Java-based On-line Monitoring of Heterogeneous Resources and Systems

Paolo Bellavista, Antonio Corradi
Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna
Viale Risorgimento, 2 - Bologna - Italy
Ph.: +39-051-2093001 - Fax: +39-051-2093073
E-mail: {pbellavista, acorradi}@deis.unibo.it

Cesare Stefanelli
Dipartimento di Ingegneria, Università di Ferrara
Via Saragat, 1 – Ferrara - Italy
Ph.: +39-0532-293831 - Fax: +39-0532-768602
E-mail: cstefanelli@ing.unife.it

Abstract

The diffusion of Web-based multimedia services and the emerging competition among service providers require to enrich the Internet infrastructure with mechanisms to manage and control service quality and availability. These goals require monitoring mechanisms that ascertain the state of resources and applications in the global distributed system, and that should be a core functionality of any infrastructure for Web service provision. The paper describes the design and the implementation of a Java-based Application Programming Interface (API) to monitor uniformly heterogeneous resources and systems over the Internet. The monitoring tool operates at different levels of abstraction. On the one hand, it can instrument the Java Virtual Machine (JVM) to handle several types of events produced by Java applications. On the other hand, it can inspect the state of machine specific information (e.g., CPU and memory utilization) typically hidden by the JVM, and available via platform-dependent modules (currently developed for WindowsNT, Solaris and Linux). The implemented monitoring tool can be integrated in any Java-based Web service infrastructure and is currently part of the SOMA mobile agent platform.

1. Introduction

The Internet behaves as an open and global distributed system that can support service execution and provision to an increasing number of users, connected via very different and heterogeneous devices, e.g., PCs, personal digital assistants, WAP phones [1]. The diffusion of multimedia Web services and the emerging competition among service providers stress to the limit some critical service properties that are obviously considered more and more important by service providers, network operators and final customers. One of the most important property is to grant and guarantee negotiated levels of ***Quality of Service*** (QoS), whichever is the current attachment point of users and the location of service providers, and independently of the dynamic conditions in the involved networks and systems [2, 3]. In this scenario, the providers that offer services with some controlled and differentiated QoS levels are interested in ***accounting*** consumed services to users in a non-repudiable way, in order to enforce a correct billing policy. The complete support of QoS-enabled service provision imposes to ensure also service dependability, even in case of failure of the communication infrastructure and/or of some of the involved distributed service components. In the public (untrusted) Internet environment, another important service property is security, that permits to identify and face all forms of misuse and attack, such as the case of ***denial-of-service*** when

service availability is threatened by overloading the distributed resources that offer service execution.

The above properties require monitoring mechanisms and tools to obtain the visibility of the state of all heterogeneous resources that compose the network infrastructure. The significant monitoring information are at very different abstraction levels, from the state of execution resources at each node (e.g., CPU, memory and bandwidth utilization), called ***kernel state*** in the following, to the state of service components at the application level (e.g., the state of a service-specific daemon process), referred as ***application state*** [4]. In addition, the monitoring component should work during service execution, without service suspension and with minimal performance intrusion: only in this case, this information can be of any help in dynamic service adaptation, in the detection of denial-of-service attacks, in performance enhancements, and in the accounting of subscribed users.

Several research activities have investigated the implementation of ***monitoring*** tools for distributed system [4-6]. In addition, some researchers have focused on ***on-line*** monitoring tools that are forced to collect a restricted set of kernel and application state indicators. Their goal is to provide tools that detect the current conditions of service provision, in order to make possible run-time corrective actions. These tools are forced to carefully select the synthetic information to be collected, thus minimizing their intrusion towards the observed system [7-9].

Monitoring tools generally exploit native features either provided by dedicated hardware/software probes or available in the hosting operating systems. Although most operating systems generally agree on the types of data to monitor (to infer general information about processes, threads, memory, network and file system), the collecting and accessing mechanisms are very heterogeneous and platform-dependent. This has produced different and sometimes clashing mechanisms (e.g., the `/proc` directory in Solaris and Linux, the registry keys in WindowsNT) which are difficult to integrate in a unique monitoring framework. The platform-dependent monitoring mechanisms have achieved interesting results [4, 5, 7, 8], but can provide only the initial deployment step in the Internet environment, that requires also to have visibility and control on the application state of any middleware component while it supports service execution.

We claim that a general monitoring component plays a basic role in the design and implementation of any distributed infrastructure for Web-compatible service provision [10]. In this scenario, the Java technology is a convenient choice to develop new applications and services for the Internet. Java can be considered an obvious programming environment in the implementation of distributed monitoring tools [11, 12]. However, the Java Virtual Machine (JVM) hides platform-dependent characteristics and imposes a level of abstraction that seems an obstacle in providing Java-based monitoring solutions. Also these monitoring considerations have recently motivated some significant extension proposals of the JVM at different abstraction levels, both platform-independent and platform-dependent: SUN has introduced the JVM Profiler Interface (JVMPI) [13], to export some JVM internal event occurrence for debugging and monitoring purposes, and the Java Native Interface (JNI) [14], to integrate Java programs with platform-dependent executable code.

The paper presents the design and implementation of a Java-based Application Programming Interface (API) for the on-line monitoring of kernel and application state. The monitoring API collect information at two different levels of abstraction. At the Java level, the API makes possible to dynamically instrument the JVM, to deal with several events produced by the execution of Java-based services, e.g., object allocation and method invocation. At the low platform-dependent level, the API permits also to overcome the boundaries of the JVM and to obtain the visibility of the needed kernel indicators of the monitored host, such as percentage of CPU and memory utilized by all active processes, whether Java-based or not.

The implementation of the monitoring API exploits the novel features of the Java 1.2 platform: the JVM Profiler Interface (JVMPI) and the Java Native Interface (JNI). The former permits to inspect the JVM internals in order to collect several types of application-level predefined events produced by the execution of Java applications, and the API exploits it to obtain Java-level

monitoring information on the application state. The latter technology allows the invocation of native code within Java programs, and has been used to integrate the Java-based monitoring API with platform-dependent monitoring mechanisms that we have implemented as dynamic link libraries for WindowsNT and shared object libraries for Solaris and Linux.

Finally, the paper reports some experimental measurements of the overhead introduced on service execution by the implemented monitoring API. All results refer to the most spread targets, i.e., Solaris, Linux and WindowsNT platforms. They show that the intrusion of the monitoring tool can be very limited by tuning the sampling frequency of kernel and application indicators.

2. The Java Technology for Monitoring

The Java technology plays a fundamental role in the design, implementation and deployment of Web services over the global and heterogeneous Internet infrastructure. Apart from Java portability, dynamic class loading and easy integration with the Web, the main motivation of Java success is its homogeneous run-time support. The JVM hides the peculiarities of the operating system and presents a uniform vision of all available computing resources, middleware facilities and application components.

However, several application domains require a deep and low level visibility of both the JVM internal and the underlying platform. This is particularly critical when Java is adopted for the implementation of monitoring tools. These considerations have suggested SUN researchers to extend the JVM via the JVM Profiler Interface, in order to propagate the visibility of some internal details of the JVM state to the application level. In addition, the necessity of observing both kernel state and application state outside the JVM requires to integrate native monitoring components in the JVM by exploiting the functions of the Java Native Interface.

2.1. The Java Virtual Machine Profiler Interface

The JVMPI is an interface provided as an experimental feature in the Java 2 platform and is mainly designed to help application developers to monitor the behavior of Java-based applications during debugging and deployment. The JVMPI is a two-way API between the JVM and a dedicated profiler agent, usually implemented as a platform-dependent native library for the sake of performance. In one direction, the JVM notifies several internal events to the profiler agent that has registered its interest. In the other direction, the profiler agent can enable/disable the notification of specific types of events and can perform some limited management actions on the JVM. Figure 1 depicts the described two ways of utilization of the JVMPI.

All notifiable events are fired by changes in the state of Java threads (started, ended, blocked on a locked monitor), beginning/ending of invoked methods, class loading operations, object allocation/deallocation, and the beginning/ending of JVM garbage collection. Any event notification carries full information about the entities that have generated that event. For instance, the allocation of a new object generates the `JVMPI_EVENT_OBJECT_ALLOC` event, and the profiler agent receives the identifiers of the new object and of its class, together with the size of allocated heap memory.

The JVMPI can be exploited also in the opposite direction from the profiler agent towards the JVM. Apart from notification enabling/disabling, the agent can intervene on the JVM via JVMPI functions to suspend/resume Java threads, and to modify the behavior of the garbage collector (enable, disable, force its immediate execution).

Figure 1 shows also a profiler process, possibly external to the JVM. The profiler can provide application developers with an immediately readable form of the monitoring data collected by the profiler agent. The profiler process interrogates the agent to obtain the information on notified events, and can elaborate these data, either on-line or off-line, to obtain traditional average system

indicators, such as the total size of heap memory allocated to a specified service thread or group of threads.

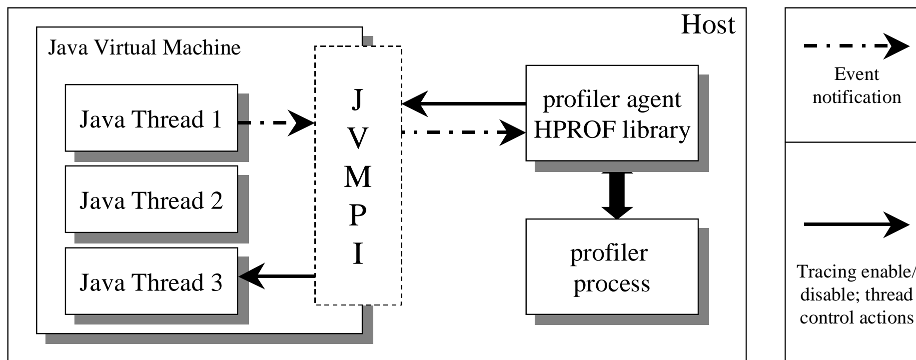

Figure 1. JVMPI-based architecture for JVM monitoring.

The SUN distribution of the Java 2 platform provides a simple implementation of the profiler agent as a native library for both Solaris and WindowsNT operating systems. The agent is called HPROF [13], and has been mainly designed for final debugging and performance tuning of applications. HPROF is a general-purpose event collector with simple configuration features. It works better as an off-line post-mortem monitoring of Java threads than as an on-line one because it tends to collect a large volume of monitoring data that cannot be given without a heavy filtering and processing to obtain significant and synthetic service indicators. This is the main reason of several recent implementations of profiler processes to organize HPROF data in user-friendly graphic interfaces [15, 16].

Although it permits to instrument dynamically the JVM, the JVMPI has several limitations in furnishing monitoring functions. In fact, the JVMPI reports only some of the events generated in the JVM, and imposes a coarse-grained specification of the events to be notified to the agent, with no possibility of fine selections and dynamic refinements. For instance, a profiler agent can only choose to enable/disable the notification of all events related to all Java classes (or objects/methods/monitors), but it does not allow any specific interest in the events generated by a particular class. In addition, the JVMPI cannot give any direct information about the current conditions of the monitored host outside the JVM (e.g., the number of non-Java active processes, the set of files opened by a non-Java process, …). Therefore, it cannot be used as it is to obtain traditional indicators of system load, such as CPU idle time and total size of allocated memory. Some load indicators can only be obtained by indirect (and usually less accurate) measures, e.g., by collecting the statistics of the time interval needed to execute a predefined CPU-bound test method (`JVMPI_EVENT_METHOD_ ENTRY` and `JVMPI_EVENT_METHOD_EXIT` events).

2.2. Exploiting Platform-dependent Functions via the Java Native Interface

The development and deployment of QoS-enabled Java services require the possibility to observe dynamically both the kernel state, e.g., the current percentage of total available memory, and the application state of service components outside the JVM, e.g., the total number of non-Java active processes. To obtain these monitoring indicators, it is necessary to exploit platform-dependent monitoring mechanisms and tools. The JNI interface specifies how to dynamically integrate Java programs with any platform-dependent code, and, in particular, can permit the extension of Java-based monitoring tools with platform-specific monitoring solutions.

The JNI is a two-way API interface that permits Java threads to invoke *native methods*, i.e., functions typically written in C/C++, compiled for a specific platform and provided in the form of Dynamic Link Libraries (DLL) under WindowsNT and of Shared Object (SO) libraries under Solaris and Linux (see figure 2). In one direction, any Java program can invoke native methods, simply by

declaring methods with the keyword `native` and with no body. During the execution, the JVM uses the JNI to call the requested function in the native library, previously loaded via the `System.loadLibrary()` method. The JNI specification defines the modes of method invocation. For instance, it rules how to perform parameter marshalling/unmarshalling between the Java invoking thread and the called native method.

In the other direction (from the native library towards the JVM), the JNI allows native methods to interwork with their invoking Java objects and methods. To this purpose, the JNI provides native code with callbacks to the invoking Java environment and, in particular, to the invoking Java object. This way permits native methods to interwork with all Java entities accessible to the Java class that has invoked the native function. For instance, a native method can access and modify the value of Java objects, can call Java methods, and can raise Java exceptions.

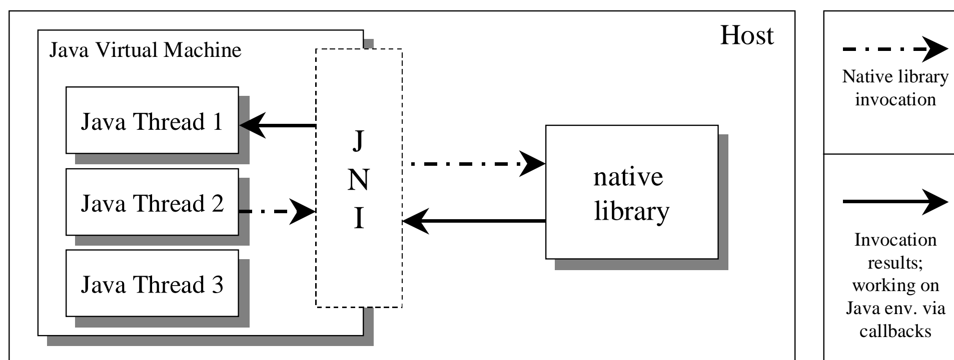

Figure 2. The two-way JNI for native library invocation.

The motivations of the JNI stems from the necessity to prevent incompatibility between JVM implementations. Dynamically loaded native libraries can contain platform-dependent executable code and cannot be ported to heterogeneous operating systems. The JNI can ensure native code portability over different implementations of the JVM for different platforms, even provided by different vendors.

If applied to the monitoring domain, the JNI permits to integrate the JVM with native monitoring libraries, thus working at a lower level than the JVMPI, with visibility of kernel and application indicators. For instance, if a Java-based monitoring applications wants to collect information about any active process on one host (e.g., to register which process occupies the most the CPU in a specified time interval), it can invoke the execution of a C-based native library to extract the needed information from either the WindowsNT registry keys or the Solaris `/proc` directory.

This widened visibility of low-level indicators has the obvious drawback that these monitoring tools are not portable over different platforms. On the opposite, a standard interface should be provided and requires a certain degree of homogeneity. A possible solution is to furnish different implementations of native monitoring functions with analogous monitoring information over different operating systems, to permit the dynamic integration of Java-based tools with the suitable native implementation. In that way, the multi-platform implementation guarantees transparency and portability. We have adopted this solution in the development of the on-line monitoring API described in the following section.

3. A Java API for On-line Heterogeneous Monitoring

We have worked to design and implement a Java-based infrastructure for on-line distributed monitoring, which permits maximum visibility in the observation of dynamic properties of the managed systems, such as resource utilization and application state, over possibly heterogeneous platforms. To achieve this goal, we have implemented a Java API (`ResourceManager` class) for the uniform local monitoring of a single host.

On the one hand, the `ResourceManager` class permits the monitoring of application state indicators at the application level, in order to export the visibility of JVM events. Our monitoring API exploits the JVMPI to gather information about any Java-based thread running on the monitored host; these JVMPI-based functions are immediately portable on any host that runs the JVM version 2. On the other hand, the `ResourceManager` class provides also monitoring indicators for the kernel state of the observed host. We have used JNI to integrate our Java-based `ResourceManager` with platform-dependent monitoring functions implemented as native libraries for several different platforms (`WindowsRM` DLL on Microsoft WindowsNT 4.0, `SolarisRM` SO on SUN Solaris 2.7, and `LinuxRM` SO on SuSE Linux 6.2). The native libraries can be loaded at run-time in a completely transparent way, dependently on the platform of the current monitored host. Figure 3 shows the `ResourceManager` architecture.

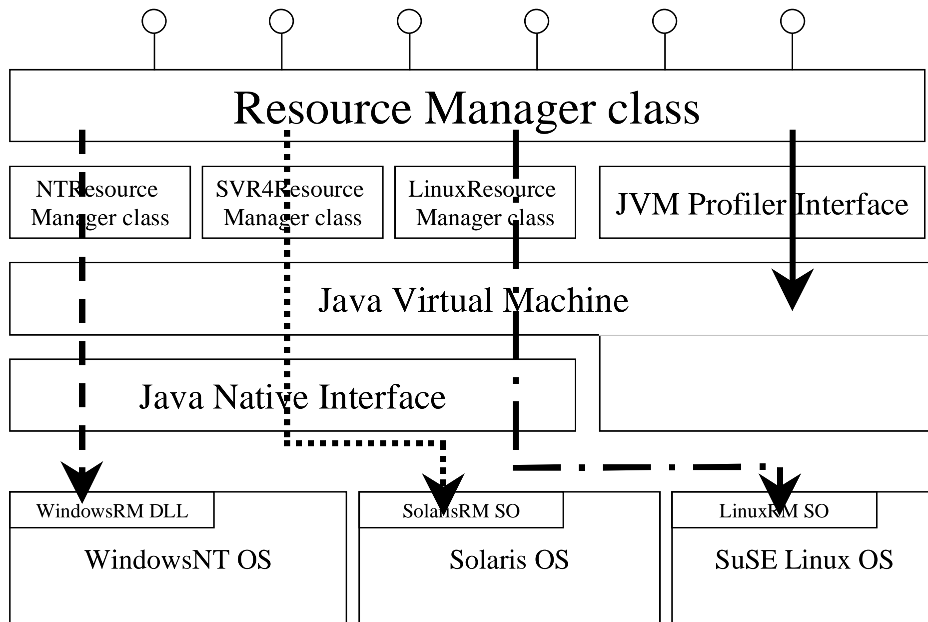

Figure 3. The architecture of our Java-based monitoring API.

The `ResourceManager` class makes available a limited set of synthetic monitoring parameters sufficient to summarize the current state of the monitored host. We have defined synthetic parameters because our main objective is on-line monitoring: service administrators (or even automatic software-based service managers) should use our API to guide their management operations on adaptation during service provision. Therefore, the overhead imposed by the monitoring tool is critical and monitoring results should be “ready to use”, i.e., without the need of any additional, possibly complex and time-consuming, off-line processing.

The monitoring indicators are organized in three main classes (`ProcessInfo`, `NetworkInfo`, and `FileSystemInfo`), and `ResourceManager` provides them in a uniform way, independently of the operating system of the observed host. The `getOS()` API method is the only way to understand which platform the `ResourceManager` class is currently running on. Apart from that result, the monitoring functions are completely transparent, independently of the hosting operating system. To minimize the intrusion of the monitoring API depending on service-specific time constraints of observation, all API methods require a `msec` invocation parameter that indicates the milliseconds to refresh the desired monitoring indicators. The `msec` interval affects the time of the periodic invocation of native libraries and of the refreshment of JVMPI collected events, with different consequences on the introduced overhead, as described in Section 4.

All methods return either an object or an array of objects of the three classes described in the following. `ProcessInfo` is a class that maintains all the data related to a process identified by `pid`

in the `getProcessInfo()` invocation. Monitored data include the utilization percentage and the effective time of the CPU consumed by the specified process, its allocated memory (either physical or virtual), and various data on its composing threads. In the case of JVM processes, a very large collection of monitoring data are available for any Java thread, such as the reference to the Java thread object, the number of loaded classes, contended monitors, allocated objects, invoked methods, and network operations. Otherwise, the monitoring API provides, for any non-Java thread, the thread identifier and the percentage/effective time of CPU utilization.

The `NetworkInfo` class reports aggregated monitoring data about the utilization of the communication infrastructure made by the target host as a whole. Monitored data include the total number of sent/received UDP/IP packets, of TCP connections and entering/exiting segments, and of received UDP/IP packets with errors. These synthetic parameters permit to evaluate traffic and congestion in the network locality of the monitored host.

Finally, the `FileSystemInfo` class maintains information about the file system on the target host (space available and its percentage on total disk size) and, in particular, about the current state of open files: for any active process and for any file opened by it, the class can return the opening time and its opening modes (read/write/both/locked), and the number of current read/write operations. We are extending this function to maintain also data on how many files a process has worked on in a specified time window, and on how many times it has opened and closed any of these files; the aim is to prevent possible denial-of service attacks based on the repetition of open and close operations on the file system.

The monitoring API has been designed and implemented to simplify dynamic extension and tailoring to service-specific needs. In particular, we are working to extend the API with a method `newMethod()`to permit to authorized administrators to introduce new functions to the monitoring interface, by dynamically injecting Java bytecode. For instance, a multimedia service component could require to observe the network latency obtained in transmitting a specific video stream between two adjacent hosts. To provide this monitoring feature, currently not provided in the `NetworkInfo` class, the multimedia service provider can securely inject the needed code at run-time, thus dynamically extending the monitoring API.

4. Experimental Measurements of the Monitoring Overhead

Any on-line monitoring tool should minimize its intrusion on observed hosts, not to introduce sensible overload during service provision. We have followed this guideline in the design and implementation of our Java-based monitoring, and we have accurately measured the overhead it introduces, in order to validate the effectiveness of the proposed solution. We have tested our `ResourceManager` class over all the different supported platforms, to evaluate also the different costs due to the adoption of either the JVMPI technology or the native methods.

We have installed an *ad-hoc* Java benchmark application that stresses both CPU and memory usage: it simply instantiates a number of different threads and objects. The simple benchmark has first been executed on target hosts with neither applications nor the `ResourceManager` running, and we have taken several hundred measurements. The average completion time of the benchmark execution, called T_{noMon}, has been compared with the average time (T_{Mon}) measured with our monitoring tool in execution. The graphs in Figure 4 and 5 report the overhead percentage (*Overhead%*) introduced by the monitoring tool, defined as:

$$Overhead\% = \frac{T_{mon} - T_{noMon}}{T_{noMon}} * 100 = \left(\frac{T_{mon}}{T_{noMon}} - 1 \right) * 100$$

All reported costs have been measured with different sampling frequencies on Microsoft WindowsNT4 and SuSE Linux6.0 running on Intel PentiumIII PCs, and on SUN Solaris7 running on UltraSPARC10 workstations.

Let us note that the test case of only the benchmark application running, is the worst possible case for the *Overhead%* parameter. In fact, when the monitored host is either loaded on the average or overloaded, T_{noMon} tends to increase faster than T_{mon} in absolute terms, and consequently their ratio *Overhead%* tends to lower. We have taken *Overhead%* measurements with several general-purpose benchmark tests running, and the gathered costs confirm the above claim, with an average decrease of about 1.0-1.5% of the *Overhead%* parameter in conditions of medium load.

Figure 4 depicts the *Overhead%* introduced by the JNI operations to monitor respectively `ProcessInfo`, `NetworkInfo` and `FileSystemInfo`, for the three supported platforms and depending on the sample time period. *Overhead%* has shown to depend linearly on the invocation frequency of the native monitoring libraries via the JNI. The set of graphs shows that the monitoring tool causes similar and acceptable overhead (interference is lower than 3%) in all cases when the sampling frequency goes under 0.3 Hz.

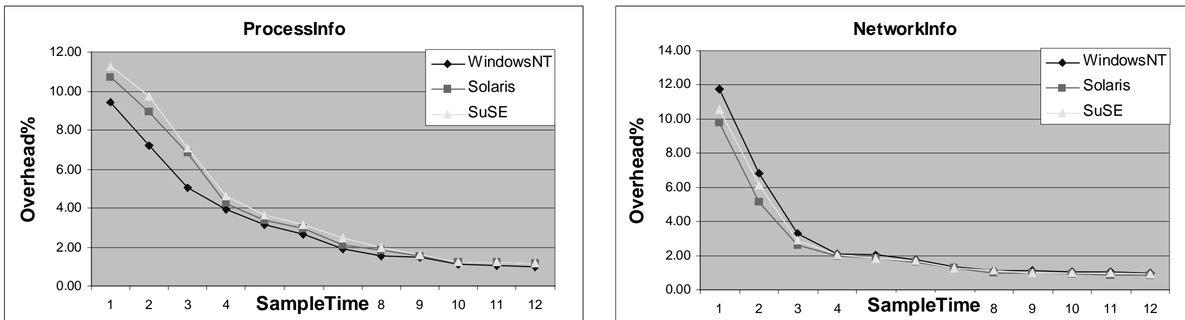

Figure 4. *Overhead%* of the native monitoring functions invoked via JNI.

In addition, we have measured the overhead of the JVMPI instrumented to notify continuously (not only at polling times) certain kinds of events. In this case, the sampling frequency represents only the frequency to refresh the observed data and to transmit them to the human/automatic manager. Figure 5 depicts the *Overhead%* for the different supported platforms, by separating the contributions stemming from the different kinds of events that the JVMPI is enabled to notify (either monitor or method or object tracing). The JVMPI notification mechanism is non-intrusive in itself (lower than 0.7%) under different load conditions and their interference has confirmed to be independent of sampling frequencies. Only object tracing can become heavy as a weak point in our current prototype implementation. In fact, our profiler agent maintains information about objects instantiated by Java threads in a table that is implemented, for the sake of simplicity, in a C++ dynamic list of pointers. This data structure can make expensive the retrieval depending on table size. We are completing a new version of the profiler agent that exploits a hashed table, and this implementation achieves a reduction of object tracing *Overhead%* of a 4 factor.

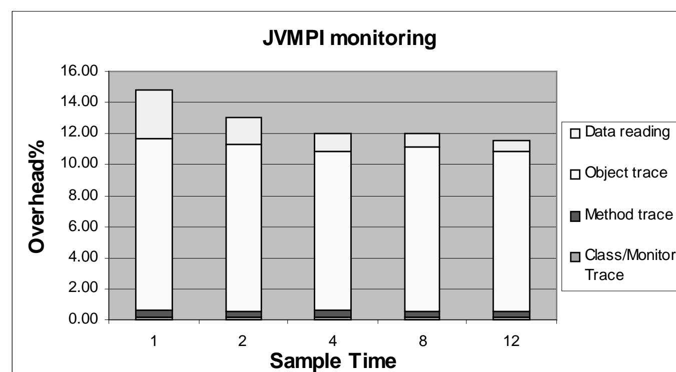

Figure 5. *Overhead%* of the JVMPISlave agent to monitor Java threads.

The sampling frequency affects only the overhead introduced by the tool that reads and processes the collected monitoring information to provide final users with their graphic visualization: in any case, the overhead can be maintained under 2.0% when the sampling frequency goes under 0.3 Hz and object tracing is disabled.

5. Concluding Remarks

The necessity of monitoring tools to achieve full visibility of kernel and application state information calls for an extension of the JVM that is capable of overcoming the boundaries imposed by its abstraction level. We have exploited two different directions, the JVMPI and the JNI technologies, to implement a Java-based API to monitor heterogeneous resources and systems in the Internet scenario.

The API provides service administrators with indicators that are particularly suitable for on-line monitoring: it tends to furnish synthetic and not overhead-prone information. The implementation has exhibited an acceptable overhead on executing services. At the moment, we work on the integration of the presented monitoring component within our mobile agent framework. This integration is twofold: on the one hand, to provide mobile agents with a local monitor component needed for network and systems management, and, on the other hand, to compose a basic framework for the accountability of resource consumption in Internet service provision.

References

[1] T. Lewis, "Information Appliances: Gadget Netopia", *IEEE Computer*, Vol. 31, No. 1, Jan. 1998.

[2] D. Chalmers, M. Sloman, "A Survey of Quality of Service in Mobile Computing Environments", *IEEE Communications Surveys & Tutorials*, Vol. 2, No. 2, 2nd Quarter 1999.

[3] D. Hutchison, et al., "QoS Management in Distributed Systems", in *Network and Distributed Systems Management*, M. Sloman (ed.), Addison-Wesley, 1994.

[4] R. Buyya, "PARMON: a Portable and Scalable Monitoring System for Clusters", *Software – Practice and Experience*, Vol. 30, pp. 723-39, 2000.

[5] S.H. Russ, et al., "Hector: an Agent-based Architecture for Dynamic Resource Management", *IEEE Concurrency*, Vol. 7, No. 2, pp. 47-55, Apr.-June 1999.

[6] E. Al-Shaer, H. Abdel-Wahab, K. Maly, "HiFi: a New Monitoring Architecture for Distributed Systems Management", 19th *Int. Conf. on Distributed Computing Systems*, IEEE Computer Society, 1999.

[7] B.A. Schroeder, "On-Line Monitoring: A Tutorial", *IEEE Computer*, Vol. 28, No. 6, June 1995.

[8] G. Weiming, G. Eisenhauer, K. Schwan, J. Vetter, "Falcon: On-line Monitoring for Steering Parallel Programs", *Concurrency - Practice and Experience*, Vol. 10, No. 9, pp. 699-736, Aug. 1998.

[9] A. Corradi, C. Stefanelli, "HOLMES: a Tool for Monitoring Heterogeneous Architectures", 4th *Int. Conf. on High Performance Computing*, IEEE Computer Society, Los Alamitos, pp. 486-491, 1997.

[10] P. Bellavista, A. Corradi, C. Stefanelli, "An Integrated Management Environment for Network Resources and Services", *IEEE Journal on Selected Areas in Communication*, Special Issue on "Recent Advances in Network Management and Operations", Vol. 18, No. 5, May 2000.

[11] J. Lee, "Enabling Network Management Using Java Technologies", *IEEE Communications*, Vol. 38, No. 1, Jan. 2000.

[12] Z. Liang, Y. Sun, C. Wang, "ClusterProbe: An Open, Flexible and Scalable Cluster Monitoring Tool", *IEEE Int. Workshop on Cluster Computing*, pp. 261-268, 1999.

[13] Sun Microsystems - Java Virtual Machine Profiler Interface (JVMPI), http://java.sun.com/products/jdk/1.3/docs/guide/jvmpi/jvmpi.html.

[14] R. Gordon, *Essential Java Native Interface*, Prentice Hall, 1998.

[15] G. Pennington and R. Watson, jProf - a JVMPI based profiler, http://starship.python.net/crew/garyp/jProf.html.

[16] N. Meyers, "PerfAnal: A Performance Analysis Tool", http://developer.java.sun.com/developer/technicalArticles/GUI/perfanal/index.html, April 2000.