Monitoring of resource consumption in Java-based application servers

Jarle G. Hulaas[1]
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
Jarle.Hulaas@epfl.ch

Dimitri Kalas
CUI
University of Geneva
CH-1211 Genève 4, Switzerland
kalas8@etu.unige.ch

Abstract. To realize dependable and profitable information and communication systems, computing resources (CPU, memory, network bandwidth) should be taken into account to a much wider extent than they are today. E-commerce infrastructure, such as application servers, are especially concerned by this issue. This paper shows how an existing Java-based code transformation framework was extended to enable transparent monitoring of resource consumption within the open source Apache Tomcat servlet engine.

1. Introduction

Today's communication and information systems are growing steadily to encompass the many application fields that are emerging both in the consumer and the corporate segments. The quest for ever more functionality puts a strong pressure on underlying infrastructures, which, as a result of a fast-paced expansion, often become deeply distributed, heterogeneous and hence difficult to manage. It is therefore necessary to develop tools for monitoring and controlling such distributed systems in their entirety, otherwise it will be harder and harder to provide guarantees concerning security, reliability and availability of services.

Our vision is a distributed e-commerce environment, possibly run by a service provider like a telecom operator, where behavioural data is collected continuously and transparently, thus supporting the provision of security, Quality-of-Service and billing. This will help ensuring an optimal exploitation of existing infrastructure and its adequacy to business needs before planning investments in further technology. Moreover, the data gathered to these ends will also help better scrutinizing and understanding the clients' behaviours, thus providing marketing departments with very comprehensive user profiles. As a first step towards this, we address here the issue of monitoring a Java servlet engine, which streams the collected data into a SQL database.

Heterogeneous and distributed systems are often programmed in Java, due to the inherent portability of applications implemented in this language. One aspect which is however missing in Java (as well as in most mainstream environments) is the ability to monitor the amount of resources (CPU, memory, network bandwidth) consumed during a program's execution. We have developed a tool called J-RAF (the *Java Resource Accounting Framework,* http://www.jraf2.org), which processes and transforms Java programs in their compiled form, "forcing" applications to provide real-time information about their resource consumption, and, to a certain extent, to conform to related restrictions. Our approach relies on a 100% portable transformation technique applied directly to executable Java byte-code; we have demonstrated that it was possible to dynamically add accounting of CPU and memory to any Java2 application without incurring an excessive performance overhead.

In this paper we first describe the basic J-RAF tool (section 2), then the extensions needed for applying it to the Apache Tomcat Java servlet engine (section 3), which is to be used as a monitored application server. Finally, we discuss the perspectives and the limitations of our approach (section 4), before concluding.

1. Part of this work was performed while the author was at the CUI - University of Geneva.

2. The J-RAF Tool

Prevailing approaches to provide resource control in Java-based systems rely either on a modified Java Virtual Machine (JVM), on native code libraries, or on program transformations. For instance, the Aroma VM [5] is a specialized JVM supporting resource control. JRes [4] is a resource control library for Java, which uses native code for CPU control and rewrites the bytecode of Java programs for memory control. On the other hand, the *Java Resource Accounting Facility* J-RAF [3] is exclusively based on program transformations in order to facilitate portability. In this approach the bytecode of applications is rewritten in order to make their CPU and memory consumption explicit (we call it CPU and memory reification). Programs rewritten with J-RAF keep track of the number of executed bytecode instructions (CPU accounting) and update a memory account when objects are allocated or reclaimed by the garbage collector.

Resource control realized with the aid of program transformations offers an important advantage over the other approaches, because it is independent of a particular Java runtime system. It works with standard JVMs and may be integrated into existing mobile code environments. Furthermore, this approach enables resource control within embedded systems based on modern Java processors, which provide a JVM implemented in hardware that cannot be easily modified.

2.1. Basic Principles of Program Transformation

In our approach, the bytecode of each Java method is rewritten to expose its CPU and memory utilization. Every thread has an associated Account Object (AccOb) for CPU and memory. These AccObs are updated while the thread is executing the rewritten methods. For CPU accounting, in each basic block of code the number of executed bytecode instructions is added to a counter within the *CPU AccOb*. For memory accounting, instructions that update the *Memory AccOb* are inserted directly before all memory allocation sites. The details of these transformations are covered in [1],[2]. It must be noted that our initial motivation was to prevent Denial-of-Service attacks in mobile code environments, hence the strategy is to always account for resources immediately *before* they are consumed.

```
public class <X>AccOb {
   private static final ThreadLocal
      currentAccOb = new ThreadLocal();

   public static <X>AccOb getOrCreate() {
      <X>AccOb a = (<X>AccOb)currentAccOb.get();
      if (a == null) {
         a = new <X>AccOb();
         currentAccOb.set(a);
      }
      return a;
   }
...
}
```

Figure 1. Associating AccObs to Threads

```
void f(MemAccOb mem, CPUAccOb cpu) {
   ... // method code with accounting
   g(mem, cpu); // relay parameters in methods
}
```

Figure 2. Method Rewriting Scheme, Passing AccObs as Additional Arguments.

Figure 1 illustrates how AccObs can be bound to threads. The meta-variable X may be replaced either with Mem (for a memory AccOb) or with CPU (for a CPU AccOb). A thread-local variable associates an AccOb with each thread. The static `AccOb.getOrCreate()` method returns the AccOb associated with the calling thread. The first time a new thread invokes this method, a new AccOb is created. For the sake of readability, this paper shows all code transformations at the level of the Java language, whereas our implementation operates at the JVM bytecode level.

Each method needs to access an AccOb, at least the CPU AccOb; therefore they are rewritten as shown in Figure 2, by passing the required AccObs as additional arguments. Typically, rewriting the bytecode of an application is not sufficient to account for and control its resource consumption, because Java applications use the comprehensive APIs of the Java Development Kit (JDK). Therefore, it is necessary to prepare resource-aware versions of the JDK classes in order to calculate the total resource consumption of an application. Ideally, the same bytecode rewriting algorithm should be used to rewrite application classes as well as JDK classes. However, the JDK classes are tightly interwoven with native code of the Java runtime system, which causes

subtle complications for the rewriting of JDK classes, as reported in [3]. Basically, the trick consists in adding various wrapper methods, in order to cope with the different invocation schemes that are possible (i.e. with or without the right to modify parameter passing as in Figure 2). Figure 3 shows a complete example of method rewriting (as above, we represent in Java source form the operations that are actually performed at the bytecode level).

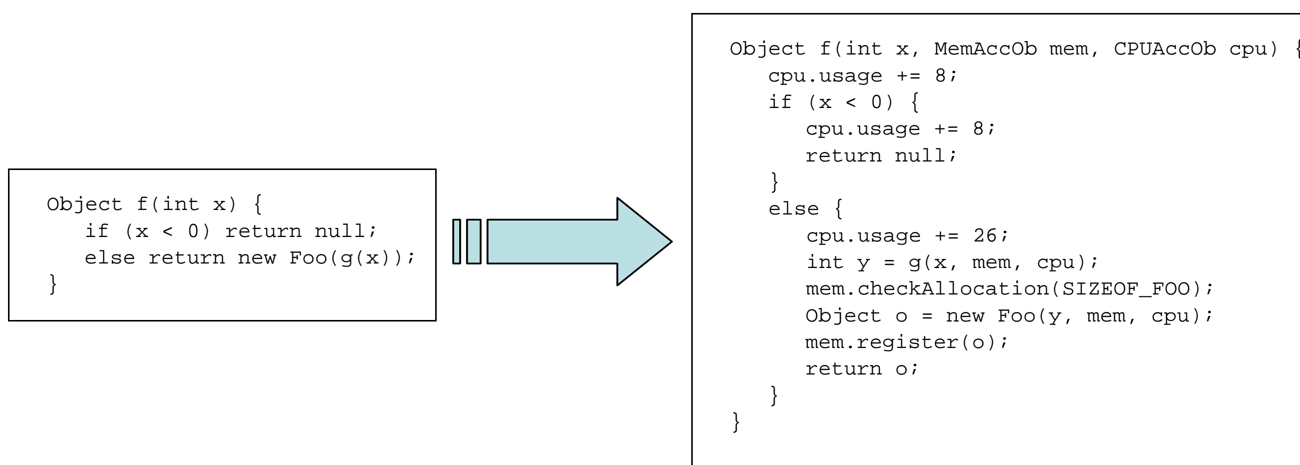

Figure 3. A Sample Method and the Corresponding Rewritten Version

This approach has been assessed in several environments, using JVMs and associated JDKs from Sun and IBM. For our evaluation we use the standard SPEC JVM98 benchmark suite (http://www.spec.org/osg/jvm98/), run on a machine with a 1800 MHz Intel Pentium IV processor and 512 MB of RAM, under Linux 2.4.7-10. The results for CPU accounting are summarized in Table 1. Our memory accounting implementation is not yet ripe for benchmarks. Mainly because of the wrappers, but also because of the actual accounting operations, the size of the code currently increases by about 80% for the different JDKs, and by about 47% for the applications.

JVM + JDK version	Average overhead
IBM JDK 1.3.1 Classic VM	17%
Sun JDK 1.3.1 Hotspot Client VM	35%
Sun JDK 1.4.0 Hotspot Client VM	32%

Table 1: Execution Time Overhead due to CPU Accounting

3. Extensions Needed for Monitoring a Servlet Engine

The basic J-RAF tool, as described and tested above, can have no prior knowledge about multi-threaded applications, and therefore considers by default that each thread is a separate application, agent or servlet. This is of course not sufficient if multithreaded applications are to co-exist inside a single JVM, as often is the case with servlet engines. In the following sub-sections we describe how this issue was addressed, as well as several other extensions that are required for monitoring application servers, taking the case of the Apache Tomcat 4.x servlet engine (http://jakarta.apache.org/tomcat/).

3.1. Addition of Bandwidth Accounting Objects

Companies that rent out application servers will often want to be able to verify that their clients' servlets do not exceed some network bandwidth that was fixed by contract. Following the same basic principles as described above, it is possible to obtain a fine-grained view of the respective bandwidth consumption of servlets inside a shared JVM. This may also serve statistical purposes.

The implementation of Bandwidth (BW) accounting is realized in a more straightforward manner than what was required for CPU and memory, because BW consumption spots are easier to locate in the code. Considering the simple example of a servlet receiving an HTTP `GET` request and sending back a finite Web page (in HTML) in response, inbound communication is represented by the predefined (`Http`)`ServletRequest` classes (of which the `getContentLength` method returns the size of the request in bytes), and outbound communication is performed through predefined classes like `ServletOutputStream` and `PrintWriter`. By creating wrappers around such predefined classes, it is possible to intercept all communication, in order to take note of the size of the packets, and even to prevent their execution if a given limit is exceeded. To make this transparent to the application (i.e. the servlet), a new implementation is created for the (`Http`)`ServletResponse` interfaces, from which references to the response output stream are obtained. If we now consider more complex examples, where input and output streams of a priori unknown (or even infinite) sizes are required, the same principle of wrapping of predefined classes is employed.

Another issue that had to be addressed while incorporating BW AccObs into J-RAF, was to choose how the transformed servlet code is to update the corresponding '`usage`' attribute (see Figure 3). For CPU accounting, one such variable is created for each thread, in order to avoid race conditions, and a special thread has to wake up periodically to gather the values and reset[1] the variables of all threads belonging to a given group and implementing e.g. an application, an agent or a servlet. For memory accounting, a notion of thread group is supported, allowing '`usage`' variables to be shared and updated through synchronized methods. This approach shows a performance penalty, as invocation of synchronized methods is an expensive operation, but it is less error-prone, and reasonable if we consider that allocations of (dynamic) memory are less frequent than executions of bytecodes by the CPU. The same reasoning was applied to the case of BW AccObs, and they are therefore shared inside thread groups.

Contrarily to CPU and memory AccObs, BW AccObs are not passed as additional arguments to Java methods. This is because it is cheaper to link them to instances of the above mentioned wrapper classes.

3.2. The RequestAccount Class

We have seen until now that each thread is by default considered an independent application. In order to support multi-threaded servlets, we need a `RequestAccount` class to keep track of all threads "belonging to" the same request.

Additionally, we have to determine exactly when a thread starts and finishes working for a given request. This is difficult since the Tomcat servlet engine implements thread pooling, i.e. it recycles threads instead of creating and deleting instances dedicated to single requests. To this end, the new J-RAF tool has to recognize specific methods, like `doGet`, `doPost` and `doService`, that define request processing entry points in servlets. The bytecode of these methods is modified in order to signal the start and the end of a request processing. Moreover, if auxiliary worker threads are created by the servlet in order to serve a given request, the modified bytecode will take care of automatically registering them in the group, associating them with the adequate `RequestAccount` object.

3.3. Client-level Accounting

Our goal is to be able to know the total resource consumption of each client. Until now, we have presented the `RequestAccount` class, which manages such information on a per request basis. Obviously, we also need an entity which periodically sweeps all parallel instances of `RequestAccount` created on behalf of the same client. That's the job of the `ClientAccount` class.

---

1. Currently we use standard, 32 bit integers, and therefore this periodicity must be calibrated with respect to the speed of the underlying processor to avoid overflow of individual variables, and hence loss of information. This problem will disappear for a long time once we decide to switch to long, 64 bit integers. The same remark applies to bandwidth consumption.

We need to determine how to distinguish between the various clients. We consider here that each client is associated with a unique set of servlets, resource limits and access rights. Clients are nevertheless difficult to identify, since each servlet programmer decides what level of security to apply, and, if applicable, what authentication mechanism to use (e.g. by IP addresses or by login/passwords). In order to overcome that issue, the servlet programmer will (exceptionally!) have to collaborate with J-RAF and explicitly give the identity of the user (as a `ClientID` object that can be customized to allow additional, application-specific attributes to be recorded and logged in the SQL database as described below). Thus we may be confident that the user has been correctly authenticated, and that the corresponding rights will be granted by our so-called `RightsManager`.

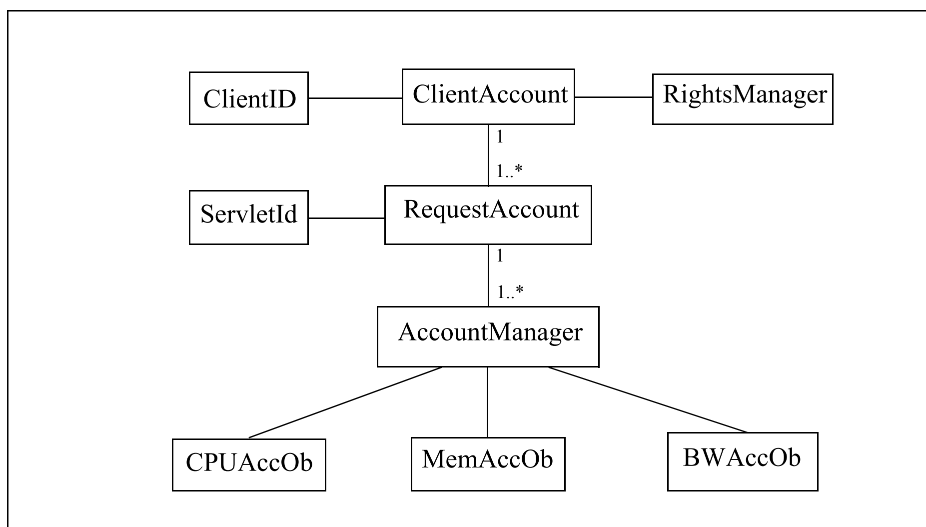

Figure 4. Simplified Overall Class Diagram

3.4. SQL Database for Accounting Data

For persistently storing resource monitoring logs and retrieving access rights on a client-level basis, we use the open source MySQL database (http://www.mysql.com/). The monitoring data that needs to be stored is structured by records as follows: one for each request initiated, plus one periodically, for each time the `ClientAccount` class activates. This data may be buffered instead of being written directly to the database, as long as we do not support load-balancing schemes in clustered application servers. The actual data contained in each record are: `ClientId`, absolute time, CPU/memory/bandwidth consumed, plus optional fields added by the servlet programmer in the `ClientId` object. An example of such additional fields is the session identifier, which is not directly used for resource monitoring.

4. Perspectives and Limitations

This paper described a basic infrastructure for monitoring, and partly controlling resources like CPU, memory and network bandwidth. The J-RAF tool was initially designed with security in mind. In the frame of application servers, it can however also be used as a building block for implementing *usage-based* (as opposed to *content-based*) billing; the issue of pricing is however out of the scope of this paper. Another application area is to monitor application servers for ensuring Quality-of-Service (QoS): when requests take longer to execute than normally, the detailed logs can help identifying the source of the problem. By implementing appropriate scheduling policies, it is even possible to realize more active QoS mechanisms. There is however no definitive application-level API for manipulating resources explicitly, and this makes it difficult to share a pool of resources among affiliated servlets.

Some of our limitations are directly related to the J-RAF approach, e.g. that we require a Java 2 platform and that the use of Java's reflexive capabilities are partly forbidden in order to prevent the application programmer from circumventing our security mechanisms. Moreover, as J-RAF works at the application level, it is not possible to provide guarantees which require ac-

cess to the lower levels, e.g. real-time scheduling (see our discussion in [2]). We might mention that we have until now developed and tested our tools exclusively around the standard Java 2 (J2SE) platform, whereas J2EE (Java 2, Enterprise Edition) might be more appropriate in the frame of application servers. Finally, from a legal standpoint, it is not impossible that application providers might object to their servlets being transformed by a tool like ours. To prevent this, some kind of formal proof of correctness would be useful.

5. Conclusion

We are convinced that, to realize dependable and profitable information and communication systems, computing resources (CPU, memory, network bandwidth) should be taken into account to a much wider extent than they are today. This paper presented J-RAF, a Java-based code transformation framework, and how it was extended to enable transparent monitoring of resource consumption within the Apache Tomcat servlet engine. Some of the perspectives and limitations of this approach were also discussed.

6. Acknowledgements

The authors would like to thank Walter Binder, Alex Villazón and Vladimir Calderón for their substantial collaboration. This work was partly supported by the Swiss National Science Foundation.

7. References

[1] W. Binder, J. Hulaas, and A. Villazón, “Resource control in J-SEAL2”, Technical Report Cahier du CUI No. 124, University of Geneva, Oct. 2000. ftp://cui.unige.ch/pub/tios/papers/TR-124-2000.pdf.

[2] Walter Binder, Jarle Hulaas, Alex Villazón, and Rory Vidal, “Portable resource control in Java: The J-SEAL2 approach”, In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001), Tampa Bay, Florida, USA, October 2001. ftp://cui.unige.ch/pub/tios/papers/OOPSLA2001.pdf

[3] Vladimir Calderón, "J-RAF - The Java Resource Accounting Facility", Master's thesis, CUI, University of Geneva, June 2002. ftp://cui.unige.ch/pub/tios/papers/Diploma-Calderon.pdf

[4] G. Czajkowski and T. von Eicken, “JRes: A resource accounting interface for Java”, In Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98), volume 33, 10 of ACM SIGPLAN Notices, pages 21-35, New York, USA, Oct. 18-22 1998. ACM Press.

[5] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith, “NOMADS: toward a strong and safe mobile agent system”, In C. Sierra, G. Maria, and J. S. Rosenschein (Eds.), Proceedings of the 4th International Conference on Autonomous Agents (AGENTS-00), pages 163-164, NY, June 3-7 2000. ACM Press.