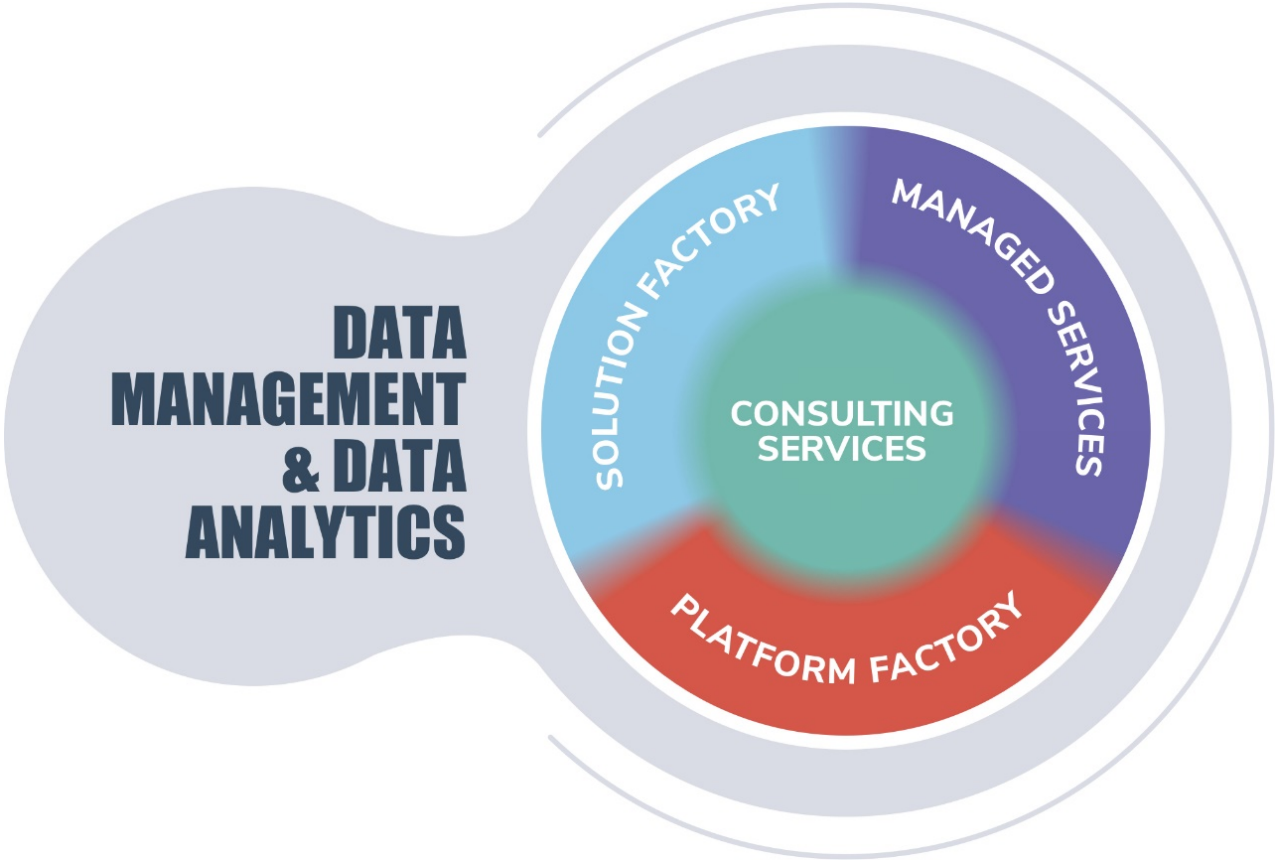
Monitoring of JVM in Docker to Diagnose Performance Issues

Jonatan Kazmierczak

@Trivadis # doag2018

trivadis
makes IT easier.

We help to generate added value from data



trivadis
makes IT easier.

With over 650 specialists and IT experts in your region

- 16 Trivadis branches and more than 650 employees
- Experience from more than 1,900 projects per year at over 800 customers
- 250 Service Level Agreements
- Over 4,000 training participants
- Research and development budget: CHF 5.0 million
- Financially self-supporting and sustainably profitable

trivadis
makes IT easier.

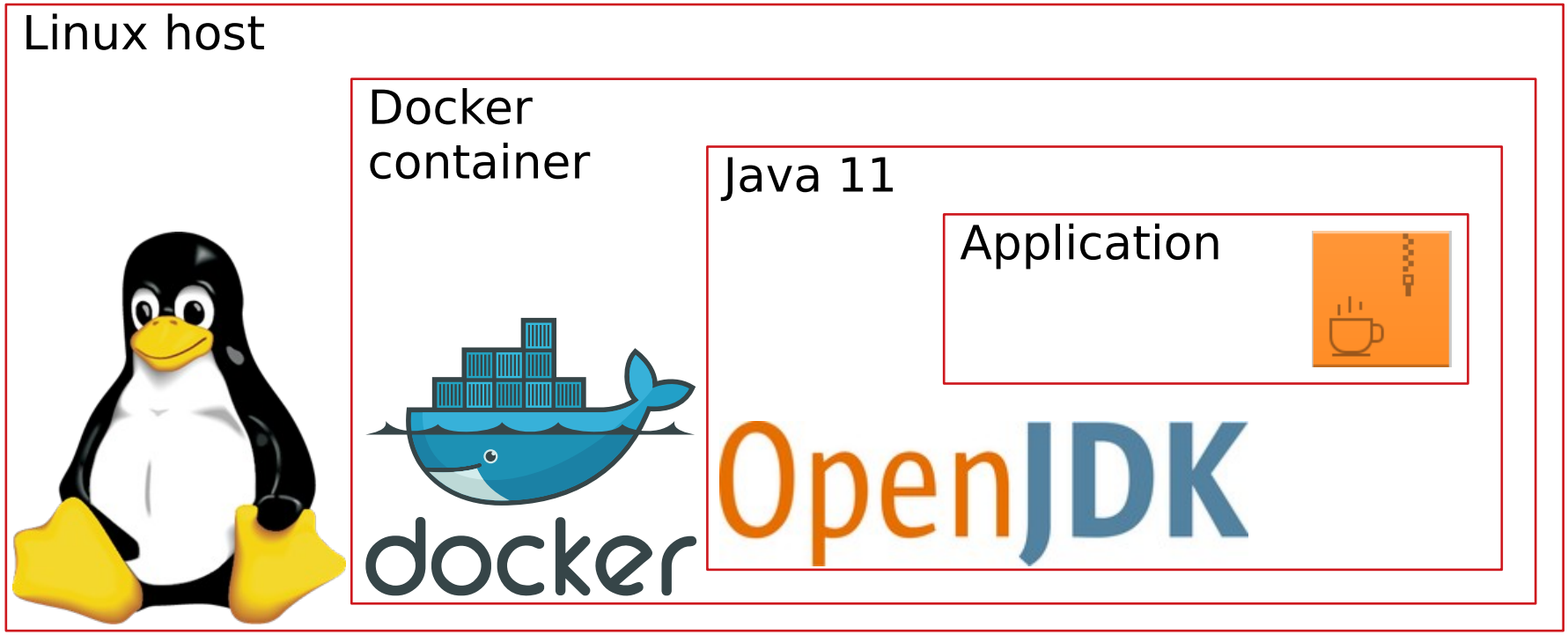
Jonatan Kazmierczak

- Senior Consultant at Trivadis
- Creator of Class Visualizer
- Contributor to Graal (JDK module)
- Uses Java for 20+ years
- Fan of chiptunes (POKEY and SID) and demos from Atari XL/XE

Agenda

- Run application having performance issues on OpenJDK 11 in Docker on Linux host
- Monitoring with OpenJDK 11 toolset: jinfo, jstat, jstack, jcmd, JFR
- Diagnostic with JMC 7
- Identification and fix of performance issues
- Summary and Q & A

Environment



Docker-vocabulary

- Image – template of deployed and configured application
- Container – running instance of an image

Discussed activities are equally relevant to Java running directly on a host (without Docker).

Get the application

```
# get the source code
git clone \
  https://github.com/jonatan-kazmierczak/simple-microservice.git

cd simple-microservice

# checkout correct branch
git checkout demo_performance_problem
```

Build Docker image containing app

```
# build the project
./gradlew build

# build a Docker image named "simple-microservice:slow"
docker build -t simple-microservice:slow .
```

Start the app in a Docker container

```
# run Docker image "simple-microservice:slow" in a new container
#   with memory limit 128 MB
#   with container's port 80 published as 3000 on the host
#   with container's /app/logs mounted at /tmp/applogs on the host

docker run \
  -m128M \
  -p 3000:80 \
  -v /tmp/applogs:/app/logs \
  simple-microservice:slow
```

Test the app

```
# measure total response time for 10 requests
time ./test_endpoints.sh 10
# Result: 0m5.261s
# Shortest single response time: 481ms

# measure total response time for 10 requests
#   sent by 2 parallel processes
time ./test_endpoints.sh 10 &
time ./test_endpoints.sh 10 &
# Result process 1: 0m9.076s
# Result process 2: 0m9.539s
# Shortest single response time: 483ms
```

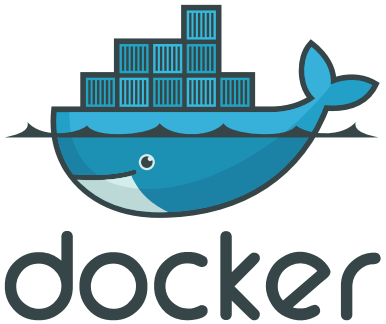
Observations – app behavior

- Very low throughput: 2 requests / second
- No parallelism

We have to do JVM monitoring
in order to diagnose observed performance issues.

About Docker container

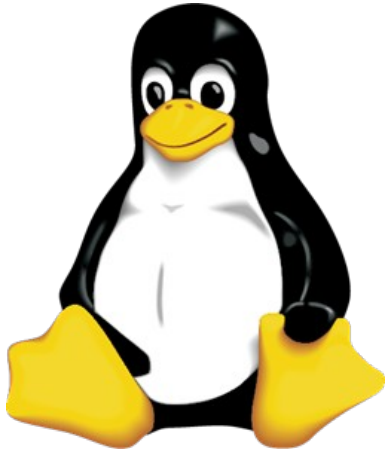
- Limitations
  - stripped down OS (Linux)
  - (possibly) stripped down JDK
  - should run only one process - in our case Java



→ not suitable for monitoring

About Linux host

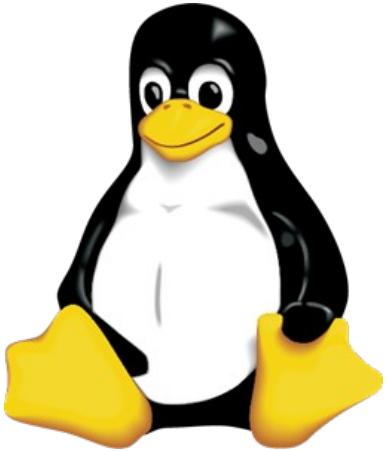
- No limitations
  - complete OS
  - complete JDK
- Executes processes from Docker containers



→ fully suitable for monitoring

How to diagnose observed performance issues?

We have to perform JVM monitoring on Linux host using tools from OpenJDK 11



OpenJDK

External monitoring

Find JVM running the app

```
# find on the host OS the PID of JVM running "simple-microservice"
ps -ef | grep java | grep simple
```

```
root      3953  3935  4 18:58 ?        00:00:04 java ...
```

Prepare shell session

```
# the JVM is executed by root
# - so we have to be logged in as root to be able to monitor it
sudo su -

# use correct JDK
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk

# save identified PID of JVM for future references
export PID=3953
```

Get basic info about JVM

```
# get basic info about JVM
$JAVA_HOME/bin/jinfo $PID
```

```
Java System Properties:
...

VM Flags:
... -XX:MaxHeapSize=6291456 ...

VM Arguments:
jvm_args: ... -Xmx5m
java_command: simple-microservice.jar 80
java_class_path (initial): simple-microservice.jar
```

Monitor heap usage and GC activities

```
# monitor heap usage and garbage collector activities
$JAVA_HOME/bin/jstat -gc $PID 2s
```

```
YGC     YGCT  FGC    FGCT  CGC    CGCT     GCT
 21    0.040    3   0.041    5   0.005   0.086
 28    0.057    7   0.206    6   0.008   0.271
 36    0.073   11   0.320    6   0.008   0.401
 44    0.091   15   0.430    6   0.008   0.529
 52    0.123   19   0.527    6   0.008   0.658
 60    0.149   23   0.582    6   0.008   0.739
 66    0.165   26   0.690    6   0.008   0.863
 74    0.186   30   0.833    6   0.008   1.027
 82    0.202   34   0.979    6   0.008   1.189
```

Perform threads dump

```
# perform threads dump
$JAVA_HOME/bin/jstack $PID
```

```
"HTTP-Dispatcher" #17 prio=5 os_prio=0 cpu=547.46ms elapsed=43.08s
tid=0x00007f82642e7000 nid=0x1a waiting on condition  [0x00007f8248dda000]
   java.lang.Thread.State: TIMED_WAITING (sleeping)
        at java.lang.Thread.sleep(java.base@11.0.1/Native Method)
        at demo.Simulator.handleRequest(Simulator.java:93)
        at demo.Simulator.lambda$new$1(Simulator.java:65)
        at demo.Simulator$$Lambda$74/0x0000000840095c40.handle(Unknown Source)
        at com.sun.net.httpserver.Filter$Chain.doFilter(jdk.httpserver@11.0.1/
Filter.java:77)
        ...
```

Internal monitoring

How to start JVM with JFR

```
# start Java
#   - with flight recorder
#   - save flight recording to disk to the given filename on JVM exit
#   - limit recording to the recent 24h
java \
-XX:+FlightRecorder \
-XX:StartFlightRecording=disk=true,filename=r.jfr,dumponexit=true,maxage=1d\
...
```

Control of JFR at runtime

```
# monitoring and control of JFR at runtime
$JAVA_HOME/bin/jcmd $PID help
```

```
JFR.check
JFR.configure
JFR.dump
JFR.start
JFR.stop
...
```

JFR start options

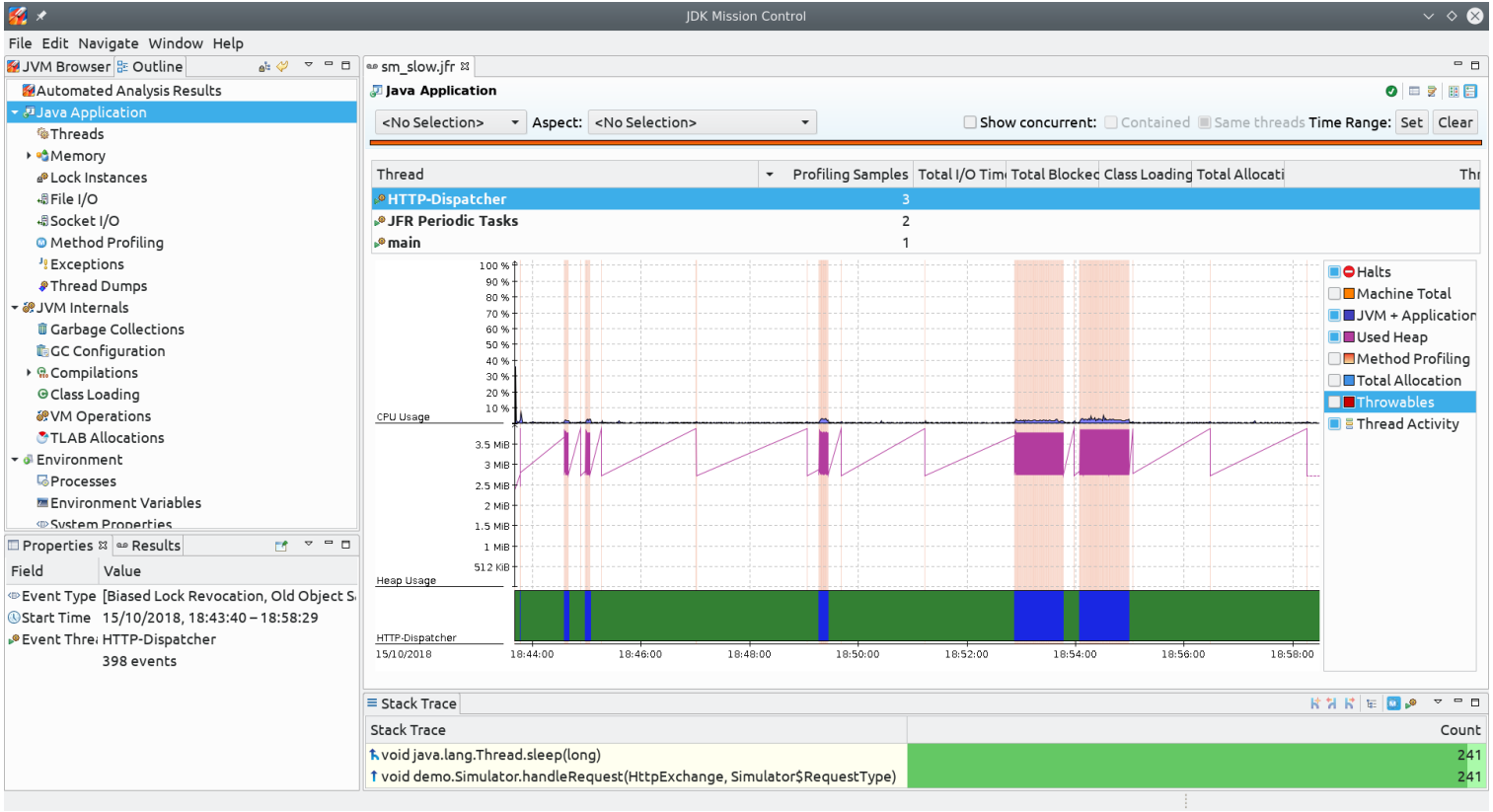
```
# documentation of JFR start options
#   - there are differences from one JDK version to another
$JAVA_HOME/bin/jcmd $PID help JFR.start
```

```
JFR.start
Starts a new JFR recording

Syntax : JFR.start [options]

Options:
...
```

Inspect flight recording in JMC (Desktop)



Identified performance issues

- Sleep in HTTP request handler
  → jstack, JFR+JMC
- Single-threaded HTTP request handler
  → jstack, JFR+JMC
- Frequent full garbage collections caused by too low heap limit
  → jstat, jinfo, JFR+JMC

Fix the app

```
# checkout git branch with fixed solution
git checkout demo_performance_problem_fixed

# build the project
./gradlew build

# build a new Docker image "simple-microservice:fixed"
docker build -t simple-microservice:fixed .
```

Take a look at Dockerfile (Docker image description)

```
FROM openjdk:11-oracle

WORKDIR /app
ADD ./build/libs/simple-microservice.jar .
ADD ./api-responses ./api-responses
RUN mkdir logs

ENV LANG en_US.UTF-8
EXPOSE 80

CMD [ "java", \
"-XX:+FlightRecorder", \
"-XX:StartFlightRecording=disk=true,filename=r.jfr,dumponexit=true,maxage=1d",\
"-jar", "simple-microservice.jar", "80" ]
```

Start fixed app in a Docker container

```
# run Docker image "simple-microservice:fixed" in a new container
#   with memory limit 128 MB
#   with container's port 80 published as 3000 on the host
#   with container's /app/logs mounted at /tmp/applogs on the host

docker run \
  -m128M \
  -p 3000:80 \
  -v /tmp/applogs:/app/logs \
  simple-microservice:fixed
```

Test fixed app

```
# measure total response time for 1000 requests
time ./test_endpoints.sh 1000
# Result: 0m4.547s
# Shortest single response time: 61µs

# measure total response time for 1000 requests
#   sent by 2 parallel processes
time ./test_endpoints.sh 1000 &
time ./test_endpoints.sh 1000 &
# Result process 1: 0m5.538s
# Result process 2: 0m5.515s
# Shortest single response time: 60µs
```

The app is fixed

Docker overhead

```
# check usage of host resources
top
```

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
19612 root      20   0  918876   7160   2292 S  89.4   0.0   1:03.84 docker-proxy
19637 root      20   0 2995200  91892  29892 S   9.2   0.6   0:33.22 java
```

Summary

- Environment:
  OpenJDK 11 in Docker on Linux host
- Monitoring with OpenJDK 11 toolset:
  jinfo, jstat, jstack, jcmd, Java Flight Recorder
- Inspection with JDK Mission Control 7

References

- Good News for us: Son-of-God.info
- OpenJDK 11 homepage: jdk.java.net/11/
- JDK Mission Control homepage: jdk.java.net/jmc/
- JDK 11 monitoring and diagnostic tools: docs.oracle.com/en/java/javase/11/troubleshoot/diagnostic-tools.html
- Java in Docker: blog.docker.com/2018/04/improved-docker-container-integration-with-java-10/

Questions & Answers

?

?

Trivadis @ DOAG 2018 #opencompany

- Booth: 3rd Floor – next to the escalator
- We share our Know how!
  Just come across, Live-Presentations and documents archive
- T-Shirts, Contest and much more
- We look forward to your visit