Monitoring of WS-Based Applications

Lechoslaw Trebacz[1], Piotr Handzlik[2], Wlodzimierz Funika[3,⋆], and Marcin Smetek[3]

[1] Department of Computer Methods in Metallurgy, AGH, Krakow, Poland
[2] Department of Physical Chemistry and Electrochemistry, AGH, Krakow, Poland
[3] Inst. Comp. Science, AGH, Krakow, Poland
ltrebacz@metal.agh.edu.pl, piohandz@interia.pl
{funika, smetek}@uci.agh.edu.pl
Tel.: (+48 12) 617 44 66; Fax: (+48 12) 633 80 54

Abstract. The paper presents a Java-related monitoring platform for distributed applications which are oriented towards the use of Web Services (WS). We focus on the building a monitoring platform based on a well-defined interface (OMIS) between a monitoring system organized as middleware and tools that use the facilities provided by the monitoring middleware for WS-based applications. We aim at the observation and analysis of SOAP messages (data) exchanged with a Web Service (requests and responses). Our system is intended to monitor the characteristics of the remote call, especially: request time, transport time, cache access time, response time.

Keywords: monitoring tools, Web Services, OMIS, J-OCM, Java.

1 Introduction

With the increased possibilities of distributed Java programming, e.g. Web Services (WS), the demand for tool support (performance analyzers, debuggers etc.) for efficient programming increases as well. Nowadays, there are few environments which provide efficient monitoring support for distributed Java programs. One of the opportunities to solve this problem is an approach exploited in the On-line Monitoring Interface Specification [1]. A universal, open interface between tools and a monitoring system and an OMIS compliant monitoring system (OCM) allowed to specify such a Java-oriented monitoring interface (J-OMIS) [2] and monitoring infrastructure, which enables an extensible range of functionality intended for various kinds of tools and programming paradigms.

In this paper we consider extending the functionality of the J-OCM system [3] by the monitoring of WS-based applications. This extension is intended to:

- provide information about running Web Services and their number,
- provide information about the main stages of a WS life cycle, e.g.: receiving a request from the client side, starting particular services, time used to parse an SOAP message, etc.

⋆ Corresponding author.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 549–556, 2006.

- make possible to manipulate particular services, for example: stopping any operation at any moment, calling a method of a Web Service from within the monitoring system etc.
- enable access to available information about a Web Service,
- make possible to check, which stages of a WS life cycle take most of time, to analyze the performance of this application,
- provide information about errors when running a Web Service.

The paper is organized as follows: In Section 2 we focus on the principal features of WS, interesting from the performance point of view. In Section 2 (Related work) we show what issues connected withe the operation of WS are not covered by the existing tools. In Section 3 we briefly describe the J-OCM monitoring system with showing its some features aimed at the monitoring of distributed applications. In Section 4 we focus on the concept of support for WS monitoring in the J-OCM monitoring system and further on implementation details of the WS-related extension to it. We give also some results of overhead measurements when monitoring sample WS-based applications.

2 Related Work

There are a number of monitoring tools aimed at providing performance information and testing of the operation of WS. Such tools as `internetvista` [1] or `Parasoft SOAtest` [2] provide for the user a lot of useful features. They allow the user to verify all aspects of WS, from WSDL validation, to unit and functional testing of the client and server, to performance testing. SOAtest addresses key Web service issues such as interoperability, security, change management, and scalability. On the other side, each of these tools runs as a client application. They test WS by sending requests and waiting for a response. However, an issue is that they do not allow the user to get insight into what really happens *inside* the WS. There is another group of monitoring tools like TAU [15] which allow for advanced performance visualization of distributed applications, but do not provide monitoring information on WS.

Our goal was to overcome this constraint and to provide that our approach can enable to locate which part of WS (initialization, request processing, operation invocation or response) is responsible for performance problems. The system under discussion should be easily extended by new functionality to meet the emerging needs of the WS user. Moreover, we aim at adapting the existing performance analysis mechanisms like this exploited in TAU for the goals of WS monitoring.

3 Web Service

Web Services are a programming paradigm of distributed systems [4, 5, 6], being a programmable application logic accessible using standard Internet protocols.

[1] [`http://www.internetvista.com`]
[2] [`http://www.parasoft.com/`]

Web Services combine the best aspects of component-based development and the Web. As components, Web Services represent the functionality that can be easily reused without knowing how the service is implemented. Web Service features the following:

- WS is accessible over the Web. Web Services communicate using platform-independent and language-neutral Web protocols,
- WS supports loosely coupled connections between systems, by passing messages to each other.
- WS provides an interface that can be called from within another program. The WS interface acts as a liaison between the Web and the actual application logic that implements the Service.

WS can communicate by the Extensible Markup Language (XML) [7]. Web Services use XML to describe their interfaces and to encode their messages. XML underlies three standards used by WS: SOAP which defines a standard invocation protocol for WS [8], WSDL which defines a standard mechanism to describe a WS [9], and UDDI which provides a standard mechanism to register and discover WS [10].

The needs in the monitoring of Web Services are mainly motivated by the performance problems when using this programming paradigm. So the main goal of the monitoring of WS is to obtain as many information as possible to improve WS performance, to discover places where a WS-based application has bottlenecks, memory leaks, and errors.

4 J-OCM Monitoring System vs. Accessing JVM

J-OCM is a monitoring system for Java applications[3], compliant with the OMIS specification [1] extended by support for distributed Java programs. The idea of OMIS (On-line Monitoring Interface Specification) is to separate the functionality of a monitoring system from monitoring tools. J-OMIS is a monitor extension to OMIS for Java applications intended to support the development of Java distributed applications with on-line tools. The work dates back to 1995 when OMIS, a monitor/tool interface specification was released. J-OMIS specifies three types of services: *information services* (providing information about an object), *manipulation services* (allowing to change the state of an object), *event services* (trigger arbitrary actions whenever a matching event is raised).

J-OCM comprises the following components: (1) *Node Distribution Unit* is part, which is responsible for distributing requests and assembling replies; (2)*Local Monitor* is a monitor process, which resides on a node. LM's extensions provide new services defined by J-OMIS, which control Java Virtual Machine via agent. LM stores information about the target Java application's object, such as JVMs, threads, classes, interfaces, objects, methods, etc., referred to by tokens; (3) *Agent* uses JVM native interfaces to access a low-level mechanism for interactive monitoring of JVM.

In Java 1.5 Sun Microsystems has incorporated a new native programming interface for use by tools, JVMTI (JVM Tool Interface) [11], to replace less efficient interfaces JVMPI and JVMDI, which were used formerly in J-OCM.

JVMTI is intended to provide a VM interface for the full range of tools that need access to VM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools. It is a two-way interface. A client of JVMTI, hereafter called an *agent*, can be notified of interesting events. JVMTI can query and control the application through many functions, either in response to events or independent of them. Agents run in the same process with and communicate directly with the virtual machine executing the application being examined. A native in-process interface allows maximal control over the application with minimal intrusion on the part of a tool. Agents can be controlled by a separate process which implements the bulk of a tool's function without interfering with the target application's normal execution. JVMTI provides support for bytecode instrumentation which can be applied in three ways:

- Static Instrumentation: The class file is instrumented before it is loaded into the VM - e.g., by creating a duplicate directory of `*.class` files which have been modified to add the instrumentation.
- Load-Time Instrumentation: When a class file is loaded by the VM, the raw bytes of the class file are sent for instrumentation to the agent. The `ClassFileLoadHook` event provides this functionality.
- Dynamic Instrumentation: A class which is already loaded (and possibly even running) is modified. This feature is provided by the `RedefineClasses` function. Classes can be modified multiple times and can be returned to their original state.

5 Concept and Implementation of WS Monitoring in J-OCM

Prior to coming to the proper discussion of performance issues of WS, we will mention the software platform used to build WS. Next, we present our approach to building a WS-oriented monitoring mechanism, followed by some details on the implementation aspects, usage, and the overhead induced by the system.

Software Platform for Building WS. For building WS to be monitored, we use Jakarta Tomcat [14] and AXIS[12, 13]. AXIS (the Apache eXtensible Interaction System) is an Open-Source product from the Apache Software Foundation and its tools for writing client Java programs that use WS (including Microsoft `.NET` WS) and tools for deploying Java programs as WS. AXIS provides transparent access to WS for Java programmers, which allows to focus on the business logic of their applications rather than to worry about low-level network protocols (like SOAP), to use WS. AXIS also allows for automated deployment of Java programs as WS by generating a WSDL description from Java code directly.

To use a WS, a client program (typically, another WS-based application in Java, C++, C#, etc.) sends a request to the WS and receives a reply much

like a web browser requests a web page and receives HTML in reply. However, the requests and replies are encoded by SOAP. With AXIS, the developer can build client programs in Java that use WS as if they used methods in any Java class. AXIS automatically generates an additional "glue" code that hides the details of SOAP from the client program. In AXIS, existing Java programs can be deployed quickly as WS by installing AXIS e.g. on a J2EE application server engine.

Tomcat, an official reference implementation for the Java Servlet 2.2 and JavaServer Pages 1.1 technologies , is a servlet container with in a JSP environment, which is a runtime shell that manages and invokes servlets on behalf of users. Tomcat runs with any web server that supports servlets and JSPs. The Tomcat servlet engine often appears in combination with an Apache webserver. Tomcat can also function as an independent web server: it operates in development environments with no requirements for speed and transaction handling.

WS Performance Issues. In WS-based applications, many issues need monitoring, especially, those directly connected with the SOAP protocol and with the life-cycle of WS activities. Usually, WS performs the following basic operations:

1. receives a request via SOAP,
2. parses XML contained within the SOAP request,
3. executes the functionality specified by the XML,
4. formats the results in XML,
5. transmits the reply via SOAP.

The above operations which may cause performance problems are intended to be monitored by a WS extension to J-OCM.

As a basis of our system, we use J-OCM , but prior to using this system we needed to reimplement it from Java 1.4 to Java 1.5, where the issue of monitoring of Java applications is realized via the JVMTI interface. This enables to capture much more information from VM compared to the earlier interfaces of JVM (JVMPI, JVMDI).

Architecture of the J-OCM/WS System. The monitoring system (Fig. 1) comprises three layers of components (darked elements in Fig. 2 are parts of the system): (1)*application monitor* (AM) - agent like in J-OCM, (2)node's *local monitor* (LM), and (3)*service monitor* (SM). AM is embedded into the AXIS, which manages SOAP messages. It is used to perform monitoring activities in the context of the application. A node's LM is created on each node, where WS to be monitored resides. LM receives requests from SM and distributes them to AMs. LM assembles replies from AMs to send an integrated reply to SM. SM is a permanent component and exposes monitoring services to tools.

Metrics. The J-OCM/WS system is intended to provide metrics on particular operations performed within a WS life-cycle, e.g., the time used to parse a SOAP message, time of computation, the whole time of the activity of a particular WS, etc. An important element to be monitored is the SOAP message and its content. It is done in an event-driven fashion. At the start and the end of each

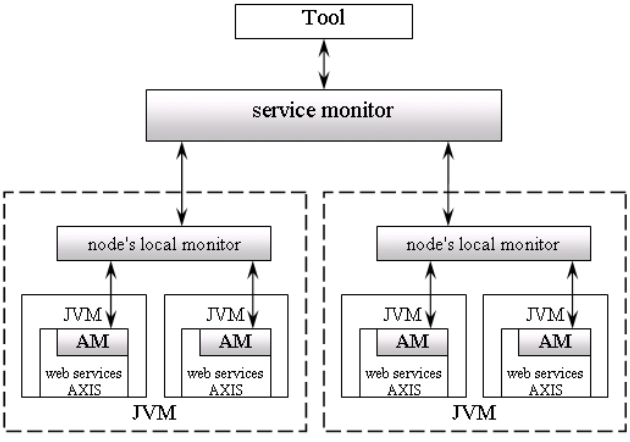

Fig. 1. Architecture of J-OCM/WS

above mentioned stage of WS operation, there are placed sensors which generate events and convey them to the AM. The sensors are dynamically placed into the class images of AXIS classes. The time stamps obtained from these events are used to produce *dynamic metrics*, by which we understand metrics related to the life-cycle of the WS-based application. The second group of metrics comprise *static metrics*, i.e. metrics related to the WS provider. There will be monitored how many WS are used at any moment, how many calls to the same WS are performed. The system provides information about the usage of WS (e.g. name, end-point URL) and about the used method (operation) of WS, e.g. the name of this operation.

J-OCM/WS provides information about:

- start/end of the whole WS execution or of its operation,
- start/end of a coming in/going out request/response message,
- start/end of parsing an SOAP message,
- calls to non-existing operation by the user or application,
- errors during an operation execution,
- number and names of WS,
- number, names, and signatures of WS operations,
- number of currently running WS,
- number of calls to each WS.

Use of the Monitoring System. In order to start the monitoring of a WS-based application, one must perform the following commands:

- to start AXIS on Tomcat with the agent in JVM:

```
java -jar agentlib:jvmlm bootstrap.jar
```

where:
- `jvmlm` - application monitor
- `bootstrap.jar` - `jar` with AXIS classes

A query from a tool to J-OCM/WS can look like the following (in compliance to the OMIS interface specification):

– a request for getting all WS tokens[3] on the JVM that runs the WS-container (AXIS):

```
:jvm_ws_get_tokens([jvm_j_1_n_1])
```

– a request for information on events (with the time, name, and token of WS and the name and token of an operation of WS), when any operation of any WS on JVM 3 on node 2 begins:

```
jvm_ws_event_operationStart([jvm_j_3_n_2]):print(["Operation
Starts at",$time, $wsStr, $ws, $opStr, $op])
```

Monitoring Overhead. The monitoring overhead measured in our tests on Intel 2.4 GHz platform can be broken down into a start-up overhead and that on a per WS basis. The start-up overhead occurs only when the software platform Tomcat and Axis), resulting from the dynamic instrumentation of all classes in JVM, which is performed by JOCM (Tomcat includes ca. 1400 classes). The start-up time increases from 6.343 s to 12. 17 s (increase by ca. 90%). The influence of monitoring on the duration of WS operations depends on the duration of the operations. In case of a 'Hello World' WS which lasted 0.142 s the overhead was 30%, while in case of a WS which lasted 1.741 s the overhead was more negligible, only 5%. Thus, the longer is WS, the less is the relative overhead.

6 Concluding Remarks

The monitoring system J-OCM/WS we presented in this paper is an implementation of our concept of the monitoring of Web Services [16], based on the inherent extendibility of the OMIS specification and its Java-bound implementation, the J-OCM monitoring system, towards new paradigms and new tools.

The WS-related monitoring system under discussion is open source and offers a functionality required in case of the monitoring of distributed Java applications and also Web Services implemented in Java.

For monitoring Web Services we used the dynamic instrumentation of AXIS classes. At the start and the end of each stage of processing SOAP message there are placed sensors (in AXIS) which generate events and convey them to the agent. We used the JVM Tool Interface and JNI to obtain information about the state of Java Virtual Machine (e.g. information about a class of WS). As a result we keep to a minimum a overhead to the monitored application performance.

[3] *Token* is a string reference to a monitored object, i.e. in this case it is a reference to JVM 1 on node 1.

Due to the fact that our system extends J-OCM, commands of this extension comply to the same syntax as the commands of J-OCM, our system should be easy to use for the users of tools which are compliant with OMIS. Within our recent research we have adapted the J-OCM/WS monitoring system to cooperate with the SCIRUN/TAU framework [17, 18]

Acknowledgements. This research was partially supported by the EU CoreGRID IST-2002-004265 project and the corresponding SPUB-M grant.

References

1. Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series. 1997.
`http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`
2. Bubak, M., Funika, W., Wismüller, R., Mętel, P., Orłowski. Monitoring of Distributed Java Applications. In: Future Generation Computer Systems, 2003, no. 19, pp. 651-663. Elsevier Publishers, 2003.
3. W. Funika, M. Bubak, M.Smętek, and R. Wismüller. An OMIS-based Approach to Monitoring Distributed Java Applications. In: Yuen Chung Kwong (ed.) Annual Review of Scalable Computing, volume 6, chapter 1. pp. 1-29, World Scientific Publishing Co. and Singapore University Press, 2004.
4. http://www.webservices.org/index.php/ws/content/view/full/1390/
5. http://www.xml.com/pub/a/ws/2001/04/04/webservices/index.html
6. http://www.w3.org/TR/2002/WD-ws-arch-20021114/
7. http://www.w3.org/XML/
8. *http* : *//www.w*3*schools.com/soap/soap*$_i$*ntro.asp*
9. http://www.w3.org/TR/wsdl
10. http://www.uddi.org/
11. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html
12. http://ws.apache.org/axis/java/user-guide.html
13. http://ws.apache.org/axis/java/architecture-guide.html
14. http://jakarta.apache.org/tomcat/
15. A. D. Malony, S. Shende, and R. Bell, "Online Performance Observation of Large-Scale Parallel Applications", Proc. Parco 2003 Symposium, Elsevier B.V., Sept. 2003.
16. P. Handzlik, L. Trebacz, W. Funika, M.Smętek, Performance Monitoring of Java Web Service-based Applications. Proc. Cracow Grid Workshop'2004, December 2004, Cracow, CYFRONET, Krakow, 2005
17. Funika, W., Koch, M., Dziok, D., Malony, A. D., Shende, S., Smetek, M., Wismuller, R. An Approach to the Performance Visualization of Distributed Applications, in: Bubak, M., Turaa, M., Wiatr, K. (Eds.), Proceedings of Cracow Grid Workshop - CGW'04, December 13-15 2004, ACC-Cyfronet UST, 2005, Krakw, pp. 193-199.
18. Funika, W., Koch, M., Dziok, D., Smetek, M., Wismuller, R. Performance Visualization of Web Services Using J-OCM and SCIRun/TAU, in: Laurence Tianruo Yang, Omer F. Rana, Beniamino Di Martino, Jack Dongarra (Eds.), Proc. HPCC 2005, pp. 666-671, Lecture Notes in Computer Science, no. 3726, Springer, 2005.