JBossProfiler Guide (Under review)

2006, Clebert Suconic at JBoss Inc (Under review)

Table of Contents

Acknowledgements

I have had an amazing support from my wife, who encourage me investing part of my time in "the open source dream". I wouldn't be able to create this without her support.

But at a first place also, I have to thank God who gave me the idea during a prayer time. Yes, I'm religious although I don't like the definition of the word religion. Someone told me that Thomas Edison always did that before his investions. I only tried to immitate him on that nice idea. I made a promise that I would say that in my acknoledgements at first moment I had the idea.

Preface

JBossProfiler uses an innovative way of profilling. It uses log files instead of direct communication between the analysis tool and the interceptor layer (the agent responsible to hook between method calls).

Because of this approach, instead of having to turn of firewal rules, to enable graphical terminals and other factors found in production or QA systems, this gives more flexibility when analyzing applications in high end servers.

Thinking about these difficulties we have constructed the analysis toold as a WEB application. It would be pretty easy to create a Swing application that analyses that data, but we have choosed primarly to support the WEB application for those reasons. Of course if someone wants to contribute with a swing version, we would be more than pleased in accept such contribution.

1

Concept

1.1. How it Works

The idea is simple, an interceptor sends events to a LOG file. Another application (WEB application) reads its contents and shows analysis.

Most of the work is done by the Analysis tool, and that's a good thing as the work of analyzing the application is done outside of the JVM's runtime. What I have seen from users is that this increases the possibility of analysis and the user ends up capturing more information.

Also, the idea of the interceptor is to stay "sleeping" until someone wakes him up. This is always done by MBeans and there is a MBean that manages each interceptor.

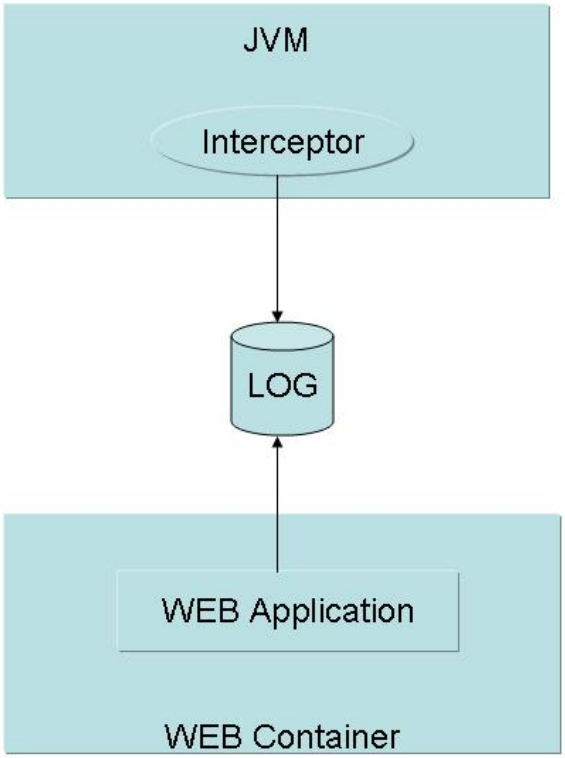

Figure 1.1. Flow of Log Files

1.2. JVMPI - Java Virtual Machine Profiler Interface

JVMPI is the interface defined for profiler events until JVM 1.4. It's still supported in JVM 1.5, but will be removed soon. JVMPI is an experimental C interface where you can capture events like enter method, exit method, class-loading.

When the interceptor is active, these events are saved in binary files for later processing:

- ClassLoad
- Enter Method / Enter Exit
- Object Alloc/Object Release
- GCStart/GCFinish

1.2.1. Options

- Add the -XrunjbossInspector:<directory>,<options...> as a parameter to your JVM. (Sun's or IBM's JVM)

```
Possible options include:
   start=<prefix name>
   include=<prefix name>
   ignore=<prefix name>
   socket=<server|IP>:<port>
   uniqueNames=true
   wakeupOnStartup=true
      Start the profiler always after the JVM start.
      This is useful for running testcases.
   memory=true|false (default=true)
      Disable ObjectAlloc and ObjectRelease events.
   tracer=true|false (default=false)
      Enable MethodEntry and MethodExit events of Exception and Erorr classes
```

To weak up this interceptor you would have to use the MBean

1.3. JVMTI - Java Virtual Machine Tool Interface

JVMTI is the replacement for JVMTI after Java 5.

We will re implement events defined at JVMPI(Section 1.2), but for now we are only using this to capture memory

profiler snapshots (Section 3.2)

1.3.1. Options

- Add the -agentlib:jbossAgent

```
There are no options at this time
```

The MBean responsible for the control of the life cycle for the memory profiler has also some utility methods

- forceGarbageCollector
- getLoadedClasses
- getObjects(Class)

1.4. How it's compared to commercial tools

The first thing that differs JBossProfiler is its conception based in LogFiles instead of wired protocols. This gives the tool more power plus enabling remote teams to analyze remote scenarios with minimal setup on servers.

Usually I say that JBossProfiler has **enough** features to find bottlenecks.

Features that both JBossProfiler and Commercial Tools will have:

- Tree Navigation over the graph of callings
- Graph visualization of the graph of callings
- % Identification of Bottlenecks

There are of course features unique to JBossProfiler. Original ideas:

- LogFile based

  This is idea to support teams, where the problem is away from your analysis tool.

  Also, this seems to increase the power of analysis. Usually you can analyze more data with JBossProfiler.

- Memory Profiler based in Snapshot using only a server MBean

  A commercial profiler requires a front-end installed in order to create the snapshot.

- Object allocations and Object Releases referenced to methods

  I haven't seen that feature in any other profiler

Usually commercial profilers are developer's tools, and of course they have more developers features like nice integrations with IDEs and Debuggers and views that can and sell lot of functionality. Well, we don't want to compete with commercial profilers but we want to offer a nice view about what is happening with your code and understand what is happening.

Sometimes of course is more comfortable to use commercial tools but you can always have the information you need with JBossProfiler

2

Runtime Profiler

2.1. Events covered during runtime analysis

When using the Runtime Profiler, you are interested in the behavior of your methods during execution.

We extract information about method calls showing % information relative to CPU and elapsed time. We also show information about memory consumption from these methods.

Say, if a method allocates an object, you will have an enter-method event, and a objectalloc during its execution.

Later on, when you have a GC operation we will have release objects events. As we identified the object during its creation we can determine where the object was created, so we can show if a method is generating leaks or not.

2.2. Capturing Log Files

The interceptor stays sleeping, not consuming any resources from the JVM until it receives a weak-up method.

This is done by the MBean at the WebConsole. (e.g. http://localhost:8080/jmx-console).

You have to look for the bean mbean=Native-profiler

After you have selected the MBean you will be able to:

- pause - Temporarily stops data collection
- stop - Definitely close all the files

  After this point, you can't collect more data and you need to restart your application server if you want to collect more.

  After this point also, the application keeps running.
- activate - Start/Resume capturing data

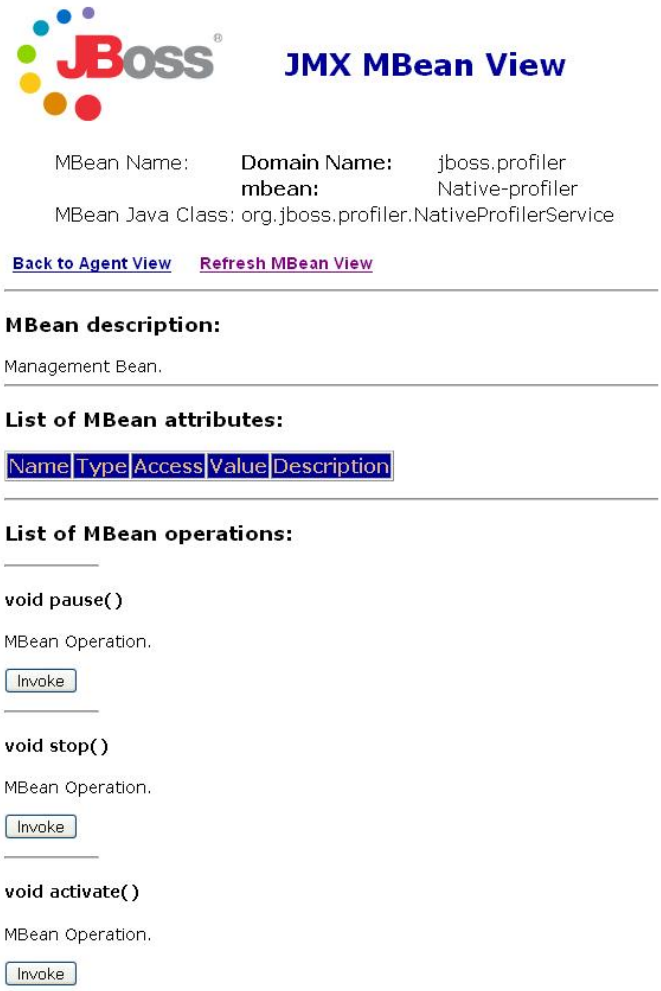

Figure 2.1.

2.3. Installing the MBean

To install the Native MBean you need to copy jboss-profiler-noAOP.sar into your /server/deploy directory

2.4. Installing the Native Library (DLL/SO)

The only thing needed to have access to the native library, is to have defined LD_LIBRARY_PATH (if Linux) or under the PATH (for Windows).

Although sometimes is hard to distribute native code under Linux. On that case you would have to recompile the native code in your library.

You can test if the installation works by doing a simple:

```
java -XrunjbossInspector:/tmp Foo

If the only error message you get is Class Foo not found, your native library installation is working
```

2.4.1. Compiling the Native Library

You have of course to download the source code from CVS first.

After you have download it, you need to:

- Make sure you gnu/gcc works
- Define JAVA_HOME to a JDK as we need access to some includes
- Under <jboss-profiler-src>/native/<platform>, execute the compilation script

2.5. Start/Stop/Pause using the API

It is possible to directly use a org.jboss.profiler.threadcheck.StartupProfilerCheck to control the life cycle of your profiling.

StartupProfilerCheck is defined at profilerConsole.jar. You have three methods defined:

```
private native static void startProfilerInternal();
private native static void pauseProfilerInternal();
private native static void stopProfilerInternal();
```

These three methods are defined at jbossInspector.dll (or .so for Solaris or .jnilib for apple)

3

Memory Profiler

3.1. Concept

The concept of the memory profiler is almost the same as the runtime profiler. The only difference is that you can take several snapshots of your memory during an execution.

You can extract a snapshot by using the MBean(???)

The CPU is frozen while the snapshot is being extracted, what usually takes

3.2. Capturing a Snapshot

Steps to extract a snapshot:

- Add jbossAgent.dll to your path or libjbossAgent.so to your LD_LIBRARY_PATH

- Add -agentlib:jbossAgent to your JVM 5 command line. (Java 5 at least is required as JVMTI was introduced on Java 5)

- Go to JMX Console (http://localhost:8080/jmx-console) and look for JBossProfiler:JVMTI MBean (mbean=JVMTIClass)

- Call heapSnapshot from JVMTIClass MBean. use P1 as a prefix, and P2 as the extension. For instance P1=/tmp/log, and P2=mem. This will create three files starting with log under /tmp, with extension=.mem

- Use the JbossProfiler application (http://localhost:8080/jboss-profiler) after you have installed jboss-profiler.war to your /deploy directory.

3.3. File formats

All the files used in the memory profiler are simple text files, CVS like.

This way you can write your own tool to analyze the snapshot generated.

3.3.1. Classes

- tagClass – An unique count id for the class

- signature – The JNI signature for the class
- tagClassLoader – The unique object id for the classLoader

Example:

```
tagClass,signature,tagClassLoader
1,Ljava/io/BufferedWriter;,0
2,Ljava/util/Collections$ReverseComparator;,0
3,Ljava/lang/StringCoding$StringDecoder;,0
```

3.3.2. Objects

- ObjectTag – The Unique Count Id of an object
- ClassTag – The Unique Count Id of the declaring class for this object
- Size – The size in bytes for this object

Example:

```
objectTag,classTag,size
1338,59,480
1339,106,88
1340,106,88
1341,106,88
```

3.3.3. References

- tagReferrer – ObjectTag for the object owning a reference
- tagReferred – ObjectTag for the object referenced
- index – JVMTI sends this count that I didn't find any usage for this. It’s here for future usage.

Example:

```
tagReferrer, tagReferee,index
0,134,0
0,168,0
0,16,0
0,270,0
```

4

Webinterface

4.1. Runtime Profiler

4.1.1. Running Application

To start analyzing after the deployment, open this URL into your WEB-Browser.

http://localhost:8080/jboss-profiler

Figure 4.1.

You will have to type the directory where you added the log files. It was not possible to create a select button as this is a Web application. The good thing about a WebApplication for a profiler, is that you can have people analyzing data remotely. (This is a good tool for consultants. support staff and architects that may not be on site)

4.1.2. Selecting the Process ID

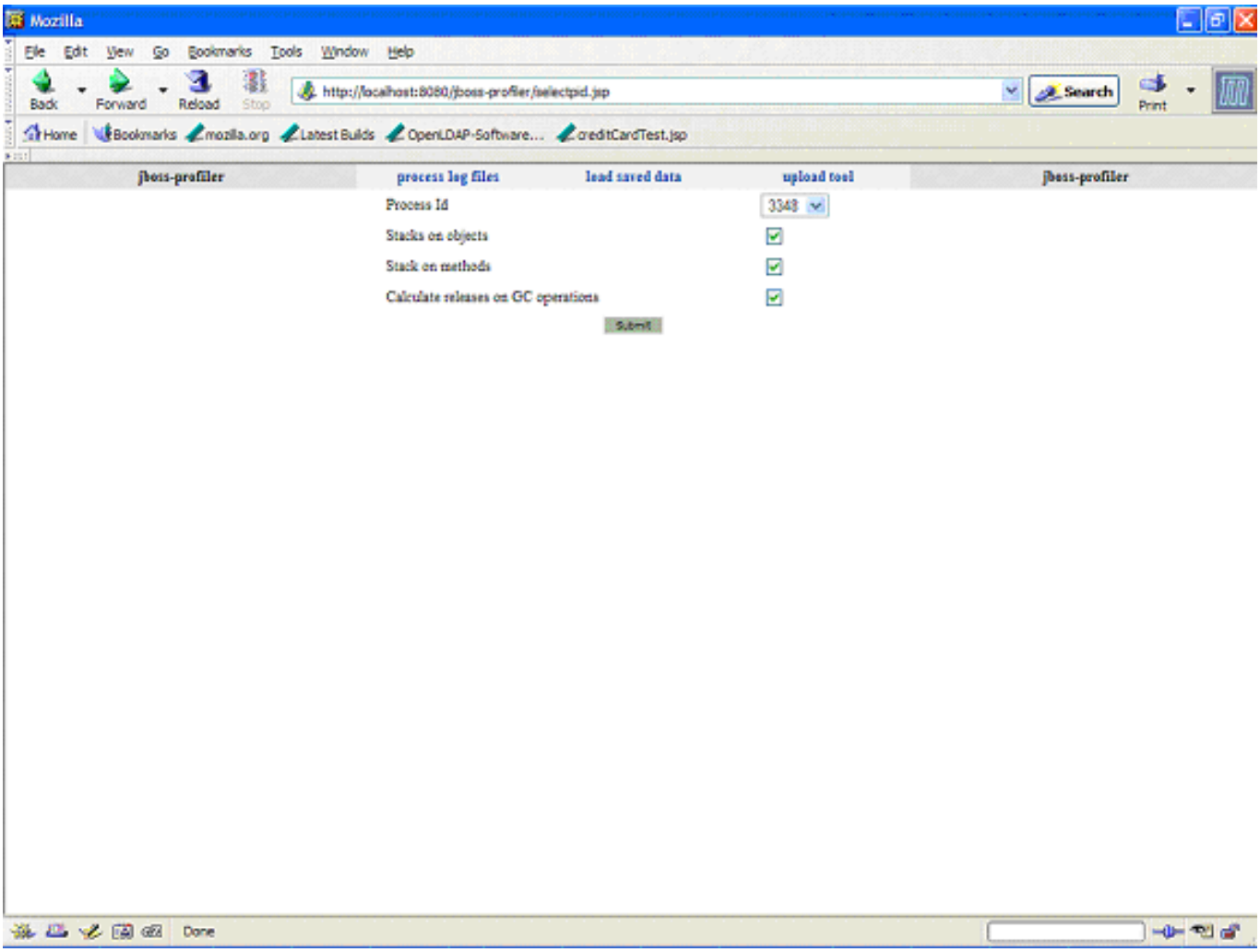

Figure 4.2.

Besides the option to select any process id you might need, you have also three extra options that will control how to analyze the log-files into profiler's object-model:

- Stack on objects

  You can locate where objects are created if this feature is enabled

- Stack on methods

  You can generate the iteration between methods callings. If not activated you won't have reference information

between methods.

- Calculate releases on GC operations

  If activated a complete reference between object ids and methods ID. It's a good option to locate memory-leaks

After submit this following processing screen will be viewed.

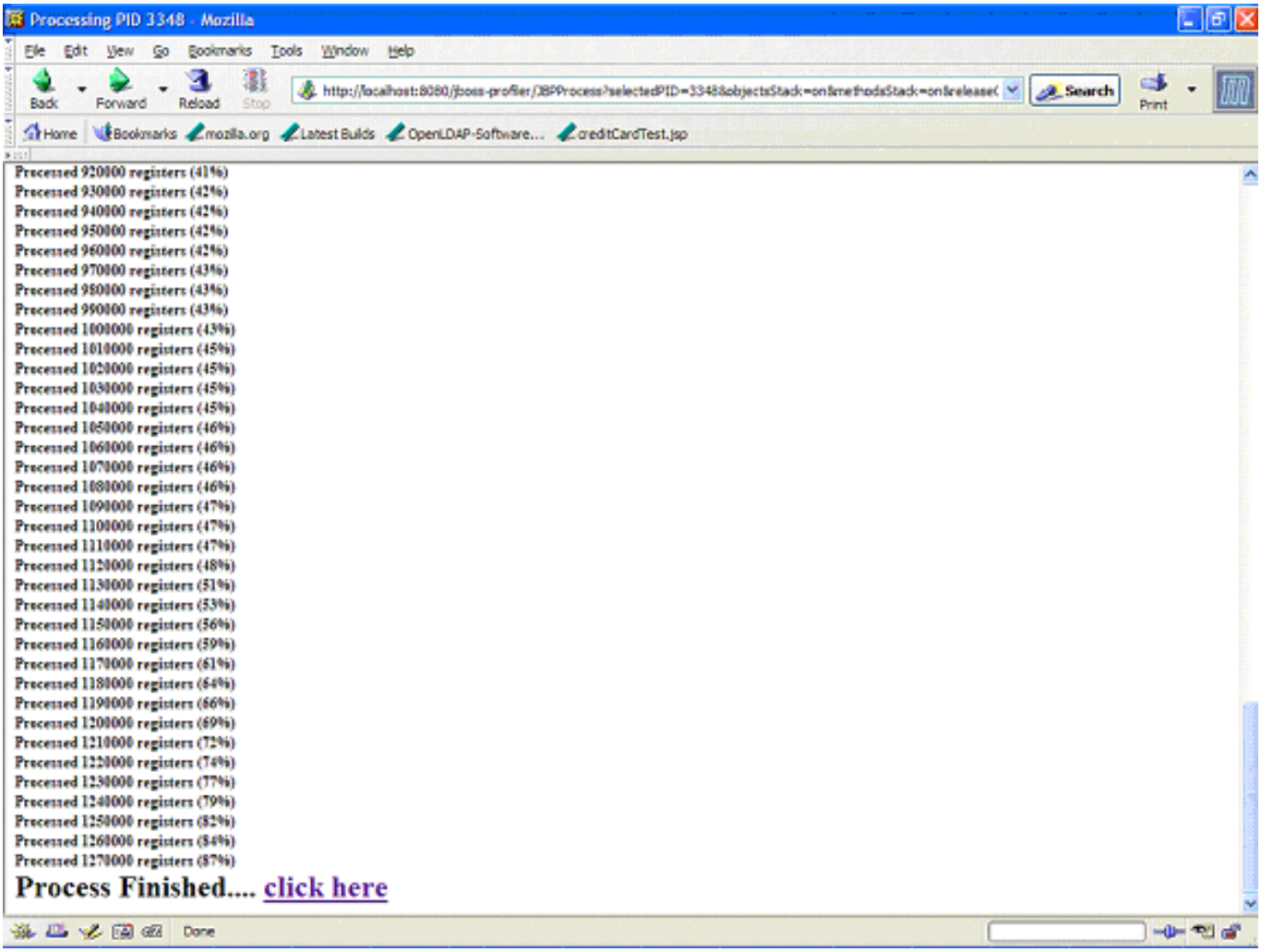

Figure 4.3.

Follow the link, and you will see Process-View/Methods view as default: You can have this view by clicking on ProcessView-methods.

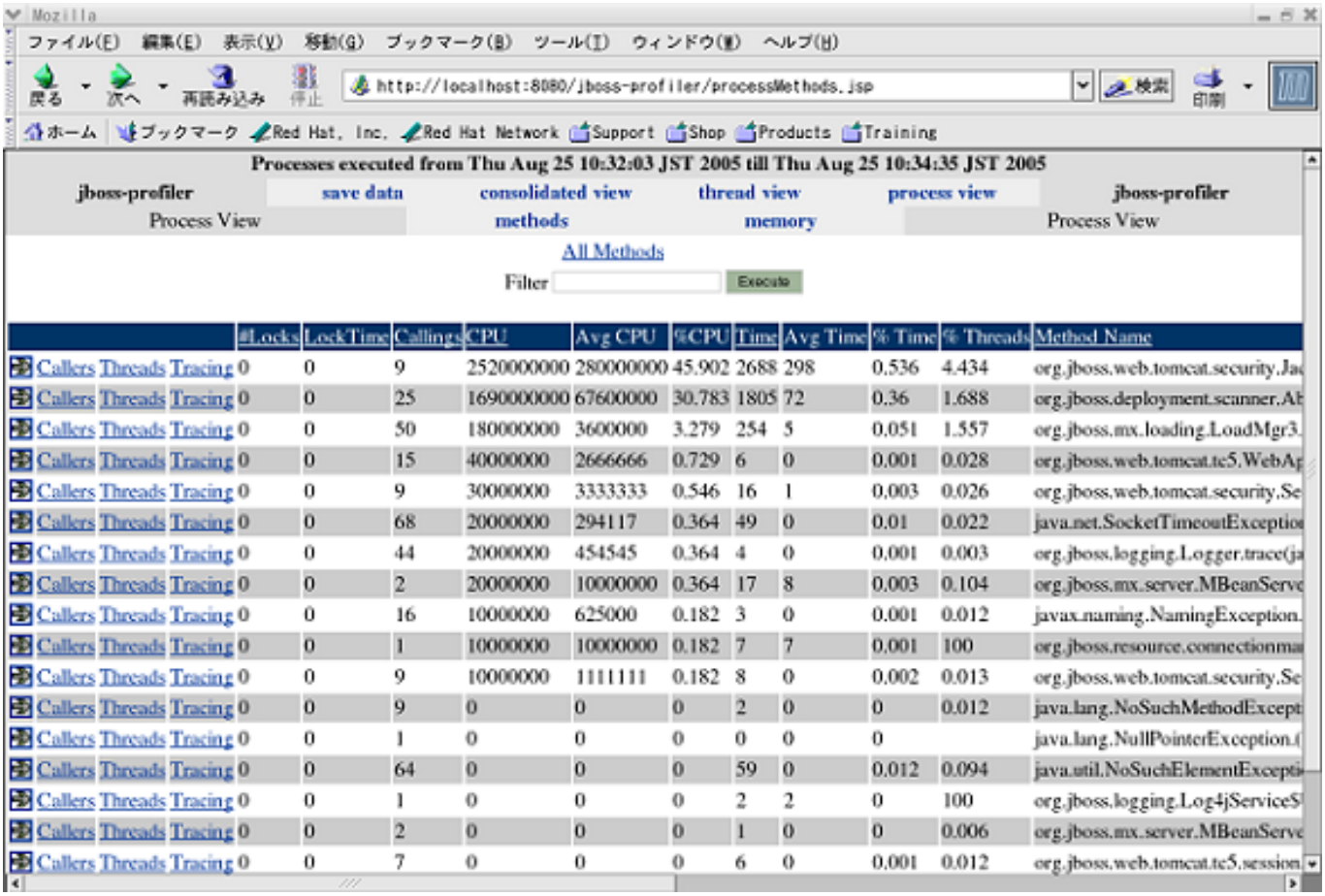

Figure 4.4.

You can proceed from this view to each of steps described in the following sections respectively.

4.1.3. Detailing method’s execution

You can click on the arrow icon to detail method’s execution: You can repeat that operation as many times you want inside method’s execution. Clicking on method’s name (a hyperlink) has the same effect.

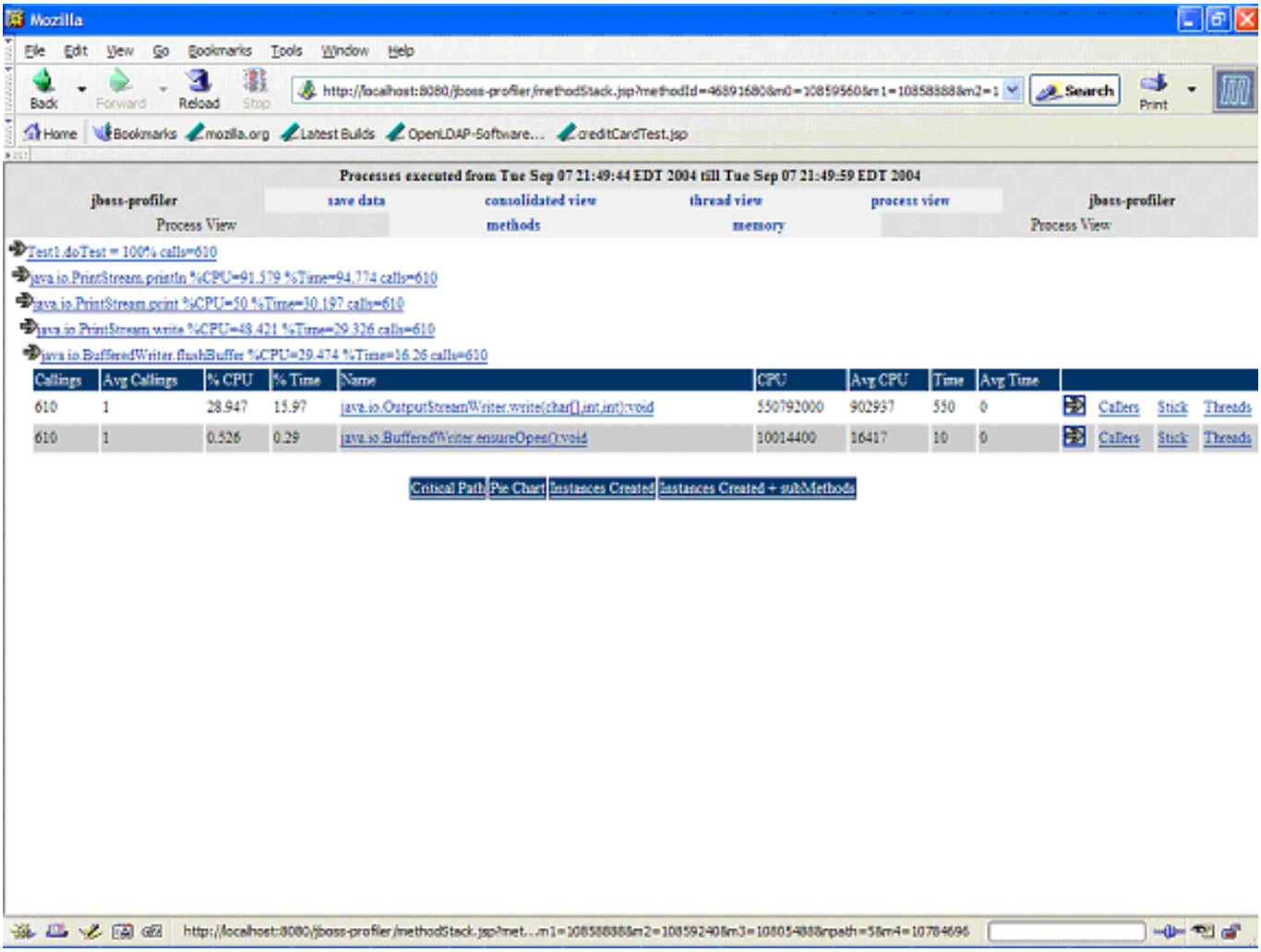

Figure 4.5.

After you got on methods detail, you can have four views about the current graph:

- Critical Path

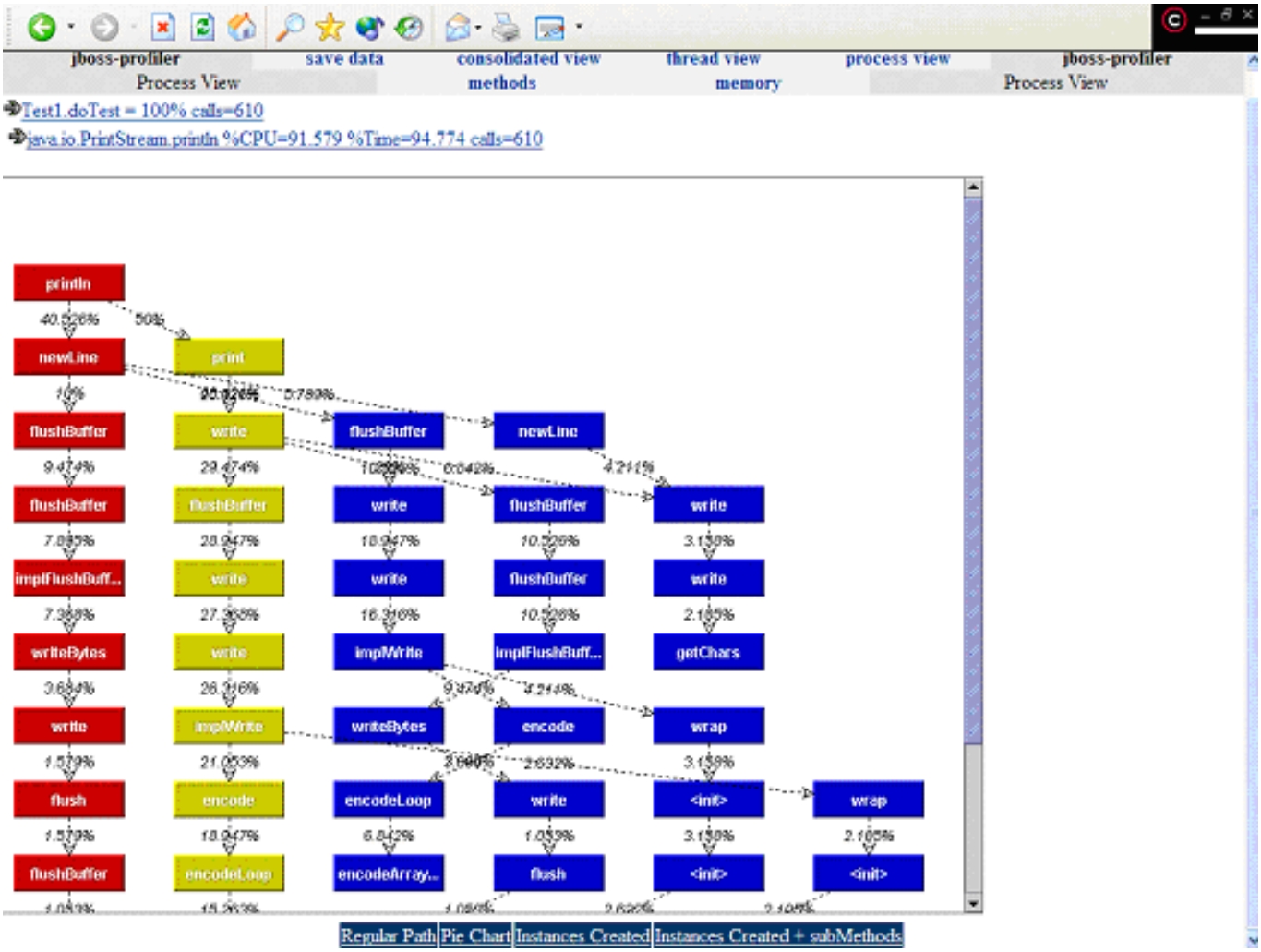

Figure 4.6.

- Pie Chart

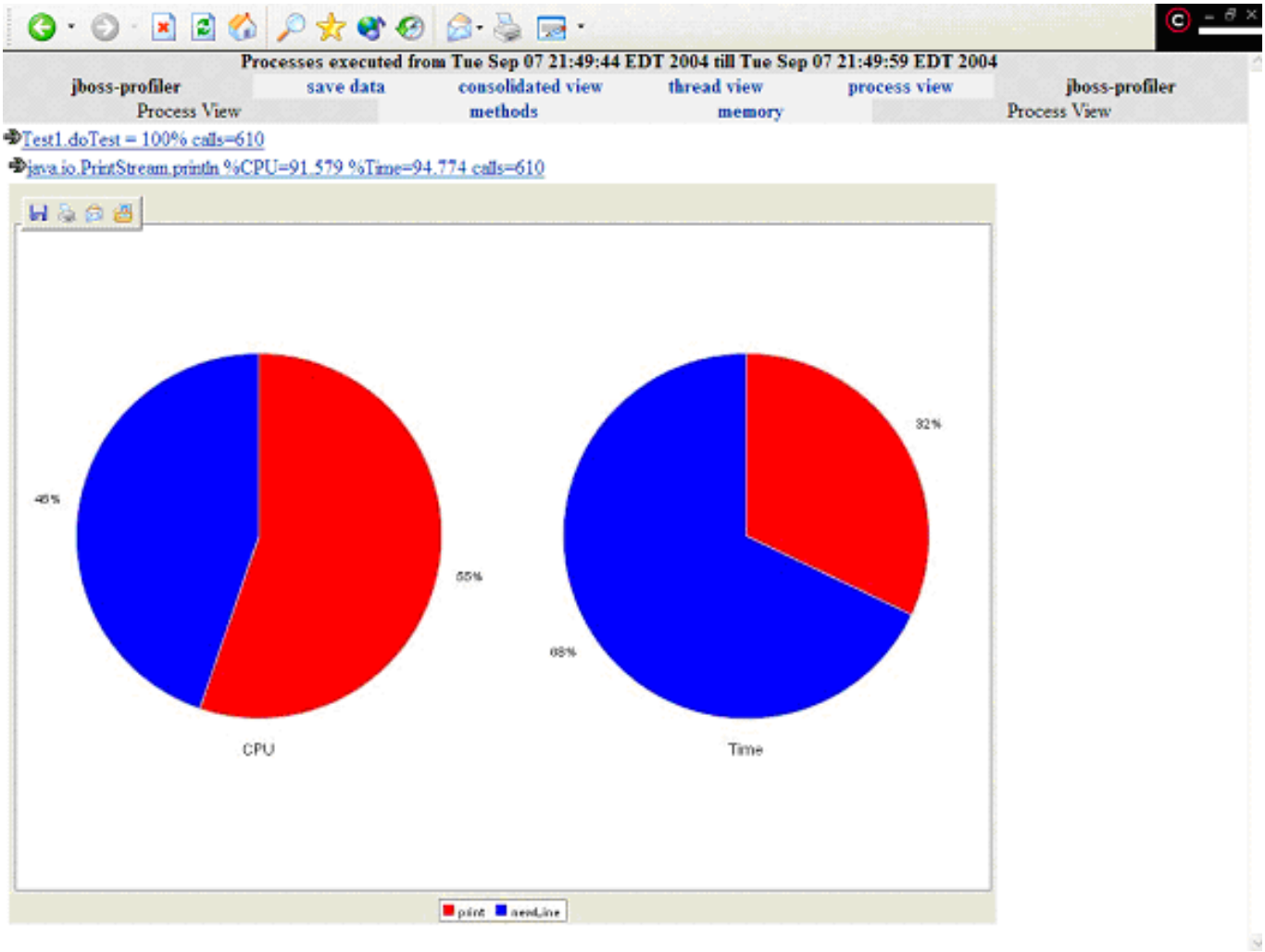

Figure 4.7.

- Instances Created:

  (There is an extra view that adds sub-methods events)

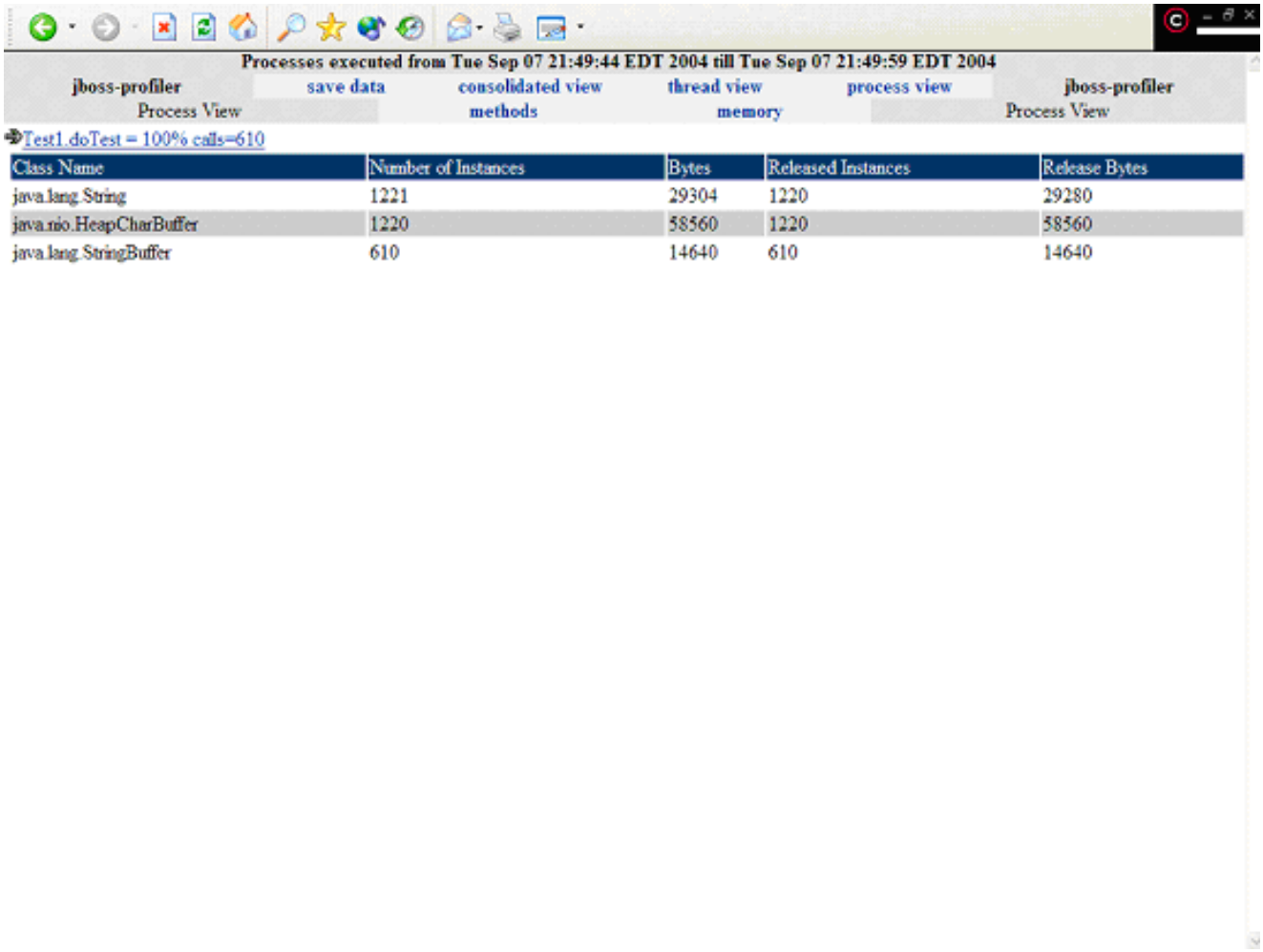

Figure 4.8.

On this view you have information about how many objects were created and **released** on that slice of the graph. You have a **complete** information about memory-leaks.

- Memory operations (ProcessView->Memory)

  On Process-View/Memory you can look at GC operations:

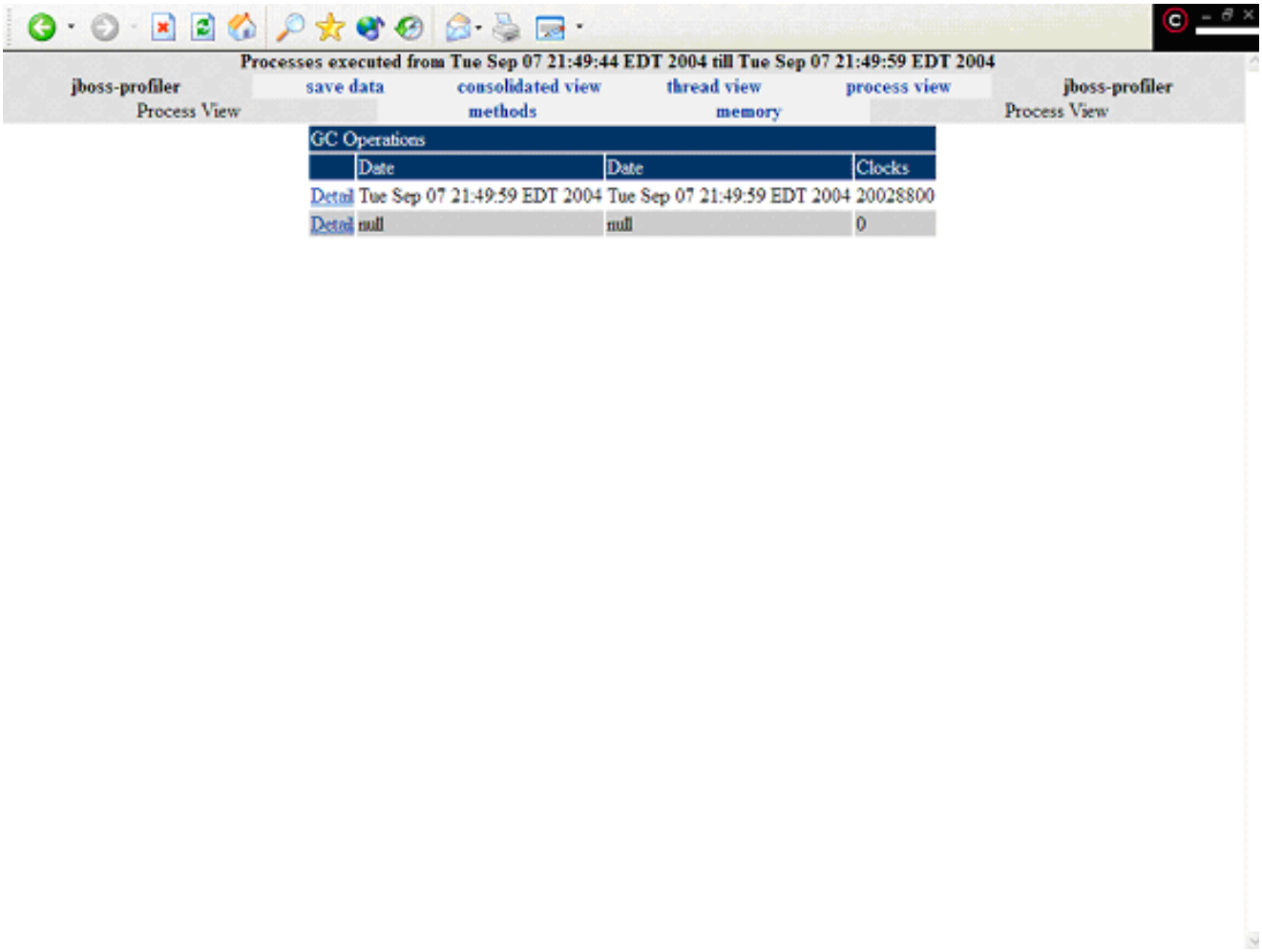

Figure 4.9.

You can detail a GC operation:

Byou

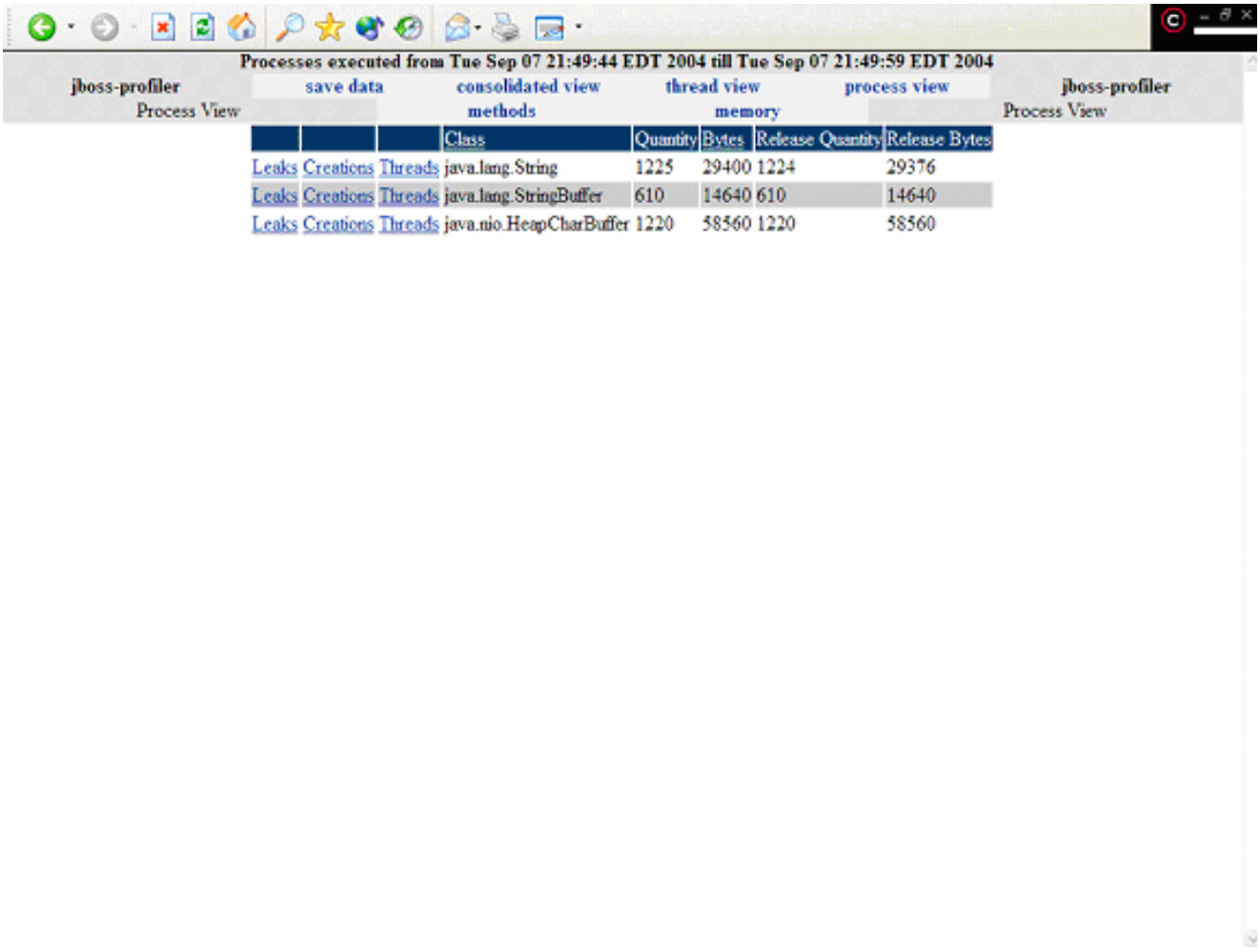

Figure 4.10.

You can look for Leaks, have information about Creations or Threads on this view.

- Consolidated View

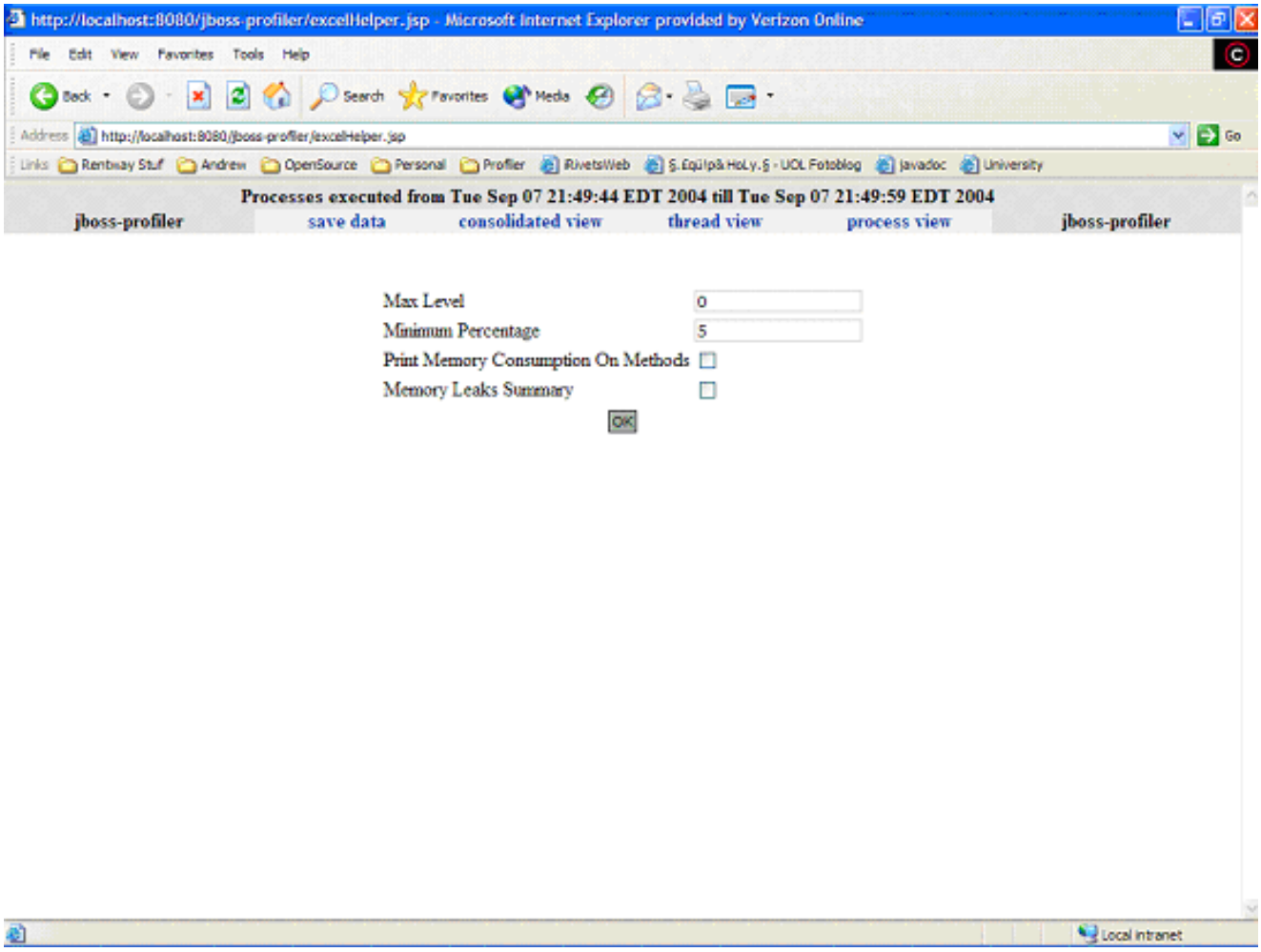

Figure 4.11.

On consolidated view, after you set filter parameters, you can generate a XML, XLS or zipped XML about the model:

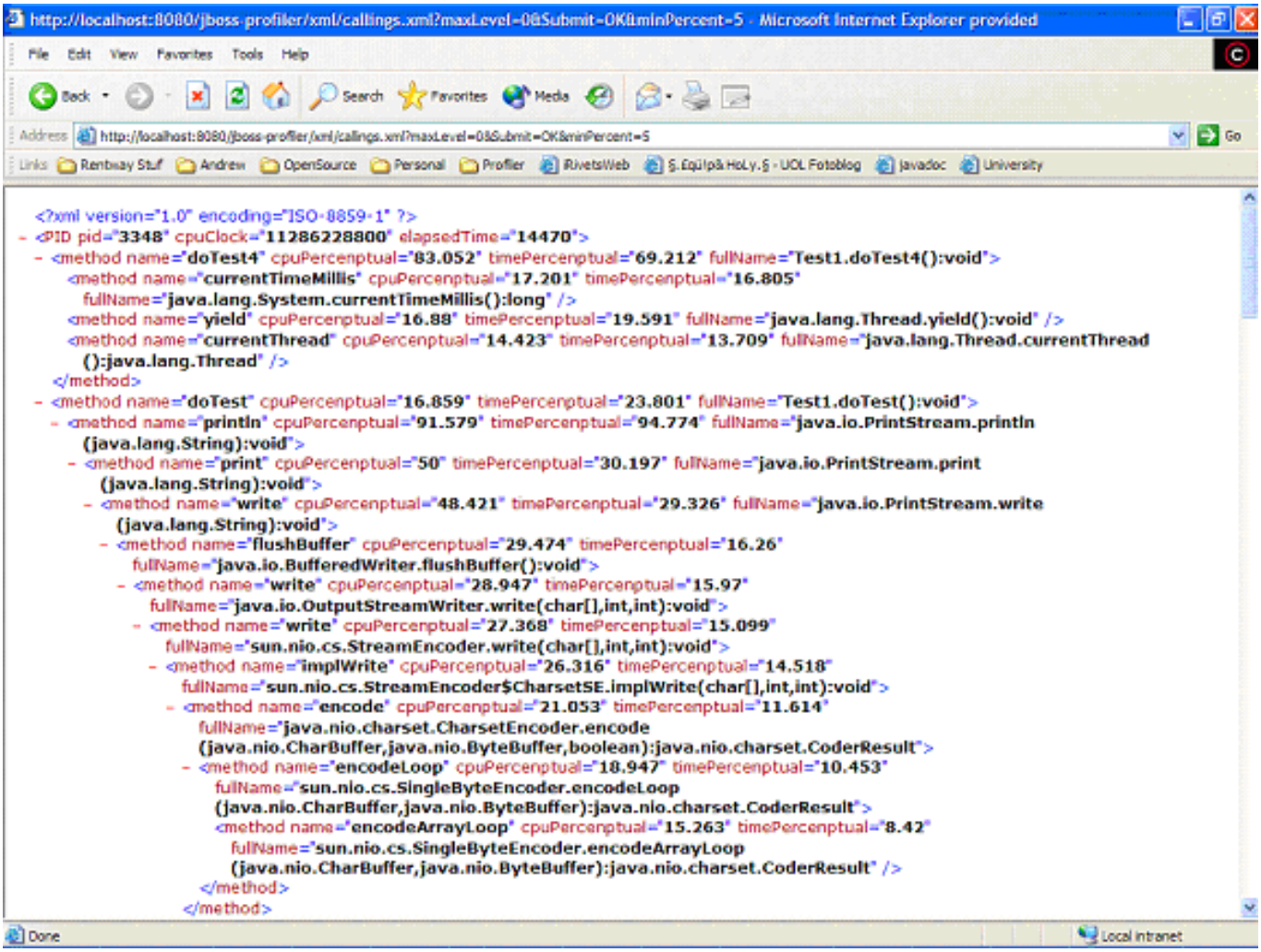

Figure 4.12.

4.1.4. Tracing

You can click on the "Tracing" field to detail the flow of "transaction" which is a sequence of methods executed in each method, and this following processing screen will be viewed.

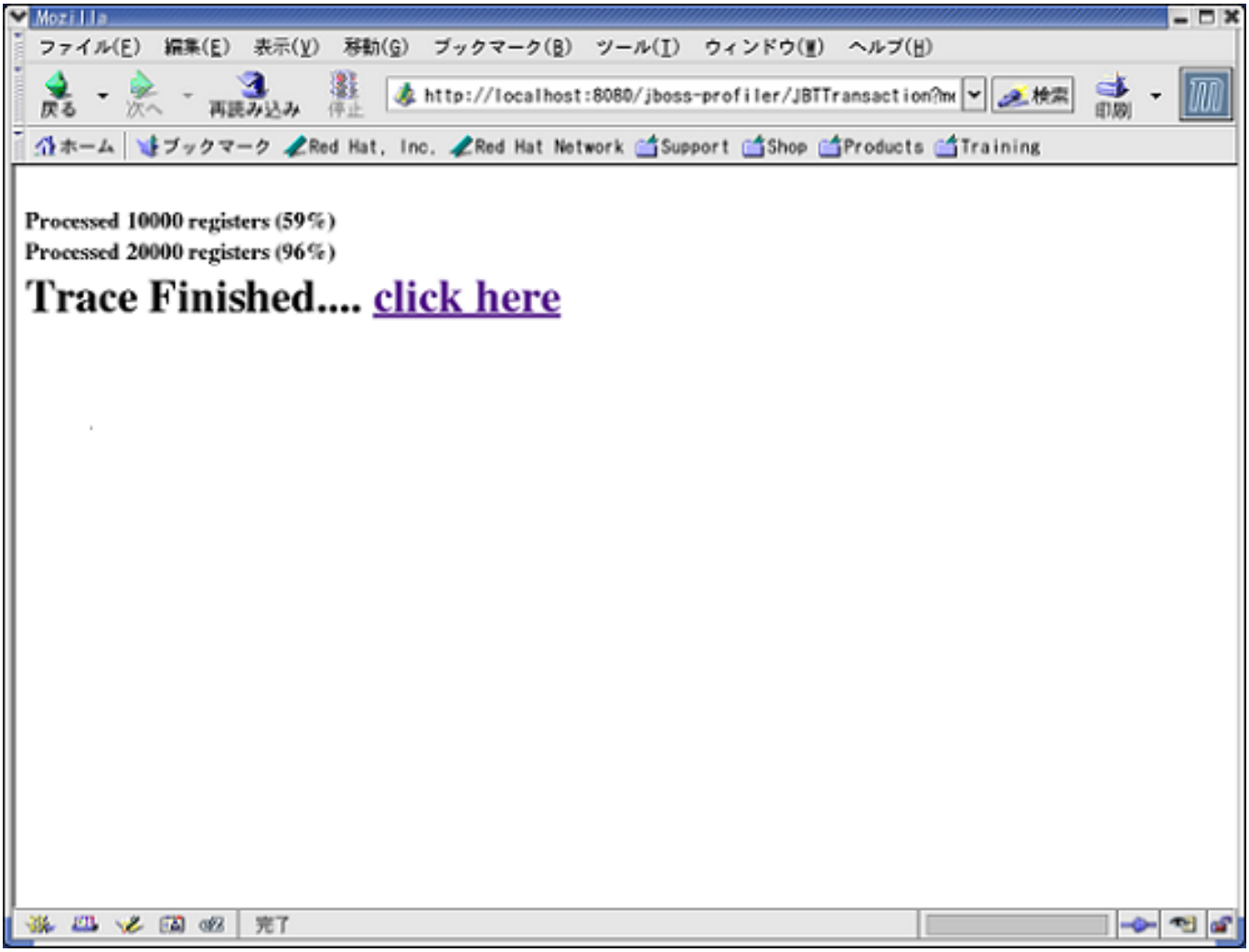

Figure 4.13.

Follow the link, and you will see the following view.

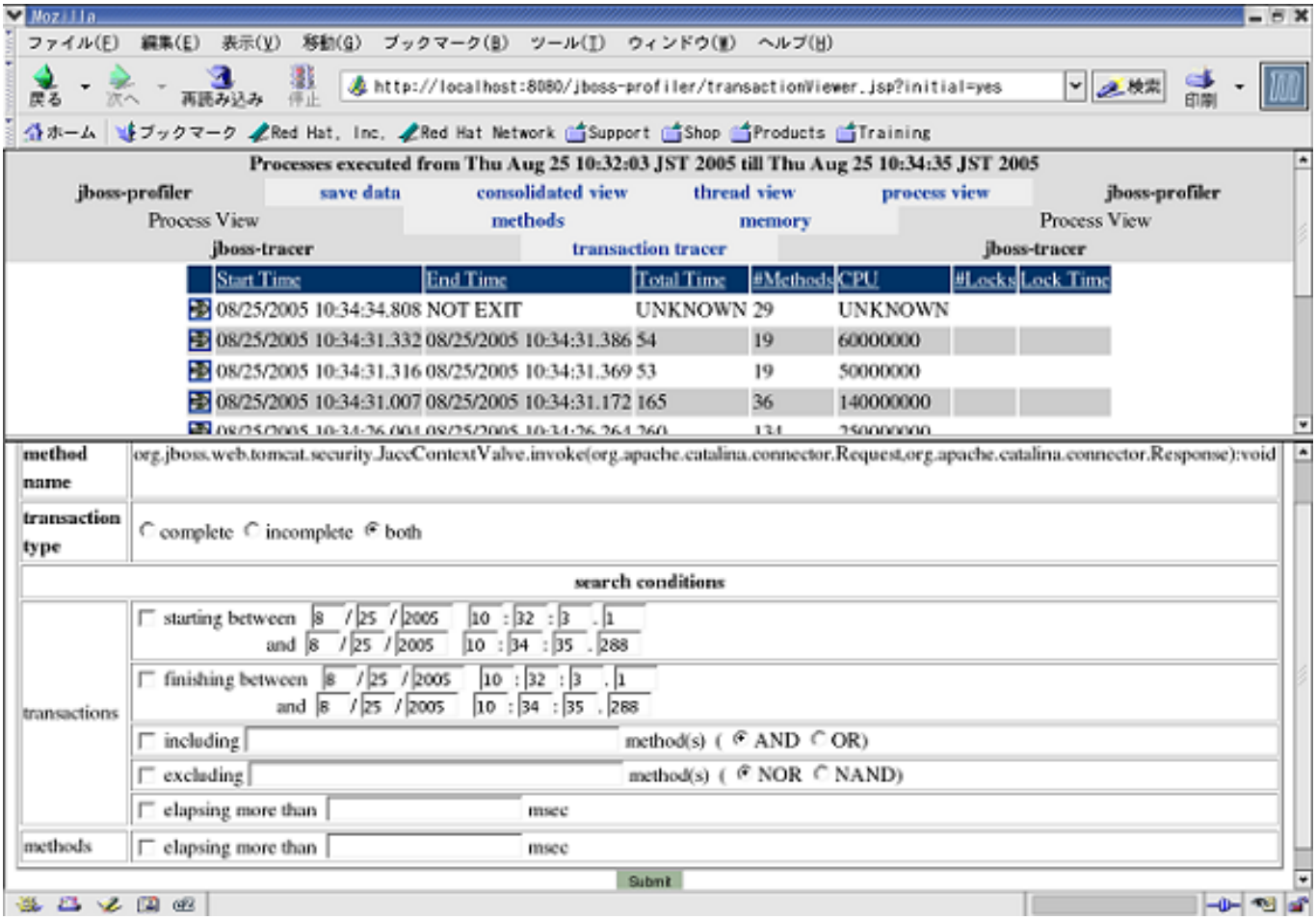

Figure 4.14.

The upper part (list view) of the view lists all transactions of the clicked method. The lower part (search view) provides an interface where you can set conditions for searching specific transactions. The "search view" lists the following items.

- start method name

  Name of the clicked method.

- transaction type

  You can specify whether target transactions are complete or not. "Transaction type" has 3 choices.

  - complete

    This choice picks up transactions which have completed from start to finish.

  - incomplete

    This choice picks up transactions which have aborted halfway.

- both

  This choice picks up both "complete" and "incomplete" transactions. (default)

- search conditions

  For setting conditions, you have 6 options. If you select more than 1 option, target transactions will have to meet conditions of all the selected options. "Search conditions" has 6 choices.

  - Period where target transactions have started

    Specify start time and finish time of the period as follows.

    MM/dd/yyyy hh:mm:ss:SSS

  - Period where target transactions have finished

    Specify start time and finish time of the period as follows.

    MM/dd/yyyy hh:mm:ss:SSS

  - Any method(s) executed in target transactions

    Specify any string which is a part of the method's name. You can specify more than one string by separating them by space(" "). If you do so, the tracer picks up transactions executing all of them (AND condition), or transactions executing at least one of them (OR condition).

  - Any method(s) **not** executed in target transactions

    Specify any string which is a part of the method's name. You can specify more than one string by separating them by space(" "). If you do so, the tracer picks up transactions not executing any of them (NOR condition), or transactions not executing all of them (NAND condition).

  - Duration of target transactions

    Specify duration by millisecond (SSS), and the tracer picks up transactions elapsing more than the duration.

  - Duration of any method executed in target transactions

    Specify duration by millisecond (SSS), and the tracer picks up transactions including any method(s) elapsing more than the duration.

Set conditions as you like and submit, and the tracer picks up transactions meeting the conditions. The "list view" will be refreshed to list them. The table has the following fields.

- Start Time

  Start time of the transaction.

- End Time

  Finish Time of the transaction.

- Total Time

  Duration of the transaction by the millisecond.

- #Methods

  The number of methods executed in the transaction.

- CPU

  CPU usage time of the transaction in nanosecond.

- #Locks

  The number of locks in the transaction if any. If not, this field is empty.

- Lock Time

  Duration of the locks in the transaction in millisecond. If there is are no locks, this field is empty.

You can click on the arrow icon to detail each transaction. You will see "transaction details view".

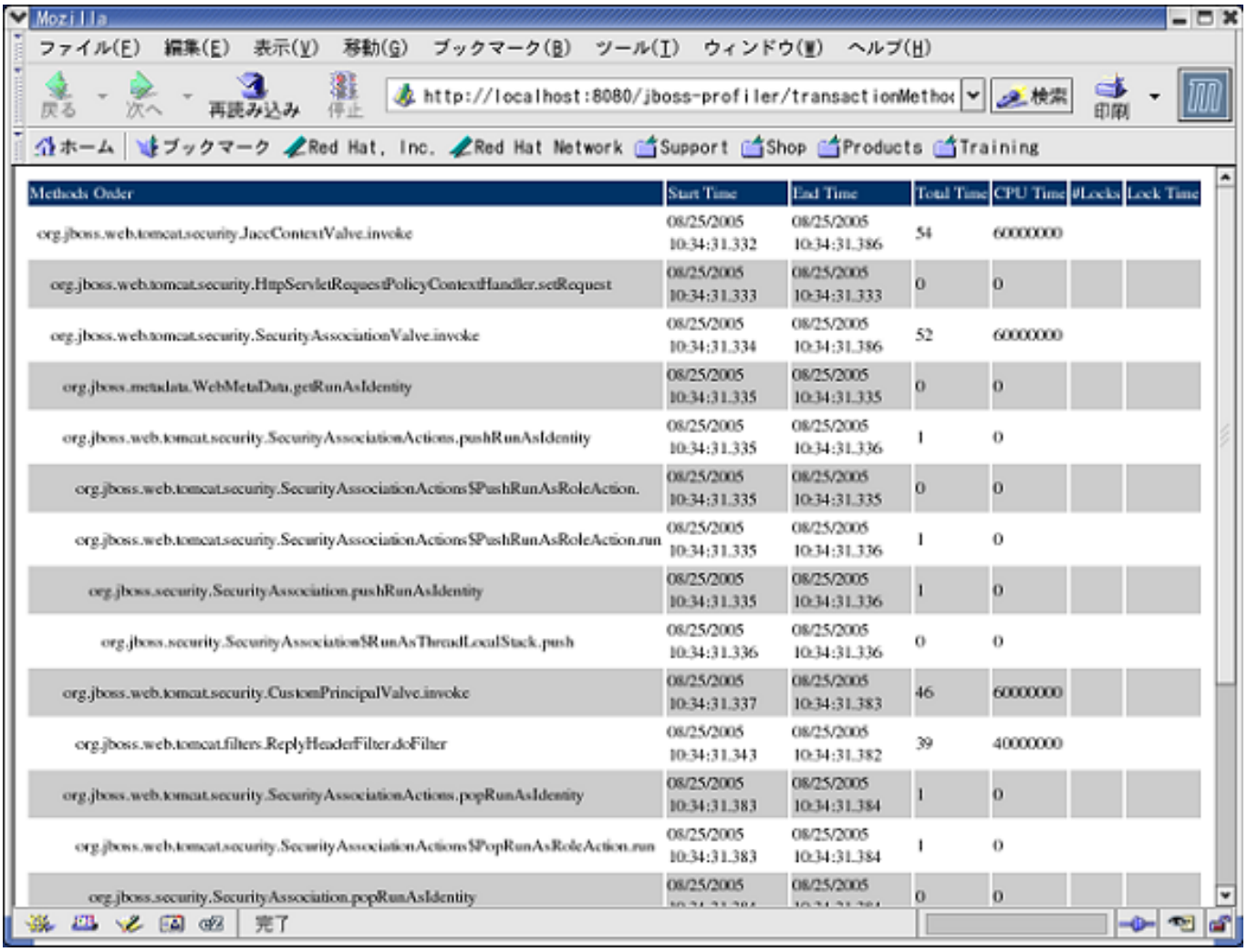

Figure 4.15.

The table has the following fields.

- Methods Order

  Methods executed in the transaction are listed in sequence. Indented methods mean nests. If you have specified any executed method(s) on "search view" mentioned above, it/they is/are drawn in different color from other methods.

- Start Time

  Start time of each method.

- End Time

  Finish time of each method. If you have chose "both" or "incomplete" of "transaction type" item on "search view", this field says "NOT EXIT" for any incomplete methods.

- Total Time

  Duration of each method by the millisecond. If you have chose "both" or "incomplete" of "transaction type" item on "search view", this field says "UNKNOWN" for any incomplete methods.

- CPU Time

  CPU usage time of each method.

- #Locks

  The number of locks in each method if any. If not, this field is empty.

- Lock Time

  Duration of the locks in each method in millisecond. If there is are no locks, this field is empty.

4.2. Memory Profiler

5

JVMTIInterface

5.1. Features

JVMTIInterface is class wrapping most features of JVMTI through a Java interface. Using this interface you can inquiry things you wouldn't be able using any regular Java API.

Things like:

- What are the current classes?
- What are the current objects of a given class?
- What is the memory state?

This is very useful to improve testcases. You could "lock" your memory setting an alarm if a test consumed more memory than expected. This is relatively easy to use and this chapter is going to show examples in how to do it.

5.2. The API

This section will show the most important methods part of the public interface. These methods are very useful on JUnit tests.

Class **org.jboss.profiler.jvmti.JVMTIInterface**

Table 5.1.

MethodName and Signature	Description	Exposed through MBean
Class getClassByName(String className)	search for all classes and return the first class maching className	N
Object[] getReferenceHolders(Object [] objects)	return all the objects holding references to the objects passed by parameter	N
Class[] getLoadedClasses()	return all loaded classes	N
void notifyInventory(boolean notifyOnClasses, String temporaryFileReferences, String temporaryFileObjects, JVMTICallBack call-	navigate through the entire HEAP sending results to JVMTICallBack interface. This process needs temporary files during its collection.	N

back)		
Object[] getReferenceHolders(Object [] objects)	return all the referenceHolders for the objects passed by parameter.	N
Class[] getLoadedClasses()	return all loaded classes on the JVM.	N
Object[] getAllObjects(Class clazz)	return all objects on a given class.	N
Object[] getAllObjects(String className)	return all objects on all classes where class.getName().equals(className). This is in case more than one classLoader loads the same class.	N
void heapSnapshot(String basicFileName, String suffix)	Exposes heap to files. Look at manual for more details.	Y
String exploreClassReferences(...)	Report method. This method reports references to classes.	Y
String exploreObjectReferences(String className, int maxLevel, boolean useToString)	Report method. Will show a reference tree for objects instanceof className.	Y
void forceReleaseOnSoftReferences()	Will allocate memory until an OutOfMemoryException happens, release it, and forceGC. This will force SoftReferences to go away.	Y
void forceGC()	Will force a FullGC by a JVMTI method (not System.gc) what guarantees a FullGC execution.	Y
String listClassesHTMLReport()	Report Method. Will show duplications and every single loaded class on the JVM	Y
String inventoryReport()	Report Method. Will show a summary of Objects allocations per class. This method is very useful to inspect number of objects and number of bytes for every single class on the system.	Y
String printObjects(String className)	Report Method. Will show a summary of every single object instanceof ClassName.	Y
Map produceInventory()	Returns a WeakHashMap<Class,InventoryDataPoint> summarizing the current JVM's inventory. This could be used by	N

	compareInventories and be used on JUnit tests making sure the system is not allocating more objects than expected.	
boolean compareInventories(...)	Compares two distinct Inventories validating expected boundaries defined by parameters.	N

5.3. Report example

if you use inventoryReport for example, you would have this output:

Class	#Instances	#Bytes
[C	75194	7294512
[I	2641	3226560
java.util.HashMap$Entry	128603	3086472
java.lang.String	111955	2686920
[Ljava.util.HashMap$Entry;	31239	2639088
[Ljava.lang.Object;	10456	1798312
java.util.HashMap	31199	1247960
[B	718	1247680
java.lang.reflect.Method	12799	1023920
javax.management.modelmbean.DescriptorSupport$CaseIgnoreString	45076	721216
org.jboss.mx.server.InvocationContext	9054	579456
javax.management.modelmbean.DescriptorSupport	19205	307280
[Ljava.lang.String;	14340	298104

5.4. Testing MemoryLeaks

5.4.1. ClassLoader Leakage

This page shows why a classLoader reference would leak. Instead of putting all these details here, I will just point the URL as a reference:

http://wiki.jboss.org/wiki/Wiki.jsp?page=ClassLeakage

As you can see on the URL, the classLoader won't be released untill all your strong references are cleared. The only strong references to a class or classLoader you don't need to release are references to your own classes or your own classLoader. Like, if a user makes a reference to a User's class on its JAR, GC will be able to determine the whole group is self contained thus the package can be released from the memory.

It's okay to make a reference for your user class, but if you are writting a framework or anything that will be used by super classLoaders using any sort of meta data and reflection, than you are highly ellegible of creating what we call a redeployment leak. That means a classLoader won't be garbage collected thus the class will never leave the memory.

ideally all your references to a Class, ClassLoader needs to be a WeakReference, and references to reflection will need to be a SoftReference.

If you read the wiki page in detail you will realize that is really hard to keep up with these constraints and if you don't have a testcase is very likely you will create a redeployment leak some day.

First thing you need in your test is an artificial classLoader. JBossTest project has a super class (org.jboss.test.JBossMemoryTestCase) you can use as base for this kind of test. In particular you need to use the newClassLoaderMethod:

```
   protected static ClassLoader newClassLoader(Class anyUserClass) throws Exception
   {
      ... This will create a regular classLoader
   }
```

Than you have to execute your methods using this classLoader. You could call a super method (using reflection). At the end, release every single instance, you could call a forceGC and the classLoader would have been released.

The secret on this type of test is to not use any strong reference (do not use imports to your class), and use get-ClassByName(String name). The class then is supposed to be released.

The following list is a simplified but real example:

```
/**
 * This class requires -agentlib:jbossAgent on JVM arguments on the JVM for
 * working properly
 *
 * @author csuconic
 *
 */
public class MemoryLeakTestCase extends TestCase {

  public void testSample() throws Exception {
    JVMTIInterface jvmti = new JVMTIInterface();
    ClassLoader loader = newClassLoader();
    Class theClass = loader.loadClass("org.jboss.serial.memory.test.SomePojo");

    jvmti.forceGC();

    // The class still on the memory
    assertNotNull(jvmti.getClassByName("org.jboss.serial.memory.test.SomePojo"));

    theClass = null;
    loader = null;
```

```
    jvmti.forceGC();

    assertNull(jvmti.getClassByName("org.jboss.serial.memory.test.SomePojo"));

  }

  /** Will create a ClassLoader to be used by the test */
  private static URLClassLoader newClassLoader() throws MalformedURLException {

    String dataFilePath = MemoryLeakTestCase.class.getResource("test")
        .getFile();
    String location = "file://"
        + dataFilePath.substring(0, dataFilePath.length()
            - "org.jboss.serial.memory.test".length());

    StringBuffer newString = new StringBuffer();
    for (int i = 0; i < location.length(); i++) {
      if (location.charAt(i) == '\\') {
        newString.append("/");
      } else {
        newString.append(location.charAt(i));
      }
    }
    String classLocation = newString.toString();

    URLClassLoader theLoader = URLClassLoader.newInstance(new URL[] { new URL(
        classLocation) }, null);
    return theLoader;
  }

}
```

With JVMTI you can inquire about the objects/classes inventory and assert about its expected state. As the classLoader wouldn't tell if you if a class is loaded or not, the only way to do so is using an API exposing DEBUG properties like JBossProfiler does through JVMTIInterface.

If you look at the testcase above you will see these operations:

- ClassLoader creation
- loadClassOperation
- forceGC
- assertionNotNull (using jvmti.getClassByName)

  So if the class was released by accident, like if something is not holding the reference by accident. (maybe a too weak scenario)
- releasing references
- forceGC
- assertNull operation (using jvmti.getClassByName)

  At this point there isn't any reference to the classLoader (or to the class). So if a GC happens the class will be released from the memory.

This example is relatively simple and you could improve a lot on top of that. If you need more concrete examples look at projects like JBoss AOP, JBoss Serialization and JBoss MC. All these projects will have a MemoryLeakTestCase validating scenarios like this.

5.4.2. Objects Leakage

Objects leakages are much easier to understand. But the problem is, everybody think it's so easy to understand that every programmer in the universe think that the GarbageCollection is always sufficient to capture our garbage. Well, in few words the GC is capable to capture our garbage, not our crap :-)

Most source of MemoryLeaks are Collections holding references when they are supposed to release the information. Most times this happens through publish/subscriber routines loosing some communication. Some occasions the linked list scenario could also happen by accident.

The API to test Object Leakages is pretty simple, just two methods:

- WeakHashMap<Class,InventoryDataPoint> produceInventory
- boolean compareInventories

  Parameters:

  - reportOutput You could set System.out here. The location where logging information is going to be sent.
  - map1 The first snapshot
  - map2 The second snapshot
  - ignoredClasses Classes you want to ignore on the comparisson. Used to ignore things you know are going to be produced and you don't have control over the testcase
  - prefixesToIgnore Same thing as classes, but every classes starting with these prefixes are going to be ignored
  - expectedIncreases An array of InventoryDataPoint with the maximum number of instances each class could be generating.

This is an example on how to validate memory leaks:

```
public void testNotifyOnObjects() throws Exception {
  class TestClass {

  }

  JVMTIInterface jvmti = new JVMTIInterface(); // The JVMTIWrapper used to
                                               // produce inventories

  TestClass keepr = new TestClass(); // at least one instance, so point2
                                     // won't be null

  Map firstMap = jvmti.produceInventory(); // The inventory of classes, this
                                           // is
                                           // HashMap<Class,InventoryDataPoint>
```

```
  TestClass[] tests = new TestClass[1000];
  for (int i = 0; i < 1000; i++) {
    tests[i] = new TestClass(); // allocating 1000 objects
  }

  Map secondMap = jvmti.produceInventory(); // the second inventory

  InventoryDataPoint point1 = (InventoryDataPoint) secondMap
      .get(TestClass.class);
  InventoryDataPoint point2 = (InventoryDataPoint) firstMap
      .get(TestClass.class);

  assertEquals(1000, point1.getInstances() - point2.getInstances()); // you
                                                                     // can
                                                                     // manually
                                                                     // compare
                                                                     // it

  assertTrue(jvmti
      .compareInventories(System.out, firstMap, secondMap,
          new Class[] { WeakHashMap.class }, new String[] {
              "[Ljava.util.WeakHashMap$Entry;", "java.lang.ref" },
          new InventoryDataPoint[] { new InventoryDataPoint(TestClass.class,
              2000) }));
  assertFalse(jvmti
      .compareInventories(System.out, firstMap, secondMap,
          new Class[] { WeakHashMap.class }, new String[] {
              "[Ljava.util.WeakHashMap$Entry;", "java.lang.ref" },
          new InventoryDataPoint[] { new InventoryDataPoint(TestClass.class,
              100) }));

}
```

Basically this test is:

1. Saving a first memory inventory snapshot, calling produceInventory
2. Doing some operations. (allocating memory)
3. Doing comparissons on the snapshot

Anything allocating more memory than expected is going to generate a failure on this testsuite.

With a testcase like that you are locking the doors of memory leaks in your system.

6

How to Compile Native Libraries

There are two native libraries for JBossProfiler, JVMTI and JVMPI.

JVMPI is a API released for JVM 1.4, and is being replaced by a newer and better version called JVMTI.

We are still supporting both versions as JVMPI is the only native way to extract performance metrics from JVM 1.4.

6.1. Requirements

In case you are using windows, you will need cygwin installed in order to compile either JVMPI or JVMTI.

You will then need g++ and gcc installed.

You also need JAVA_HOME defined to a valid JDK 1.4 (JVMPI) or JDK 1.5 (JVMPI and JVMTI), as some includes will be used under JDK directory.

6.2. Compiling JVMPI Module

Under jvmti-lib there are one directory for each platform. (linux, solaris, macos and win32).

Go the directory that represents your platform and call ./compile.sh.

6.3. Compiling JVMTI Module

Under jvmpi-lib there are one directory for each platform. (linux, solaris, macos and win32).

Go the directory that represents your platform and call ./compile.sh.