Troubleshooting Frozen Deployed Instances

This article was written by Andrew Lindesay (http://www.lindesay.co.nz) around February 2005. It first appeared as LaTeX PDF and has been transcribed into this Wiki. You use the information contained in this document at your own risk. Please contact the author if you feel there may have been an error in the conversion to Wiki markup.

Contents

Applicability

The material discussed here has been used with WebObjects 5 and Java 1.4 on MacOS-X Server. It may or may not work on older or newer versions of WebObjects or Java. It is strongly suggested that you test this on a non-production server first. Note that there may be security issues with this technique if your system is exposed on the internet.

Introduction

Java provides the substrate upon which WebObjects 5 applications operate. Java has it's good and bad points. One good point is the ease with which one can achieve threaded operation of a software system. This can often be used to make the most of your hardware assets, but also means you need to ensure that things are locked to prevent two threads getting at the same thing at the same time. Locking failures can lead to data getting damaged or when one thread won't let a lock go for some reason, other threads can be left waiting indefinitely.

When "things go wrong" in a multithreaded system, obtaining a solid diagnosis can be arduous. Problems are often very hard to repeat, can be very sporadic and are often infrequent. It may require a very specific series of events with very specific timing to reproduce an issue and often these issues are hard to reproduce in "the lab". In a production environment such bugs are liable to "freeze" up instances of your application and eventually render the live system unresponsive.

It is also possible that more banal errors such as infinite loops and the like can sneak into production systems and cause unresponsive instances as well.

This article focuses on a technique that can be used to ascertain what is going on inside a frozen instance in a production environment. In particular this technique will obtain for you a stack-trace of the instances' threads. Armed with this information, you are much better placed to diagnose the issue and fix it quickly.

Before you Start

A WebObjects system which is deployed in the traditional manner consists of a number of copies of the program running separately, each carrying some of the inbound load from users. Each of these 'copies' is termed an instance. The `SiteConfig.xml` file defines the instances. This configuration file is located at the following place in your MacOS-X Server's file system.

```
/Library/WebObjects/Configuration/SiteConfig.xml
```

Before you modify it, make a backup of the `SiteConfig.xml` file in case anything goes wrong.

Setup

The instances are modified such that they are able to be connected to remotely using the `jdb` debugging tool. Some "additional arguments" need to be inserted into the configuration in the `SiteConfig.xml` file for each instance in order to achieve this. These additional arguments are inserted as shown below in the text of the element `additionalArgs`. You will need to choose a different address for each instance - choose addresses from 8000 - 8999. This is a TCP/IP port. Note that the additional arguments should all appear on one continuous line. The author has split this up here to improve readability.

```
...
<instanceArray type="NSArray">
  <element type="NSDictionary">
    <id type="NSNumber">1</id>
    <port type="NSNumber">2001</port>
    <cachingEnabled type="NSString">YES</cachingEnabled>
    <additionalArgs type="NSString">
      -Xdebug
      -Xrunjdwp:transport=dt_socket,address=8121,server=y,suspend=n
    </additionalArgs>
...
  </element>
```

The `element` tag will repeat here for all the instances inside the `instanceArray` tag.

The `id` tag gives the instance number and you need to remember the mapping from the instance number to the `address` in the additional arguments. Jot this information down on a piece of paper. For example, one can see above that the instance 1 is mapped to address 8121.

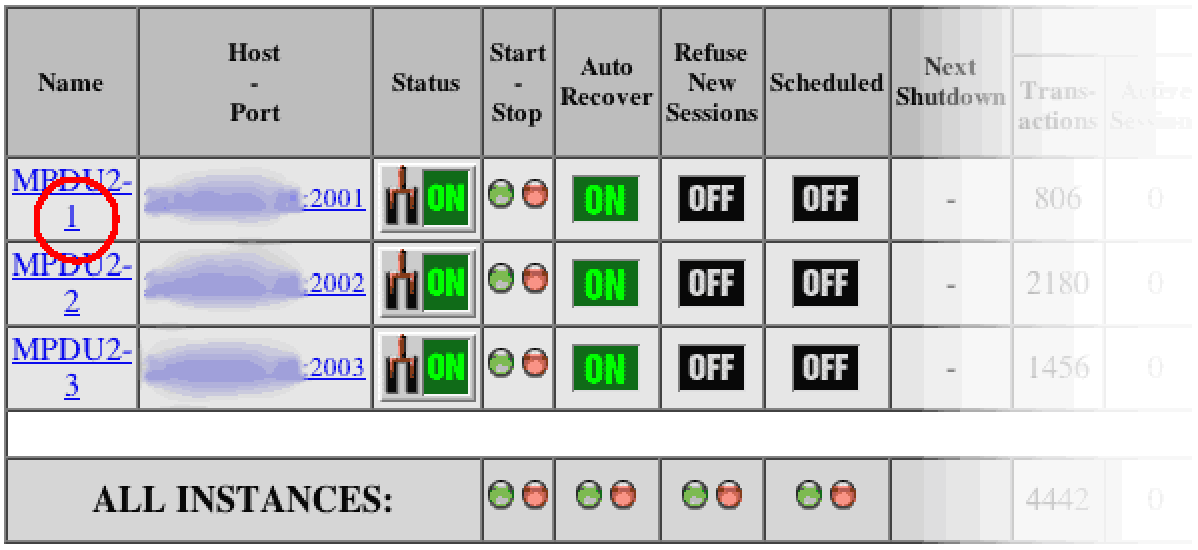
Now restart your instances.

Note that some JVM's on other platforms expect the "jdwp" to be specified as follows;

```
-agentlib:jdwp=transport=dt_socket,address=8121,server=y,suspend
```

When Something Goes Wrong...

The instances are listed in the JavaMonitor. A screenshot is shown below with the instance number circled in red. You need to first identify which instance has frozen.



Name	Host - Port	Status	Start - Stop	Auto Recover	Refuse New Sessions	Scheduled	Next Shutdown	Trans-actions	Active Sessions
MPDU2-1	[redacted]:2001	ON	🟢 🔴	ON	OFF	OFF	-	806	0
MPDU2-2	[redacted]:2002	ON	🟢 🔴	ON	OFF	OFF	-	2180	0
MPDU2-3	[redacted]:2003	ON	🟢 🔴	ON	OFF	OFF	-	1456	0
ALL INSTANCES:			🟢 🔴	🟢 🔴	🟢 🔴	🟢 🔴		4442	0

Once you have the instance number, using your instance to address mapping from the setup, identify the address you want to connect to.

Use the `jdb` command line tool that comes with the java environment to connect to the instance and debug it. To do this, enter a command of the following form on the application server.

```
fooserver$ jdb -attach 8121
```

If you want to debug a remote machine you can use the following command.

```
foodev$ jdb -attach woserverhost:8121
```

This may be useful in situations where your WebObjects application server does not have the `jdb` tool installed and so you need to run the jdb tool from a host where the `jdb` tool is installed.

You will now be using the java debugger. There are a slew of commands that can help you work with the debugged java system, but this article is just going to focus on getting the thread stack traces. Issue the command `suspend` to freeze all of the threads so they can be dumped and then the command `where all` in order to get all the stack traces of the threads. Finally when you wish to resume the threads again, issue the command `resume`. You're advised to actually quit the `jdb` environment as soon as you have the information you need.

What to Look For

`jdb`'s `watch all` command will give you a stack trace for all threads in your app. But your process needs first to be suspended in order to get a coherent stack trace. Use it like so:

```
> suspend
All threads suspended.
> where all
...
> resume
All threads resumed.
>
```

An example of a stack trace is shown below. You'll notice the java class name and source-code line number at the end of a particular entry in the stack. Here we can see that the thread called "WorkerThread103" has stuck trying to get a session from the session store. In this situation another thread will most likely have the session store locked and is not releasing the lock.

```
WorkerThread103:
  [1] java.lang.Object.wait (native method)
  [2] java.lang.Object.wait (Object.java:429)
  [3] com.webobjects.appserver.WOSessionStore.checkOutSessionWithID
(WOSessionStore.java:207)
  [4] com.webobjects.appserver.WOApplication.restoreSessionWithID
(WOApplication.java:1,546)
  [5]
com.webobjects.appserver._private.WOComponentRequestHandler._dispatchWithP
reparedApplication
(WOComponentRequestHandler.java:314)
  [6]
com.webobjects.appserver._private.WOComponentRequestHandler._handleRequest
(WOComponentRequestHandler.java:358)
  [7]
com.webobjects.appserver._private.WOComponentRequestHandler.handleRequest
(WOComponentRequestHandler.java:432)
  [8] com.webobjects.appserver.WOApplication.dispatchRequest
(WOApplication.java:1,306)
  [9] nz.co.orcon.osm.webobjects.Application.dispatchRequest
(Application.java:428)
  [10] com.webobjects.appserver._private.WOWorkerThread.runOnce
(WOWorkerThread.java:173)
  [11] com.webobjects.appserver._private.WOWorkerThread.run
(WOWorkerThread.java:254)
  [12] java.lang.Thread.run (Thread.java:552)

WorkerThread101:
  [1] java.net.PlainSocketImpl.accept (PlainSocketImpl.java:351)
  [2] java.net.ServerSocket.implAccept (ServerSocket.java:448)
  [3] java.net.ServerSocket.accept (ServerSocket.java:419)
  [4] com.webobjects.appserver._private.WOWorkerThread.run
(WOWorkerThread.java:238)
  [5] java.lang.Thread.run (Thread.java:552)
...
```

If you have a look at what the other threads are doing at the same time, it is hopefully possible to ascertain what area might be at fault. At the very least, one can figure out what part of the application is at fault.

Conclusion

Despite the simplicity of this approach, it provides for a means by which you can find out what is going on inside frozen instances rather than playing laborious guessing games.

Alternative Approaches

http://www.gvcsitemaker.com/gvc.webobjects/faq&mode=single&recordID=41413