

Liberty Performance Lab

Liberty Performance Lab

- Author: Kevin Grigorenko
- Version: V18 (March 20, 2022)
- Source: <https://github.com/kgibm/dockerdebug/tree/master/fedorawasdebug>

Table of Contents

- Introduction
- Core Concepts
- Installation
 - With podman
 - With Docker Desktop
- Starting the lab
 - Start with podman
 - Start with Docker Desktop
- IBM Java and OpenJ9 Thread Dumps
- Garbage Collection
- Health Center Sampling Profiler
- WebSphere Liberty
 - Request Timing
 - HTTP NCSA Access Log
- Methodology
 - Performance Tuning Tips
 - Liberty Performance Tuning Recipe
- Appendix
 - Windows Remote Desktop Client

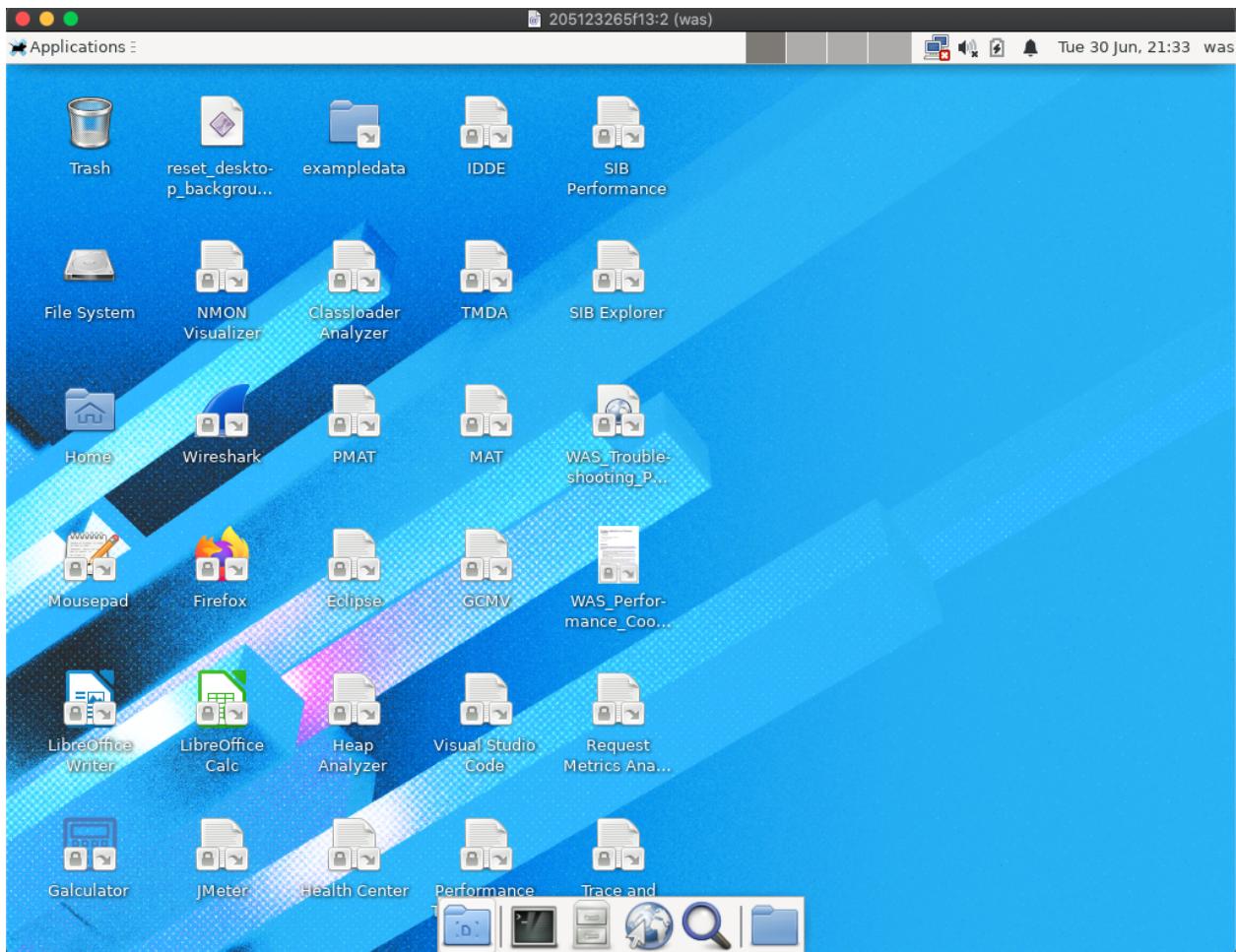
Introduction

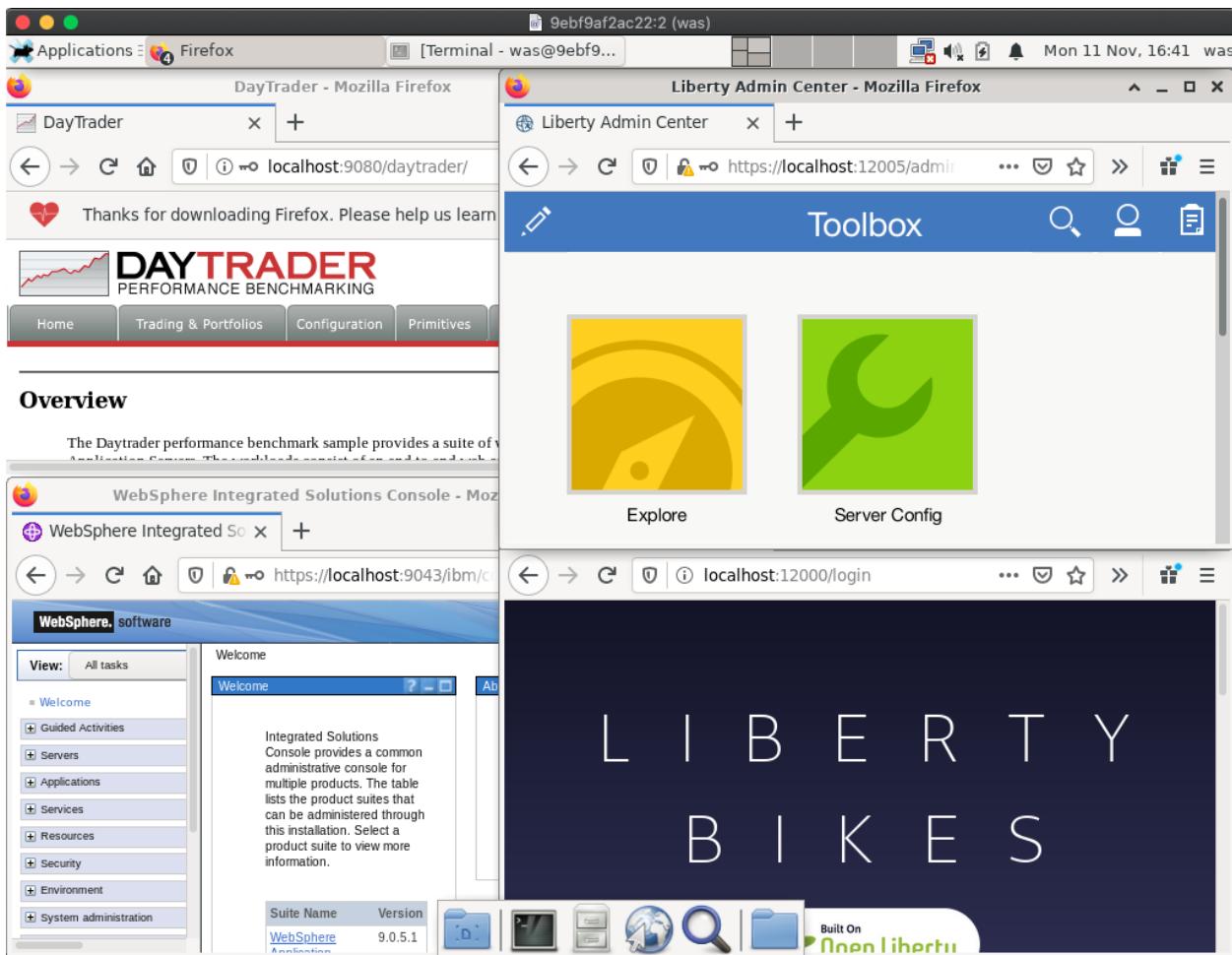
IBM® WebSphere® Application Server (WAS) is a platform for serving Java™-based applications. WAS comes in two major product forms:

1. WAS traditional (colloquially, tWAS): Released in 1998 and still fully supported and used by many.
2. WebSphere Liberty: Released in 2012 and designed for fast startup, composability, and the cloud. The commercial WebSphere Liberty product is built on top of the open source OpenLiberty. The colloquial term ‘Liberty’ may refer to WebSphere Liberty, OpenLiberty, or both.

WAS traditional and Liberty share some source code but differ in significant ways.

Lab Screenshots





Lab

What's in the lab?

This lab covers the major tools and techniques for performance tuning WebSphere Liberty. This is a subset of the WebSphere Performance and Troubleshooting Lab which also covers WAS traditional, and troubleshooting labs.

This lab container image comes with WebSphere Liberty pre-installed so installation and configuration steps are skipped.

The way we are using these container images is to run multiple services in the same container (e.g. VNC, Remote Desktop, WebSphere Liberty, a full GUI server, etc.) and although this approach is valid and supported, it is generally not recommended for real-world application deployment usage. For labs that demonstrate how to use WebSphere in containers in production, see WebSphere Application Server and Docker Tutorials.

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 may vary significantly and are covered elsewhere.

Java

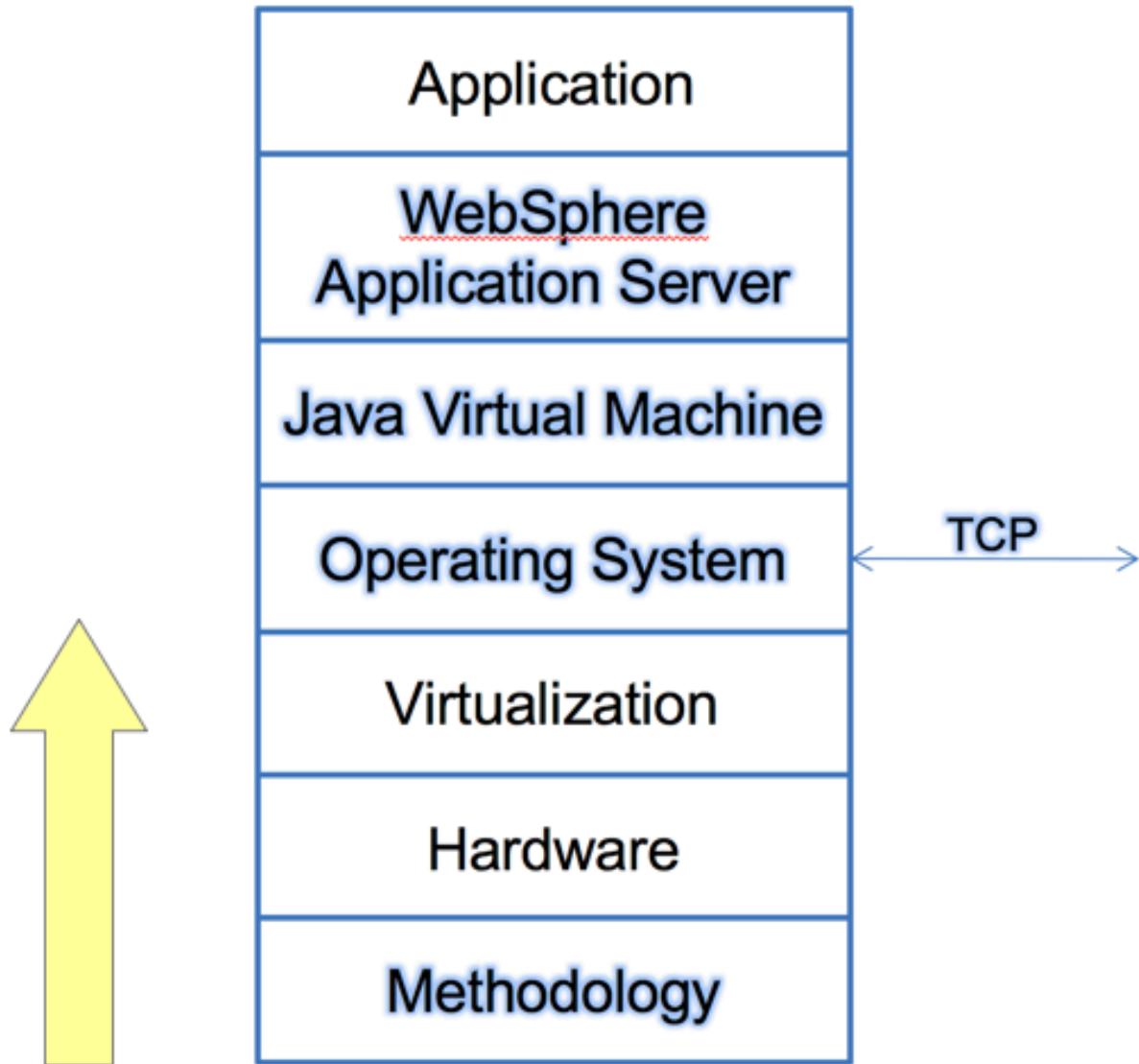
WebSphere Liberty supports any Java 8, 11, or 17 edition.

This lab uses IBM Java 8. The concepts and techniques apply generally to other Java runtimes although details of other Java runtimes (e.g. HotSpot) vary significantly and are covered elsewhere.

The IBM Java virtual machine (named J9) has become largely open sourced into the OpenJ9 project. OpenJ9 ships with OpenJDK through the IBM Semeru offering. OpenJDK is somewhat different than the JDK that IBM Java uses. WebSphere Liberty supports running with newer versions of OpenJDK+OpenJ9, although some IBM Java tooling such as HealthCenter is not yet available in OpenJ9, so the focus of this lab continues to be IBM Java 8.

Core Concepts

Performance tuning is best done with all layers of the stack in mind. This lab will focus on the layers in bold below:



Lab environment

Installation

This lab assumes the installation and use of `podman` or Docker Desktop to run the lab. Choose one or the other.

The image is about 20GB. If you plan to run this in a classroom setting, perform the installation steps beforehand which includes downloading the image.

Installing podman

If you are using `podman` instead of Docker Desktop, perform the following steps to install `podman` and then perform the `podman` post-installation steps. If you are using Docker Desktop, skip down to Installing Docker Desktop.

`podman` installation instructions:

- Windows: <https://podman.io/getting-started/installation#windows>
- macOS: <https://podman.io/getting-started/installation#macos>
- For a Linux host, simply install `podman`

podman post-installation steps

1. On macOS and Windows:

1. Create the `podman` virtual machine with sufficient memory (at least 4GB and, ideally, at least 8GB), CPU, and disk. For example (memory is in MB): `podman machine init --memory 10240 --cpus 4 --disk-size 100` If you already have a `podman` machine and you'd like to resize it to make it bigger, it's usually simplest to just delete it using `podman machine rm` and then re-create it using the above command.
2. Start the `podman` virtual machine: `podman machine start`
3. Switch to a “root” `podman`: `podman system connection default podman-machine-default-root`
4. Run the following commands to allow producing core dumps within the container: `podman machine ssh ln -sf /dev/null /etc/sysctl.d/50-coredump.conf` `podman machine ssh sysctl -w kernel.core_pattern=core`

2. Download the image:

```
podman pull quay.io/kgibm/fedorawasdebug
```

This command may not show any output for a long time while the download completes.

The following section on Docker Desktop should be skipped since you are using `podman`. The next section for `podman` is Start with `podman`.

Installing Docker Desktop

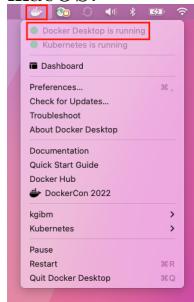
If you are using Docker Desktop instead of `podman`, perform the following steps to install Docker Desktop and then perform the Docker Desktop post-installation steps:

- 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/desktop/windows/install/>
- macOS (“must be version 10.15 or newer”)
 - Download: <https://hub.docker.com/editions/community/docker-ce-desktop-mac>
 - For details, see <https://docs.docker.com/desktop/mac/install/>
- For a Linux host, simply install and start Docker (e.g. `sudo systemctl start docker`):
 - For an example, see <https://docs.docker.com/engine/install/fedora/>

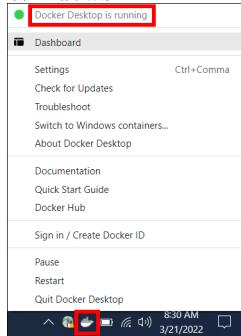
Docker Desktop post-installation steps

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

macOS:



Windows:

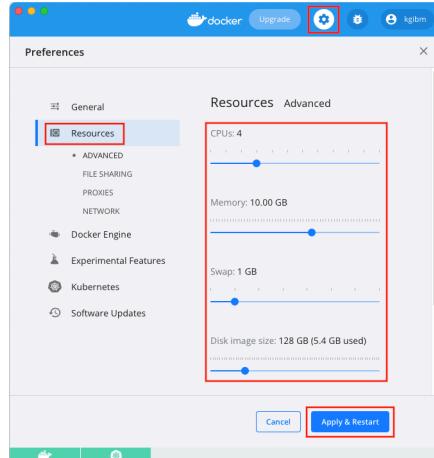


2. Ensure that Docker receives sufficient resources, particularly memory (at least 4GB and, ideally, at least 8GB), CPU, and disk:

1. macOS:

1. Click the Docker Desktop icon and select **Dashboard**
2. Click the **Settings** gear icon in the top right, then click **Resources** on the left.
3. Configure sufficient memory (at least 4GB and, ideally, at least 8GB), CPU, and disk.
4. Click **Apply & Restart**

macOS:



2. Windows uses the WSL2 backend which defaults to 50% of RAM or 8GB, whichever is less, and the same number of CPUs as the host. This may be overridden with a %UserProfile%\wslconfig file with, for example:

```
[ws12]
memory=10GB
processors=4
```

3. Open a terminal or command prompt and download the image: `docker pull quay.io/kgibm/fedorawasdebug`

The following section on `podman` should be skipped since you are using Docker Desktop. The next section for Docker Desktop is Start with Docker Desktop.

Start the container

Start with `podman`

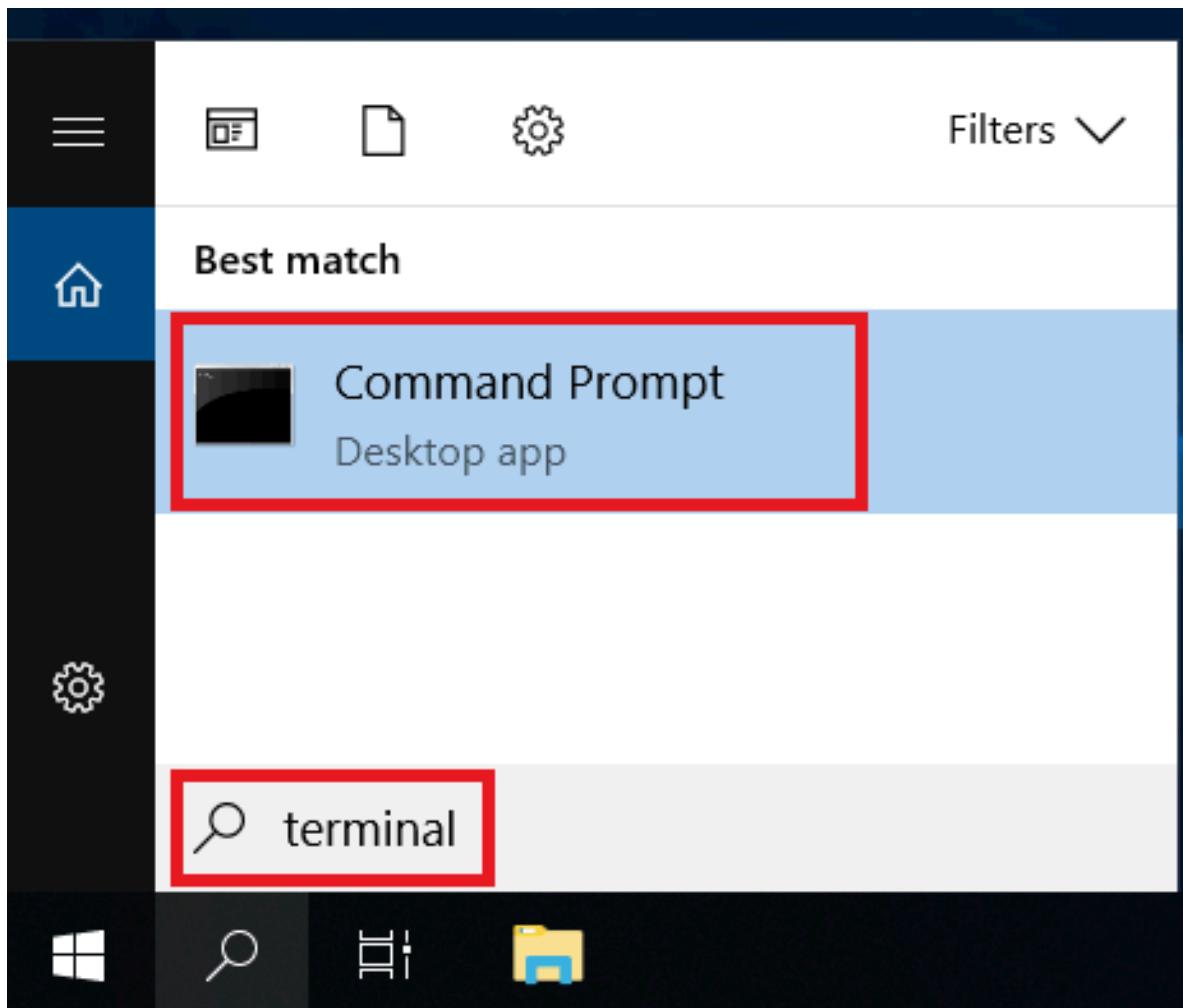
The following section is the start of the lab. If you were only preparing for the lab by installing and downloading the lab before the lab begins, then you may stop at this point until the instructor provides further direction.

If you are using `podman` for this lab instead of Docker Desktop, then perform the following steps. If you are using Docker Desktop, skip down to Start with Docker Desktop.

1. Open a terminal or command prompt:



Windows:



2. Start the lab:

```
podman run --cap-add SYS_PTRACE --cap-add NET_ADMIN --ulimit core=-1 --ulimit memlock=-1  
--ulimit stack=-1 --shm-size="256m" --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 9082:9082 -p  
9083:9083 -p 9445:9445 -p 8080:8080 -p 8081:8081 -p 8082:8082 -p 12000:12000 -p  
12005:12005 -it quay.io/kgibm/fedorawasdebug
```

3. Wait about 2 minutes until you see the following in the output (if not seen, review any errors):

```
=====  
= READY =  
=====
```

4. VNC or Remote Desktop into the container:

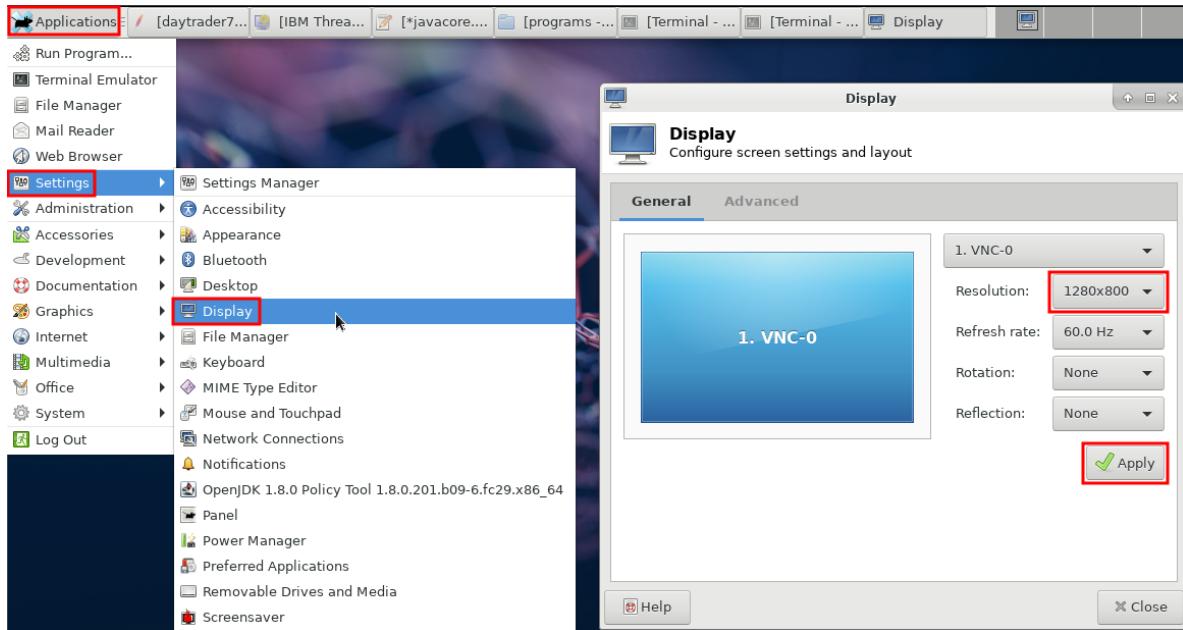
1. macOS built-in VNC client:

1. Open another tab in the terminal and run:
 1. **open vnc://localhost:5902**
 2. Password: **websphere**

2. Linux VNC client:

1. Open another tab in the terminal and run:

1. **vncviewer localhost:5902**
 - You may need to install `vncviewer` first.
2. Password: **websphere**
3. Windows 3rd party VNC client:
 - i. If you are able to install and use a 3rd party VNC client (there are a few free options online), then connect to **localhost** on port **5902** with password **websphere**.
4. 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.
5. When using VNC, you may change the display resolution from within the container and the VNC client will automatically adapt. For example:



The following section on Docker Desktop should be skipped since you are using `podman`. The next section for `podman` is Apache JMeter.

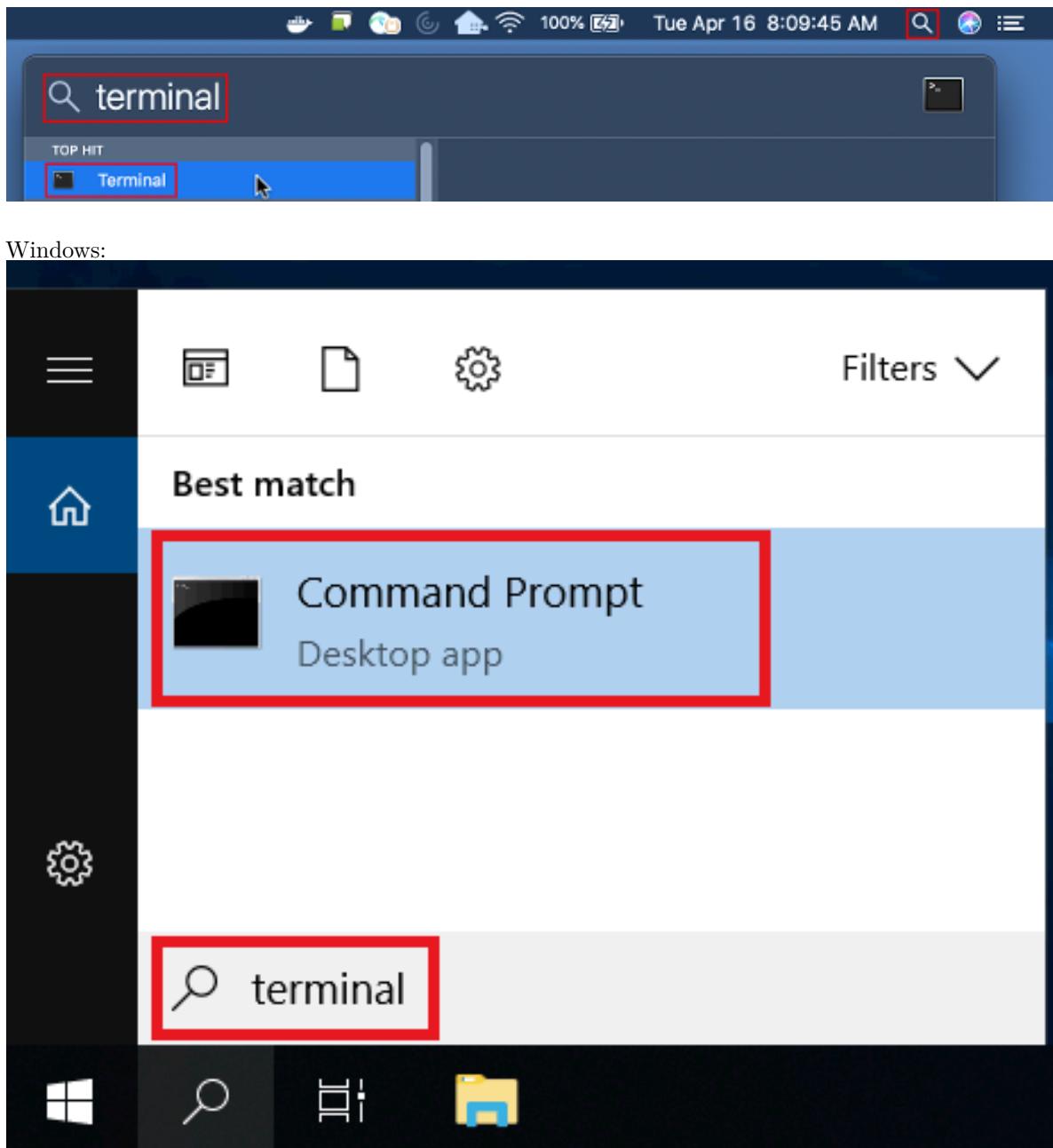
Start with Docker Desktop

The following section is the start of the lab. If you were only preparing for the lab by installing and downloading the lab before the lab begins, then you may stop at this point until the instructor provides further direction.

If you are using Docker Desktop for this lab instead of `podman`:

1. Open a terminal or command prompt:

macOS:



2. Start the lab by starting the Docker container from the command line:

```
docker run --cap-add SYS_PTRACE --cap-add NET_ADMIN --ulimit core=-1 --ulimit memlock=-1  
--ulimit stack=-1 --shm-size="256m" --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 9083:9083 -p 9445:9445 -p 8080:8080 -p 8081:8081 -p 8082:8082 -p 12000:12000 -p  
12005:12005 -it quay.io/kgibm/fedorawasdebug
```

3. Wait about 2 minutes until you see the following in the output (if not seen, review any errors):

```
=====  
= READY =  
=====
```

4. VNC or Remote Desktop into the container:

1. macOS built-in VNC client:

1. Open another tab in the terminal and run:

1. **open vnc://localhost:5902**

2. Password: **websphere**

2. Linux VNC client:

1. Open another tab in the terminal and run:

1. **vncviewer localhost:5902**

• You may need to install **vncviewer** first.

2. Password: **websphere**

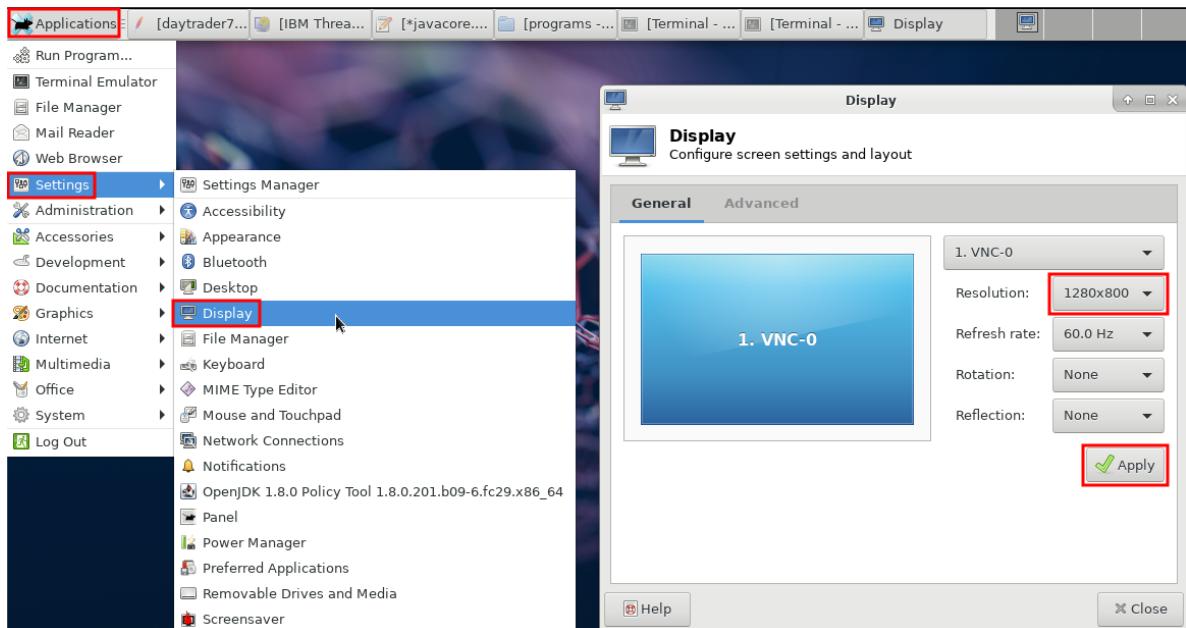
3. Windows 3rd party VNC client:

i. If you are able to install and use a 3rd party VNC client (there are a few free options online), then connect to **localhost** on port **5902** with password **websphere**.

4. 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.

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

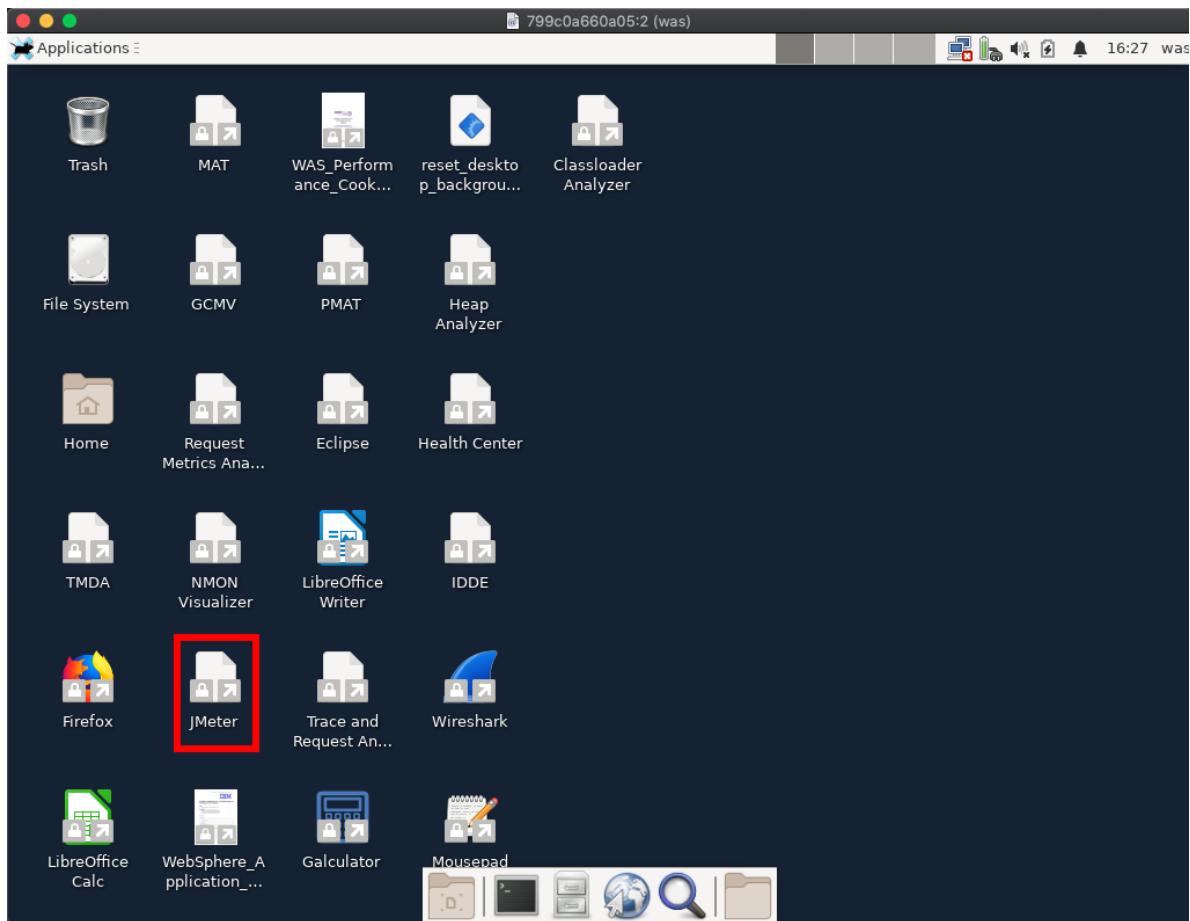


Apache Jmeter

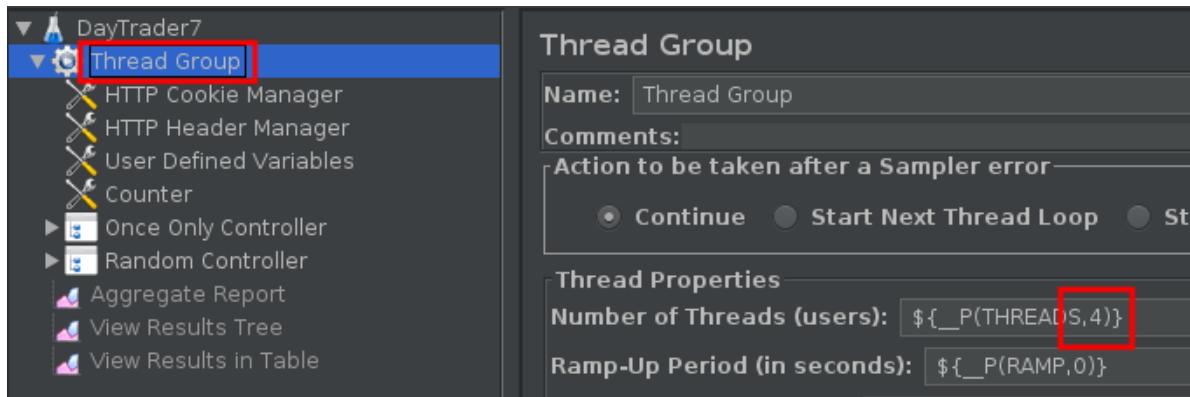
Apache JMeter is a free tool that drives artificial, concurrent user load on a website. The tool is pre-installed in the lab image and we'll be using it to simulate website traffic to the DayTrader7 sample application pre-installed in the lab image.

Start JMeter

1. Double click on JMeter on the desktop:



2. Click **File → Open** and select:
 1. `/opt/daytrader7/jmeter_files/daytrader7.liberty.jmx`
3. 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):



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

Aggregate Report

Label	# Sam...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throu...	Receive...	Sent K...
TOTAL	25517	8	3	18	31	76	0	1077	0.00%	463.8/...	4298.81	194.61

5. It will take some time for the responses to start coming back and for all of the pages to be exercised.
6. Ensure that the **Error %** value for the **TOTAL** row at the bottom is always 0%.

Aggregate Report

Label	# Sam...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throu...	Receive...	Sent K...
TOTAL	25517	8	3	18	31	76	0	1077	0.00%	463.8/...	4298.81	194.61

1. If there are any errors, review the WAS logs:

1. `/logs/messages.log`

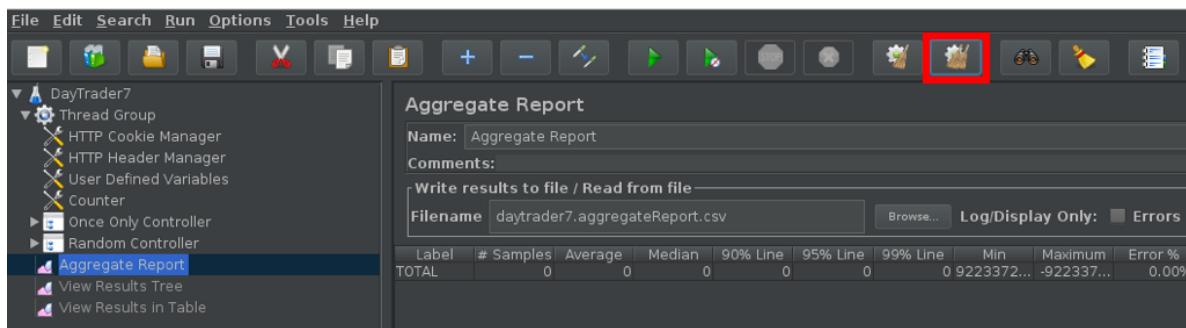
Stop JMeter

1. You may stop a JMeter test by clicking the STOP button:

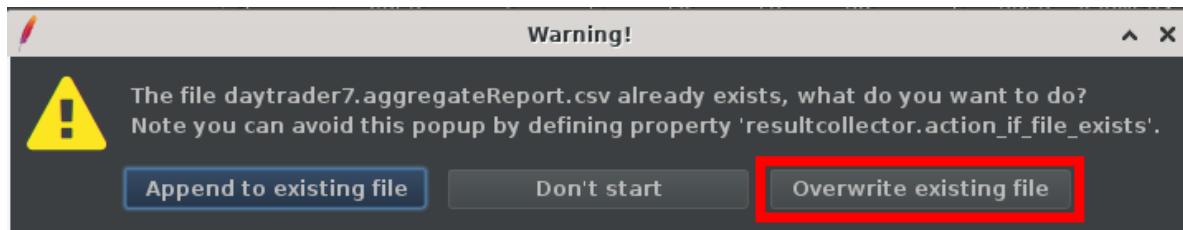
Aggregate Report

Label	# Sam...	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throu...	Receive...	Sent K...
TOTAL	25517	8	3	18	31	76	0	1077	0.00%	463.8/...	4298.81	194.61

2. You may click the broom button to clear the results in preparation for the next test:

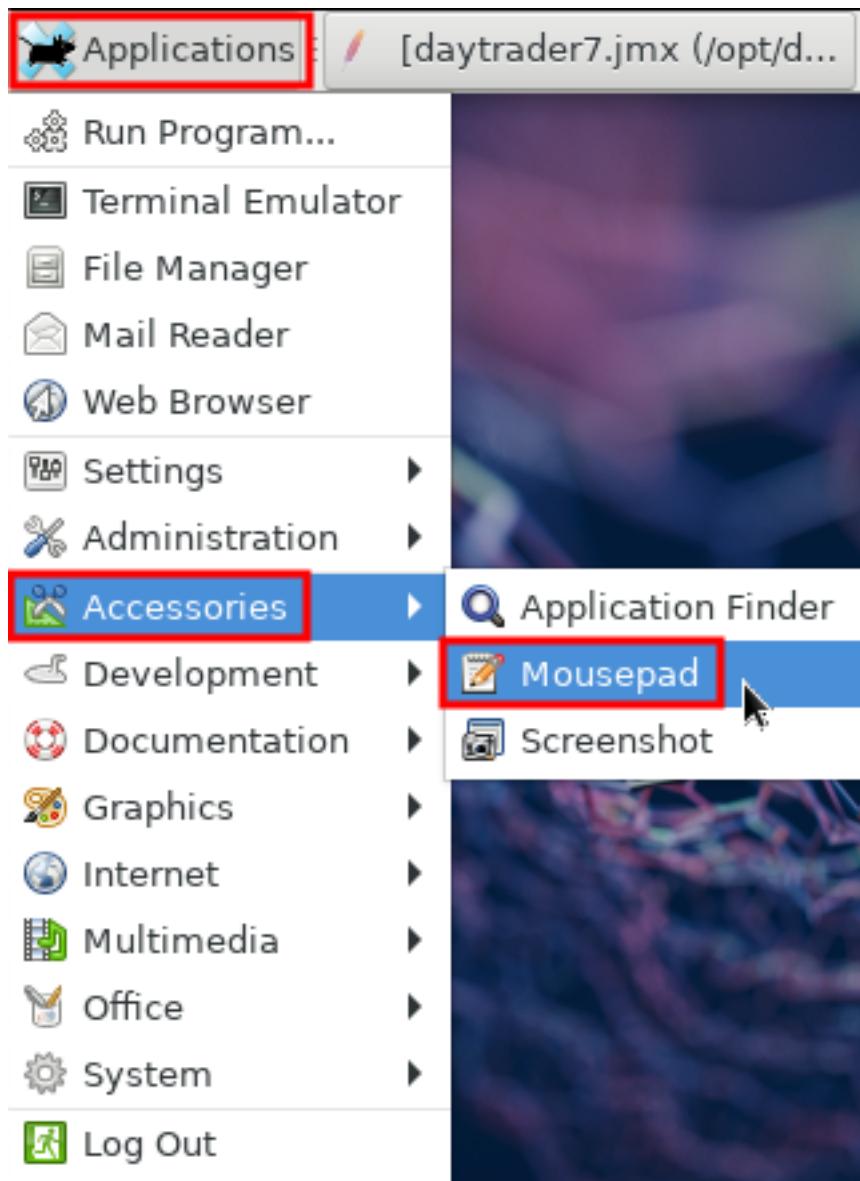


3. If it asks what to do with the JMeter log files from the previous test, you may just click **Overwrite existing file**:



Mousepad

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



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*.txt` or `javadump`. HotSpot-based thread dumps are covered elsewhere.

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

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.
- 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.

linperf.sh

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” (similar scripts exist for other operating systems). This script will be used to gather thread dumps in this lab. Such a script should be pre-installed on all machines where you run Liberty and it should be run when you have performance or hang issues and the resulting files should be uploaded if you open such a support case with IBM.

The **linperf.sh** script is pre-installed in the lab image at **/opt/linperf/linperf.sh**.

Thread Dumps Lab

We will gather and review thread dumps:

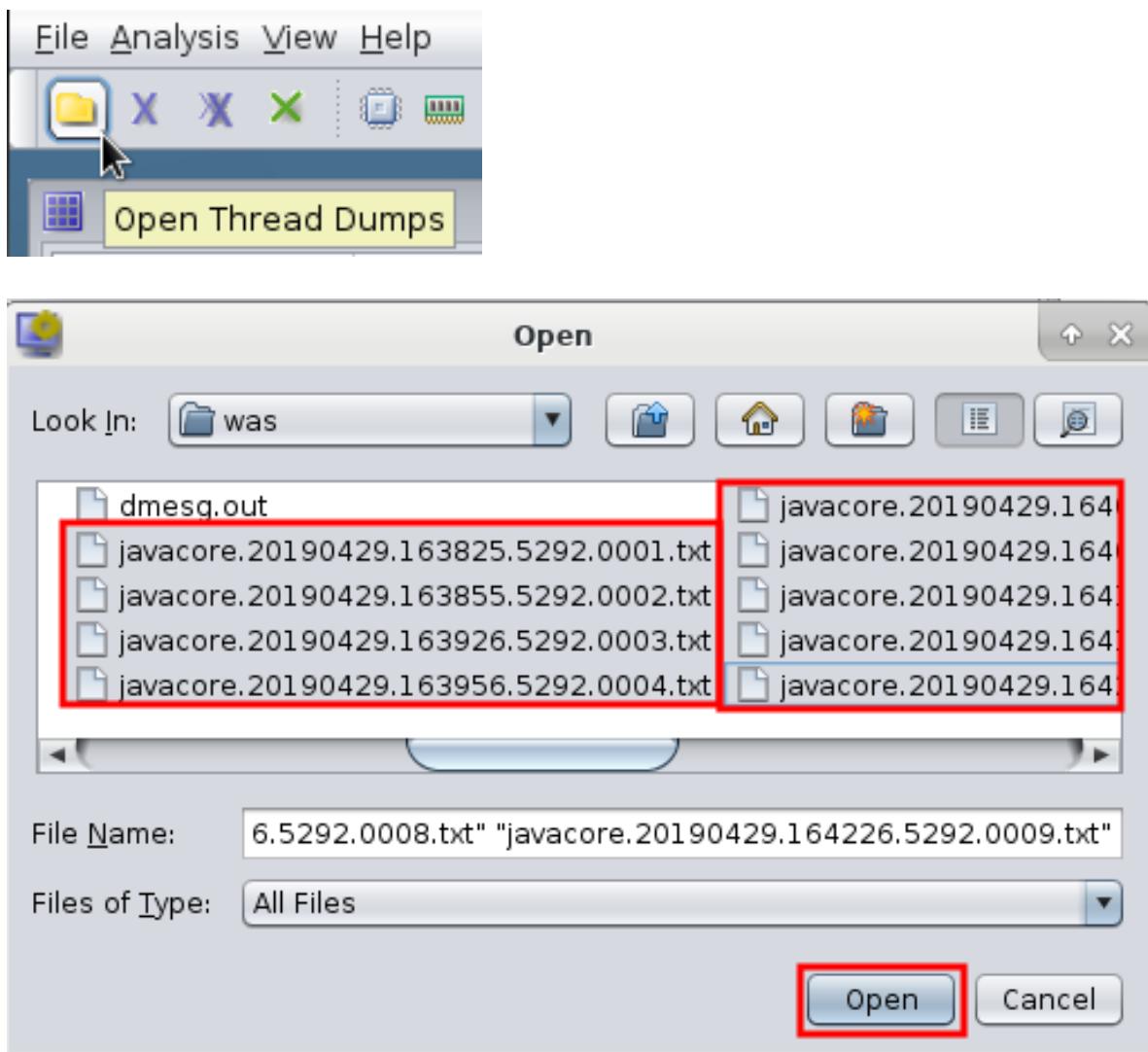
Note: You may skip the data collection steps and use example data packaged at **/opt/dockerdebug/fedorawasdebug/support**

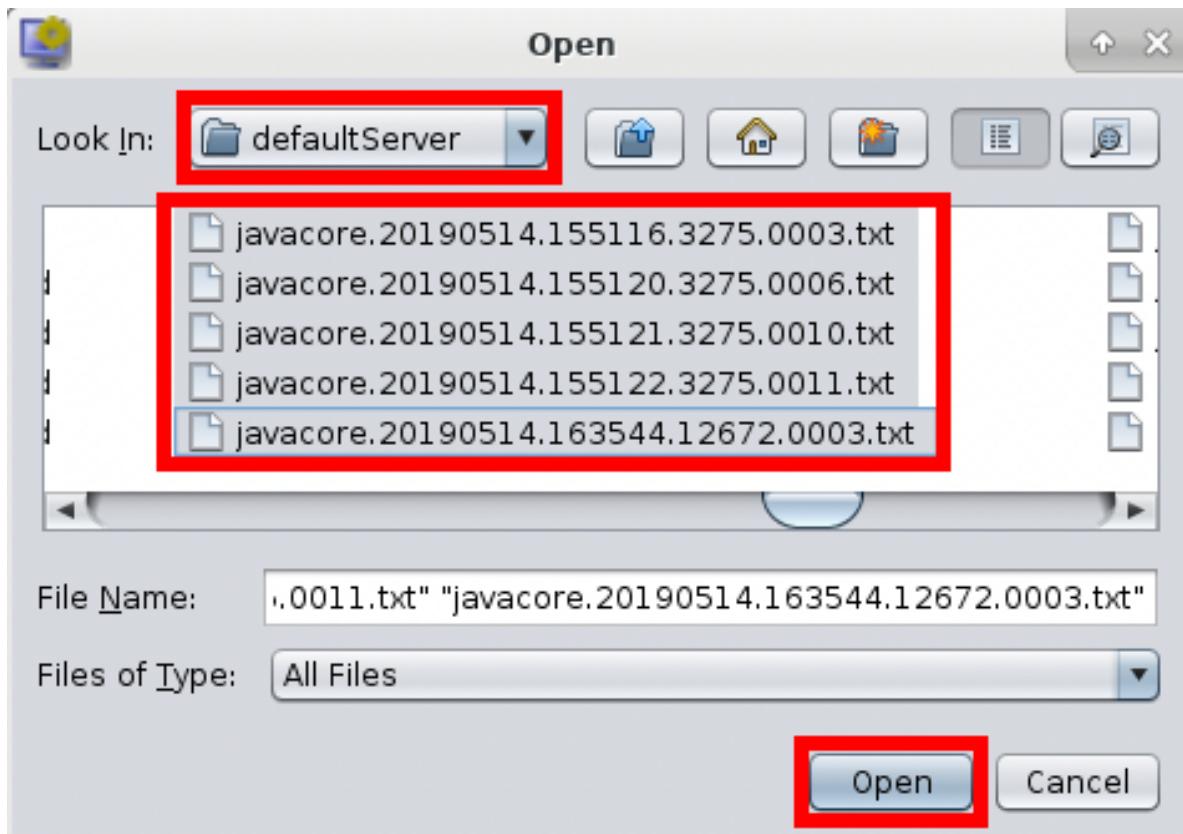
1. Start JMeter
2. Open a terminal on the lab image.
3. First, we'll need to find the PID(s) of Liberty. There are a few ways to do this, and you only need to choose one method:
 1. Show all processes (**ps -elf**), search for the process using something unique in its command line (**grep defaultServer**), exclude the search command itself (**grep -v grep**), and then select the fourth column (in bold below):
 2. Search for the process using something unique in its command line using **pgrep -f**:
4. Execute the **linperf.sh** command and pass the PID gathered above (replace 1567 with your PID from the output above):


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


```
mv /opt/ibm/wlp/output/defaultServer/javacore.* .
```
7. Stop JMeter
8. At this point, if you were creating a support case, you would upload **linperf_RESULTS.tar.gz**, **javacore***, and all the Liberty logs; however, instead, we will analyze the results.
9. From the desktop, double click on **TMDA**.

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



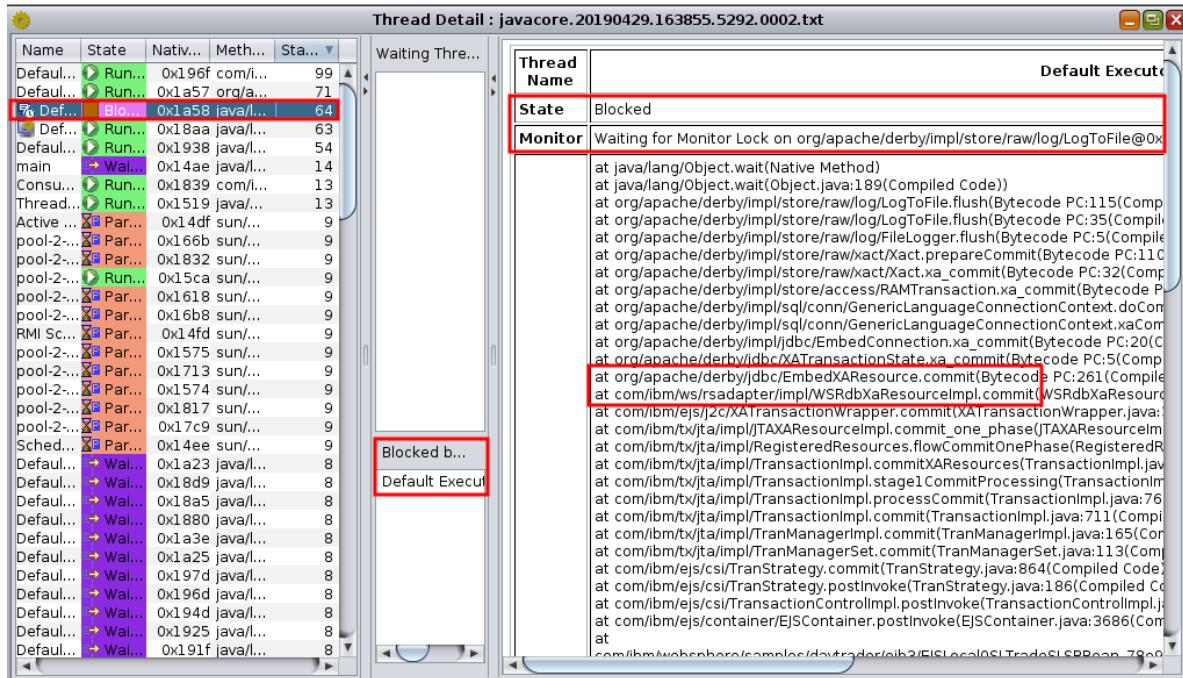


11. Select a thread dump and click the **Thread Detail** button:

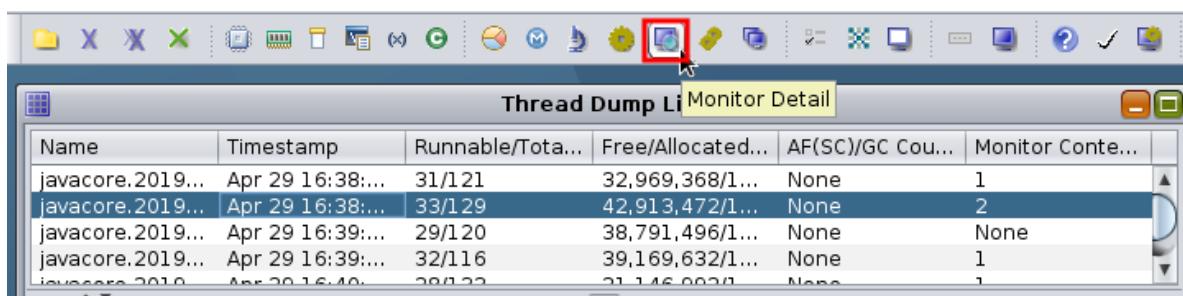
Name	Timestamp	Runnable/Tota...	Free/Allocated...	AF(SC)/GC Cou...	Monitor Conte...
javacore.2019...	Apr 29 16:38:...	31/121	32,969,368/1...	None	1
javacore.2019...	Apr 29 16:38:...	33/129	42,913,472/1...	None	2
javacore.2019...	Apr 29 16:39:...	29/120	38,791,496/1...	None	None
javacore.2019...	Apr 29 16:39:...	32/116	39,169,632/1...	None	1
javacore.2019...	Apr 29 16:40:...	28/122	21,146,002/1...	None	1

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

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

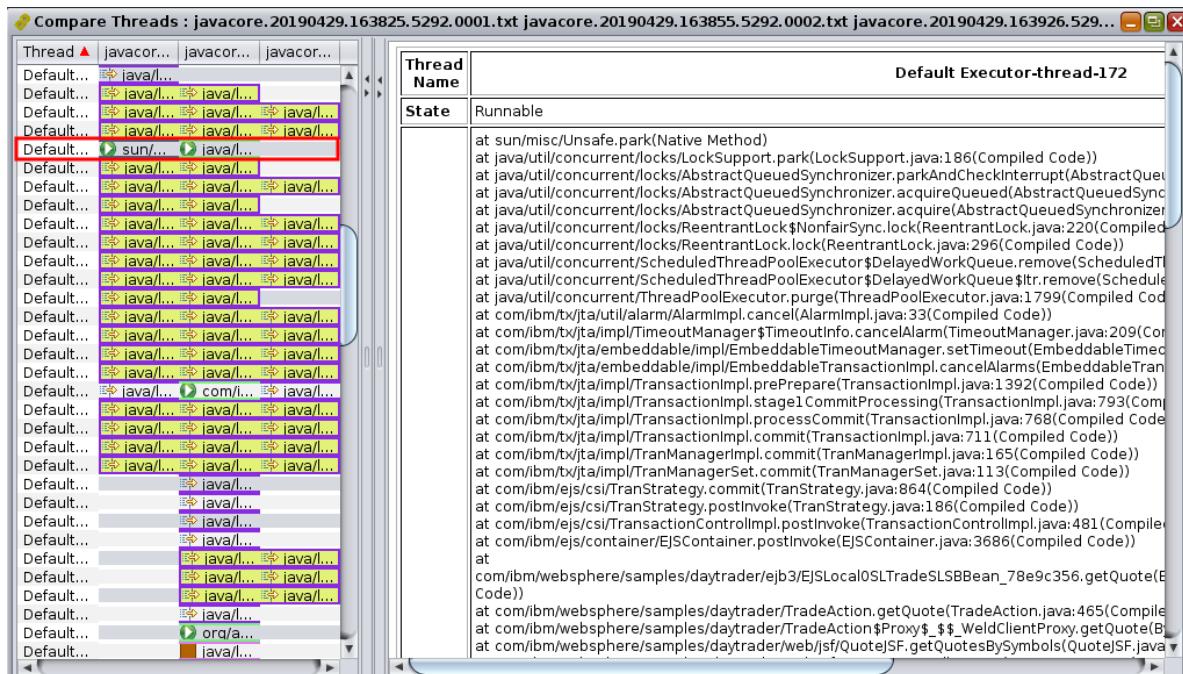
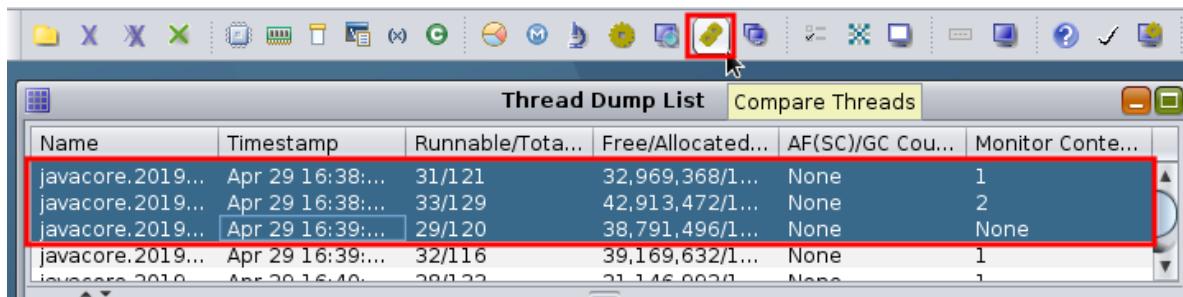


1. The **Monitor** line shows which monitor this thread is waiting for, and the stack shows the path to the request for the monitor. In this example, the application is trying to commit a database transaction. This lab uses the Apache Derby database engine which is not a very scalable database. In this 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.
16. An alternative way to review lock contention is by selecting a thread dump and clicking **Monitor Detail**:





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



- Each column is a thread dump and shows the state of each thread (if it exists in that thread dump) over time. Generally, you're interested in threads that are runnable (Green Arrow) or blocked 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:

Note: You may skip the data collection steps and use example data packaged at /opt/dockerdebug/fedorawasdebug/support/

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

Deadlocker (Dining Philosophers)	/Deadlocker	None	Attempt to create a deadlock with an algorithm that emulates the Dining Philosophers problem . You will know a deadlock has occurred if messages stop being written to the HTML output. To confirm if a deadlock has occurred, take a javadump and search for "deadlock." It is possible, based on CPU availability, OS timing, and thread dispatching that a true deadlock will not occur.
----------------------------------	-----------------------------	------	---

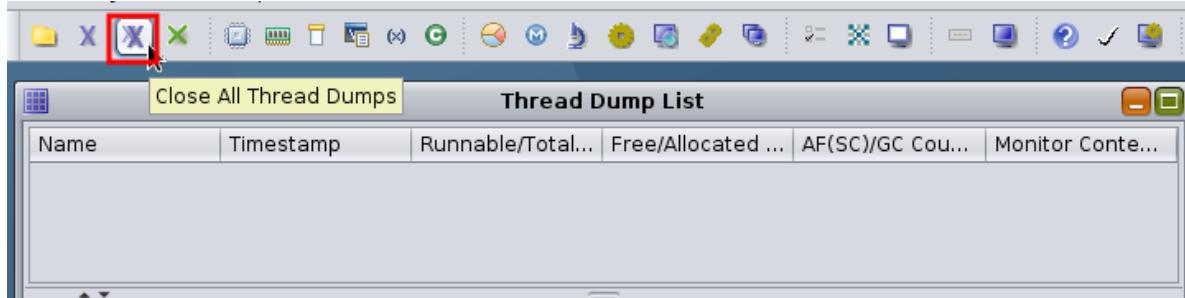
3. Wait until the continuous browser output stops writing new lines of "Socrates [...]" which signifies that the threads have become deadlocked and then gather a thread dump of the WAS process by sending it the **SIGQUIT (3)** signal. Although the name of the signal includes the word "QUIT", the signal 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)
```

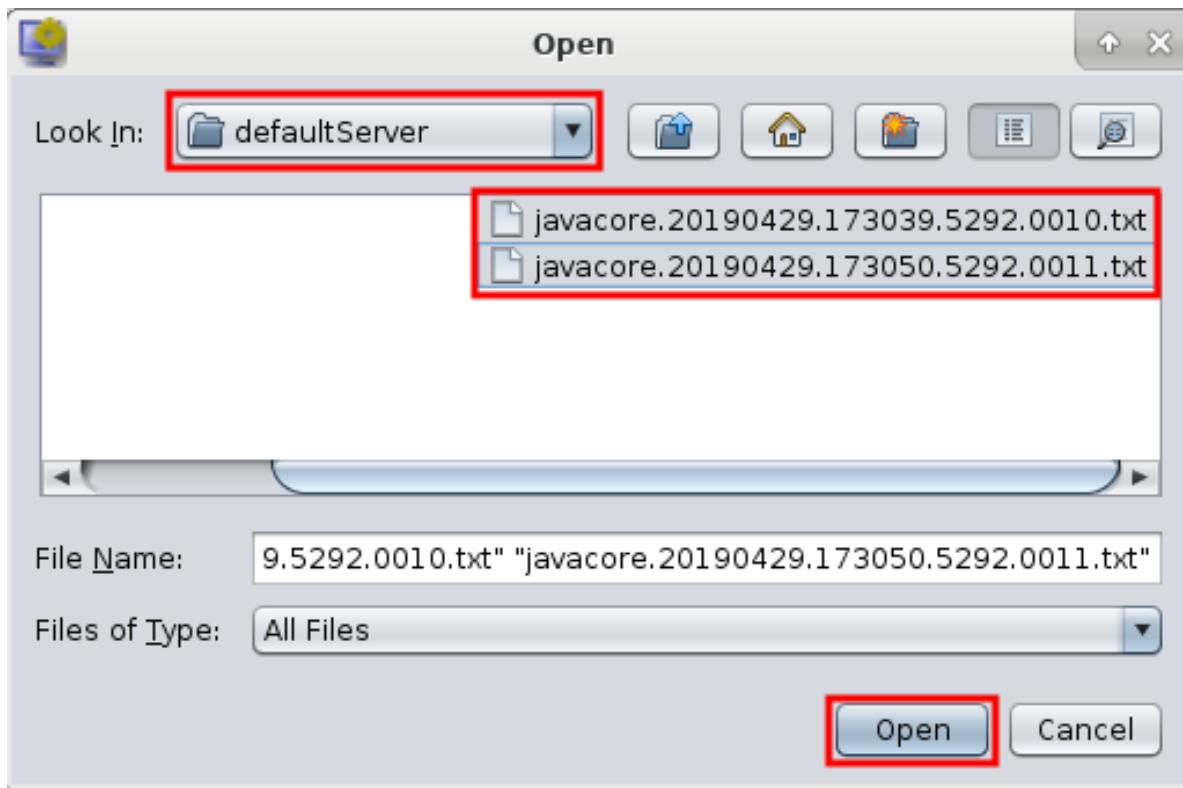
1. Note that here we are using a sub-shell to send the output of the pgrep command (which finds the PID of WAS) as the argument for the kill command.
2. This can be simplified even further with the **pkill** command which combines **pgrep** functionality:

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

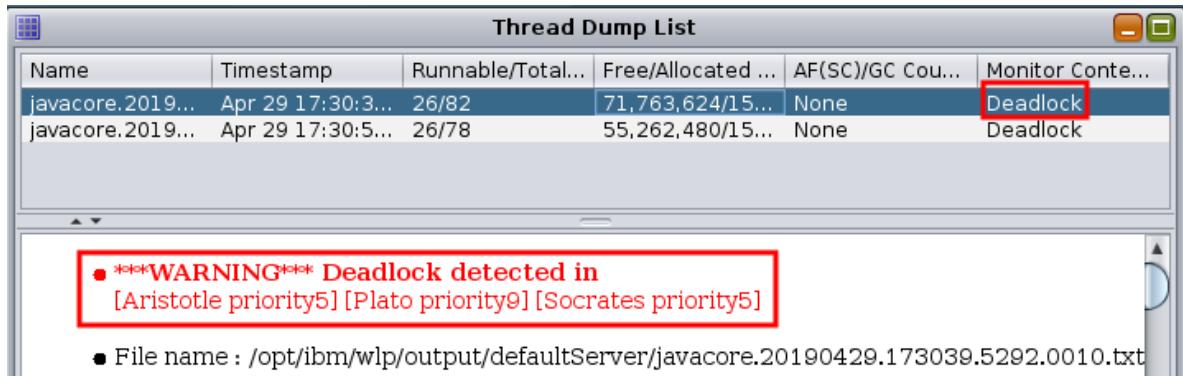
4. In the TMDA tool, clear the previous list of thread dumps:



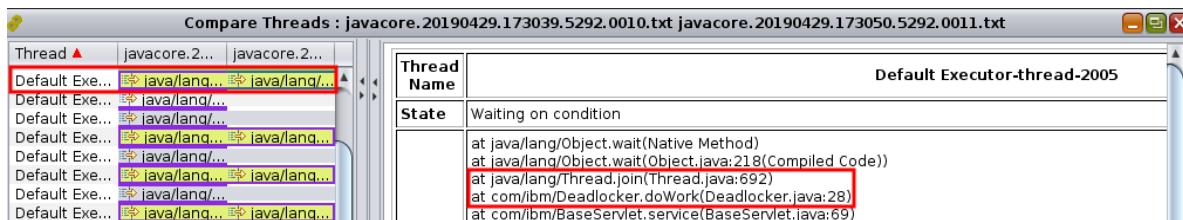
5. Click **File > Open Thread Dumps** and navigate to /opt/ibm/wlp/output/defaultServer and select both new thread dumps and click **Open**:



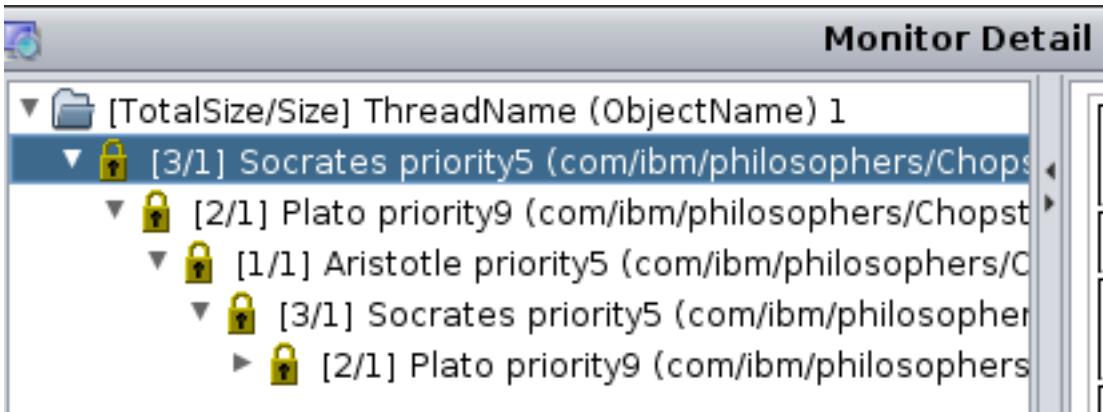
- When you select the first thread dump, TMDA will warn you that a deadlock has been detected:



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



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



Next, let's simulate a thread that is using a lot of CPU:

Note: You may skip the data collection steps and use example data packaged at /opt/dockerdebug/fedorawasdebug/support/

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

Infinite Loop Servlet	InfiniteLoop	• threshold: After threshold/2 iterations, break. Maximum value=2147483647	Simply loops infinitely on a servlet thread in a while loop with some trivial math operations. WARNING: You will lose this thread until the JVM is restarted and it will consume 100% of 1 CPU/core!
-----------------------	------------------------------	--	---

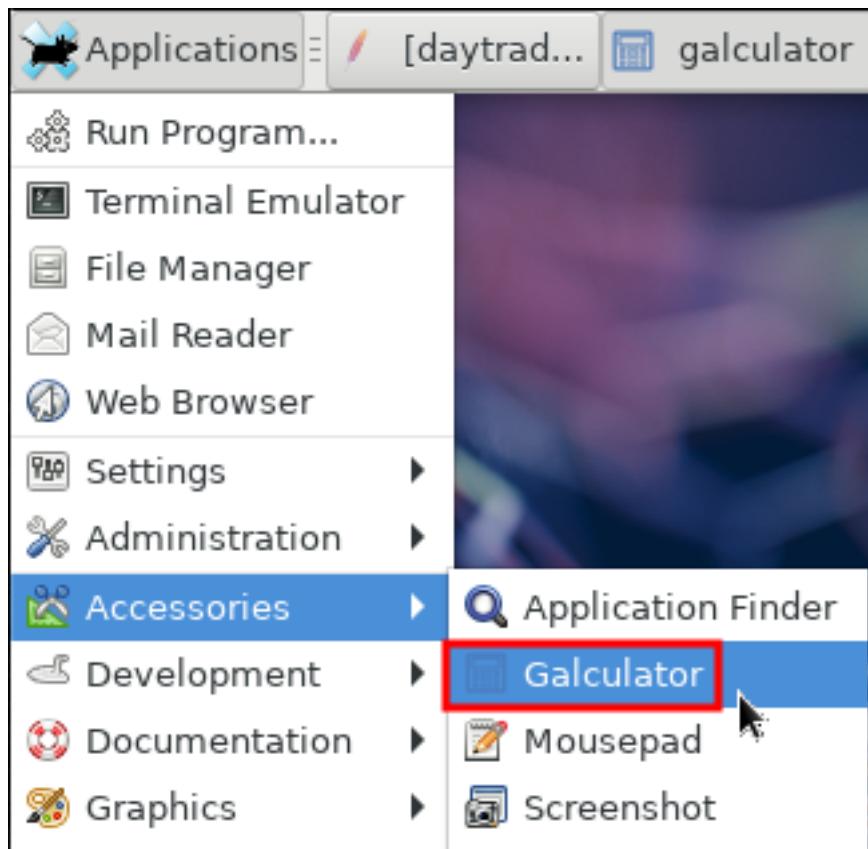
3. Go to the container terminal and start **top -H** with a 10 second interval:

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

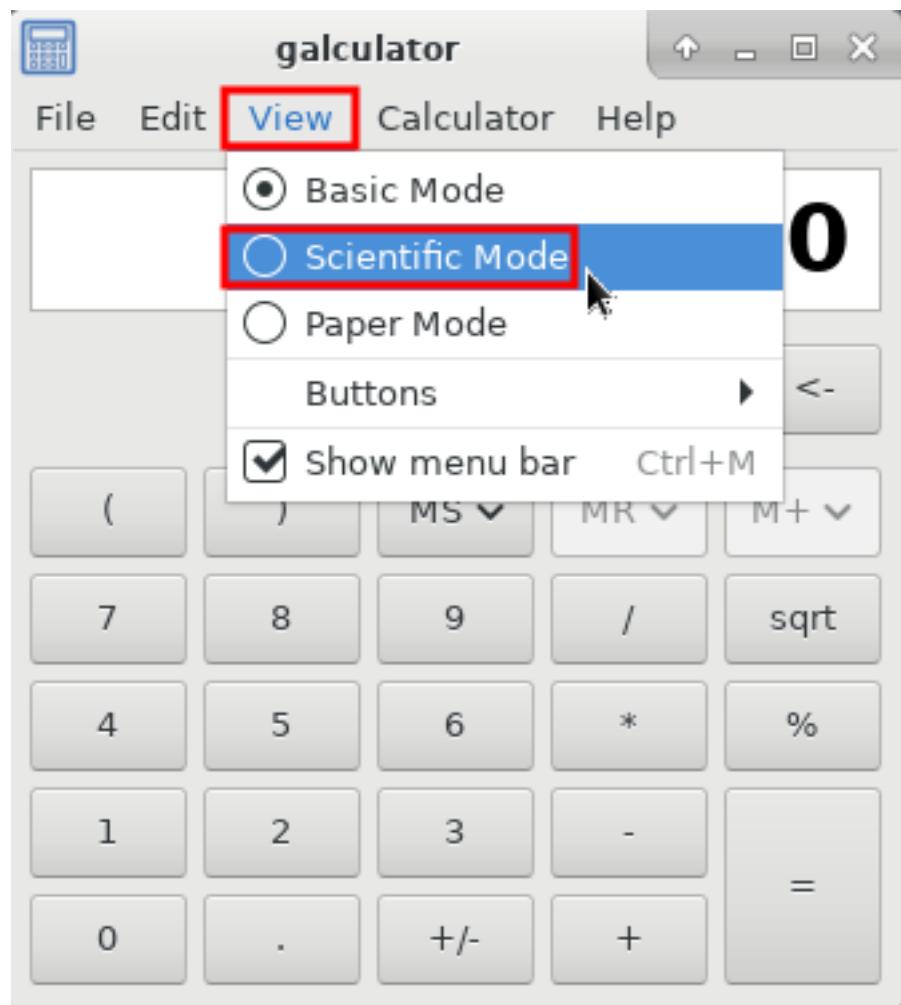
top - 17:48:13 up 2:34, 0 users, load average: 0.92, 0.42, 0.28											
Threads: 485 total, 2 running, 483 sleeping, 0 stopped, 0 zombie											
%Cpu(s): 25.6 us, 0.3 sy, 0.0 ni, 74.2 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st											
MiB Mem : 11993.4 total, 1454.5 free, 1774.4 used, 8764.5 buff/cache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

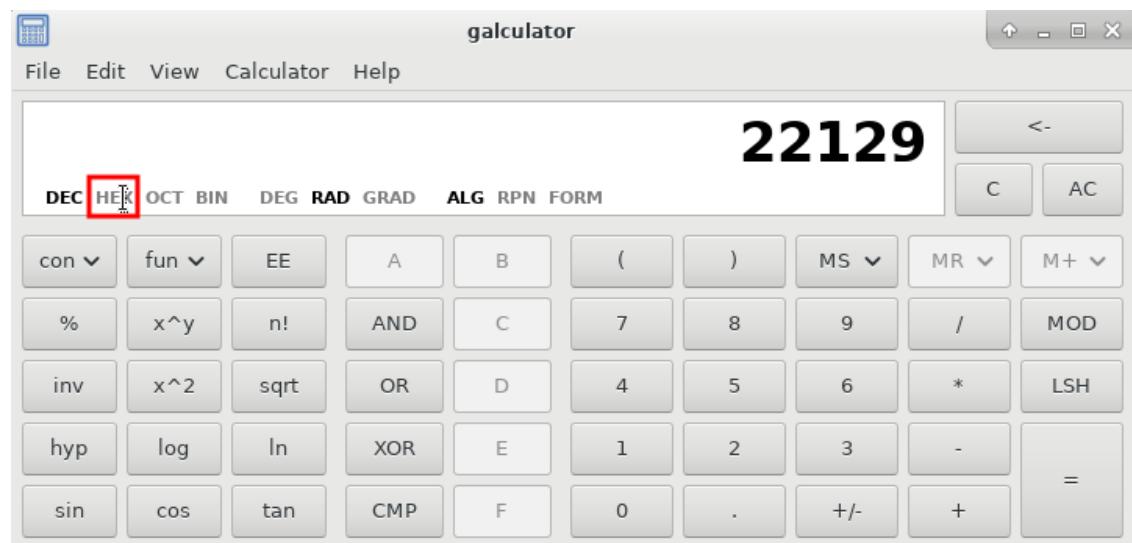
1. In the container, open Galculator:



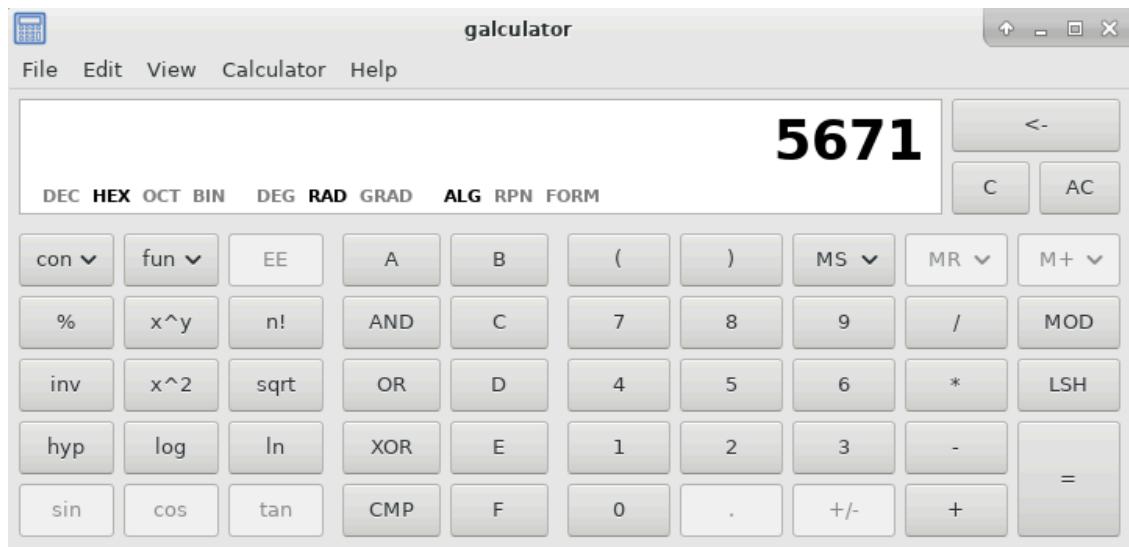
2. Click View > Scientific Mode:



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



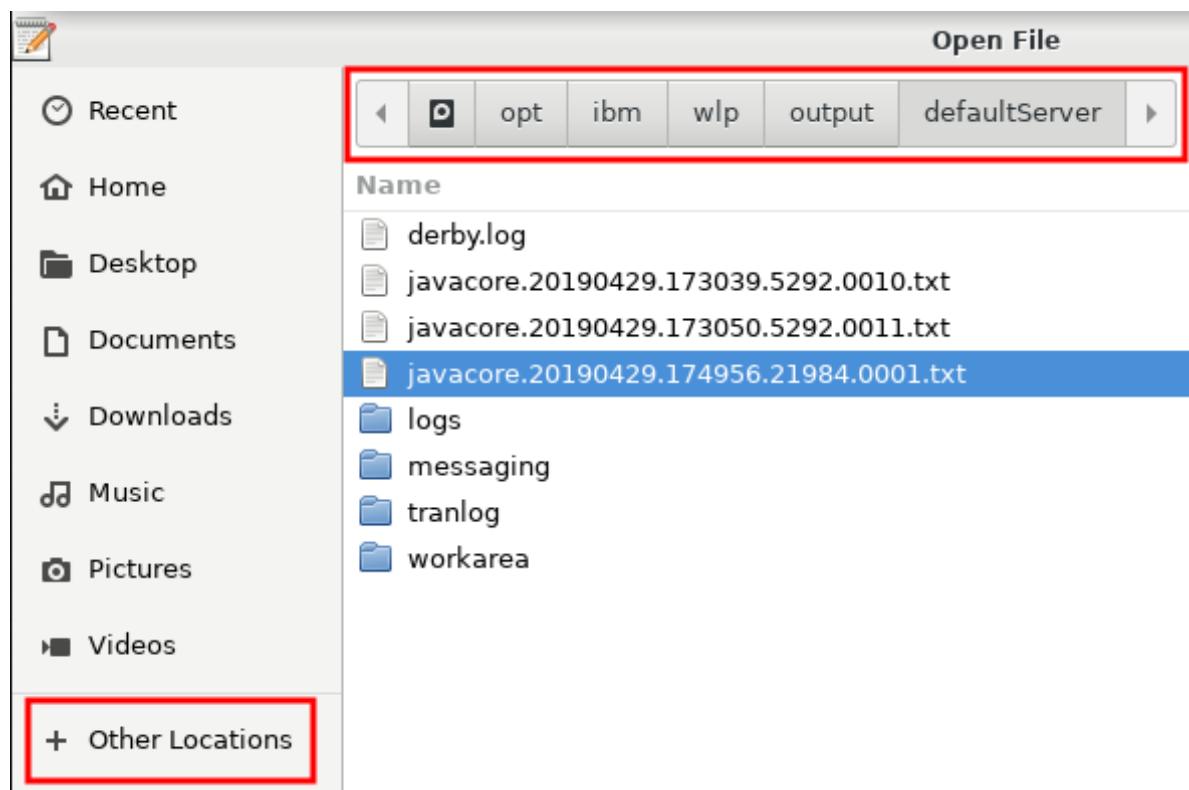
4. The result is **0x5671**:



- Take a thread dump of the parent process:

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

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



- 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:
3XMJAVATHREAD     (java/lang/Thread_getId:0x5C, isDaemon:true)
3XMTHREADINFO01   (native thread ID:0x5671, native priority:0x5, native policy:UNKNOWN, vmstate:CW, vm thread flags:0x00
3XMTHREADINFO02   (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)
3XMTHREADINFO03   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(HttpServletRequest.java:790)
4XESTACKTRACE    at com/ibm/ws/webcontainer/servlet/ServletWrapper.service(ServletWrapper.java:1255)

```

- Finally, kill the server destructively (**kill -9**) because trying to stop it gracefully will not work due to the infinitely looping request:

```
pkill -9 -f defaultServer
```

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%.

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.

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), using the options to rotate the verbosegc logs. For IBM Java - 5 historical files of roughly 20MB each:

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

Garbage Collection Lab

Add the verbosegc option to the jvm.options file:

Note: You may skip the data collection steps and use example data packaged at /opt/dockerdebug/fedorawasdebug/support

- Stop JMeter if it is started.
- WebSphere Liberty:

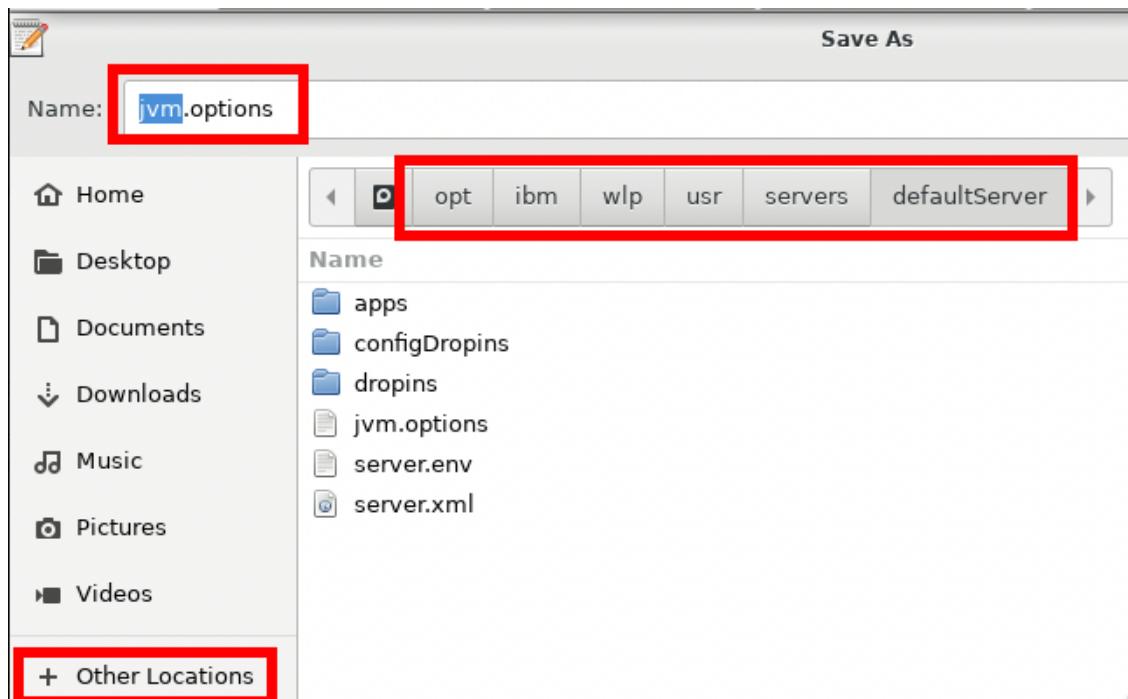
- Stop the Liberty server.

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

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

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

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



4. Start the Liberty server

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

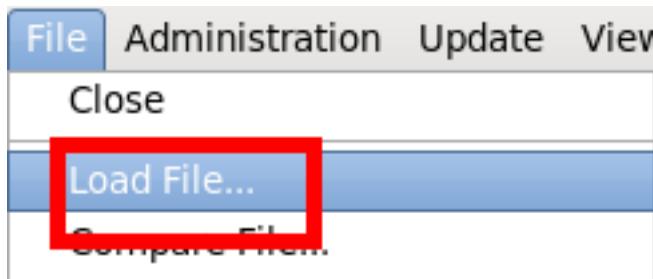
3. Start JMeter

4. Run the test for about 5 minutes.

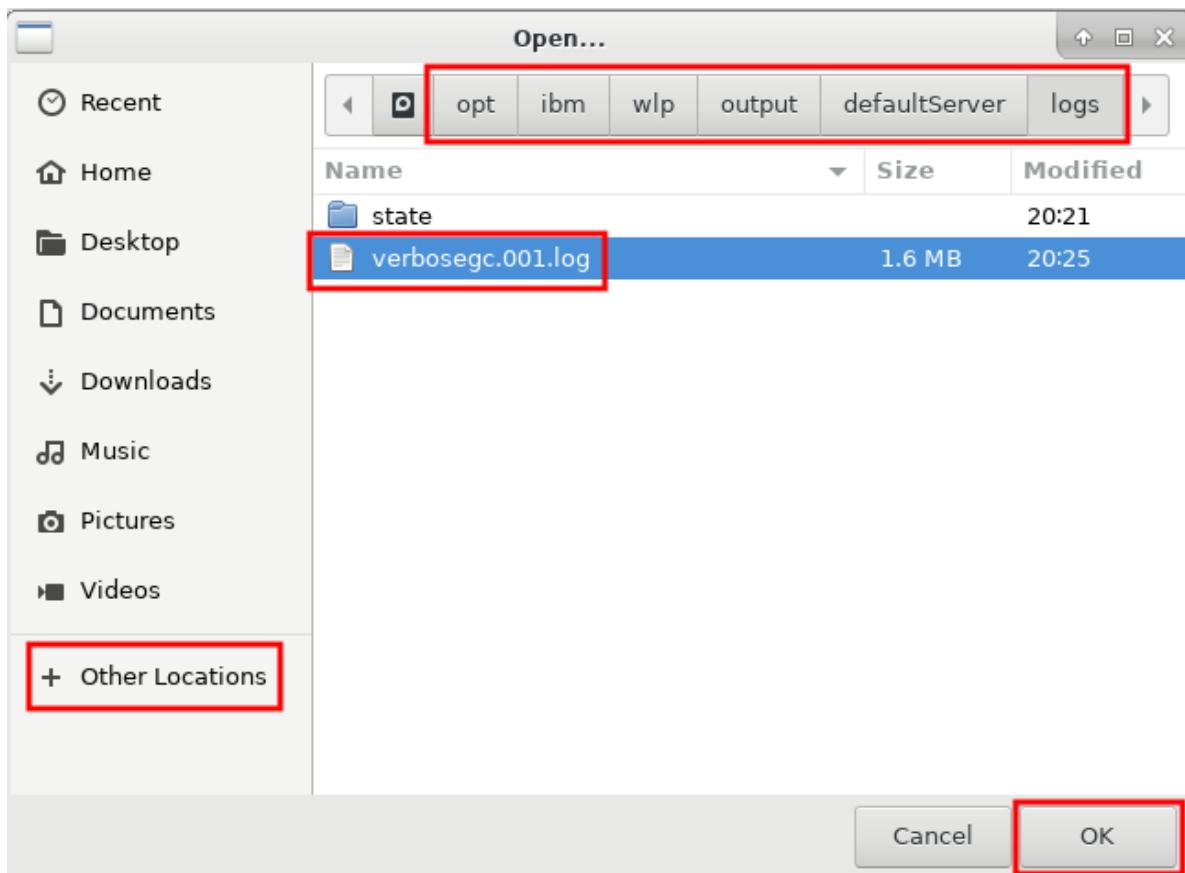
5. Stop JMeter

6. From the desktop, double click on **GCMV**:

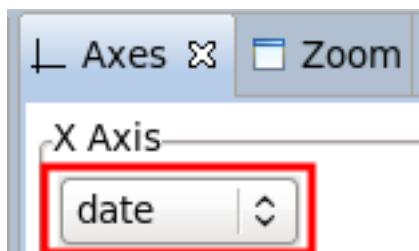
7. Click **File > Load File...** and select the **verbosegc.001.log** file. For example:



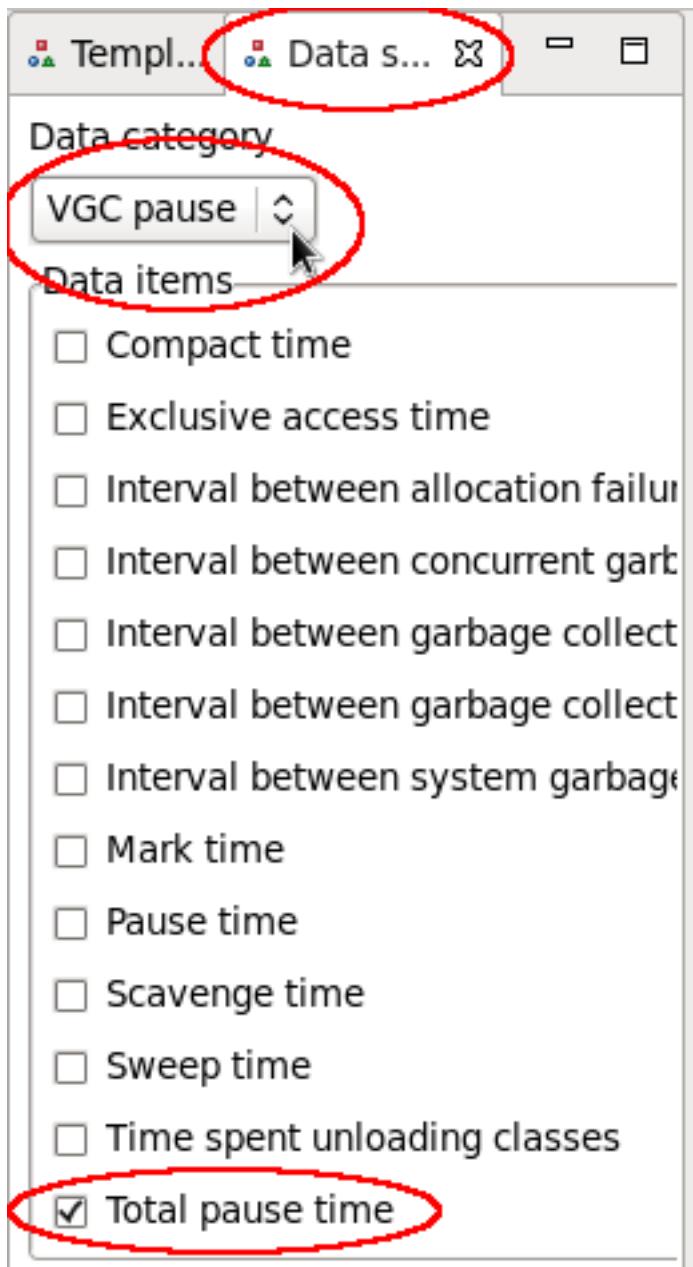
8. Select `/opt/ibm/wlp/output/defaultServer/logs/verbosegc.001.log`



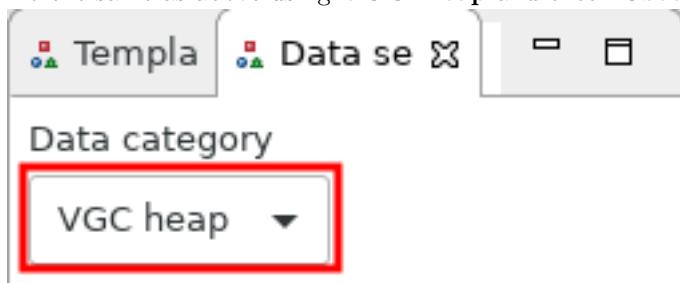
9. Once the file is loaded, you will see the default line plot view. It is common to change the **X-axis** to **date** to see absolute timestamps:

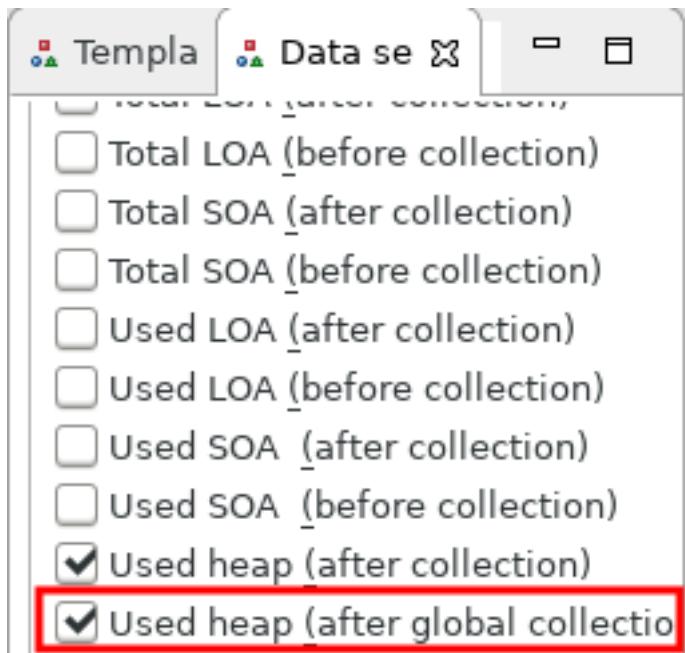


10. Click the **Data Selector** tab in the top left, choose **VGC Pause** and check **Total pause time** to add the total garbage collection pause time plot to the graph:

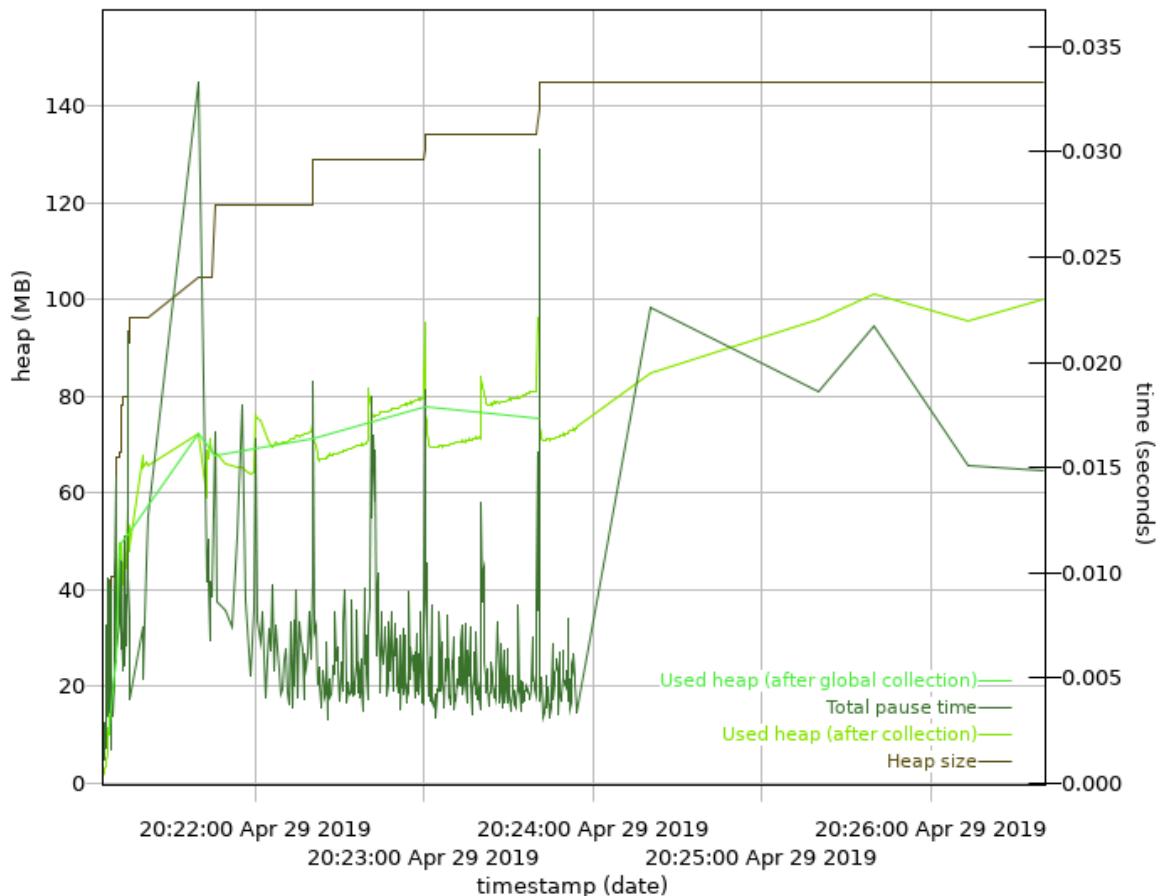


11. Do the same as above using **VGC Heap** and check **Used heap (after global collection)**:





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



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

~80MB.

13. 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 (%)**:

Data set 1	
Tuning recommendation	Summary
Summary	Concurrent collection count 12
Heap size	Forced collection count 0
Total pause time	GC Mode gencon
Used heap (after global collection)	Global collections - Mean garbage collection pause (ms) 14.0
Used heap (after collection)	Global collections - Mean interval between collections (ms) 10307
	Global collections - Number of collections 15
	Global collections - Total amount tenured (MB) 630
	Largest memory request (bytes) 131080
	Number of collections triggered by allocation failure 487
	Nursery collections - Mean garbage collection pause (ms) 4.61
	Nursery collections - Mean interval between collections (ms) 735
	Nursery collections - Number of collections 452
	Nursery collections - Total amount flipped (MB) 843
	Nursery collections - Total amount tenured (MB) 119
	Proportion of time spent in garbage collection pauses (%) 0.91
	Proportion of time spent unpause (%) 99.09
	Rate of garbage collection (MB/minutes) 3269

Heap size

Report	Table data	Line plot	Structured data	verbosegc.001.log
------------------------	----------------------------	---------------------------	---------------------------------	-------------------

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.

Note: You may skip the data collection steps and use example data packaged at /opt/dockerdebg/fedorawasdebug/support/verbosegc.001.log

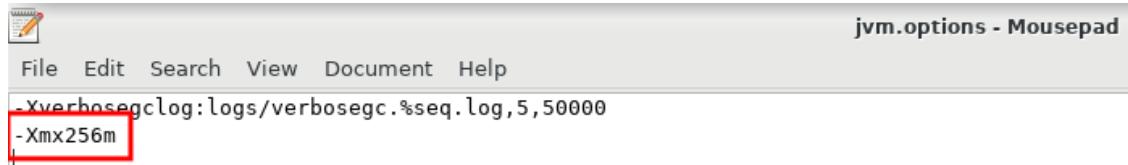
1. Stop JMeter if it is started.
2. Liberty:

1. Stop Liberty:

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

2. 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



```
jvm.options - Mousepad
File Edit Search View Document Help
-Xverboselog:logs/verbosegc.%seq.log,5,50000
-Xmx256m
```

3. Start Liberty

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

3. Start JMeter

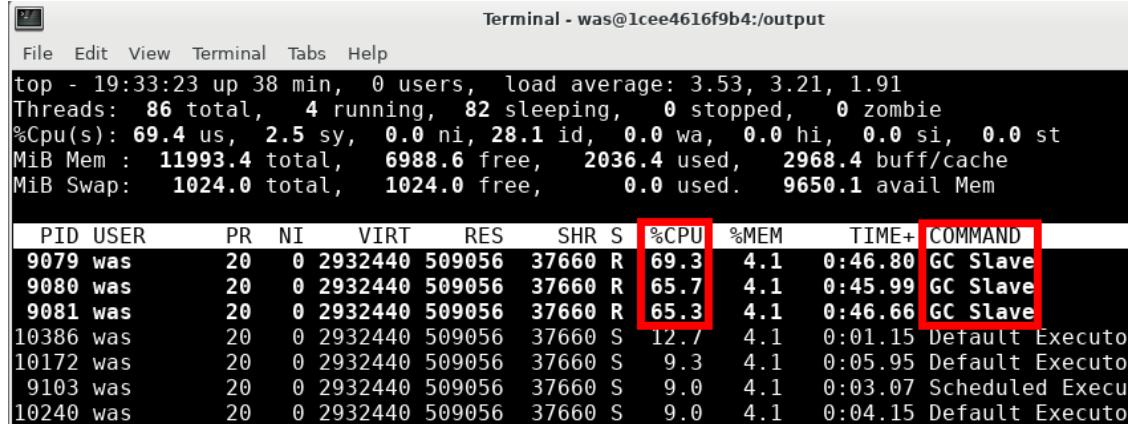
4. Let the JMeter test run for about 5 minutes.

5. Do not stop the JMeter test but leave it running as you continue to the next step.

6. Open your browser to the following page: <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
```



```
Terminal - was@lcee4616f9b4:/output
File Edit View Terminal Tabs Help
top - 19:33:23 up 38 min, 0 users, load average: 3.53, 3.21, 1.91
Threads: 86 total, 4 running, 82 sleeping, 0 stopped, 0 zombie
%Cpu(s): 69.4 us, 2.5 sy, 0.0 ni, 28.1 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 11993.4 total, 6988.6 free, 2036.4 used, 2968.4 buff/cache
MiB Swap: 1024.0 total, 1024.0 free, 0.0 used. 9650.1 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
9079 was 20 0 2932440 509056 37660 R 69.3 4.1 0:46.80 GC Slave
9080 was 20 0 2932440 509056 37660 R 65.7 4.1 0:45.99 GC Slave
9081 was 20 0 2932440 509056 37660 R 65.3 4.1 0:46.66 GC Slave
10386 was 20 0 2932440 509056 37660 S 12.7 4.1 0:01.15 Default Executo
10172 was 20 0 2932440 509056 37660 S 9.3 4.1 0:05.95 Default Executo
9103 was 20 0 2932440 509056 37660 S 9.0 4.1 0:03.07 Scheduled Execu
10240 was 20 0 2932440 509056 37660 S 9.0 4.1 0:04.15 Default Executo
```

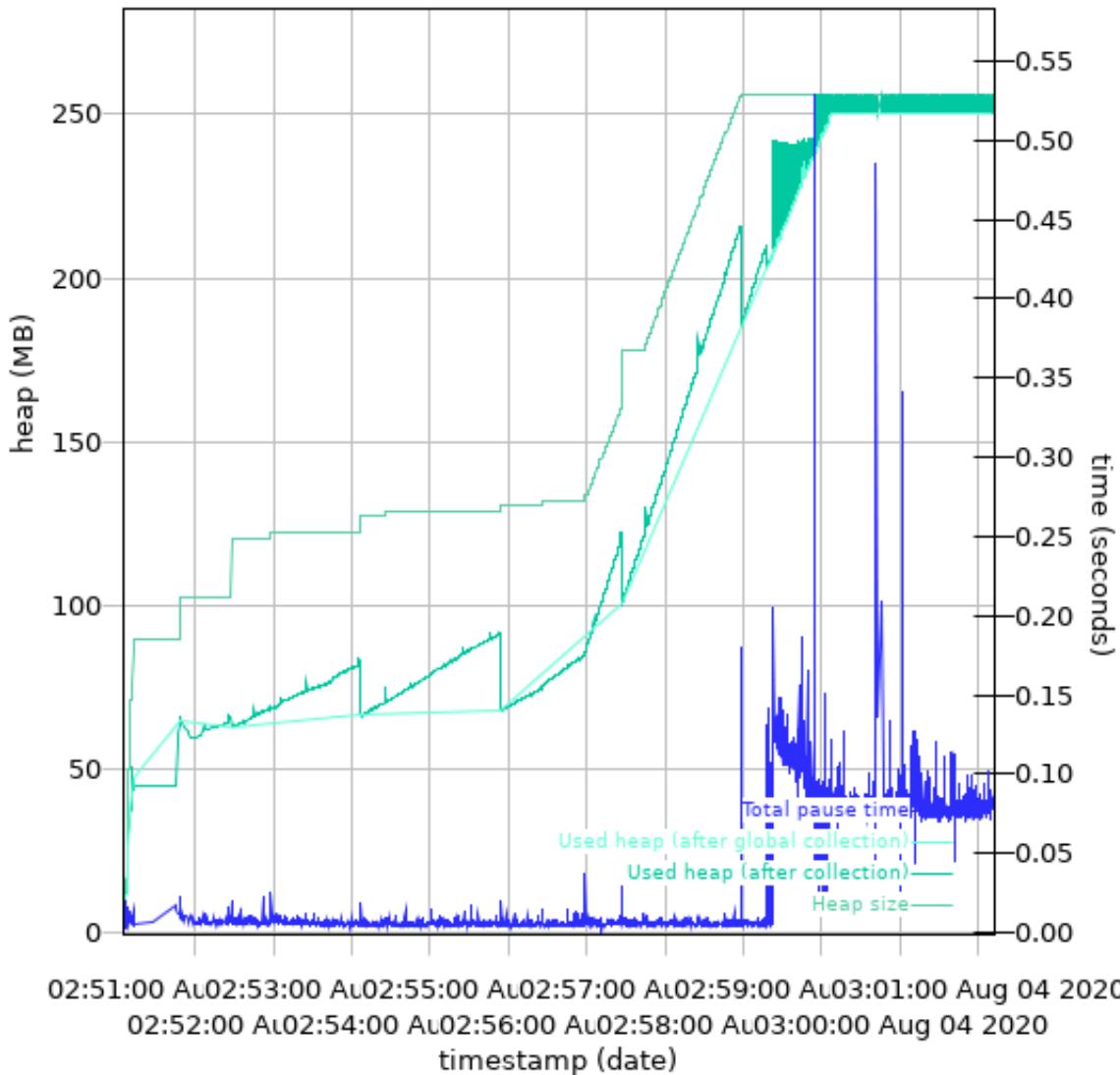
4. At some point, browser output will stop because the JVM has thrown an OutOfMemoryError.

7. Stop JMeter

8. Forcefully kill the JVM because an OutOfMemoryError does not stop the JVM; it will continue garbage collection thrashing and consume all of your CPU.

```
pkill -9 -f defaultServer
```

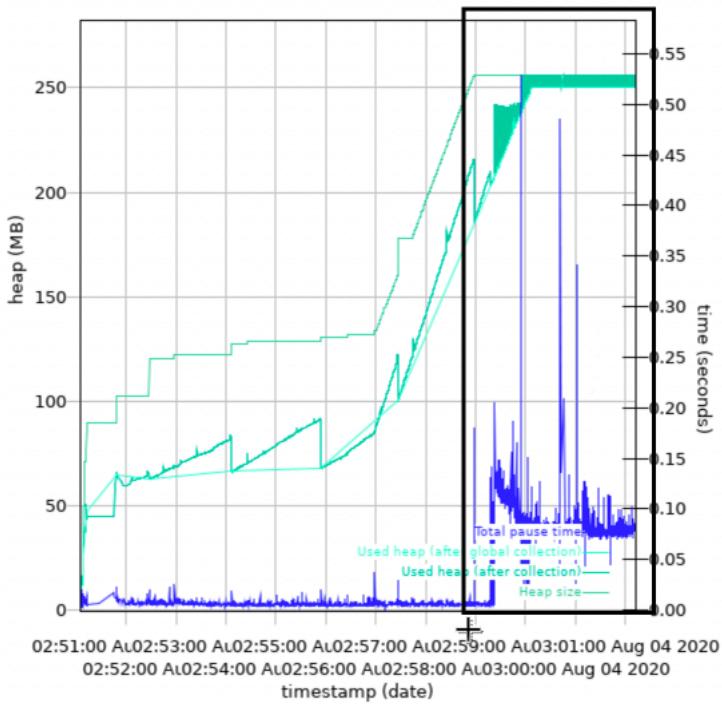
9. Close and re-open the **verbosegc*log** file in GCMV:



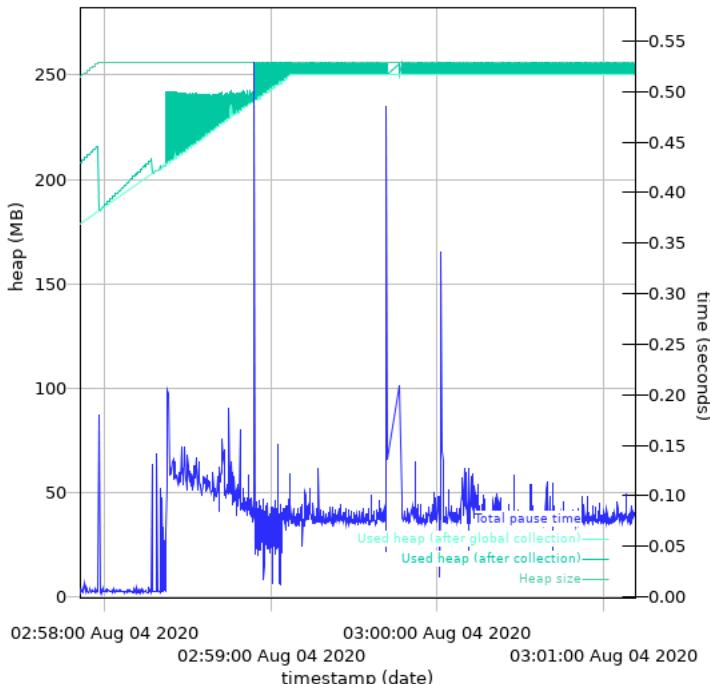
1. We can quickly see how the heap usage reaches 256MB and the pause time magnitude and durations increase significantly.
10. Click on the **Report** tab and review the **Proportion of time spent in garbage collection pauses (%)**:

Proportion of time spent in garbage collection pauses (%)	24.38
---	-------

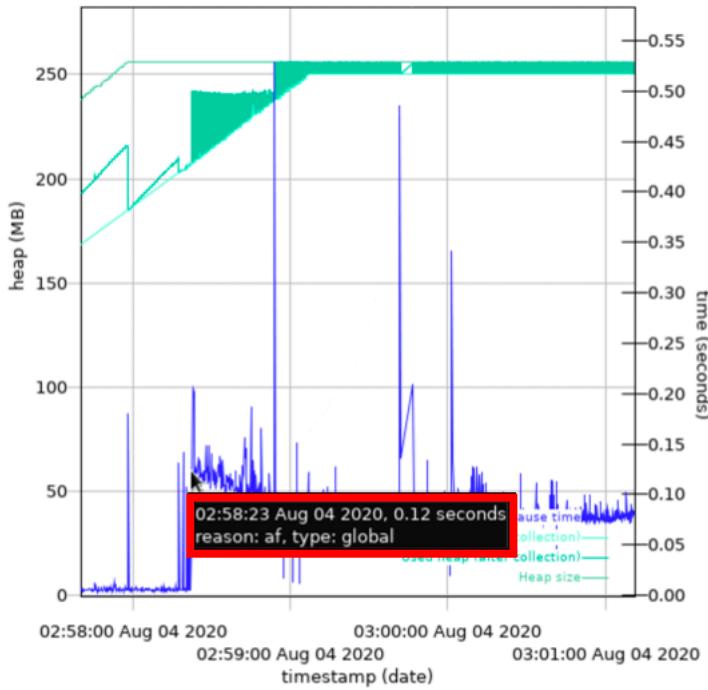
11. 24% seems pretty bad but not terrible and doesn't line up with what we know about what happened. This is because, by default, the GCMV Report tab shows statistics for the entire duration of the verbosegc log file. Since we had run the JMeter test for 5 minutes and it was healthy, the average proportion 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:



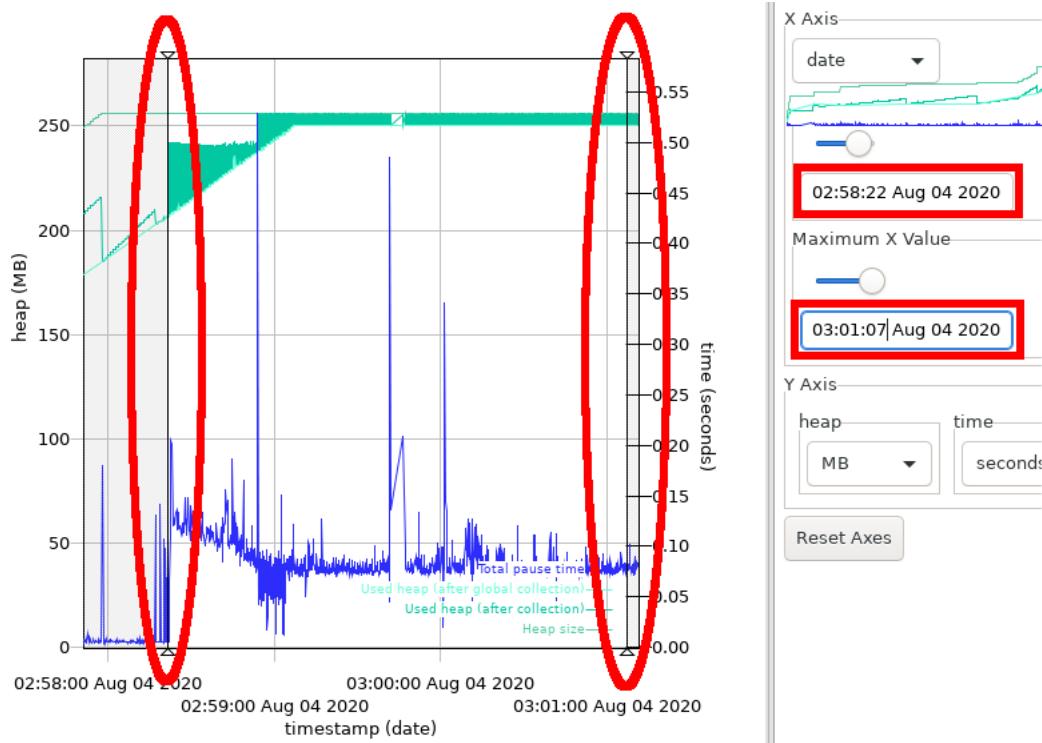
13. This will zoom the view to that bounding box:



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, ~87%).

Proportion of time spent in garbage collection pauses (%)	87.01
---	-------

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. Resolve memory leaks through heapdump analysis.
 4. Decrease the maximum thread pool size.

Health Center

IBM Monitoring and Diagnostics for Java - Health Center is free and shipped with IBM Java 8. 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.

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
- 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:

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).

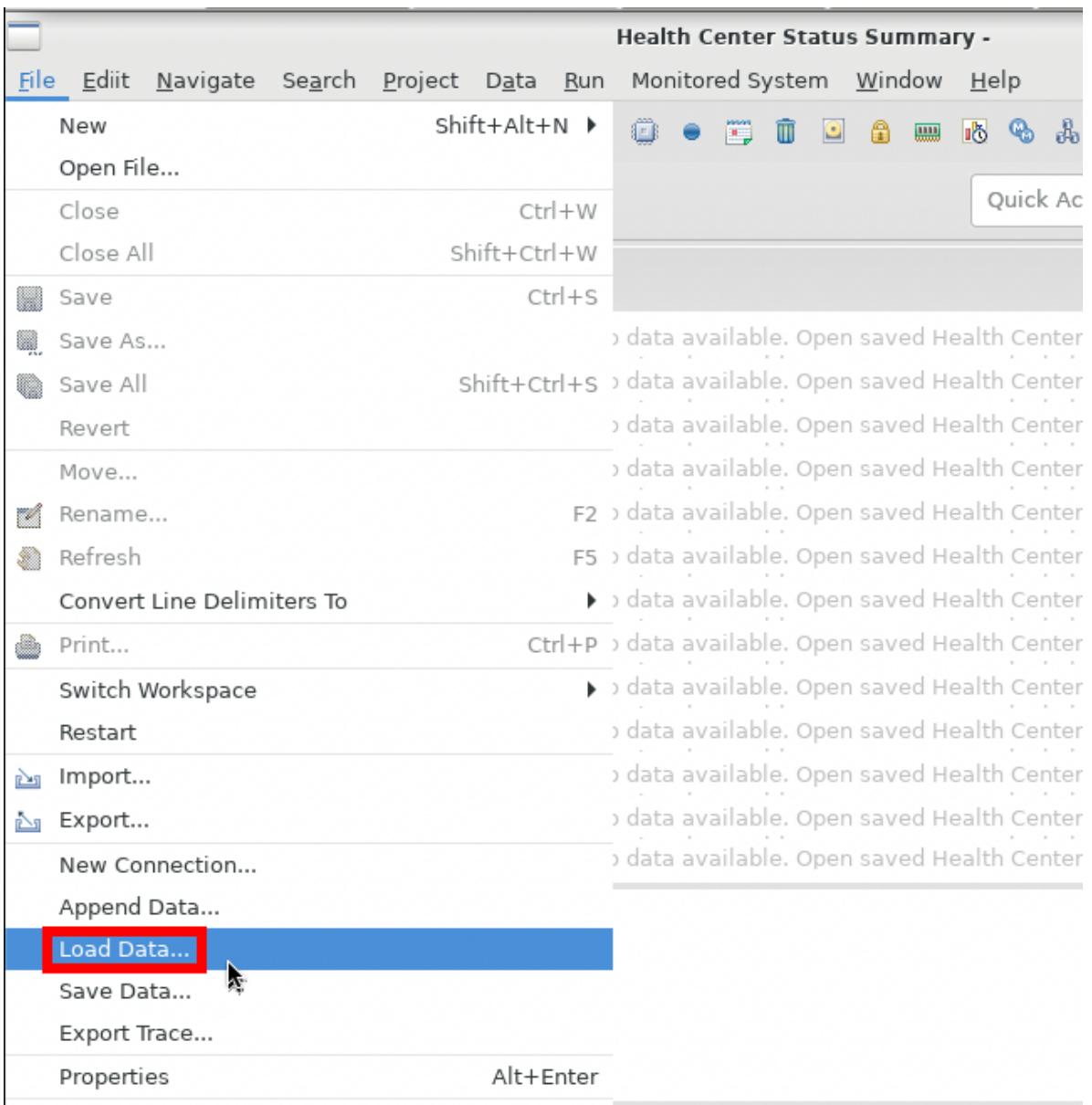
Consider always enabling HealthCenter in headless mode for post-mortem debugging of issues.

Health Center Lab

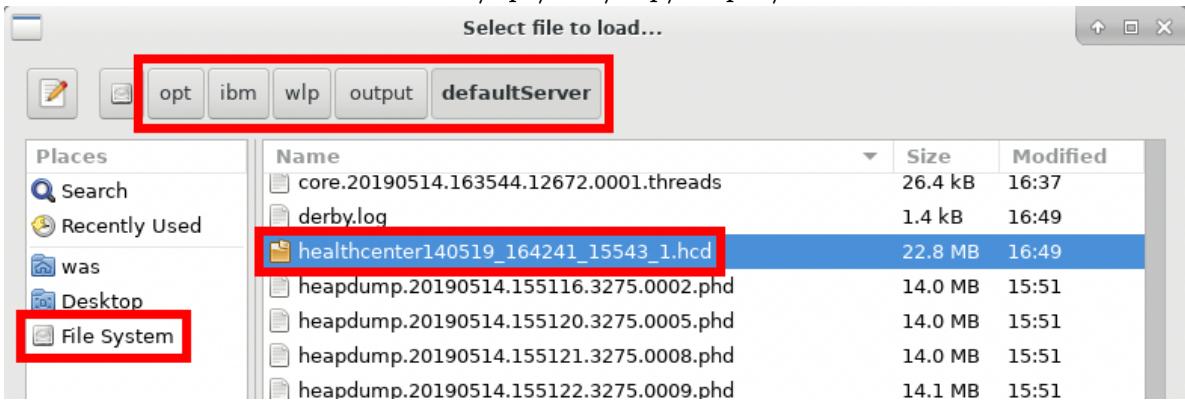
Note: You may skip the data collection steps and use example data packaged at /opt/dockerdebug/fedorawasdebug/support/

1. Stop JMeter if it is started.

2. Add Health Center arguments to the JVM:
 1. Add the following line to `/opt/ibm/wlp/usr/servers/defaultServer/jvm.options:`
`-Xhealthcenter:level=headless`
3. Stop the server:
`/opt/ibm/wlp/bin/server stop defaultServer`
4. Start the server
`/opt/ibm/wlp/bin/server start defaultServer`
5. Start JMeter and run it for 5 minutes.
6. Stop JMeter
7. Stop WAS as in step 3 above.
8. From the desktop, double click on **HealthCenter**.
9. Click **File > Load Data...** (note that it's towards the bottom of the **File** menu; **Open File** does not work):



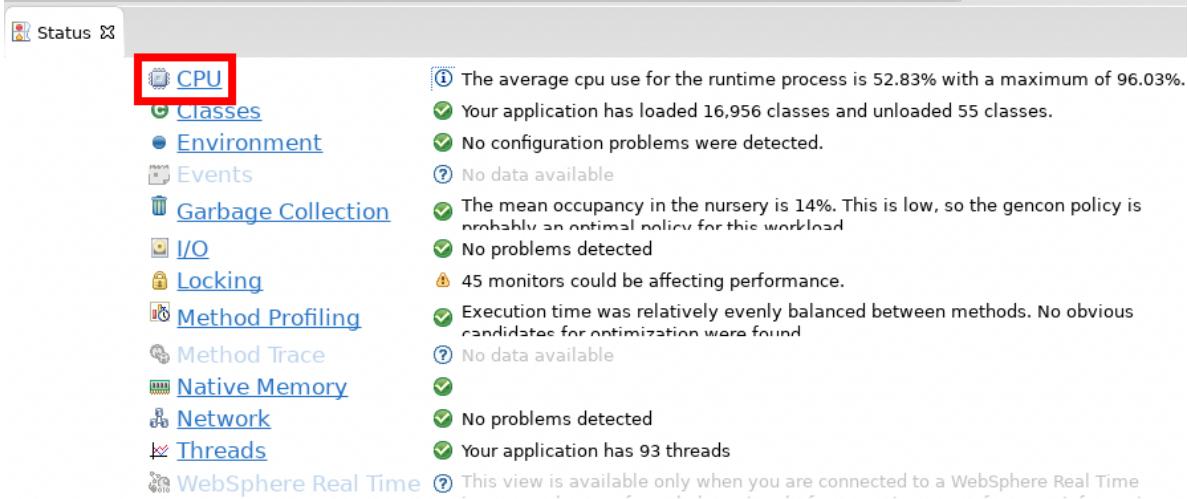
10. Select the **healthcenter*.hcd** file from **/opt/ibm/wlp/output/defaultServer**:



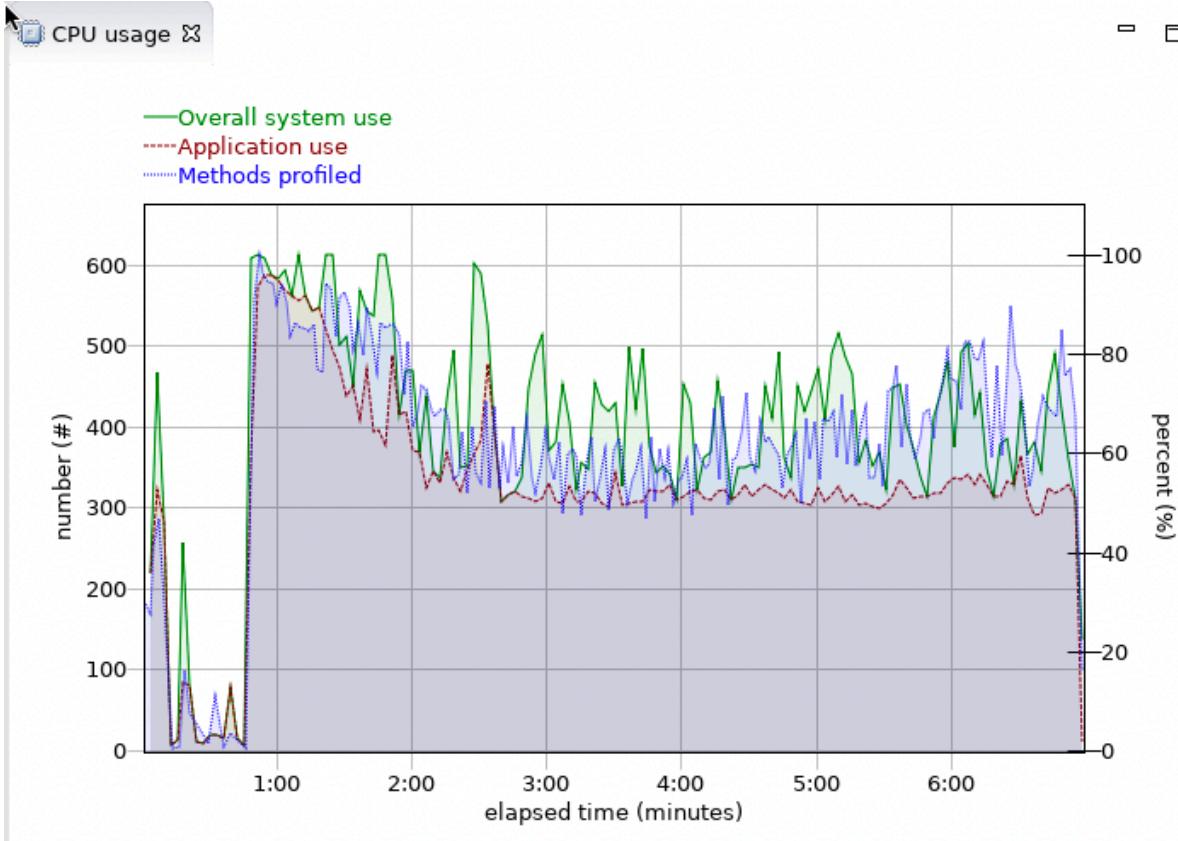
11. Wait for the data to complete loading:



12. Click on CPU:

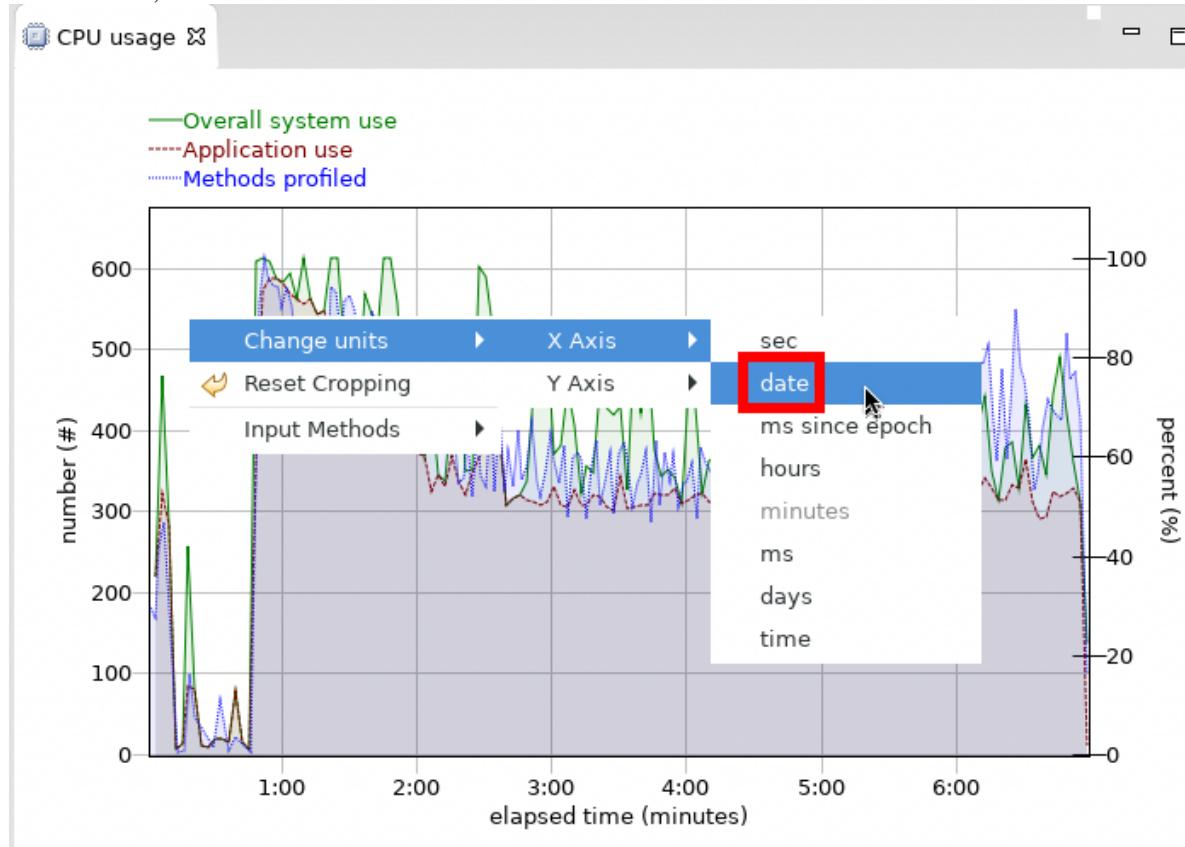


13. Review the overall system and Java application CPU usage:



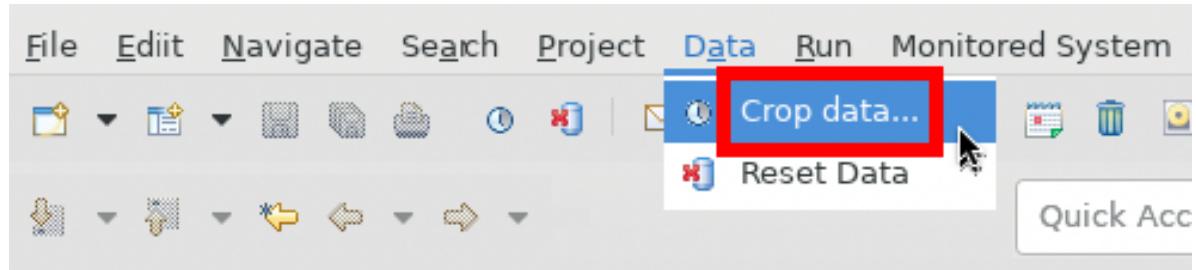
14. Right click anywhere in the graph and change the **X-axis** to **date** (which changes all other views to

date as well):

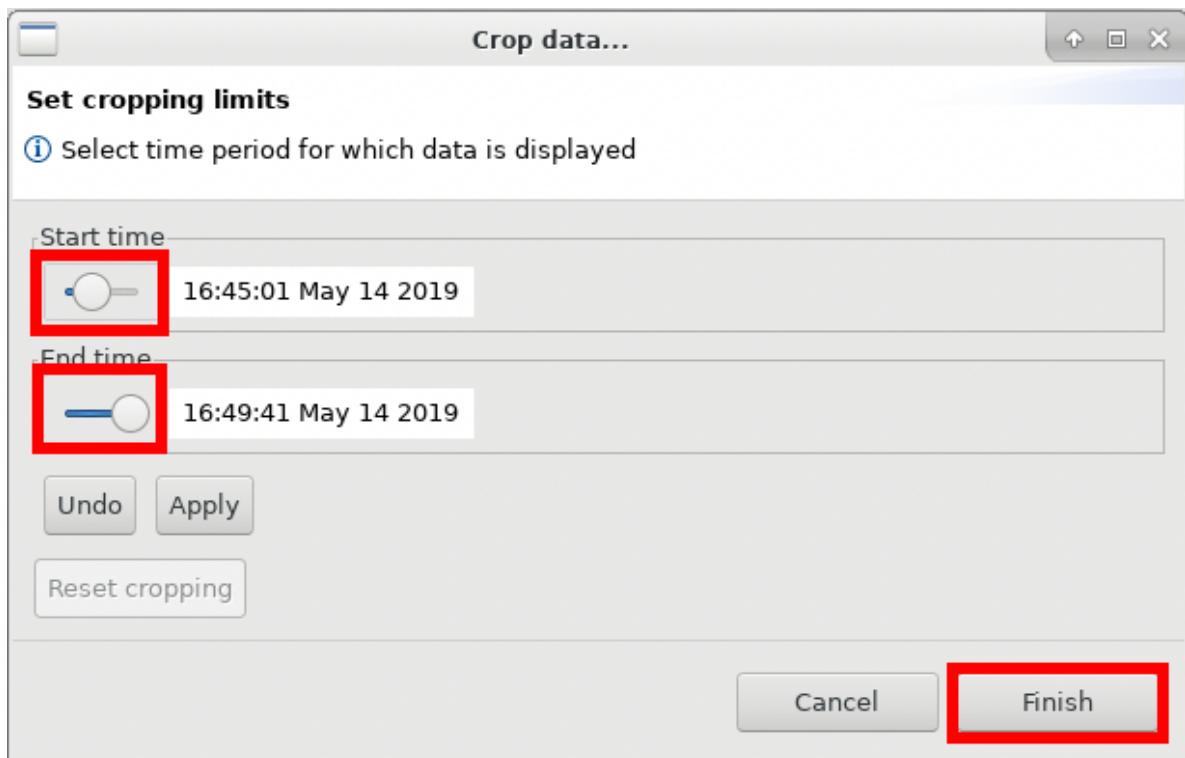


1. For large Health Center captures, this may take significant time to change and there is no obvious indication when it's complete. The best way to know is when the CPU usage of Health Center drops to a low amount.

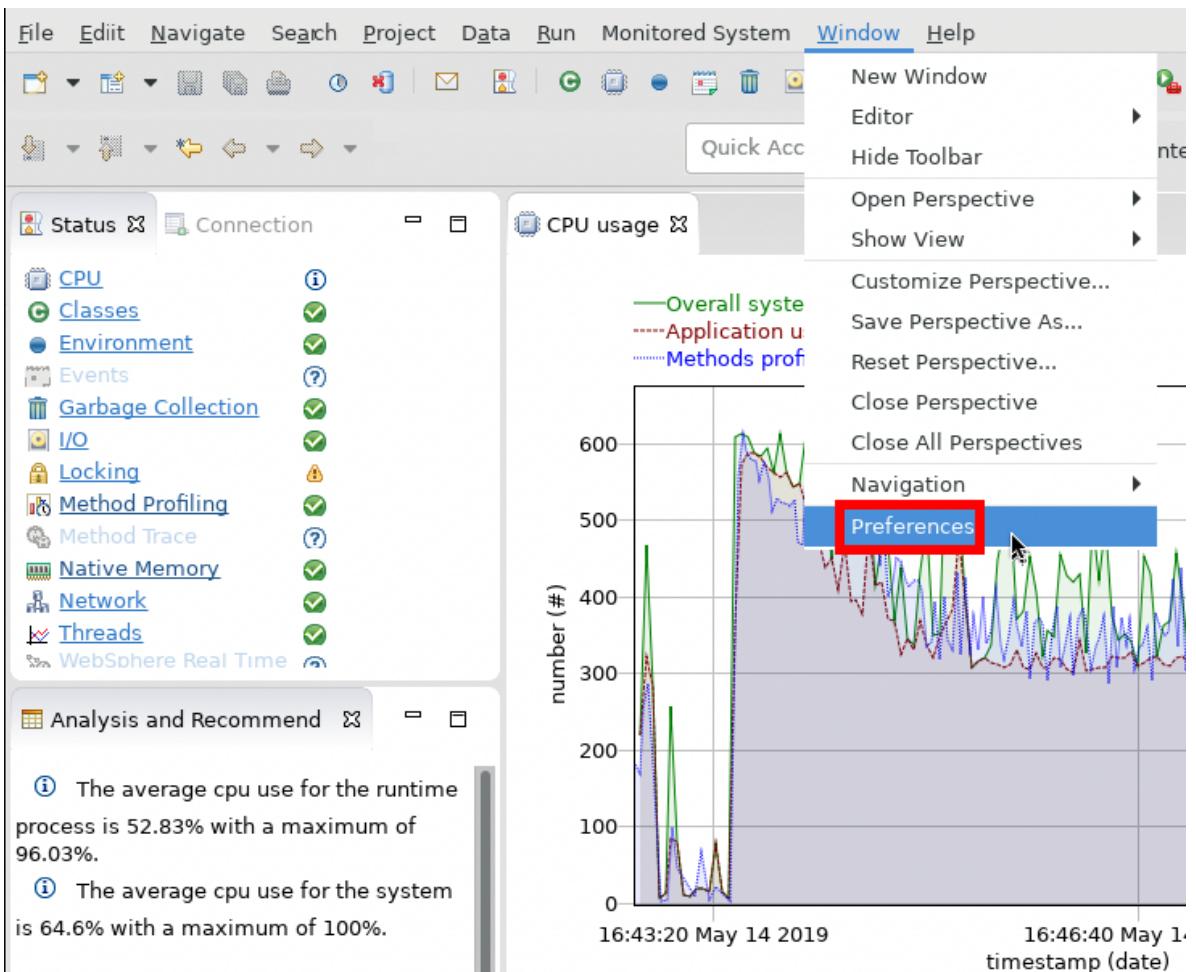
15. Click Data > Crop data...



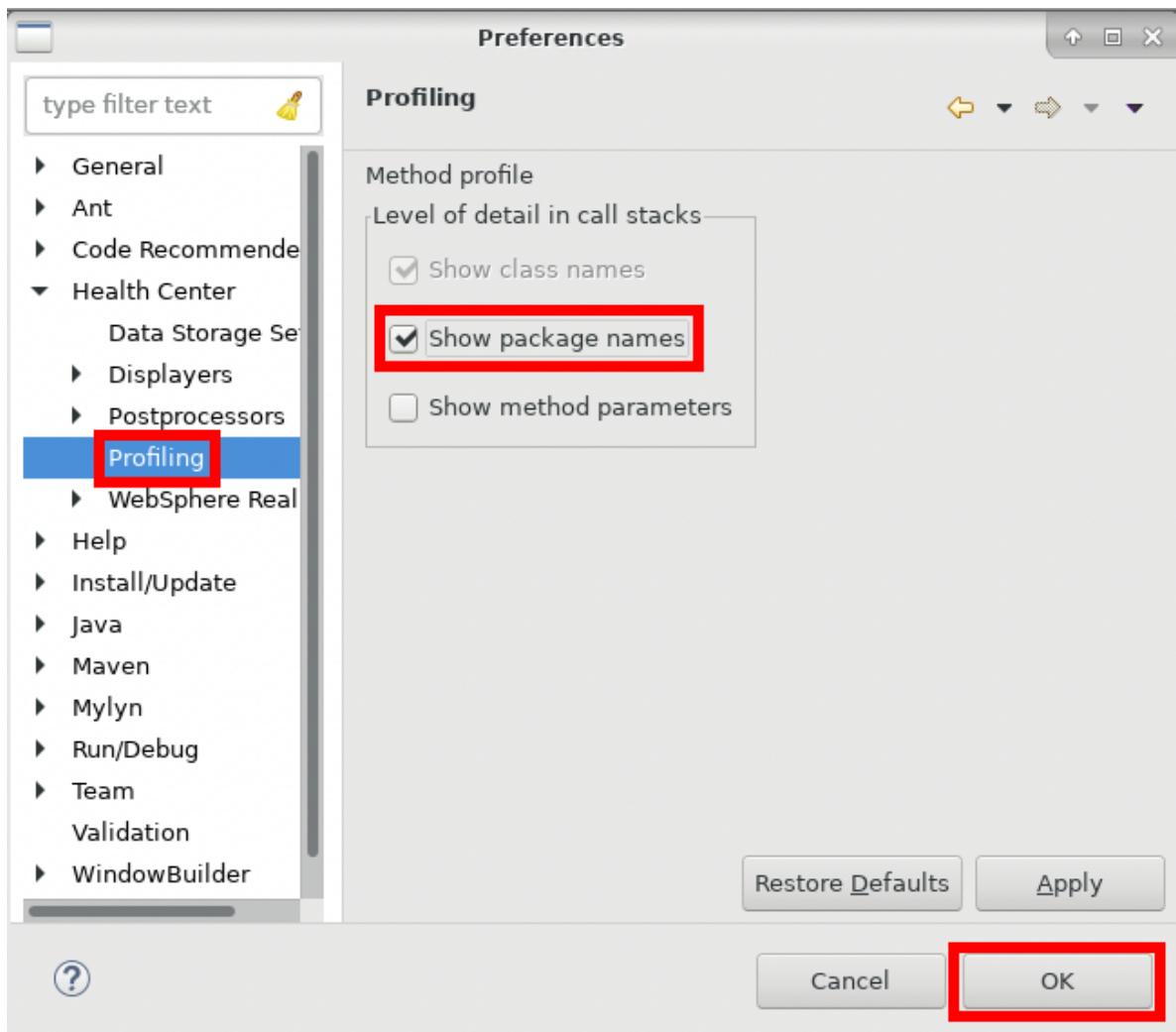
16. Change the **Start time** and **End time** to match the period of interest. For example, usually you want to exclude the start-up time of the process and only focus on user activity:



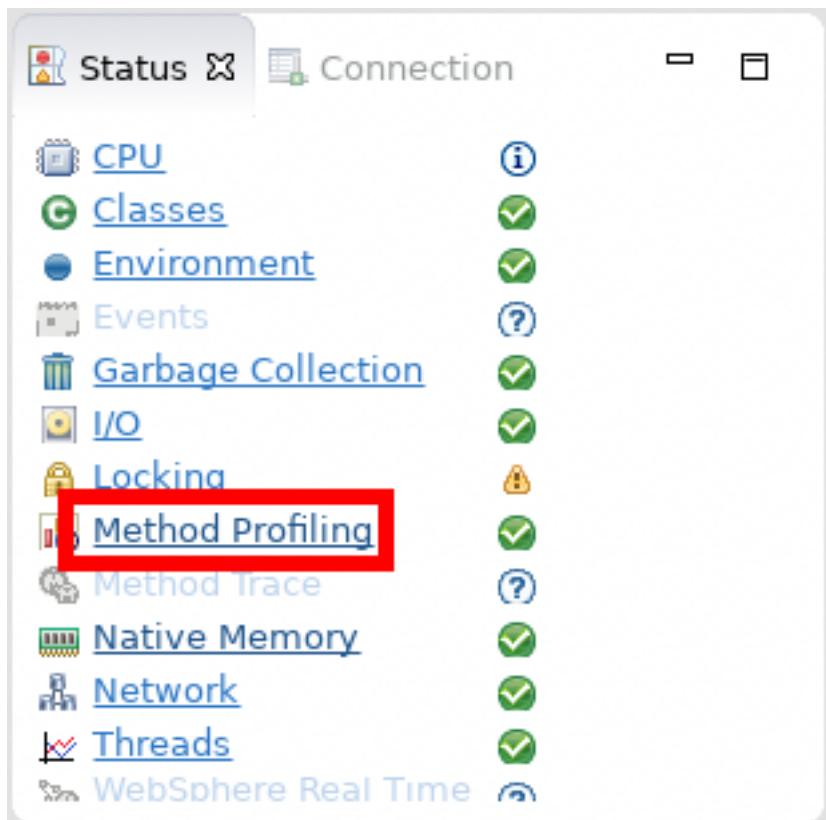
17. Click Window > Preferences:



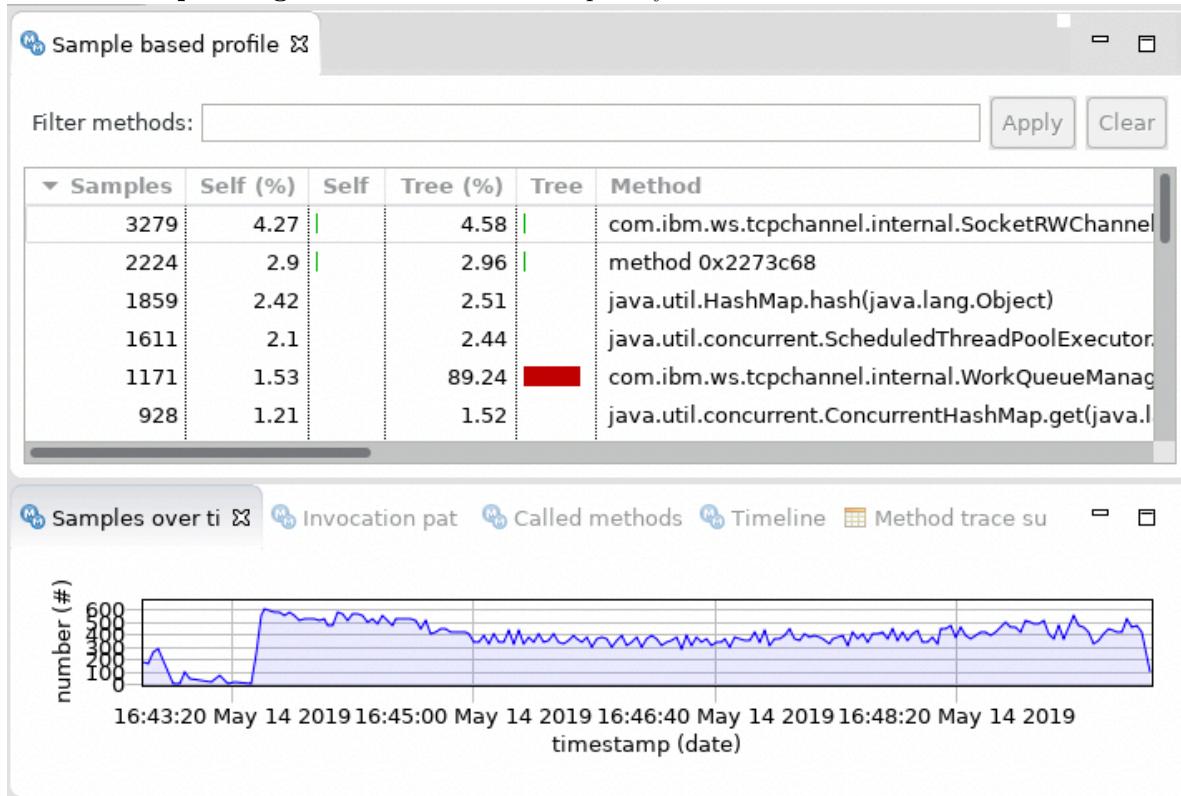
18. Check the **Show package names** box under **Health Center > Profiling** and press **OK** so that we can see more details in the profiling view:



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



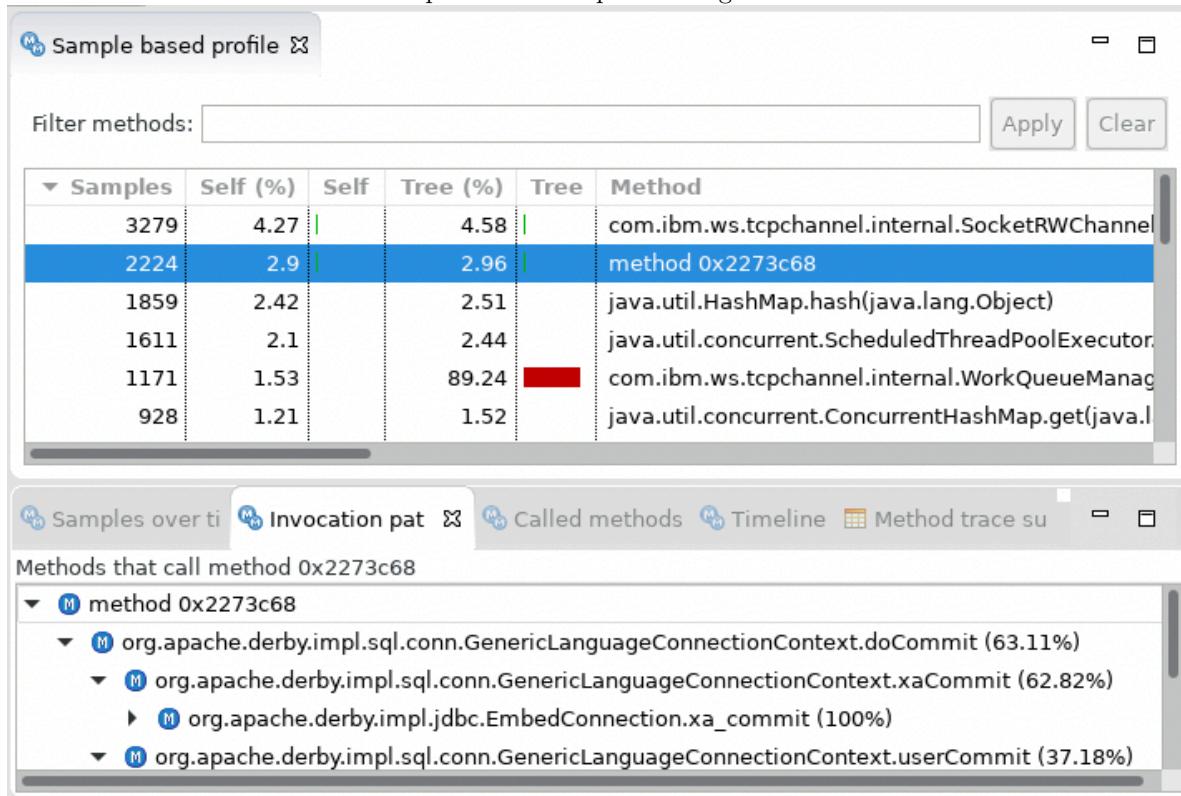
20. The **Method profiling** view will show CPU samples by method:



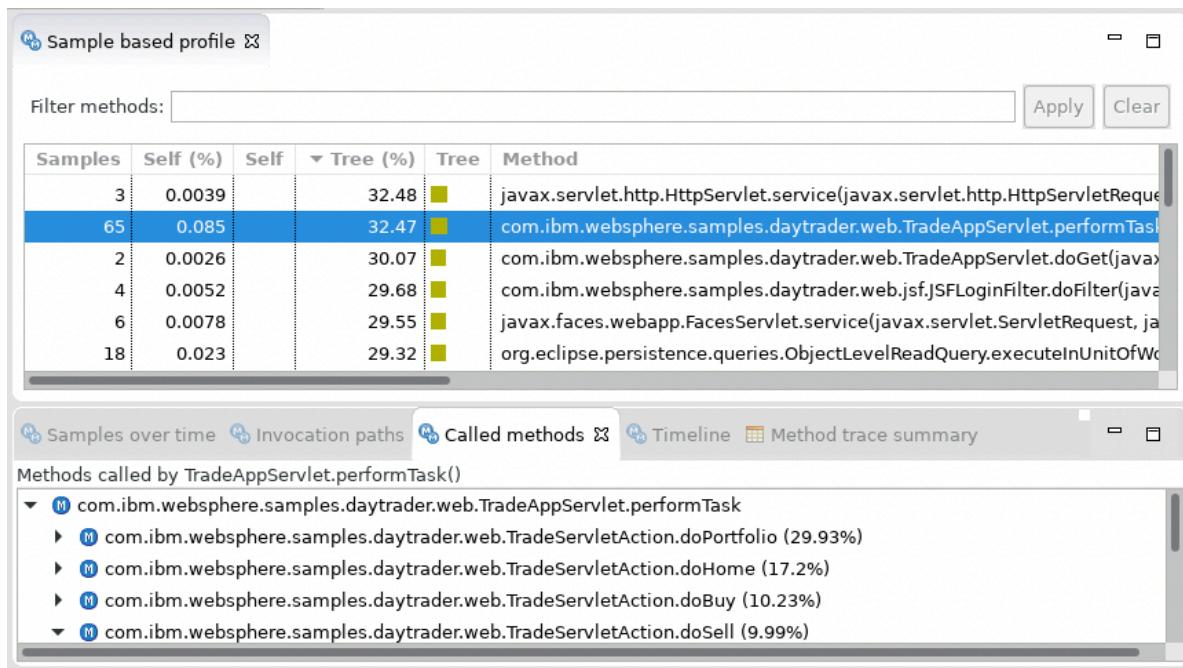
21. The **Self (%)** column reports the percent of samples where a method was at the top of the stack. The

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 0x2273c68 (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:



- 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.



WebSphere Liberty

WebSphere Liberty has many built-in troubleshooting and performance features, including:

- Request Timing
- HTTP NCSA access log
- Admin Center
- MXBean Monitoring
- Server Dumps
- Event Logging
- Diagnostic trace
- Binary logging
- Timed operations

Request Timing

WebSphere Liberty's Slow and Hung Request Detection is optionally enabled with the requestTiming-1.0 feature.

The slow request detection part of the feature monitors for HTTP requests that exceed a configured threshold and prints a tree of events breaking down the components of the slow request. The hung request detection part of the feature additionally gathers thread dumps after its threshold is exceeded.

requestTiming Lab

1. Modify `/config/server.xml` to add the following before `</server>`:


```
<featureManager><feature>requestTiming-1.0</feature></featureManager>
<requestTiming slowRequestThreshold="60s" hungRequestThreshold="180s" sampleRate="1" />
```
2. Execute a request that takes more than one minute by opening a browser to `http://localhost:9080/swat/Sleep?duration=60`
3. After about a minute and the request completes, review the requestTiming warning in `/logs/messages.log`
 - for example:

[3/20/22 16:16:52:250 UTC] 0000007b com.ibm.ws.request.timing.manager.SlowRequestManager

```
at java.lang.Thread.sleep(Native Method)
at java.lang.Thread.sleep(Thread.java:956)
at com.ibm.Sleep.doSleep(Sleep.java:35)
at com.ibm.Sleep.doWork(Sleep.java:18)
at com.ibm.BaseServlet.service(BaseServlet.java:73)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:790)
[...]
```

The following table shows the events that have run during this request.

Duration Operation

60008.080ms + websphere.servlet.service | swat | Sleep?duration=65000

1. The warning shows a stack at the time `requestTiming` notices the threshold is breached and it's followed by a tree of components of the request. The plus sign (+) indicates that an operation is still in progress. The indentation level indicates which events requested which other events.
4. Execute a request that takes about three minutes by opening a browser to `http://localhost:9080/swat/Sleep?duration=200000`
5. After about five minutes, review the `requestTiming` warning in `/logs/messages.log` – in addition to the previous warning, multiple thread dumps are produced:

[3/20/22 16:22:23:565 UTC] 0000007d com.ibm.ws.kernel.launch.internal.FrameworkManager
[3/20/22 16:22:23:662 UTC] 0000007d com.ibm.ws.kernel.launch.internal.FrameworkManager

1. Thread dumps will be captured, one minute apart, after the threshold is breached.
6. Review the thread dumps using the TMDA tool and see if you can find the captured long-running request.

In general, it is a good practice to use `requestTiming`, even in production. Configure the thresholds to values that are at the upper end of acceptable times for the users and the business. Configure and test the `sampleRate` to ensure the overhead of `requestTiming` is acceptable in production.

When the `requestTiming` feature is enabled, the server dump command will include a snapshot of all the event trees for all requests thus giving a very nice and lightweight way to see active requests in the system at a detailed level (including URI, etc.), in a similar way that thread dumps do the same for thread stacks.

HTTP NCSA Access Log

The Liberty HTTP access log is optionally enabled with the `httpEndpoint` `accessLogging` element. When enabled, a separate `access.log` file is produced with an NCSA standardized (i.e. httpd-style) line for each HTTP request, including items such as the URI and response time, useful for post-mortem correlation and performance analysis. For example,

HTTP NCSA Access Log Lab

1. Modify `/config/server.xml` to change the following element:

```
<httpEndpoint id="defaultHttpEndpoint"
    host="*"
    httpPort="9080"
    httpsPort="9443" />
```

2. To the following (note that the `<httpEndpoint>` element must no longer be self-closing):

```
<httpEndpoint id="defaultHttpEndpoint"
    host="*"
```

```

        httpPort="9080"
        httpsPort="9443">
<accessLogging filepath="${server.output.dir}/logs/access.log" maxFileSize="250" maxFiles="2" log=
</httpEndpoint>
```

3. Start JMeter
4. Run the test for a couple of minutes.
5. Stop JMeter
6. Review /opt/ibm/wlp/output/defaultServer/logs/access.log to see HTTP responses. For example:

```

127.0.0.1 - Admin1 [20/Mar/2022:16:28:23 +0000] "GET /daytrader/app?action=portfolio HTTP/1.1" 200
127.0.0.1 - Admin1 [20/Mar/2022:16:28:23 +0000] "GET /daytrader/app?action=quotes&symbol=s%3A6651 H
127.0.0.1 - Admin1 [20/Mar/2022:16:28:23 +0000] "GET /daytrader/app?action=quotes&symbol=s%3A9206 H
```

7. The second-to-last number is the response time in microseconds. In the example above, the first response time was 397.114 milliseconds. The last number is the time until the first byte of the response was sent back which may help investigate network slowdowns in front of WebSphere.

There are various scripts and tools available publicly (example) to post-process NCSA-style access logs to create statistics and graphs.

In general, it is a good practice to use `accessLogging`, even in production, if the performance overhead is acceptable.

Additional Liberty Labs

For additional labs on Liberty, review the main lab sections on WebSphere Liberty starting with Liberty Bikes.

Methodology

The Scientific Method

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.

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		Next Steps
		#	Status	
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%
2	Slow database response times	Thread stacks showed many threads waiting on the database	Gather database re-sponse 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	ConcurrentModificationException correlate with HTTP 500 response codes	Gather WAS diagnostic trace capturing some exceptions

Experiments/Tests

#	Experiment/Test	Start	End	Environment	Changes	Results
1	Baseline	2019-01-01 09:00:00 UTC	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	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	2019-01-01 14:00:00 UTC	Test server1	Increase Java heap size to 1g	Average response time 4000ms; Website error rate 15%

#	Experiment/Test Start	End	Environment	Changes	Results	
4	Test problem 1 - hypothesis 1	2019-01-04 09:00:00 UTC	2019-01-01 17:00:00 UTC	Production server1	Increase Java heap size to 1g	Average response time 2000ms; Website error rate 10%

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.).

Liberty Performance Tuning Recipe

The IBM WebSphere Application Server Performance Cookbook provides a recipe for common tuning of Liberty and underlying components.

Appendix

Stopping the container

The `podman run` or `docker run` command in this lab does not use the `-d` (daemon) flag which means that it runs in the foreground. To stop such a container, use one of the following methods:

1. Hold down the Control/Ctrl key on your keyboard and press C in the terminal window where the `run` command is running.
2. In a separate terminal window, find the container ID with `podman ps` or `docker ps` command and then run `podman stop $ID` or `docker stop $ID`.

If you add `-d` to `podman run` or `docker run`, then to view the standard out logs, find the container ID with `podman ps` or `docker ps` and then run `podman logs $ID` or `docker logs $ID`. To stop, use `podman stop $ID` or `docker stop $ID`.

Remote terminal into the container

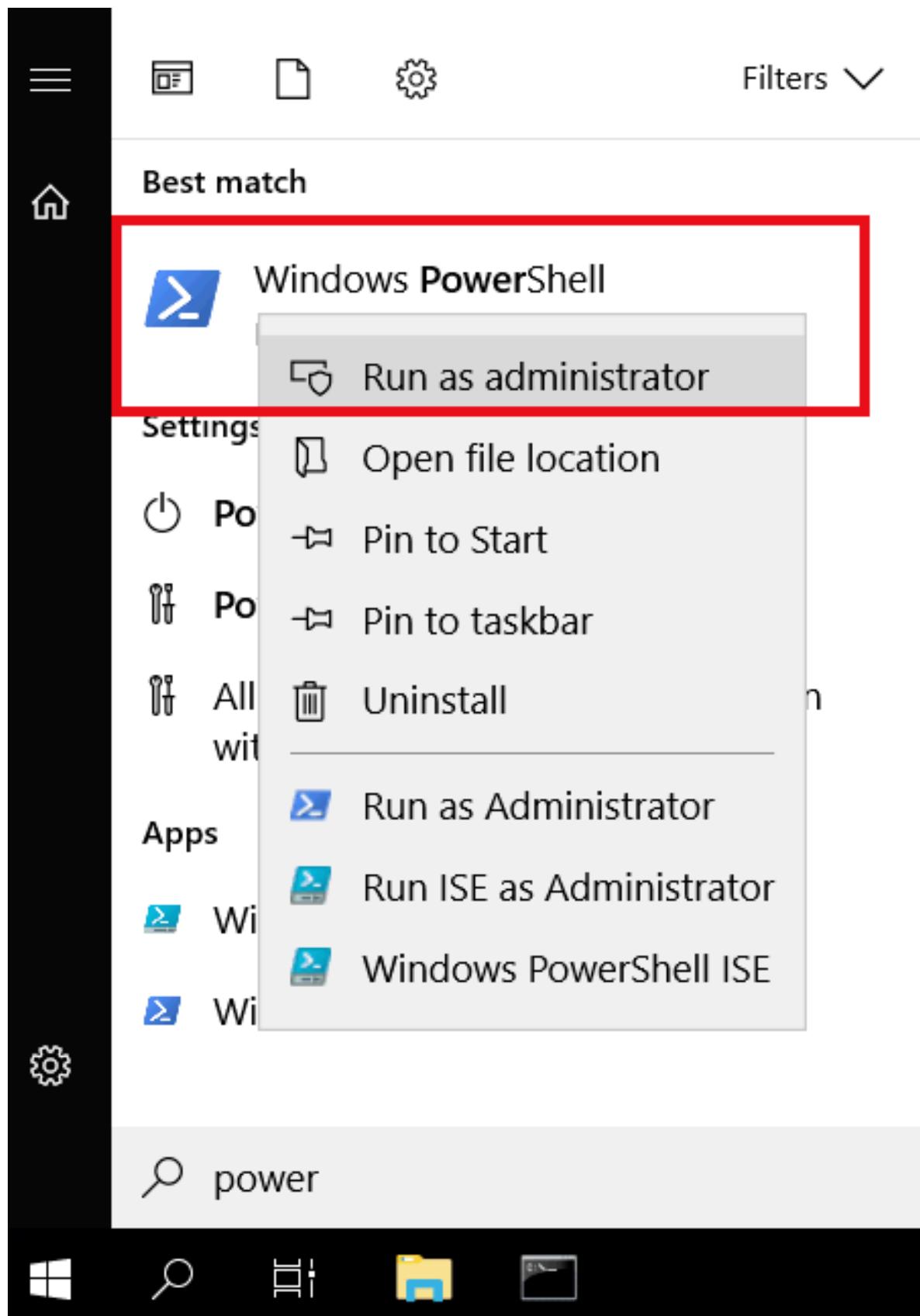
In a separate command prompt or terminal window, execute one of the following commands depending on whether you're using `podman` or Docker Desktop:

- `podman exec -u was -it $(podman ps -q) bash`
- `docker exec -u was -it $(podman ps -q) bash`

Windows Remote Desktop Client

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

1. Open **PowerShell** as Administrator:



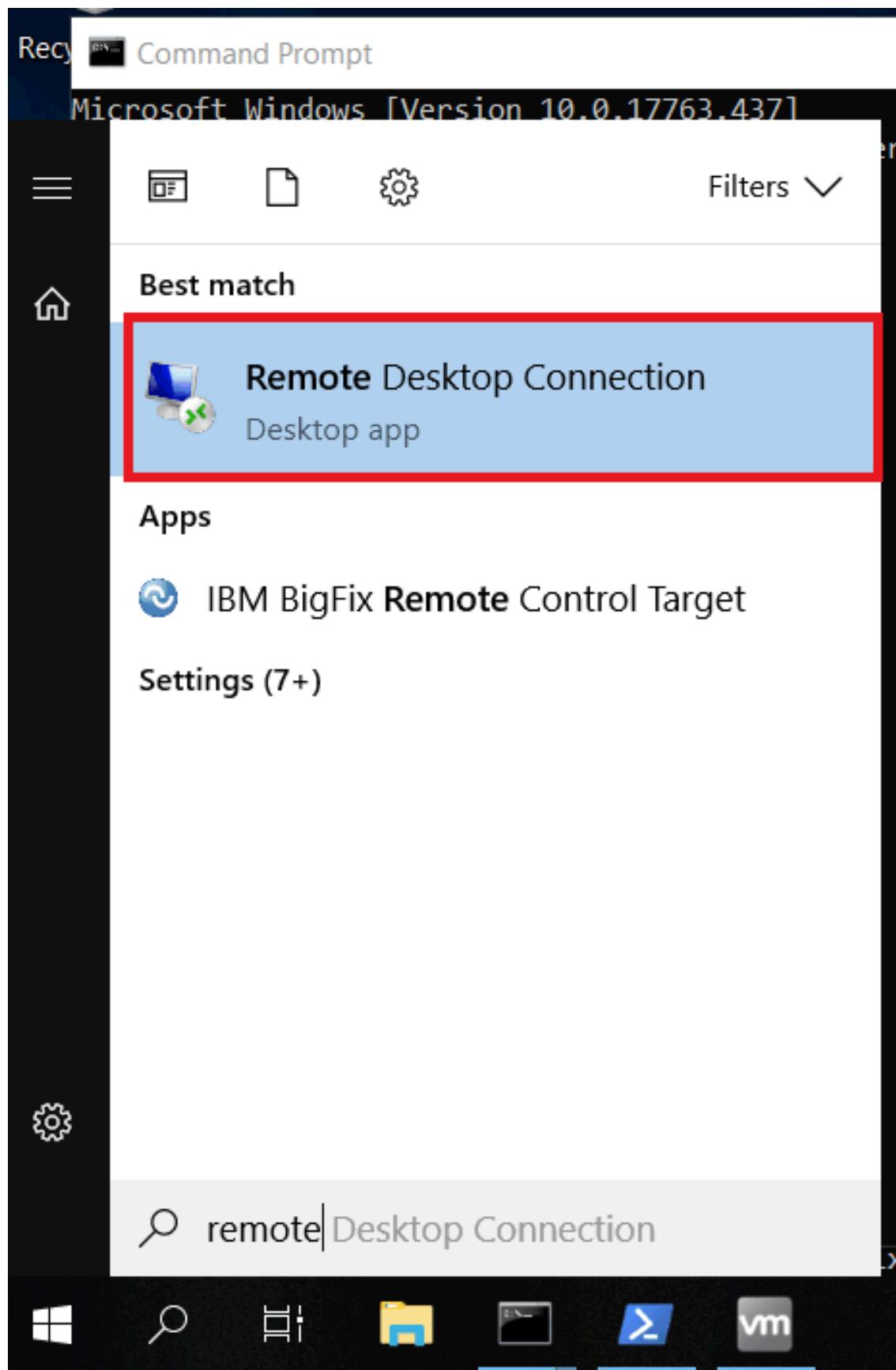
2. Run **ipconfig** and copy the **IPv4** address of the **WSL** adapter. For example, **172.24.0.1**:

```
Ethernet adapter vEthernet (WSL):  
  Connection-specific DNS Suffix . :  
  Link-local IPv6 Address . . . . . : fe80::457e:650b:7002:d26a%68  
  IPv4 Address . . . . . : 172.24.0.1  
    Subnet Mask . . . . . : 255.255.240.0  
  Default Gateway . . . . . :
```

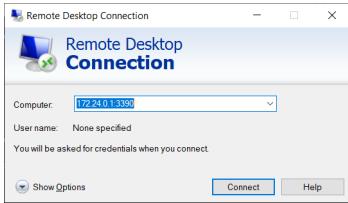
3. Run the following command in **PowerShell**:

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

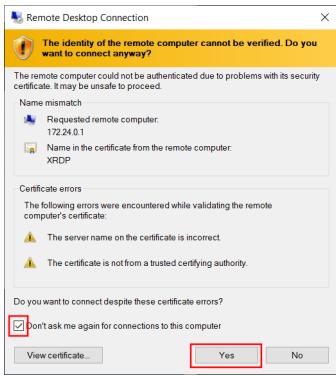
4. Run **Remote Desktop**



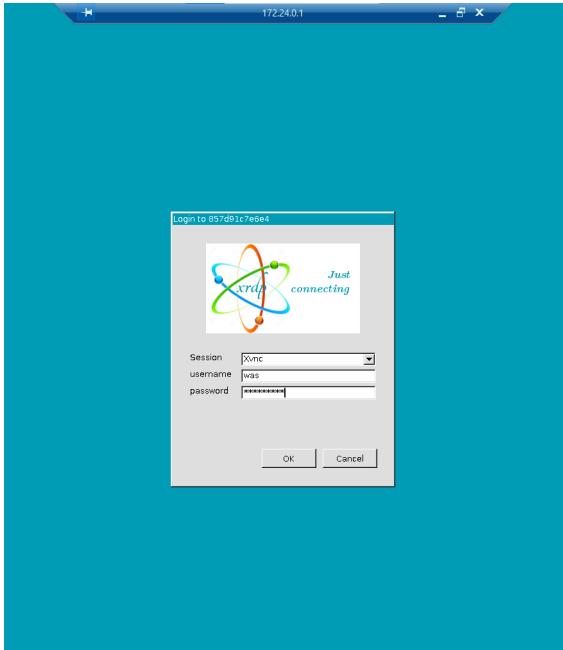
5. Enter the WSL IP address (for example, 172.24.0.1) followed by :3390 as **Computer** and click **Connect**:



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



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

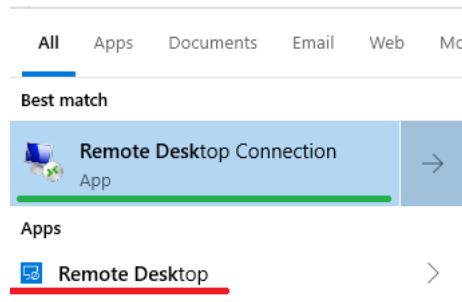


8. You should now be remote desktop'ed into the container:



9. Notes:

1. In some cases, only the **Remote Desktop Connection** application worked, and **not Remote Desktop**:



2. Microsoft requires the above steps and the use of port 3390 instead of directly connecting to 3389.

Changing Java

There are many different versions and types of Java in the image. To list them, run:

```
$ alternatives --display java | grep "^\/"
/usr/lib/jvm/java-11-openjdk-11.0.14.1.1-5.fc35.x86_64/bin/java - family java-11-openjdk.x86_64 priority
/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.322.b06-6.fc35.x86_64/jre/bin/java - family java-1.8.0-openjdk.x8
/opt/ibm/java/bin/java - family ibmjava priority 99999999
/opt/openjdk8_hotspot/jdk/bin/java - family openjdk priority 89999999
/opt/openjdk11_hotspot/jdk/bin/java - family openjdk priority 89999999
/opt/openjdk17_hotspot/jdk/bin/java - family openjdk priority 89999999
/opt/openjdk8_ibm/jdk/bin/java - family openjdk priority 89999999
/opt/openjdk11_ibm/jdk/bin/java - family openjdk priority 89999999
/opt/openjdk17_ibm/jdk/bin/java - family openjdk priority 89999999
```

To change the Java that is on the path, run the following command and enter the number of the Java that you wish to change to and press Enter. The directory name tells you the type of Java; for example, OpenJ9 or HotSpot. The directory name also tells you the version of Java (for example, openjdk17 is Java 17).

```
$ sudo alternatives --config java
sudo: unable to send audit message: Operation not permitted
```

There are 9 programs which provide 'java'.

Selection	Command
1	java-11-openjdk.x86_64 (/usr/lib/jvm/java-11-openjdk-11.0.14.1.1-5.fc35.x86_64/bin/java)

```

2      java-1.8.0-openjdk.x86_64 (/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.322.b06-6.fc35.x86_64/j
** 3      ibmjava (/opt/ibm/java/bin/java)
4      openjdk (/opt/openjdk8_hotspot/jdk/bin/java)
5      openjdk (/opt/openjdk11_hotspot/jdk/bin/java)
6      openjdk (/opt/openjdk17_hotspot/jdk/bin/java)
7      openjdk (/opt/openjdk8_ibm/jdk/bin/java)
8      openjdk (/opt/openjdk11_ibm/jdk/bin/java)
9      openjdk (/opt/openjdk17_ibm/jdk/bin/java)

Enter to keep the current selection[+], or type selection number: 9
$ java -version
openjdk version "17.0.2" 2022-01-18
IBM Semeru Runtime Open Edition 17.0.2.0 (build 17.0.2+8)
Eclipse OpenJ9 VM 17.0.2.0 (build openj9-0.30.0, JRE 17 Linux amd64-64-Bit Compressed References 2022011
OpenJ9   - 9dccbe076
OMR     - dac962a28
JCL     - 64cd399ca28 based on jdk-17.0.2+8)

```

The `alternatives` command has the concept of groups of commands so when you change Java using the method above, other commands like `jar`, `javac`, etc. also change.

Any currently running Java programs will need to be restarted if you want them to use the different version of Java (WAS traditional is an exception because it uses a bundled version of Java).

Common Issues

VNC and the clipboard

The clipboard may not be shared when using VNC. There are two workarounds:

1. From within the VNC session, open the PDF of the lab from the desktop and use that instead. Then you can copy/paste within the lab.
2. For command line steps, start a new command prompt or terminal and execute one of the following commands depending on whether you're using `podman` or Docker Desktop:
 - `podman exec -u was -it $(podman ps -q) bash`
 - `docker exec -u was -it $(podman ps -q) bash`

Cannot login after screen locked

This is a known issue that will be fixed on the next build of the lab. Until then, the workaround is:

1. Start a new command prompt or terminal and execute one of the following commands depending on whether you're using `podman` or Docker Desktop:
 - `podman exec -u root -it $(podman ps -q) bash`
 - `docker exec -u root -it $(podman ps -q) bash`
2. Then execute the following command:
`sed -i '1 i\auth sufficient pam_succeed_if.so user = was' /etc/pam.d/xfce4-screensaver`
3. Try unlocking the screen again.

Version History

- V18 (March 14, 2022):
 - Add workaround for Liberty Issue 20082
 - Add Eclipse Mission Control 8.1
- V17 (February 21, 2022):

- Add podman instructions in addition to Docker Desktop
 - Upgrade to tWAS 9.0.5.10
 - Upgrade to Liberty 22.0.0.1
 - Add IBM Semeru runtimes
 - Update MAT, GCMV, and HealthCenter
 - Update Request Metrics Analyzer Next
 - Update to Eclipse 2021-12
 - Update WAS Configuration Visualizer
 - Switch to IBM HTTP Server archive install
 - Update Performance Tuning Toolkit
 - Add IBM Service Integration Bus Destination Handler
- V16 (April 13, 2021):
 - Upgrade to Liberty 21.0.0.3
 - Update to TMDA 4.6.9
 - Add AdoptOpenJDK Java 16 (J9 and HotSpot)
 - Change Health Center to use Eclipse 2020-03 because Luna is no longer available. The known StackOverflowError on 2020-03 doesn't always happen.
 - Upgrade Request Metrics Analyzer to 2.0.20210111
 - Increase Eclipse 2020-03 max heap to 4g
- V15 (February 15, 2021):
 - Fix issue 2 by upgrading to Liberty 21.0.0.1
 - Upgrade to tWAS 9.0.5.6
- V14 (January 12, 2021): Fix issue tailing tWAS logs
- V13 (January 11, 2021): Refresh software:
 - Upgrade to Fedora 33
 - Upgrade to Liberty 20.0.0.12
 - Upgrade PTT to V1.0.20200908
 - Update to TMDA 4.6.8
- V12 (August 3, 2020): Refresh software:
 - Upgrade to Fedora 32
 - Upgrade to Liberty 20.0.0.8
 - Upgrade to tWAS 9.0.5.3
 - Upgrade to Eclipse 2020-03
 - Add OpenJDK 14
 - Upgrade to PTT V1.0.20200728
 - Upgrade Apache Ant
 - Upgrade Apache JMeter
 - Upgrade Gradle
 - Upgrade Eclipse MAT
 - Upgrade to TMDA 4.6.7
 - Increase HealthCenter -Xmx
 - Upgrade Eclipse SWT
 - Add libertymon
 - Upgrade Request Analyzer Next
 - Upgrade WebSphere Application Server Configuration Comparison Tool
 - Add PostgreSQL
 - Add -Xnoloa to tWAS as temporary workaround for crash issue
- V11 (December 16, 2019): Add Performance Tuning Toolkit and required 32-bit libraries and XUL-Runner. Fix intermittent issue where screen lock gets wrong timeout value. Upgrade Request Metrics Analyzer.
- V10 (November 27, 2019): Add tWAS SIBExplorer and SIBPerf tools. Disable Xfce desktop tooltips. Resolve rare VNC deadlock issue. Upgrade TMDA. Add IBM Channel Framework Analyzer. Add IBM Web Server Plug-in Analyzer for WebSphere Application Server (WSPA). Add Connection and Configuration Verification Tool for SSL/TLS. Add WebSphere Application Server Configuration Visualizer.

- Add Problem Diagnostics Lab Toolkit. Add Eclipse SWT.
- V9 (November 12, 2019): Fix being unable to unlock screensaver after idling 10 minutes. Change screensaver lock time to 30 minutes. Add Totem video player and VP9/webm codec.
- V8 (November 6, 2019): Fix errors re-launching Eclipse. Add example lab data.
- V7 (November 6, 2019): Minor fix for the crash lab test if the server is restarted in an unexpected way.
- V6 (November 5, 2019): Enable tWAS and Liberty Application Security for DayTrader to use local OpenLDAP. Increase recommended Docker disk space to >100GB. Enable OpenLDAP logging. Change DayTrader7 to runtimeMode=1 to avoid WebSocket security issues calling EJBs with application security enabled. Remove OpenJDK12. Upgrade to Fedora 31. Send tWAS traffic through IHS.
- V5 (October 23, 2019): Add OpenLDAP and integrate it into tWAS. Update the lab instructions to include tWAS. Add VS Code. Add JDKs to alternatives. Update Eclipse to 2019-06. Add OpenJ9 source. Add IHS connected to tWAS.
- V4 (August 14, 2019): Add tWAS DayTrader7.
- V3 (July 2, 2019): Updates based on customer feedback.
- V2 (May 20, 2019): Convert to Docker and modernize.
- V1 (December 14, 2016): First version on VMWare.

Tip: to compare between tags; for example: <https://github.com/kgibm/dockerdebug/compare/V11...V12>

Acknowledgments

Thank you to those that helped build and test this lab:

- Hiroko Takamiya
- Andrea Pichler
- Kazuma Tanabe
- Shinichi Kako