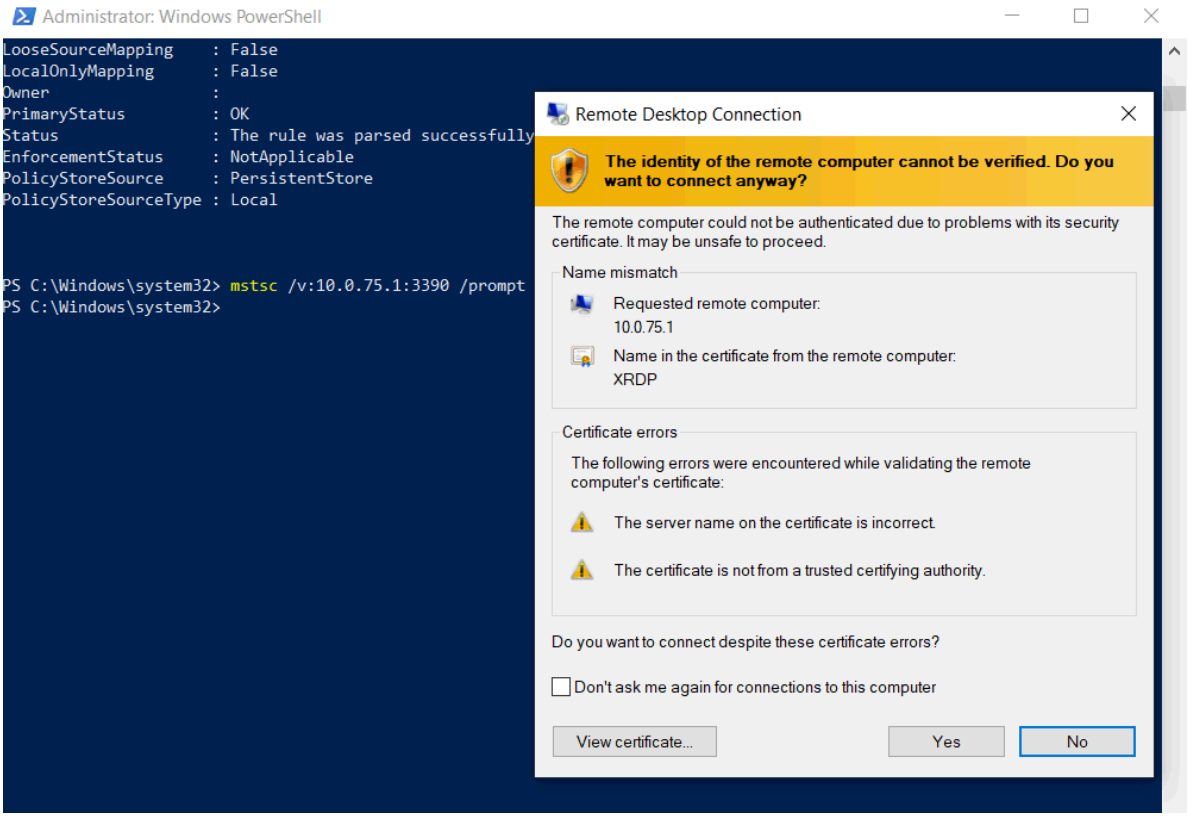
5. Run **Remote Desktop**



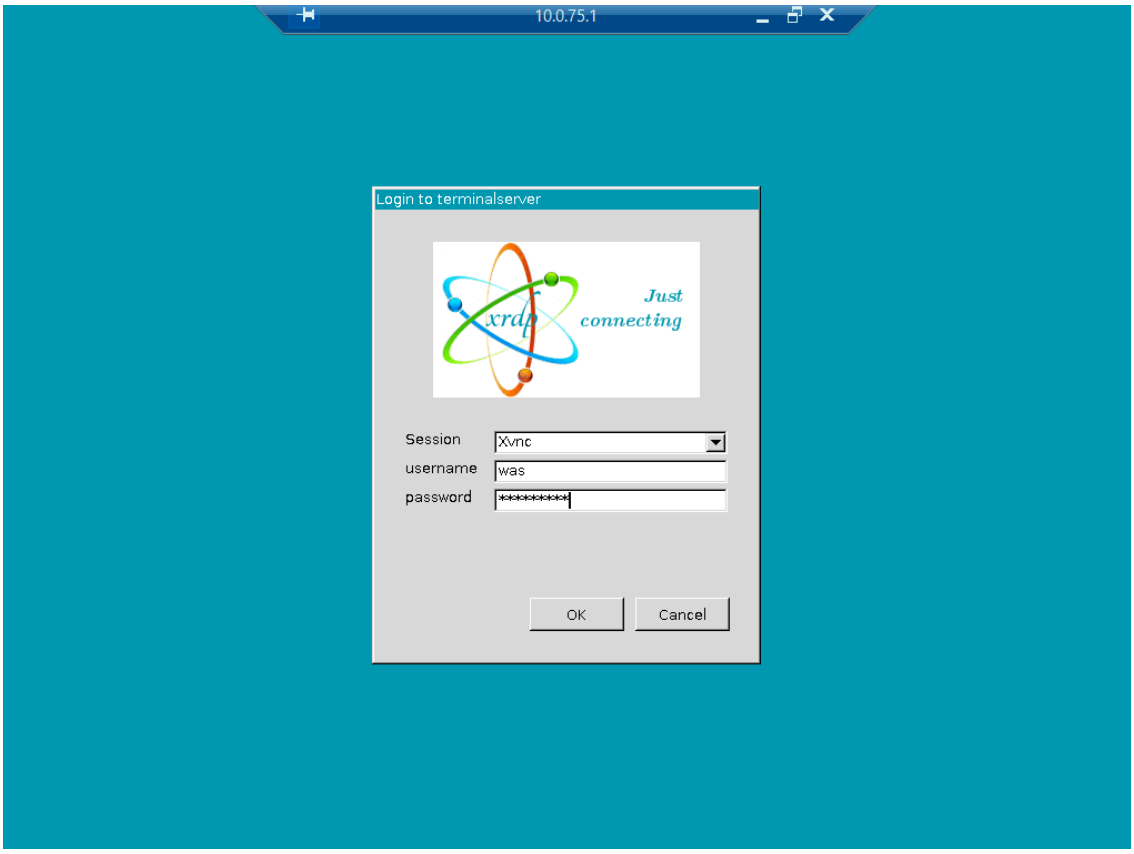
6. Enter the DockerNAT IP address (for example, 10.0.75.1) followed by :3390 as **Computer** and click **Connect**:



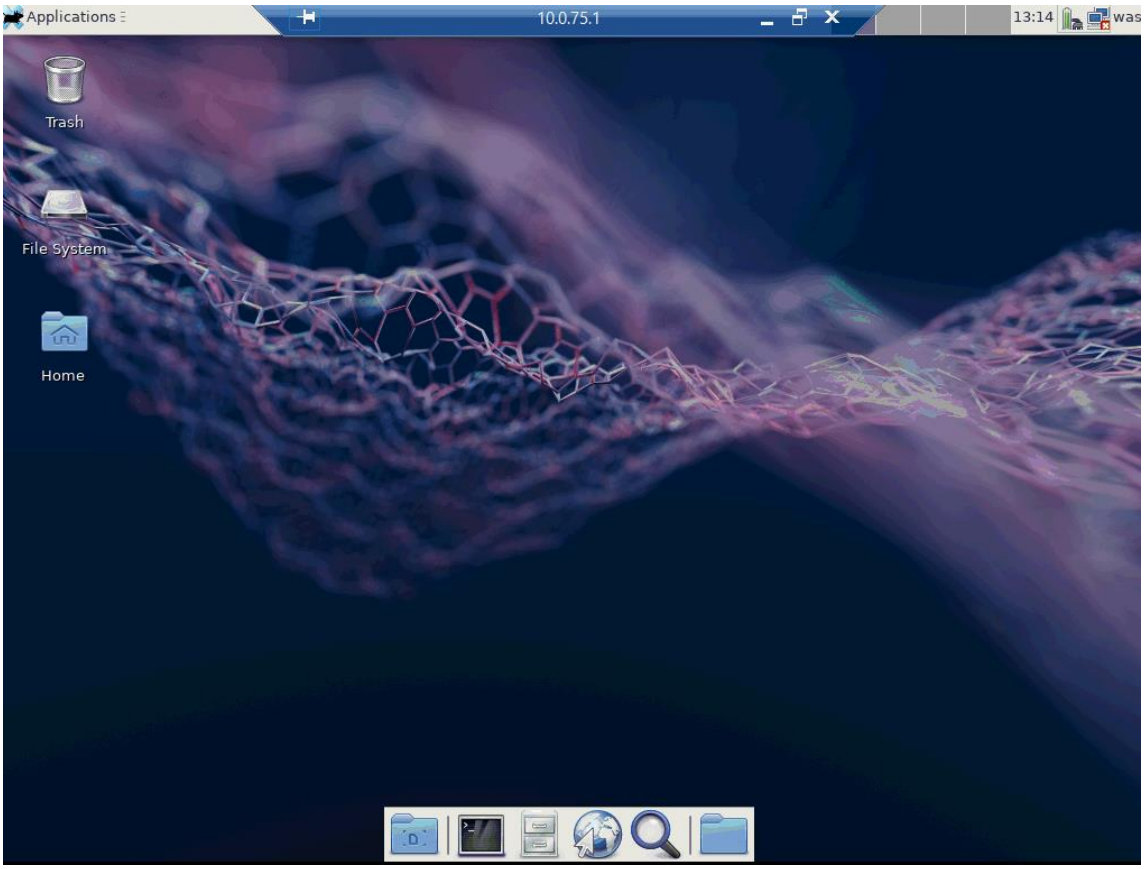
7. You'll see a certificate warning because of the name mismatch. Click **Yes** to connect:



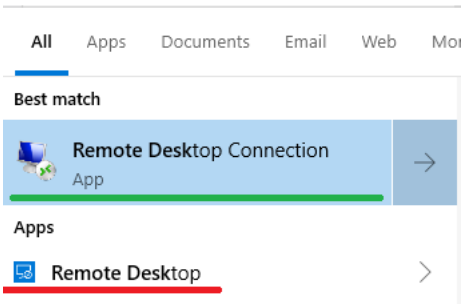
8. Type username = **was** and password = **websphere**



9. You should now be remote desktop'ed into the container:



10. Note: In some cases, only the **Remote Desktop Connection** application worked, and **not Remote Desktop**[36]:



11. Also note: Microsoft requires the above steps and the use of port 3390 instead of directly connecting to 3389[37].

[36] https://docs.microsoft.com/en-us/windows-server/remote/remote-desktop-services/clients/remote-desktop-app-compare
[37] https://social.msdn.microsoft.com/Forums/en-US/872129e4-07a5-48c3-86f7-996854e7a920/how-to-connect-via-rdp-to-container?forum=windowscontainers

16.2 Acknowledgments

Thank you to those that helped build and test this lab: Hiroko Takamiya, Andrea Pichler.