Memory Leak Analysis of Java Virtual Machine

Sameer Yadav

12th July, 2011

Abstract

This project work presents a practical experience with memory analysis on a real world complex middleware platform, being developed in the context of academic-industrial collaboration. The reported experience suggests a practical method that can help practitioners to analyze memory leaks and to adopt proper actions to mitigate these bugs, especially in the context of complex Off-The-Shelf (OTS) based software systems.

In this report I shall explain how memory management works inside application process, focusing on the main memory problem that cause software aging: Memory leakage [10]. Along with the theoretical explanation, I present an experimental study that illustrates how memory leaks occur and how they accumulate over time in order to cause system aging-related failures.

Although several approaches have already been proposed to study the development of software aging phenomena there are still some open issues, especially in the field of OTS-based software systems. All measurement-based software aging analyses consider aging introduced by long-running applications, such as web servers and mail servers, as measured at the operating system level, neglecting the contribution of intermediate layers, such as middleware, virtual machines, and, more in general third-party OTS items.

Starting from an experimental campaign on real world testbeds, this work isolates the contribution of the Java Virtual Machine to the overall aging trend, and identifies, through statistical methods, which workload parameters are more relevant to aging dynamics. Experimental results show that the Sun Hotspot JVM experiences software aging phenomena due to memory leakage. Presented results clearly indicate that much more efforts have still to be done in order to improve the dependability of the JVM.

Introduction

While the last decades have seen a lot of efforts from industry as well as academia to avoid software crisis, they still happen and their impact is increasing. According to many studies [10] [4], one of the most important causes of these outages is software aging. Software aging phenomena refers to the accumulation of errors, usually provoking resource contention, during long running application executions, like web applications and mail servers which normally cause applications/systems hang or crash. Determining the software aging root cause failure is a huge task due to the growing day by day complexity of the systems. Memory-related aging effects are one of the most important problems in this research field. Software aging effects are the practical consequences of errors caused by aging-related fault activations, such as Memory leakage [1]. The accumulating effects of successive Memory leak occurrences directly influence the memory-related aging effects. Therefore understanding their causes and how they work is a major requirement in designing dependable software systems.

A memory leak is the gradual loss of available computer memory when a program (an application or part of the operating system) repeatedly fails to return memory that it has obtained for temporary use. As a result, the available memory for that application or that part of the operating system becomes exhausted and the programs can no longer function. Memory leaks may not be serious or even detectable by normal means. In modern operating systems, normal memory used by an application is released when the application terminates. This means that a memory leak in a program that only runs for a short time may not be noticed and is rarely serious. For a program that is frequently opened and called or that runs continuously, even a very small memory leak can eventually cause the program or the system to terminate. A memory leak is the result of a program bug. Some operating systems provide memory leak detection so that a problem can be detected before an application or the operating system crashes. Some program development tools also provide automatic "housekeeping" for the developer. It is always the best programming practice to return memory and any temporary file to the operating system after the program no longer needs it.

A memory leak can diminish the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down unacceptably due to thrashing. Memory leaks in enterprise applications cause a significant number of critical situations. The cost is a combination of time and money for analysis, expensive downtime in production environments, stress, and a loss of confidence in the application and frameworks [4]. The cost of memory leaks is significant and is often associated directly with production down time, or a slipped deployment schedule.

To be clear, the best approach for solving memory leaks is to detect and resolve them in test. Ideally, there should be a testing schedule, a test environment identical to the production environment that is able to drive representative and anomaly workloads, and technical resources with appropriate skill sets dedicated to system testing. This is the best way to ensure, as much as possible, a clean transition to production [3]. However, designing and provisioning such an environment, along with the associated cultural changes, is not the focus of this paper. This paper focuses more on the diagnosis and analysis of the Memory leakage problem and their after-effects than to resolve or remove it. To detect the signs of Memory Leaks, I would test and monitor commonly-used Off-The-Shelf (OTS) software and record its behavior against time and workload.

Operating Systems such as Windows and Linux [5], Database Management Systems such as *Oracle* and *Microsoft SQL Server*, or Application Servers such as *Bea Weblogic* and *Red Hat JBoss*, are all examples of OTS items . Among the extremely wide range of available OTS items, I choose Virtual Machines, and in particular the Java Virtual Machine (JVM), as the case study to employ throughout the dissertation.

The JVM has been chosen for the following reasons:

1. Virtual Machines represent a relevant example of the above mentioned intermediate layers, since they provide a virtualization of a complete execution environment

2. The JVM is currently widely employed in a wide range of applications, including critical ones. For instance, Java has been used to develop the *ROver Sequence Editor* (ROSE), a component of the *Rover Sequencing on Visualization Program* (RSVP), used to control the Spirit Robot in the exploration of Mars

3. There is a growing interest in the scientific and industrial community toward the employment of Java in safety and mission critical scenarios, as proved by a recent issue of a Java Specification Request (JSR-302), which aims at defining those capabilities needed to use Java technologies in safety critical applications

4. There is a lack of research about the characterization of the dependability of the JVM; moreover, as regards Software Aging, a recent paper [2], discussed in the previous chapter, highlighted the presence of aging phenomena in a Java-based SOAP server.

Before I begin to demonstrate my test and analysis on JVM server I would like to present a brief overview of the architectural model of the JVM which would help us to know how a memory leakage occurs in such a High Level Language and also to get a better picture of what's exactly going on in each layer and unit during their run-time.

Overview of the JVM architecture

The JVM is a virtual machine belonging to the High Level Language VMs (HLL-VM) category. An HLL-VM is a VM which supports for cross-platform programming and provides a virtual *Instruction Set Architecture* (ISA) abstracting the *Application Binary Interface* (ABI) and the ISA exposed by the underlying Operating System and Hardware, thus making applications written for the virtual machine platform-independent.

The virtual ISA of the JVM is a set of instructions called **bytecodes**; programs written in Java are compiled into bytecodes. The JVM is composed of four main components, depicted in figure 1 [6]

- *Execution Unit* - It dispatches and executes operations, emulating a CPU. An operation could be i) a translated bytecode instruction, ii) a compiled bytecode instruction, or iii) a native instruction. The Interpreter translates single bytecode instructions into native machine code whereas the *Just-In-Time* (JIT) compiler translates entire methods into native code doing some optimizations. Instead, native instructions need no translation since they are native machine instructions. They are dynamically loaded, linked and executed by the *Java Native Interface* (JNI). Moreover, the Exception Handler handles exceptions thrown by both Java Applications and the Virtual Machine. Exceptions thrown by applications are defined checked, while exceptions thrown by the VM are defined unchecked and are related to errors originated into the virtual machine.

- *OS Virtualization Layer Unit* - It provides a platform-independent abstraction of the host system's ABI. This abstraction layer provides a common gateway for all JVM components to access host system resources.

- *Memory Management Unit* - It handles both the JVM heap area and the stack area, managing object allocation, reference handling, object finalization and garbage collection. Moreover, Fast Allocation Mechanisms are provided to allocate temporary memory areas for internal VM operations.

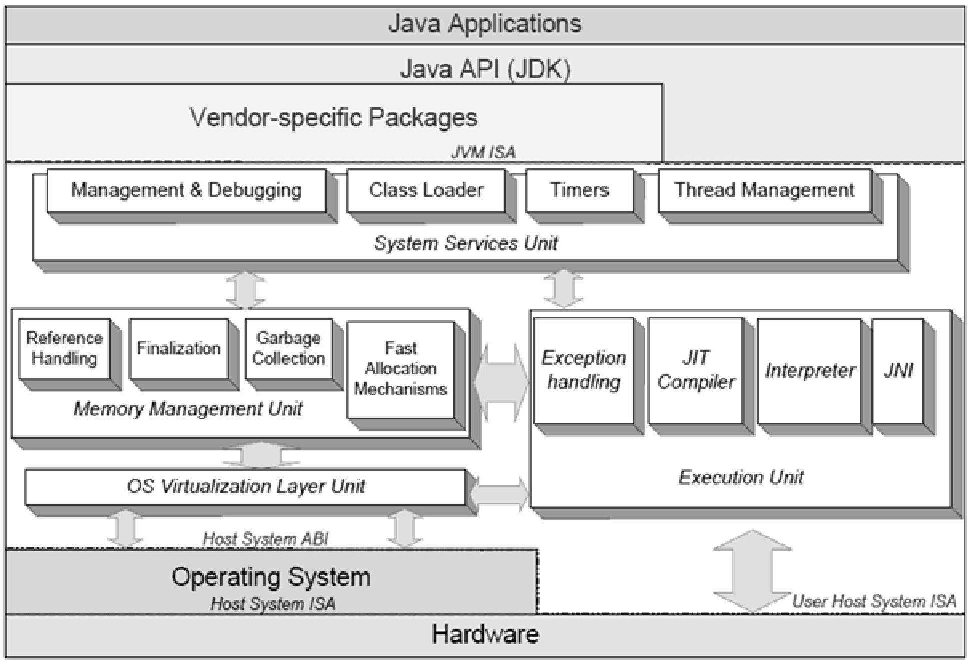


Figure 1: The JVM Architecture

- *System Services Unit* - Components included in this unit offer services to Java Applications. The Thread Management component handles thread creation and termination and it implements mechanisms for thread synchronization as specified by the Java Virtual Machine Specification and the Java Language Specification. The Class Loader is in charge of dynamically loading and verifying Java class files. Timer components expose functionalities to access system timers through the JVM. Finally, the Management and Debugging component includes functionalities for debugging Java applications and for the management of the JVM.

Java implements *Automatic Memory Management* where we no longer have to worry about allocating and freeing memory [6]. We can simply create objects, and once we stop using an object we can depend on the *Garbage Collector* (GC) to collect it. Before I get too deep into my discussion, let's begin by reviewing how garbage collection actually works. The job of the garbage collector is to find objects that are no longer needed by an application and to remove them when they can no longer be accessed or referenced. The garbage collector starts at the root nodes, classes that persist throughout the life of a Java application, and sweeps through all of the nodes that are referenced. As it traverses the nodes, it keeps track of which objects are actively being referenced. Any classes that are no longer being referenced are then eligible to be garbage collected. The memory resources used by these objects can be returned to the Java virtual machine (JVM) when the objects are deleted.

Memory leakage in Java programs

Let's start by describing how memory leaks happen in Java. In garbage collected languages like Java the unused memory is claimed by the garbage collector, thus relieving the programmer of the burden of managing explicitly the use of dynamic memory. This claim is only partially correct: technically, the garbage collector reclaims only allocated portions of memory which have become unreachable from program variables, and often, this memory does not entirely correspond to the unused memory of the system. Memory leaks occur when a program never stops using an object, thus keeping a permanent reference to it. It may also occur when objects hold references to objects that are no longer needed. For instance, it is quite common that memory is allocated, used for a while, and then no longer needed nor used by the program. However, some of this memory cannot be freed by the garbage collector and will remain in the state of the program for longer than it needs to be, as there are still references to it from some program variables. Figure 2 illustrates this concept [9].

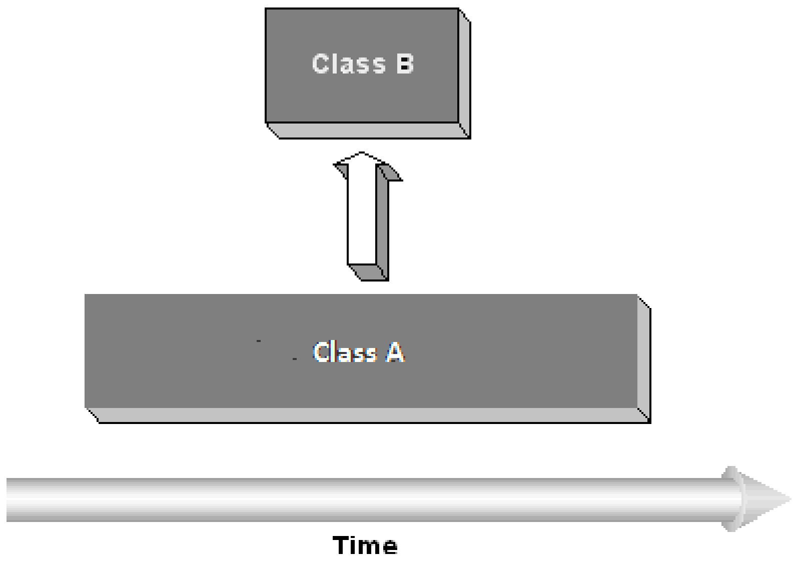


Figure 2: Unused but still referenced

The figure illustrates two classes that have different lifetimes during the execution of a Java application. Class A is instantiated first and exists for a long time or for the entire life of the program. At some point, class B is created, and class A adds a reference to this newly created class. Now let's suppose class B is some user interface widget that is displayed and eventually dismissed by the user. Even though class B is no longer needed, if the reference that class A has to class B is not cleared, class B will continue to exist and to take up memory space even after the next garbage collection cycle is executed.

This problem prevents the automatic Java garbage collection process from freeing memory, even though the Java virtual machine has a built-in garbage collection mechanism which frees the programmer from any explicit object de-allocation responsibilities. These memory leak issues manifest as increasing Java heap usage over time with an eventual *OutOfMemoryError* when the heap is completely exhausted. This type of a memory leak is referred to as a **Java heap memory leak**. Memory leaks can also occur in Java due to failure to clean up native system resources like file handles, database connection artifacts, and so on, after they are no longer used. This type of memory leak is referred to as a **native memory leak**. These types of memory leaks manifest as increasing process sizes over time without any increase in the Java heap usage.

Even though this phenomenon, typical of Java and other garbage collected languages like Python, defines a different form of "memory leakage" than in traditional languages like C, its results are equally catastrophic. If an application leaks memory, it first slows down the system in which it is running and eventually causes the system to run out of memory. There are two main sources of memory leaks in Java code:

- *Unknown or unwanted object references.* As commented above, this happens when some object is not used anymore; however the garbage collector cannot remove it because it is pointed to by some other object.
- *Long-living (static) objects.* These are objects that are allocated for the entire execution of the program.

These two possibilities appear in different forms. For example, a common simple error, such as forgetting to assign null to a live variable pointing to the object not needed anymore, leads to a memory leak. Such a leak can have serious consequences if the memory associated to it is substantial in size. There are four common categories of memory usage problems in Java[1]:

- Java heap memory leaks
- Heap fragmentation
- Insufficient memory resources
- Native memory leaks.

Common causes for memory leaks in Java applications

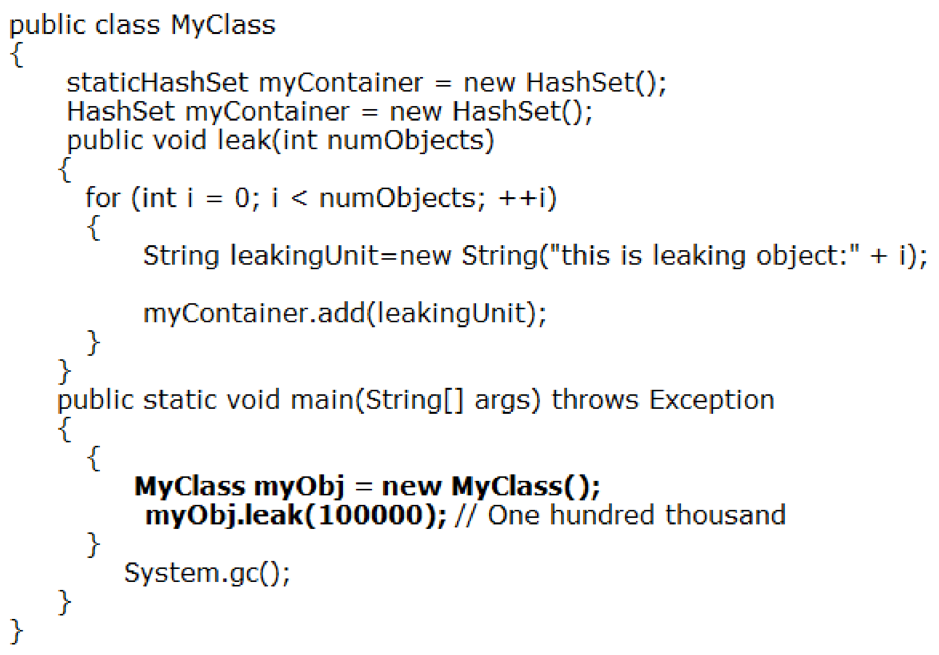
In general, a Java heap memory leak results when an application unintentionally (due to program logic error) holds on to references to objects that are no longer required. These unintentional object references prevent the built-in Java garbage collection mechanism from freeing the memory used by these objects. Common causes for these memory leaks are:

- Unbounded caches
- Un-invoked listener methods
- Infinite loops
- Too many session objects
- Poorly written custom data structures

Memory usage problems due to insufficient memory resources can be caused by configuration issues or system capacity issues. For example, the maximum heap size allowable for the Java virtual machine may be configured using the *-Xmx* parameter at a value which is too low to accommodate the total number of user sessions in memory. As mentioned above, the common underlying cause of memory leaks in Java (Java heap memory leaks) are unintentional (due to program logic error) object references holding up unused objects in the Java heap. In this section, a number of common types of program logic error that lead to Java heap memory leaks are described.

Unbounded caches

A very simple example of a memory leak would be a *java.util.Collection* object (for example, a **HashMap**) that is acting as a cache but which is growing without any bounds. Listing 1 shows a simple Java program demonstrating how a basic memory leaking data structure [3].



```
public class MyClass
{
     staticHashSet myContainer = new HashSet();
     HashSet myContainer = new HashSet();
     public void leak(int numObjects)
    {
      for (int i = 0; i < numObjects; ++i)
      {
           String leakingUnit=new String("this is leaking object:" + i);

           myContainer.add(leakingUnit);
      }
    }
    public static void main(String[] args) throws Exception
    {
      {
          MyClass myObj = new MyClass();
           myObj.leak(100000); // One hundred thousand
      }
         System.gc();
    }
}
```

Listing 1: Sample Java program leaking String objects into a static HashSet container object

In the Java program shown in Listing 1, there is a class with the name *MyClass* which has a static reference to HashSet by the name of *myContainer*. In the main method of the class: *MyClass*, there is a subscope (in bold text) within which an instance of the class: *MyClass* is instantiated and its member operation: leak is invoked. This result in the addition of a hundred thousand String objects into the container: *myContainer*. After the program control exits the subscope, the instance of the *MyClass* object is garbage collected, because there are no references to that instance of the *MyClass* object outside that subscope. However, the *MyClass* class object has a static reference to the member variable called *myContainer*. Due to this static reference, the *myContainer* HashSet continues to persist in the Java heap even after the sole instance of the *MyClass* object has been garbage collected and, along with the HashSet, all the String objects inside the HashSet continue to persist, holding up a significant portion of the Java heap until the program exits the main method. This program demonstrates a basic memory leaking operation involving an unbounded growth in a cache object.

Un-invoked listener methods

Many memory leaks result due to program errors that cause cleanup methods from not getting invoked. The Listener pattern is a commonly used pattern in Java programs which is used to implement methods for cleaning up shared resources when they are no longer required. For instance, J2EE programs often rely on the *HttpSessionListener* interface and its *sessionDestroyed* callback method to clean up any state stored in a user session when the user session expired. Sometimes, due to program logic error, the program responsible for invoking the listener may fail to invoke it, or the listener method may fail to complete due to an exception and this might lead to unused program state lying around in the Java heap even after it is no longer required.

Infinite loops

Some memory leaks occur due to program errors in which infinite loop in the application code allocates new objects and adds them to a data structure accessible from outside the program loop scope. This type of infinite loops can sometimes occur due to multithreaded access into a shared unsynchronized data structure. These types of memory leaks manifest as fast growing memory leaks, where if the verbose GC data reports a sharp drop in free heap space in a very short time leads to an *OutOfMemoryError*. For this type of memory leak case, it is important to analyze a heap dump taken within the short span of time the free memory is observed to be dropping quickly.

While it might be possible to identify the memory leaking data structure by analyzing the heap dumps, identifying the memory leaking code which is in a infinite loop is not straightforward. The method which is stuck in an infinite loop can be identified by looking at the thread stacks of all the threads in a thread dump taken during the time the free memory is observed to be dropping quickly

Too many session objects

Many *OutOfMemoryError* occur due to inappropriate configuration for the maximum heap size necessary to support maximum user loads. A simple example would be a J2EE application that uses in-memory *HttpSession* objects to store user session information. If no maximum limit is set on the maximum number of session objects that can be held in memory, then it is possible to have many session objects during peak user load time. This can lead to *OutOfMemoryError* that are not really memory leaks, but improper configuration.

When are memory-leaks a concern?

If our program is getting a *java.lang.OutOfMemoryError* after executing for a while, a memory leak is certainly a strong suspect. The perfectionist programmer would answer that all memory leaks need to be investigated and corrected. However, there are several other points to consider before jumping to this conclusion, including the lifetime of the program and the size of the leak.

Consider the possibility that the garbage collector may never even run during an application's lifetime. There is no guarantee as to when or if the JVM will invoke the garbage collector, even if a program explicitly calls *System.gc()*. Typically, the garbage collector won't be automatically run until a program needs more memory than is currently available. At this point, the JVM will first attempt to make more memory available by invoking the garbage collector. If this attempt still doesn't free enough resources, then the JVM will obtain more memory from the operating system until it finally reaches the maximum allowed.

Take, for example, a small Java application that displays some simple user interface elements for configuration modifications and that has a memory leak [9]. Chances are that the garbage collector will not even be invoked before the application closes, because the JVM will probably have plenty of memory to create all of the objects needed by the program with leftover memory to spare. So, in this case, even though some dead objects are taking up memory while the program is being executed, it really doesn't matter for all practical purposes.

If the Java code, being developed is meant to run on a server 24 hours a day, then memory leaks become much more significant than in the case of our configuration utility. Even the smallest leak in some code that is meant to be continuously run will eventually result in the JVM exhausting all of the memory available.

The Degradation Model

I developed a model for performance degradation due to the gradual loss of system resources, especially, the memory resource. In a client-server system, for example, every client process issues memory requests at varying time points. An amount of memory is granted to each new request (when there is enough memory available), held by the requesting process for a period of time, and presumably released back to the system resource reservoir when it is no longer in use. A memory leak occurs when the amount of allocated memory is not fully released. The available memory space is gradually reduced as such resource leaks accumulate over time. Consequently, a resource request that would have been granted in the leak-less situation may not be granted when the system suffers from resource leaks. This model accommodates analysis for both the leak-less case and the leak-present case. My analysis gives concrete expressions of the failure rate in each case.

Preventing memory leaks

We can prevent memory leaks by watching for some common problems. Collection classes, such as hashtables and vectors, are common places to find the cause of a memory leak [4]. This is particularly true if the class has been declared static and exists for the life of the application. Another common problem occurs when we register a class as an event listener without bothering to unregister when the class is no longer needed. Also, many times member variables of a class that point to other classes simply need to be set to null at the appropriate time.

Determining if an application has memory leaks

To see if a Java application running on a Windows NT platform is leaking memory, we might be tempted to simply observe the memory settings in Task Manager as the application is run. However, after observing a few Java applications at work, we will find that they use a lot of memory compared to native applications [7]. Some Java projects can start out using 10 to 20 MB of system memory. The other thing to note about Java application memory use is that the typical program running with the IBM JDK 1.1.8 JVM seems to keep gobbling up more and more system memory as it runs. The program never seems to return any memory back to the system until a very large amount of physical memory has been allocated to the application [8].

To understand what is going on, we need to familiarize ourselves with how the JVM uses system memory for its heap. When running *java.exe*, we can use certain options to control the startup and maximum size of the garbage-collected heap (-ms and -mx, respectively). The Sun JDK 1.1.8 uses a default 1 MB startup setting and a 16 MB maximum setting. The IBM JDK 1.1.8 uses a default maximum setting of one-half the total physical memory size of the machine. These memory settings have a direct impact on what the JVM does when it runs out of memory. The JVM may continue growing the heap rather than wait for a garbage collection cycle to complete.

So for the purposes of finding and eventually eliminating a memory leak, I am going to need better tools than task monitoring utility programs. Memory debugging programs can come in handy when we're trying to detect memory leaks. These programs typically give us information about the number of objects in the heap, the number of instances of each object, and the memory being using by the objects. In addition, they may also provide useful views showing each object's references and referrers so that we can track down the source of a memory leak.

Enterprise applications written in the Java language involve complex object relationships and utilize large numbers of objects. Although, the Java language automatically manages memory associated with object life cycles, understanding the application usage patterns for objects is important. In particular,we should verify the following:

- The application is not over utilizing objects
- The application is not leaking objects
- The Java heap parameters are set properly to handle a given object usage pattern

Understanding the effect of garbage collection is necessary to apply these management techniques.

The garbage collection bottleneck

Examining Java garbage collection gives insight to how the application is utilizing memory. Garbage collection is Java strength. By taking the burden of memory management away from the application writer, Java applications are more robust than applications written in languages that do not provide garbage collection [11]. This robustness applies as long as the application is not abusing objects. Garbage collection normally consumes from 5% to 20% of total execution time of a properly functioning application. If not managed, garbage collection is one of the biggest bottlenecks for an application.

Monitoring garbage collection

We can use garbage collection to evaluate application performance health. By monitoring garbage collection during the execution of a fixed workload, we gain insight as to whether the application is over-utilizing objects. Garbage collection can even detect the presence of memory leaks. For this type of investigation, set the minimum and maximum heap sizes to the same value. Choose a representative, repetitive workload that matches production usage as closely as possible, user errors included. To ensure meaningful statistics, run the fixed workload until the application state is steady. It usually takes several minutes to reach a steady state.

JVM Diagnosis alerts administrators on abnormalities in Java memory consumption. Administrators can use JVM Diagnostics and take heap dumps in production applications without disturbing the application. They can take multiple heap dumps over a period of time, analyze the differences between the heap dumps and identify the object causing the memory leak. Heap analysis can be performed even across different application versions. Differential Heap Analysis with multiple heap dumps makes it easy to identify memory leaks.

Detecting memory leaks

Memory leaks in the Java language are a dangerous contributor to garbage collection bottlenecks. Memory leaks are more damaging than memory overuse, because a memory leak ultimately leads to system instability. Over time, garbage collection occurs more frequently until the heap is exhausted and the Java code fails with a fatal out-of-memory exception. Memory leaks occur when an unused object has references that are never freed. Memory leaks most commonly occur in collection classes, such as Hashtable because the table always has a reference to the object, even after real references are deleted.

High workload often causes applications to crash immediately after deployment in the production environment. This is especially true for leaking applications where the high workload accelerates the magnification of the leakage and a memory allocation failure occurs.

Memory leak testing

The goal of memory leak testing is to magnify numbers. Memory leaks are measured in terms of the amount of bytes or kilobytes that cannot be garbage collected. The delicate task is to differentiate these amounts between expected sizes of useful and unusable memory. This task is achieved more easily if the numbers are magnified, resulting in larger gaps and easier identification of inconsistencies [9]. The following list contains important conclusions about memory leaks:

Long-running test

Memory leak problems can manifest only after a period of time, therefore, memory leaks are found easily during long-running tests. Short running tests can lead to false alarms. It is sometimes difficult to know when a memory leak is occurring in the Java language, especially when memory usage has seemingly increased either abruptly or monotonically in a given period of time. The reason it is hard to detect a memory leak is that these kinds of increases can be valid or might be the intention of the developer. We can learn how to differentiate the delayed use of objects from completely unused objects by running applications for a longer period of time. Long-running application testing gives us higher confidence for whether the delayed use of objects is actually occurring.

Repetitive test

In many cases, memory leak problems occur by successive repetitions of the same test case [11]. The goal of memory leak testing is to establish a big gap between unusable memory and used memory in terms of their relative sizes. By repeating the same scenario over and over again, the gap is multiplied in a very progressive way. This testing helps if the number of leaks caused by the execution of a test case is so minimal that it is hardly noticeable in one run.

I used repetitive tests at the system level or module level. The advantage with modular testing is better control. When a module is designed to keep the private module without creating external side effects such as memory usage, testing for memory leaks is easier. First, the memory usage before running the module is recorded. Then, a fixed set of test cases are run repeatedly. At the end of the test run, the current memory usage is recorded and checked for significant changes. We should remember that garbage collection must be suggested when recording the actual memory usage by inserting *System.gc()* in the module where we want garbage collection to occur, or using a profiling tool, to force the event to occur.

Concurrency test

Some memory leak problems can occur only when there are several threads running in the application. Unfortunately, synchronization points are very susceptible to memory leaks because of the added complication in the program logic [9]. Careless programming can lead to kept or unreleased references. The incident of memory leaks is often facilitated or accelerated by increased concurrency in the system. The most common way to increase concurrency is to increase the number of clients in the test driver.

Monitoring Framework

In this section I present the results of controlled experiments executed to forcedly cause memory leak. My purpose is to show how this memory-related aging effects occur internally, showing the mechanics of these effects inside the memory allocators. But before presenting the monitoring architecture proposed I need to present the technologies used to build my approach:

Used Technology

Mail Server: I chose a mail server as the benchmark application for my analysis since it represents an important class of long running server applications usually stressed by significant workloads. Now let's get a brief knowledge on how E-mail works. First we construct a message with one or more recipient addresses using a **Mail User Agent** (MUA), like *Microsoft Outlook* and *Netscape Messenger.* Each e-mail client is configured to send mail to a **Mail Transfer Agent** (MTA) and can be used to poll an MTA to fetch E-mail messages sent to the user's address. To do this, we need an E-mail account on a mail server and using standard Internet protocols we can either work with the e-mail offline (using *POP3*) or leave the e-mail on the server (using IMAP). The protocol used to send mail both from the client to the MTA and between MTAs is *SMTP* (Simple Mail Transfer Protocol). E-mail servers rely heavily on DNS and e-mail-specific records called mail transfer (or *MX*) records. MX records are slightly different from the DNS records used to resolve URLs, containing some additional priority information used to route mail more effectively. I won't delve into those details here, but it's important to understand that DNS is the key to routing e-mail successfully and efficiently. I would be using JAMES mail server which uses JVM as its underlying middleware virtual machine and monitor it through another system so as to not affect any stress on the underlying JVM on the server side. James is a 100% pure Java SMTP, POP3 Mail server, IMAP and NNTP server designed to be a complete and portable enterprise mail/messaging engine solution based on currently available open messaging protocols. Amongst the benefits of James over other mail platforms is its support for building custom mail handling applications. James is an MTA, while the *JavaMail API* provides a framework for an MUA [17]. To configure a mail server I would be running JAMES on one system and then using a DHCP server I would change the *host* file of the client system to include the address of the newly-configured server.

Java Management Extensions (JMX): The JMX technology offers a set of capabilities to manage and monitor any system component: from devices to Java objects. The JMX is based on a 3-level architecture: *Probe level, Agent level and Remote Management Level.* The Probe level is composed by the probes (called *MBeans*), and every MBean represents a Java object. The Agent level (called *MBeanServer*) is the core of the JMX technology and acts as intermediary between MBeans and the external applications. Finally, the Remote Management Level allows external applications communicate with the MBeanServer via JMX connectors or protocol adapters. So, JMX allows to connect and communicate with Java objects (MBeans) in runtime without modify the application source code allowing to interact with them, transparently. In this experiment I would be using this technology to connect and monitor a server system through a client system and collect the memory heap dump at regular intervals necessary to analyze and compare different heap dumps taken at different time samples and with different workload. To use this technology I would be using *jstatd* tool and monitor through another tool *jvisualvm*, both of which are provided in the Sun Hotspot JDK 1.6 bundle.

Used Tools

Apache JMeter: Apache JMeter is a 100% pure Java desktop application designed to load test client/server software (such as a web and mail application). It may be used to test performance both on static and dynamic resources such as static files, Java Servlets, CGI scripts, Java objects, databases, FTP servers, and more. JMeter can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load types [12]. Additionally, JMeter would help me to regression test my application by letting me creates test scripts with assertions to validate that my application is returning the results I expect. For maximum flexibility, JMeter helps me to create these assertions using regular expressions.

Jstatd: The jstatd tool displays performance statistics for an instrumented Hotspot JVM. It is a server that allows JVM monitoring tools like jps and jstat to access JVM processes from a remote machine. The target JVM is identified by its virtual machine identifier, or vmid option described below. It is an RMI server application that monitors for the creation and termination of instrumented Hotspot Java virtual machines (JVMs) and provides an interface to allow remote monitoring tools to attach to JVMs running on the local host. The jstatd server requires the presence of an

RMI registry on the local host. After jstatd is started, JVM monitoring tools, *jps* and *jstat*, can connect to it from a remote machine and access local JVM processes as shown in figure 3 below [13]:

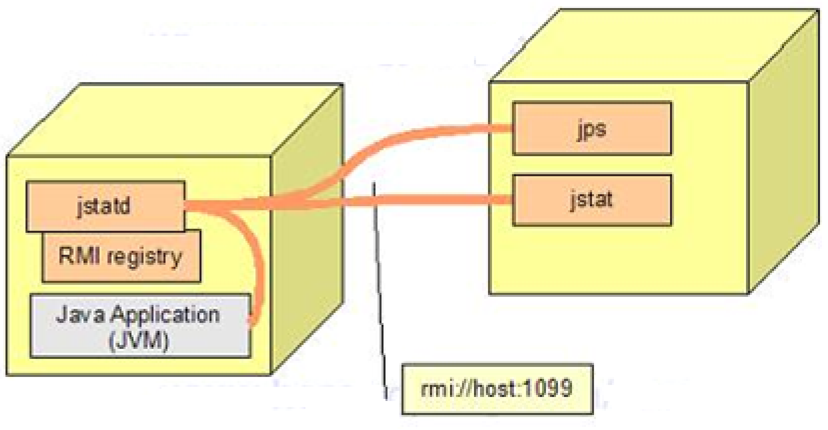


Figure 3: Connection of jps/jstat from a Remote machine using Jstatd

JVisualVM: Java VisualVM is an intuitive graphical user interface that provides detailed information about Java technology-based applications (Java applications) while they are running on a given Java Virtual Machine (JVM). The name Java VisualVM comes from the fact that Java VisualVM provides information about the JVM software visually. Java VisualVM combines several monitoring, troubleshooting, and profiling utilities into a single tool [14]. Java VisualVM is useful to Java application developers to troubleshoot applications and to monitor and improve the applications' performance. Java VisualVM can allow developers to generate and analyse heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweight memory and CPU profiling.

JVMMon: The SAP JVM Monitoring Tool JVMMon is an SAP JVM specific tool for a wide range of monitoring tasks. It is part of the SAP JVM delivery and was developed in order to monitor the internal behavior of the Java Virtual Machine. It is a distributed monitoring system based on JVMTI and JMX technologies [15]. It can attach to a running SAP JVM, no matter if it is running in an application server environment or stand-alone, and display comprehensive monitoring information or control various aspects of the VM configuration. VMs running on the local host are found and listed without further preparation. To manage SAP JVMs on a remote host, a small daemon program, the jvmmond must be running on the remote host. This daemon will provide a network connection port for the monitoring tool to connect to. Here are some of key features of JVMMon:

- Transparent execution of Java Applications on Instrumented VMs

- On-line monitoring
- Low impact on JVM Performance
- Concurrent monitoring of multiple virtual machines
- Automatic restart of the virtual machine upon crashes

The figure reported below shows the main components of JVMMon and their interconnection:

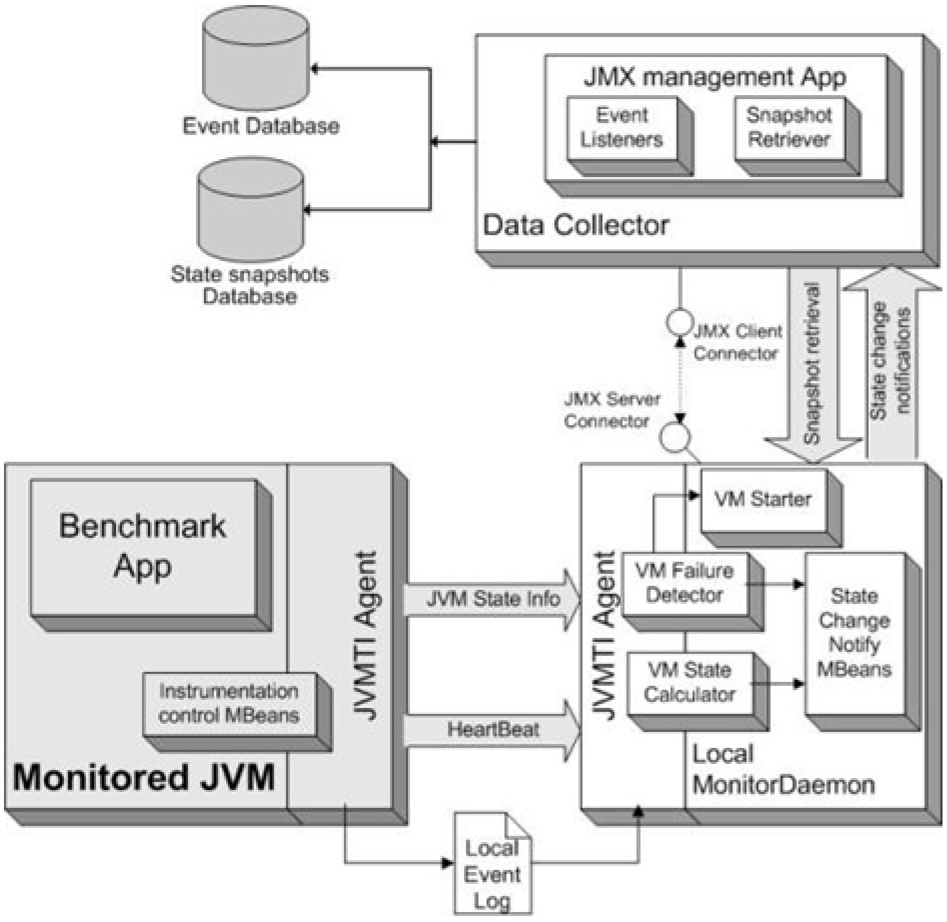


Figure 4: Main components of JVMMon and their interconnection

Eclipse Memory Analyzer: The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps us find memory leaks and reduce memory consumption. I would be using the Memory Analyzer to analyze productive heap dumps with hundreds of millions of objects, calculate the retained sizes of objects and run a report to automatically extract leak suspects.

Postage: Postage is a stand-alone pure-java application generating mail traffic on mail servers. It uses standard mail protocols to do this, currently POP3 and SMTP [16]. Therefore, it is well suited for testing any mail server supporting those protocols. It is a small stand-alone tool to put load onto a running James Server and record its performance. It was particularly created for running against Apache James Server and contains special feature for it. It is also able to monitor the Server for memory and thread consumption.

Proposed approach

In order to cope with the analysis of resources usage by complex long-running applications, I adopted an approach based on stress tests. The target application is executed under stressful environmental conditions (e.g., by requesting a high number of operations) for a long period, in order to point out resource leaks, that otherwise would not be identified during the testing phase of the system (e.g., functional and performance tests), and could lead to unpredictable failures during the operational phase. Moreover, stress tests are repeated under different environmental conditions, in order to highlight resource leaks dependencies on workload parameters (which, in turn, can be used for long-term predictions about resource consumption). During the program execution field data are collected in order to be processed by external analysis tools [8].

The memory depletion has been measured with the same techniques at two different layers: i) application, measuring the total size of reachable objects in the heap area; ii) JVM, measuring the total heap usage, including the space used to store JIT-compiled methods. At the application layer, sample collection is synchronized with garbage collection events, thus reducing noise in samples due to short-lived objects in the young generation; at the JVM layer, total heap usage has been measured taking into account usage data for each generation of the JVM Heap.

In order to estimate evidences of memory leak in JVM workload parameters, 7 experiments, each with duration of 6-8 hours, have been performed. As for application workload, I used a long running mailing application, namely the JAMES mail server [17]. The server has been stimulated with *JMeter* which is a load generator delivering a constant workload from a client server. The load generator allows me to define the workload simply specifying the number of mails per minute and the size of each mail. In order to stress the server uniformly, mail size is kept constant among all experiments. This is done using the James stress testing tool *Postage* and increasing the workload in every experiment. Thus the number of mails sent and received continuously increases among experiments, thus imposing progressively increasing workloads to the mail server and to the JVM. I would then monitor the performance and memory usage continuously with the help of *JVisualVM* through a JMX connection to the Remote server from the client side with the help of *Jstatd*.

Field data would be collected by talking heap dumps at regular intervals and comparing them using a monitoring infrastructure, named *JVMMon*. It is capable of capturing i) resource usage data at different layers: application, JVM, and OS; ii) JVM workload parameters, related to the activity of its internal components, such as object allocation rate, number of threading events, garbage collection frequency, and time spent in JIT compilation. After collecting the heap dump I would analyze the data using *Eclipse Memory Analyzer* and compare various data taken along different time period and different Workload and check for the evidence of Memory leakage in JVM.

Experimental Case Study

After presenting my proposal, I have conducted a set of experiments to evaluate the effectiveness of my approach. My idea is to test my prototype to determine the cause of a memory leak. In next subsection I present the experimental environment used in my experiments.

Experimental Setup

In this section I describe the experimental setup used in all experiments presented below. The experimental environment simulates a real web environment, composed by the mail server, the monitoring agent and the client's machine. The monitoring infrastructure was used to collect resource usage and system activity data from a workstation running JAMES mail server on a Sun Hotspot JVM v.1.6.0 24. The workstation was a Pentium IV server equipped with 2GB RAM and running Windows OS. The JVM was started with the typical server configuration and a maximum heap size of 512 Megabytes; it was configured to run serial, stop-the-world collectors on both generations. No other application is competing for system resources with the JVM and the server machine is started with just minimal system services.

To get a high level of precise and accurate data I won't be using any JVM services on the server side and even monitor it from another system. I would then setup a JMX connection to the Remote host from the Client system with the help of *jstatd* tool. This connection would allow the monitoring agent (here *JVisualVM*) to connect to the server JVM and collect its memory usage data which would provide a better insight to issues like which objects are talking how much memory, how many thread processes are alive, how much heap memory couldn't be used further, etc. I would then record the data and use another Monitoring Agent attached to the server VM to take Heap Dumps at regular time interval. This would help me a great deal in knowing about the exact cause of memory leakage by analyzing it through a memory analyzer (here *Eclipse Memory Analyzer*).

Finally I would be able to analyze through charts and histograms the signs of Memory Leaks. An example of a sample experiment is presented in the chart below in Figure 5 and 6.

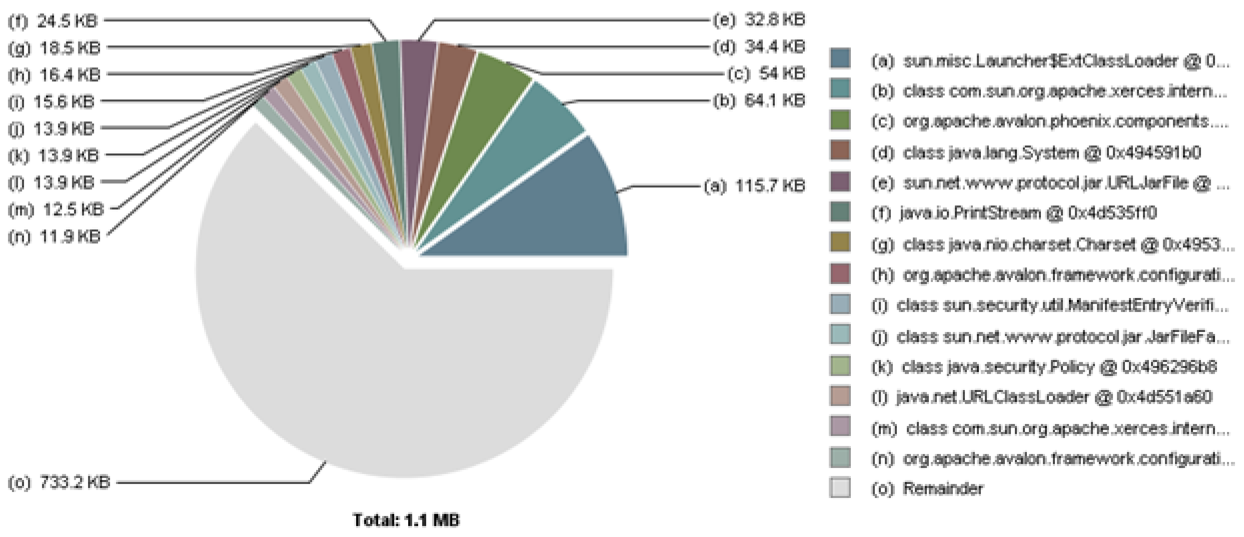


Figure 5: Chart showing Top Consumers of resources

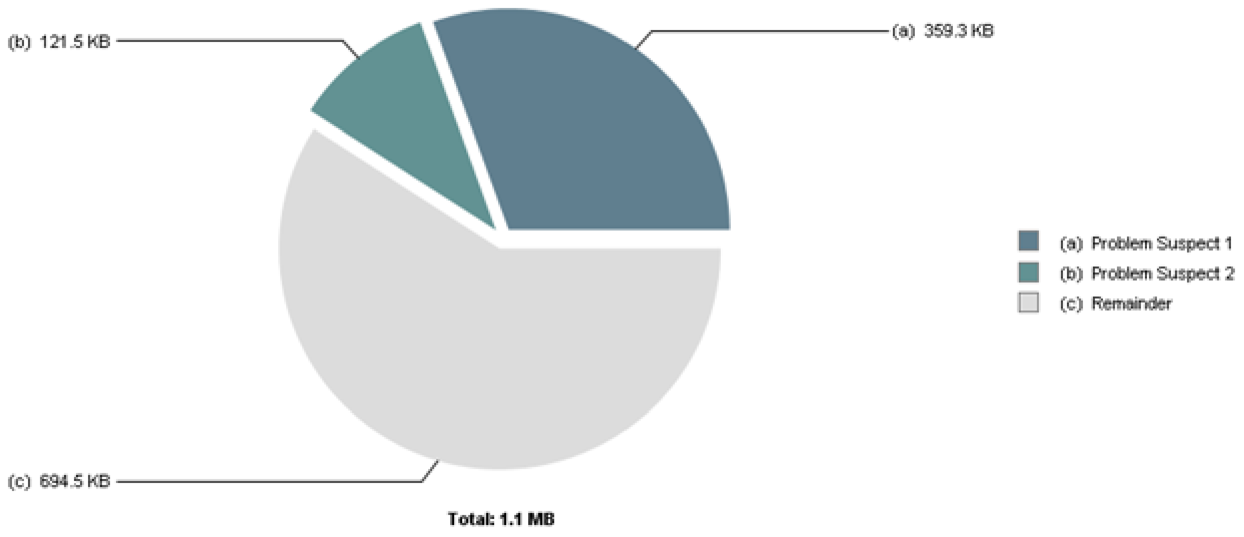


Figure 6: Chart showing Memory Leak Suspects

The first Chart depicts the Top Consumers which are talking the resources of the system. It provides an overview of the biggest objects as per their resource usage. The second Chart is used to view the Memory Leak Suspects and is of my utmost concern. It gives the overview of the possible suspects of memory depletion and shows the Remainder section which is the resource that it couldn't find but is still consuming memory. This chart would be the base of my approach because this would help me to determine the memory depletion in the entire course of my experiment.

In order to avoid accumulation of aging effects of the previous executions the entire server is restarted after each experiment. The JVM monitoring infrastructure collects the greatest part of the information required for the analysis. The monitoring agent attached to the server VM, *JVMMon*, has been extensively tested in order to guarantee that it is aging-bug free. I first evaluated the highest workload (in terms of email per minute) the mail server is capable of sustaining without refusing any connection. I estimated this limit to be about 1550 e-mails per minute. Therefore the workload applied in the 7 experiments ranges from 600 emails /min to 1500 emails /min.

Using the above proposed approach I configured both the Mail Remote Server and a Client attached to it. I monitored the server load using *JVisualVM* which uses a JMX connection to the server and got the data and results of the experiment which is explained in the next section.

Experimental Results

In this section I would present the results of the analysis of data collected in the 7 experiments. Due to my limitations I could run the servers only for few hours. I started collecting heap dumps after two hours of starting the server and after an interval of every half an hour. Although running the server for only 6-8 hours won't present me a precise result, strikingly the data that I got was sufficiently showing evidences of Memory leaks in the JVM layer of the mail server.

I first collected the data on a normal run with a workload of 120 mails /min. I used the Jmeter to stress the server with this constant rate. Using JVMMon I dumped the heaps after a regular interval of half an hour. I then used the Eclipse Memory Analyzer to analyze the heap dumps and compare it. The data collected through the Memory Analyzer is presented in the form of the table below.

Table 1: Result showing heap dump sizes when server is running with a constant workload

Time(in hr)	No. of Objects	Size (in KB)	Rem (in KB)	No. of GC roots
0200	26,694	1126.4	694.5	891
0230	27,193	1184.7	701.2	950
0300	27,537	1191.3	716.2	980
0330	27,626	1197.9	723.0	991
0400	28,469	1215.6	839.5	1113
0430	29,699	1255.6	879.0	1114
0500	30,749	1289.8	913.2	1114

Time	Interval after starting the server (say 00 hr) when the sample was taken
Size	Size of heap dump created
Rem	Memory used but has no use and can't be determined by the Memory Analyzer. This is the major suspect of the memory leak problem.
No. of GC roots	No. of Garbage Collector root classes attached

By analyzing the table we can summarize that as the time that the collected heap dumps increased, the Memory leaks were increasing as well. This trend kept on increasing and the no. of objects created was increasing as well but the Garbage collector was not able to remove it because the addresses of many objects were still referenced though the object was of no use after. Now in the second part I took the sample data after two hours of starting the server but this time I increased the workload with each experiment. I noted down the result in the table shown below.

Table 2: Result showing heap dump sizes when server is running on different workloads

WL (mails/min)	No. of Objects	Size (in KB)	Rem (in KB)	No. of GC roots
600	53766	2331.2	1331.2	1141
750	56980	2491.5	1433.6	1172
900	59397	2645.7	1536.0	1175
1050	63089	2690.2	1740.8	1181
1200	66038	2791.8	1839.0	1184
1350	69249	2892.5	1973.8	1186
1500	71839	2997.6	2150.4	1190

WL	Workload, i.e., mails per minute sent through the client system

Again, we can clearly see signs of Memory leak by observing the Remainder column here. Actually these are the memory which are still occupied by the JVM but has no use and thus, no record of it. The Garbage collector is not able to detect and remove it because it still holds reference to it.

Thus my work clearly detected the presence of Memory Depletion in the form of leakage. Also it is observed that Leakage increases as we delay time or the Workload over the JVM is increased. Since we know that Memory depletion in the form of Leakage is the major cause of Software Aging, we can hence conclude that JVM is facing serious aging issues and that it should be rigorously and precisely tested for its dependability.

These results outline the presence of software aging phenomena inside the JVM, manifested as throughput loss and memory depletion. From experimental results it has been demonstrated that:

- The JIT compiler is a source of memory depletion due to data stored in the Native Code Cache. However these depletion dynamics cannot be regarded as a serious threat for the JVM.
- Sudden downfalls in Garbage Collector activity causes free memory to decrease with a very high slope.
- The interface between the JVM and the operating system seem to be really critical from the throughput loss perspective.

Conclusion and future work

The report addressed software aging issues in the Java Virtual Machine, in terms of estimating dynamics for memory depletion, and of evaluating the impact of JVM workload parameters on aging phenomena. A massive experimental campaign was conducted by using a long running server application hosted by the Sun HotSpot JVM. Analysis of collected data revealed the presence of software aging phenomena, which manifested as memory depletion.

Although results presented are valid only for the Sun Hotspot JVM implementation, the approach adopted in this paper is general and may be applied not only to different JVM implementations, but also to completely different software systems. Provided results also outlined directions for future research. On one hand, memory depletion trend at the operating system layer should be studied in detail, taking into account also OS-related workload parameters, and performing experiments across different Operating Systems. On the other hand, the approach adopted in this paper can be refined in order to assess a methodology capable of

1. Isolating the contribution to aging phenomena given by the different layer of a software system,

2. Identifying, for each of these layers, which workload parameters are more relevant to aging trend, and

3. Evaluating the impact of these workload parameters on software aging dynamics.

This report showed that several problems can arise when coping with memory leaks in complex applications. The practical experience with the case study helped me to obtain the following issues, and leaved some questions open, which may deserve further research efforts in the near future:

- OTS items are a significant source of memory leaks.

- Developers should correctly fix known memory leaks.

Finding the cause of a memory leak can be a tedious process, not to mention one that will require special debugging tools. However, once we become familiar with the tools and the patterns to look for in tracing object references, we will be able to track down memory leaks. In addition, we'll gain some valuable skills that may not only save a programming project, but also provide insight as to what coding practices should be avoided to prevent memory leaks in future projects.

References

[1] D. Cotroneo, S. Orlando, and S. Russo, “Characterizing Aging Phenomena of the Java Virtual Machine,” in Proc. Of the 26th IEEE Symposium on Reliable Distributed Systems (SRDS), 2007, pp. 127–136.

[2] D.Cotroneo, S.Orlando, and S.Russo. “Failure Classification and Analysis of the Java Virtual Machine”. The 26th International Conference on Distributed Computing Systems (ICDCS 06), Lisboa, Portugal, July 2006.

[3] Javier Alonso and Jordi Torres, Josep Ll. Berral, and Ricard Gavald‘a, “J2EE Instrumentation for software aging root cause application component determination with AspectJ”, 2010

[4] Autran Macêdo, Taís B. Ferreira, and Rivalino Matias Jr. “The Mechanics of Memory-Related Software Aging”, 2011

[5] Q. Ni, W. Sun, and S. Ma., “Memory Leak Detection in Sun Solaris OS”, in Proc. of International Symposium on Computer Science and Computational Technology, 2008.

[6] T. Lindholm and F. Yellin. “The Java(TM) Virtual Machine Specification”, Sun Microsystems, 2nd edition, 1999.

[7] C. Erickson, “Memory leak detection in embedded systems” .Linux Journal, 2002(101):9, September 2002.

[8] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia and S. Russo, “An experiment in memory leak analysis with a mission-critical middleware for Air Traffic Control” Campania, Naples, Italy, 2008

[9] G. Xu and A. Rountev. “Precise Memory Leak Detection for Java Software using Container Profiling” in Proc. of the 30th Intl. Conf. on Software Engineering (ICSE), 2008.

[10] M. Grottke, R. Matias, and K. Trivedi, “The fundamentals of software aging,” In Proc of Workshop on Software Aging and Rejuvenation, in conjunction with IEEE International Symposium on Software Reliability Engineering. 2008.

[11] Dino Distefano and Ivana Filipovi'c "Memory Leaks Detection in Java by Bi-Abductive Inference", in Proc. of International Symposium on Computer Science and Computational Technology, 2008.

[12] JMeter User Manual Page http://jakarta.apache.org/jmeter/usermanual/

[13] Jstatd User Manual Page http://download.oracle.com/javase/6/docs/technotes/tools/sha

[14] http://download.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html

[15] http://www.mobilab.unina.it/JVMMon.htm

[16] http://james.apache.org/postage/

[17] Working with *James*, http://www.ibm.com/developerworks/java/library/j-james1/index.html