Performance Troubleshooting using Java Thread Dumps

Rainer Jung
kippdata informationstechnolgie GmbH

Introduction

- Rainer Jung, kippdata GmbH
- Committer and PMC member for Apache Tomcat, the Apache httpd server and Apache JMeter
- Apache Software Foundation (ASF) member
- Doing lots of performance analysis
- Providing support for Apache Tomcat, Apache httpd and other web infrastructure

Topics

- Which problem do we want to solve?
- What is a Java thread dump and how does it help?
- How does one create thread dumps?
- Real-world examples!

Note

- This talk is only about server applications written in Java
- Why is the talk in the Apache Tomcat track? Because we from the Tomcat project get often asked about Tomcat performance problems and they always turn out as application performance problems
- I want to introduce you into a useful methodology

Which problem do we want to solve?

- We are interested in solving performance problems
  - Response times are much too high

    and/or
  - Application throughput is not big enough
- For throughput driven applications (like most online applications) such problems typically result in
  - Total unavailability of the application due to backlog
- Our goal is: find the root cause!

Methodology Pros

- Out methodology will have the following nice features:
  - It works for any Java application
  - It can be used even in production
  - It typically does not need complex preparations
  - It does not need third-party software
  - The application does not need any adjustments

- Out methodology has the following limitations:
  - It only works for serious performance problems. You can't use it to improve your application performance by 10%.
  - You always need to first exclude Java memory use and garbage collection behavior as a possible root cause.
  - We don't have a GUI
    - Seriously: modern APM tools are very powerful. Our approach works ad-hoc. No preparation, fast turnaround.

- The most common root causes for performance problems are
  - Overloaded back end systems – the application waits for their response (database, middleware, web services, ...)
  - Locking problems – parts of application code are not allowed to execute in parallel
  - Bad memory sizing, memory leaks, or garbage collection configuration
  - Wrong sizing of software components (pool, caches, timeouts)

- Other common root causes for performance problems are
  - Resource constraints: CPU, OS memory, I/O, network
    - These were once well understood and not a frequent problem
    - Due to virtualisation they are back on the table as root causes
      - More precisely: virtualisation with thin provisioning, or should we say resource over-commitment …
        - But that's another topic

- First idea: what if we could trace everything that application code does?
  - Doing this naively will slow down the application so much, that we wont be able to distinguish this slowness from the performance problem we need to analyze
  - Doing it less naively will quickly lead to a demand in additional tool infrastructure and complex tool knowledge. In many cases also adjustments to the application (byte code instrumentation).

- Alternative idea: can we find out in which parts of the code the application runs for a long time or very frequently?
  - We are not really interested in fast and rare things.
- If the performance problems are serious, it would suffice to check every now and then, in which code the application is at the moment of the check?

- What is a Java thread dump?
  - A Java thread dump is a snapshot of the code executing at the moment the thread dump is taken
  - The snapshot is textual
  - It shows the execution stack of every thread in the process
  - This execution stack has the same format that you might know from the logging of an exception or throwable in a log file
  - Let's ´hava a look at an example!

- So how does a Java thread dump help?
  - When we have a performance problem, typically many parallel running threads gather in the same code places (the slow code places).
  - So look for thread stacks in the thread dump that occur many times in the same dump.
    - Start with the most frequent ones
    - But exclude threads in "idle" stacks (short stacks containing none of your custom code)

- Pros of Java thread dumps
  - All of the Pros we were looking for, plus
  - Any Java virtual machine provides Java thread dumps
  - They are fast and do not really disturb the application
  - They are textual and can be directly read and understood by humans
  - They can be aggregated by simple tools like scripts
  - They contain locking information. You will see the importance once we look at the real-world examples.

- Cons of Java thread dumps
  - For complex applications under high load they can be quite big (a few thousand lines)
  - You should make more than one thread dump to double check, whether they show consistently the same problem. Remember they are only a snapshot in time. I suggest taking 3 dumps with pauses of 3 seconds.
  - They contain no data. You can see where the code is, but not on which data it operates. But often you can see the type of data by code class and method names.

Disclaimer

- This is not a poor mans tracing. Do not dump every few milliseconds!
- There are two “dumps” in the Java world. Thread dumps, also known as stack dumps, that's the ones we are interested in. And: memory dumps, also known as heap dumps. These are totally different. When you request a thread dump from ops people make sure you tell them you do not want a memory dump.

- How does one create thread dumps?
  - First and best method, but available only on Linux/Unix: send a QUIT signal to the process: “kill -QUIT 12345”
    - This does NOT terminate the process. The JVM has a signal handler for QUIT registered. When it receives the QUIT signal, it writes a thread dump to STDOUT.
      - You do not get the thread dump back from the kill.
      - You need to redirect STDOUT to some file in the application start script to actually get access to the written dump.
        - Example: Tomcat redirects to logs/catalina.out.

- How does one create thread dumps?
  - Second method using the jstack command from a JDK installation: “/path/to/java/bin/jstack 12345”
    - The dump is send from the Java process to the jstack command, so you get it in the window where you execute jstack. So you should redirect its output: “jstack 12345 > jstack.out”
    - Output for older Java versions not as good as “kill -QUIT”, especially for Windows
  - Third method using jvisualvm from a JDK installation

- How does one create thread dumps?
  - Other methods might be available depending on the application. For example:
    - Tomcat Windows task bar icon has a context command to create a thread dump
    - There is a Java API to get all thread stacks, so one can write Java code running inside the app that generates the dump
- Not of use are methods, that only allow to look at stacks for individual threads!
  - These are too inefficient.

- Let's have a look at real-world examples.
  - I have extracted frequent stacks from thread dumps taken during performance problems
  - I have removed some lines by "..." to achieve a more dense stack feasible for a presentation screen

- What is this?

```
at java.net.SocketInputStream.socketRead0(Native Method)
...
at sun.net.www.http.HttpClient.parseHTTPHeader(HttpClient.java:687)
at sun.net.www.http.HttpClient.parseHTTP(HttpClient.java:632)
at sun.net.www.protocol.http.HttpURLConnection.getInputStream
(HttpURLConnection.java:1000)
at java.net.HttpURLConnection.getResponseCode(HttpURLConnection.java:373)
at com.provider.xyz.util.UserLogin.sendMessageToCallgate
(UserLogin.java:381)
at com.provider.xyz.util.UserLogin.transferClientData
(UserLogin.java:284)
...
at WICKET_com.provider.xyz.util.UserLogin$$EnhancerByCGLIB$
$b620ce.loginUser(<generated>)
at com.provider.xyz.panels.account.PanelLogin$3.onSubmit
(PanelLogin.java:231)
```

- Explanation

```
at java.net.SocketInputStream.socketRead0(Native Method)
```

Socket = network communication, here reading

```
...
at sun.net.www.http.HttpClient.parseHTTPHeader(HttpClient.java:687)
```

Protocol is HTTP, we are the client reading the response from a remote HTTP server

```
...
at java.net.HttpURLConnection.getResponseCode(HttpURLConnection.java:373)
```

We are actually waiting for the response code (first line)

```
at com.provider.xyz.util.UserLogin.sendMessageToCallgate
(UserLogin.java:381)
```

The remote system seems to be known as "Callgate"

```
at com.provider.xyz.util.UserLogin.transferClientData
(UserLogin.java:284)
```

The action seems to be triggered by a user login

- So: the HTTP calls to callgate during user logins are slow

- What is this?

```
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:129)
at java.net.SocketInputStream.read(SocketInputStream.java:182)
at net.xyz.util.InputStreamUtils.readLine(InputStreamUtils.java:74)
at net.xyz.rpc.RpcBase.readResponseHead(RpcBase.java:746)
...
at net.xyz.rpc.RpcService.invoke2008(RpcService.java:799)
at net.xyz.rpc.RpcService$$FastClassByCGLIB$$cc7d91e6.invoke(<generated>)
at net.sf.cglib.proxy.MethodProxy.invoke(MethodProxy.java:149)
at org.springframework.aop.framework.Cglib2AopProxy$CglibMethodInvocation.
invokeJoinpoint(Cglib2AopProxy.java:700)
...
at net.xyz.rpc.RpcServiceSkulD$$EnhancerByCGLIB$
$93e8acf0.invoke2008(<generated>)
at com.provider.xyz.rpc.BasicWrapper.invoke(BasicWrapper.java:183)
at com.provider.xyz.rpc.MBoxDWrapper.getFolderTree
(MBoxDWrapper.java:155)
at com.provider.xyz.util.BackendUtil.helpGetFolderList
(BackendUtil.java:503)
at com.provider.xyz.util.BackendUtil.getFolderList
(BackendUtil.java:438)
```

- Explanation

```
at java.net.SocketInputStream.socketRead0(Native Method)
```

again Socket = network communication, here reading

```
...
at net.xyz.rpc.RpcBase.readResponseHead(RpcBase.java:746)
```

No sign of HTTP, instead someone has named this protocol RPC (remote procedure call), we are reading (waiting for) the response head

```
...
at com.provider.xyz.rpc.MBoxDWrapper.getFolderTree
(MBoxDWrapper.java:155)
at com.provider.xyz.util.BackendUtil.helpGetFolderList
(BackendUtil.java:503)
at com.provider.xyz.util.BackendUtil.getFolderList
(BackendUtil.java:438)
```

The action seems to be triggered by a the need for some mail box (mbox) folder list.

- So: the RPC calls retrieving the mbox folder list are slow

- What is this?

```
at java.net.PlainSocketImpl.socketConnect(Native Method)

...
at java.net.Socket.connect(Socket.java:469)
...
at sun.net.www.protocol.http.HttpURLConnection.plainConnect
(HttpURLConnection.java:729)
at sun.net.www.protocol.http.HttpURLConnection.connect
(HttpURLConnection.java:654)
at sun.net.www.protocol.http.HttpURLConnection.getInputStream
(HttpURLConnection.java:977)
at java.net.HttpURLConnection.getResponseCode
(HttpURLConnection.java:373)
at com.provider.xyz.util.Utilities.isURLAccessible
(Utilities.java:1011)
```

- Explanation

```
at java.net.PlainSocketImpl.socketConnect(Native Method)
```

again Socket = network communication, but now not reading, instead connect. How long does a connect normally take? What does it mean, if we find it in our snapshot?

```
...
at sun.net.www.protocol.http.HttpURLConnection.plainConnect
(HttpURLConnection.java:729)
```

Protocol is HTTP, we are the client connecting to a remote HTTP server

```
...
at java.net.HttpURLConnection.getResponseCode
(HttpURLConnection.java:373)
```

We are actually waiting for the response code (first line)

```
at com.provider.xyz.util.Utilities.isURLAccessible
(Utilities.java:1011)
```

The action is triggered by a method named "isURLAccessible"

- So: an accessibility check for a URL sometimes hangs in a socket connect.

- More Explanation
  - Discussion with developers reveals: during user login they wanted to communicate with other social networks.
  - They observed some of the remote servers where not always available and implemented an additional availability check
  - BUT:
    - They did it synchronously during every login
    - Optimization: do it independent of login in different threads every N seconds, cache results and use them during logins

- What is this?

```
at de.acme.lib.client.connect.RemoteLogin.callServerInThread(RemoteLogin.java:867)
- waiting to lock <0x00002aab2a245410> (a de.acme.to30.service.api.To30RemoteLogin)
at de.acme.lib.client.connect.RemoteLogin.callServer(RemoteLogin.java:804)
at com.ticketing.framework.client.business.bridge.StatelessConnector.sendRequest\
   (StatelessConnector.java:58)
at com.ticketing.framework.client.business.bridge.DatasourceBridgeConnector.load\
   (DatasourceBridgeConnector.java:30)
at com.ticketing.framework.business.datasource.DatasourcePipe.load\
   (DatasourcePipe.java:30)
at com.ticketing.framework.business.datasource.CachedDatasource.load\
   (CachedDatasource.java:56)
```

Partial explanation

```
at de.acme.lib.client.connect.RemoteLogin.callServerInThread(RemoteLogin.java:867)
- waiting to lock <0x00002aab2a245410> (a
de.acme.to30.service.api.To30RemoteLogin)
```

A class named RemoteLogin executes the method callServerInThread.
That method uses a mutual exclusion lock to prevent concurrent execution of parts of its code.
Every thread showing the above "waiting to lock" line waits for some other thread to free the lock before it can acquire it and proceed execution.
If this happens a lot, it results in a performance problem!
In this case it did happen a lot, dozens of threads showed this stack!

```
at de.acme.lib.client.connect.RemoteLogin.callServer(RemoteLogin.java:804)
at com.ticketing.framework.client.business.bridge.StatelessConnector.sendRequest\
   (StatelessConnector.java:58)
at com.ticketing.framework.client.business.bridge.DatasourceBridgeConnector.load\
   (DatasourceBridgeConnector.java:30)
at com.ticketing.framework.business.datasource.DatasourcePipe.load\
   (DatasourcePipe.java:30)
at com.ticketing.framework.business.datasource.CachedDatasource.load\
   (CachedDatasource.java:56)
```

- Remaining explanation
  - Remember: `waiting to lock <0x00002aab2a245410>`
  - Look at the below stack from three other threads

```
    at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(SocketInputStream.java:129)
...
- locked <0x00002aab18e56768> (a java.io.BufferedInputStream)
at sun.net.www.http.HttpClient.parseHTTPHeader(HttpClient.java:681)
```

So this thread waits for an HTTP response form a remote server ...

```
...
at de.acme.lib.client.connect.RemoteLogin.callServerInThread\
   (RemoteLogin.java:893)
- locked <0x00002aab2a245410> (a
de.acme.to30.service.api.To30RemoteLogin)
```

… while it holds the lock and prevents other threads from also calling that remote server

- Remaining Explanation
  - Discussion with developers reveals: access to the back end server was limited on purpose to shield the back end from overload
  - Why are only three threads allowed to access to back end in parallel?
    - Configuration error!
      - That was the default setting supposed to be only used in development.
      - Someone simply forgot to adjust for production.
      - After this finding, they adjusted the value to 600 for production

- Special topic locking
  - Locking is important
    - Needed to ensure correctness
    - Not a performance problem if properly done
  - Locking can get problematic
    - If locked code is to big or more precisely takes to long to execute
      - Especially problematic: running remote calls while holding a lock (database request, HTTP request etc.)
    - If locked code is very hot, ie. is executed extremely often
  - IMHO bad locking is the number one reason for local performance problems

- Locking in thread dumps
  - Threads which are blocked while waiting for a lock can be found by searching for "- waiting to lock", "- waiting on" and "- parking to wait".
  - Threads which hold a lock and prevent others to use the same lock can be found by searching for "- locked"
    - Depending on the lock details, sometimes the thread holding the lock does not have this text on its stack. Then you might find it by comparing class and method names with the ones where the other threads are blocked
  - AI of the lines contain the unique address of the lock object

- Example for a lock in a thread dump
  - This thread already acquired a lock (owns the lock)
    - Lock is of type (class name) „java.net.SocksSocketImpl“

```
"http-28380-Processor2" daemon prio=10 tid=0x00968800 nid=0x1a
runnable ...
   java.lang.Thread.State: RUNNABLE
        at java.net.PlainSocketImpl.socketAccept(Native Method)
        at
java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
        - locked <0xf4490718> (a java.net.SocksSocketImpl)
        at
java.net.ServerSocket.implAccept(ServerSocket.java:450)
        at java.net.ServerSocket.accept(ServerSocket.java:421)
        at java.lang.Thread.run(Thread.java:626)
...
```

- Example for a lock in a thread dump
  - This thread waits for a lock
    - of type (class) „org.apache.tomcat.util.threads.ThreadPool$ControlRunnable“

```
"http-28380-Processor1" daemon prio=10 tid=0x00969800 nid=0x19
in Object.wait()
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
- waiting on <0xf4202130> (a org.apache.tomcat.util.
threads.ThreadPool$ControlRunnable)
        at java.lang.Object.wait(Object.java:484)
        at
org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run

(ThreadPool.java:687)
...
```

- Lock waits sometimes simply indicates idleness
  - An idle thread in a thread pool is typically blocked while waiting for something to do by letting it wait for a lock
  - Typical stack pattern for such non-problematic lock waits:
    - Thread stack is very short (about 10 lines)
    - Thread stack does only contain classes from the JVM, no business or framework code
  - Don't panic: you will quickly get used to these

- What is this?
  - I found many threads in this stack

```
at sun.misc.Unsafe.park(Native Method)
- parking to wait for  <0xHEXADDR> (a
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
```

So the threads wait for a lock ...

```
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await\
   (AbsractQueuedSynchronizer.java:1925
at java.util.concurrent.LinkedBlockingQueue.put\
   (LinkedBlockingQueue.java:254)
```

… while trying to add something to a queue ...

```
at org.jboss.cache.RegionImpl.registerEvictionEvent(RegionImpl.java:249)
```

… called by a JBoss cache class RegionImpl, method registerEvictionEvent

  - I'm not a JBoss cache expert, but ...

- Searching the logs I found 62646 log lines like the following:
  - org.jboss.cache.**RegionImpl**: putNodeEvent(): eviction node **event queue size is at 98% threshold** value of capacity: 200000 Region: /categories
    You will need to reduce the wakeUpIntervalSeconds parameter.
  - What happens often after 98% full? 100% full!
  - How does a queue behave once it is full and I want to add more items? The queue was a LinkedBlockingQueue. So: it blocks. That's what we see.
    - Either the queue is to small, or more likely the queued entries are not processed quickly enough

- Real-world examples summary
  - Looking at thread dumps we can interprete what the code does without knowing the code in advance
  - Most class names and method names are quite specific
  - To analyze performance problems, the most important part of the stack is the top 5-10 lines

- Methodology for analyzing thread dumps
  - Take a few dumps instead of just one, they are cheap
  - Concentrate on stacks that occur for many threads
    - One can write a simple text processing script to produce some statitics (most frequent stack, second frequent stack , …)
  - Concentrate on the top 5-10 lines of the stacks
  - Ignore the "idle thread" stacks
  - Look for network communication and lock waits

- What's next?
  - Create a few thread dumps of your favorite application right now
    - Dare to do it even in production
  - Look at them and familiarize yourself with the contents, even when there is no performance problem right now
  - Share your dumps and findings. Thread dumps enable joint analysis between devs and ops.
  - Include taking thread dumps in the stop method of your shutdown scripts. Thus you'll get good post-mortem information in case of emergency restarts.

- Thread dumps and Apache Tomcat
  - Everything we said of course applies to applications running in Tomcat
  - There's two nice additions:
    - Getting a thread dump via browser from the tomcat manager servlet with the URI /manager/text/threaddump
      - A bit more expensive than “kill -QUIT” and “jstack”, but still very useful in case of mergency
    - The StuckThreadDetectionvalve: automatically log threads stacks whenever a request takes longer to process than a configured threshold
- Demo

Questions?

- Hopefully time for questions …
  - … or send them to rainer.jung@kippdata.de