Java for On-line Distributed Monitoring of Heterogeneous Systems and Services

PAOLO BELLAVISTA[1], ANTONIO CORRADI[1] AND CESARE STEFANELLI[2]

[1]*Dipartimento di Elettronica, Informatica e Sistemistica, Università di Bologna, Viale Risorgimento, 2, Bologna, Italy*
[2]*Dipartimento di Ingegneria, Università di Ferrara, Via Saragat, 1, Ferrara, Italy*
Email: pbellavista@deis.unibo.it

The control and management of Web-based service quality require the extension of the Internet infrastructure with monitoring functions to ascertain dynamically the state of networked resources. We describe the design and implementation of the Monitoring Application Programming Interface (MAPI), a Java-based tool for the on-line monitoring of Internet heterogeneous resources, which provides monitoring indicators at different levels of abstraction. At the application level, it instruments the Java Virtual Machine (JVM) to notify several different types of events triggered during the execution of Java applications, e.g. object allocation and method calls. At the kernel level, MAPI inspects system-specific information generally hidden by the JVM, e.g. CPU usage and incoming network packets, by integrating with Simple Network Management Protocol agents and platform-dependent monitoring modules. MAPI is the core part of a portable tool for distributed monitoring, control and management in the Internet environment. The tool is implemented in terms of mobile agents that move close to the monitored resources to enforce distributed management policies autonomously, with a significant reduction in both reaction time and traffic overhead.

Received 1 November 2001; revised 22 April 2002

1. INTRODUCTION

The Internet is becoming an open and global distributed system for service provision to an increasing number of users, interconnected by very different and heterogeneous devices, e.g. PCs, personal digital assistants and even cellular phones [1]. In addition, the enlarging market of multimedia Web services and the competition among service providers stress some innovative service properties that providers, network operators and final customers tend to consider more and more important. One of these properties is the quality of service (QoS), i.e. the ability to control, manage and possibly guarantee negotiated service levels independently of the dynamic conditions of resources in the involved networks and systems [2]. The providers offering services with differentiated QoS levels are interested in accounting users for their real resource consumption and in enforcing the desired billing policies effectively. In the untrusted Internet environment, another crucial service property is security, which permits one to identify and face all forms of misuse and attack, such as denial-of-service obtained by overloading resources to produce unavailability.

In the last few years, several research efforts have introduced *ad hoc* protocols for QoS management at the network layer [3]. These solutions achieved interesting and effective results when applied to limited networks, but their approach clashes with the best-effort model of the Internet. They require any intermediate router traversed by service packet flows to implement the specific protocol, which is likely to pass through a long process of acceptance and diffusion. As a general consideration, network-layer solutions work at a level of abstraction that makes it difficult to embed some functions which are recognized as fundamental in the state-of-the-art of Internet service provisioning, such as application-specific adaptation and secure billing. Therefore, in order to integrate some forms of QoS differentiation/control with the standard best-effort Internet model, the service infrastructure should be informed of the current usage of heterogeneous resources at runtime. In other words, the service infrastructure should include an on-line monitoring tool able to detect the current condition of network, system and application components during execution. This is necessary to enable dynamic service management via runtime corrective operations.

The monitoring information needed in QoS management covers different abstraction levels, from system conditions at each node (the usage of CPU, memory, bandwidth, etc.), called the kernel state in the following, to the state of application-level service components (the state of a service-specific daemon process, etc.), sometimes referred to as the application state [4]. In addition, the on-line requirement makes a short response time critical as well as the need to reduce the overhead in the observed target, thus forcing one to collect only a restricted set of kernel and application state indicators.

This paper presents the design of a Java-based Monitoring Application Programming Interface (MAPI) for the on-line monitoring of Web services. MAPI overcomes Internet platform heterogeneity and permits one to observe the state of systems/applications during execution. MAPI collects monitoring data at the different levels of abstraction as required. At the application level, it dynamically interacts with the Java Virtual Machine (JVM) to gather detailed information about the execution of Java-based services. At the kernel level, it enables access to system indicators at the monitored target (either Java-based or external to the JVM), such as CPU and memory usage of all active processes.

To overcome the transparency imposed by the JVM, MAPI exploits some recent extensions of the Java technology: the JVM Profiler Interface (JVMPI) [5] and the Java Native Interface (JNI) [6]. In addition, MAPI integrates with external standard monitoring entities, particularly those diffused in the network management domain, i.e. Simple Network Management Protocol (SNMP) agents [7]. JVMPI makes it possible to instrument dynamically the JVM for debugging and monitoring purposes, and MAPI exploits it to collect, filter and analyze application-level events produced by Java applications, e.g. object allocation and method invocation. At the kernel level, MAPI collects system-dependent monitoring data, e.g. CPU usage and incoming network packets, by interrogating SNMP agents that export local monitoring data via their standard management information base (MIB). To also enable the monitoring of hosts without any SNMP agent in execution, MAPI exploits JNI to integrate with platform-dependent monitoring mechanisms, which we have currently implemented for the Windows NT, Solaris and Linux platforms.

We claim that on-line monitoring components play a central role in any distributed infrastructure for QoS-enabled service provision in the Internet environment, to achieve dynamic service adaptation, to enhance global performance, to charge subscribed users for accessed services and to detect possible denial-of-service attacks. This is the reason why we have included the MAPI tool as a core component of a Java-based distributed middleware, called Secure and Open Mobile Agent[3] (SOMA), for the design, implementation and deployment of Internet services. We have also implemented a MAPI-based distributed monitoring tool in terms of SOMA Mobile Agents that cooperate to enforce distributed management policies. Monitoring agents can migrate close to the networked resources to observe and manage them autonomously on behalf of system administrators, with a significant reduction in both reaction time and traffic overhead.

The paper also reports measurements of the overhead introduced by the local MAPI component and the MAPI-based distributed tool. The monitoring overhead depends mainly on the level of detail of monitored indicators and on the time interval for their update. The overhead can be tuned dynamically by service administrators in response to service-/system-specific runtime conditions, and has been shown to be acceptable for most classes of Web services, with good scalability results if compared with traditional SNMP-based centralized solutions.

[3]SOMA, MAPI and the MAPI-based distributed monitoring tool are available at http://lia.deis.unibo.it/Research/SOMA/.

2. JAVA TECHNOLOGIES FOR MONITORING

The Java technology plays a fundamental role in the design, implementation and deployment of Web services over the Internet infrastructure. Apart from Java portability, dynamic class loading and easy integration with the Web, the main motivation of Java diffusion is its virtual machine that hides the local operating system and presents a uniform vision of all available computing resources and middleware facilities.

However, the monitoring perspective requires a complete and low-level visibility of both JVM internals and underlying platforms. At the application level, the MAPI monitoring component exploits JVMPI to acquire visibility of the JVM internal events. At the kernel level, MAPI employs modules external to the JVM, to gather information about platform-dependent resources and non-Java application components. In MAPI, these external modules include both native monitoring mechanisms integrated via JNI and standard monitoring components, i.e. SNMP agents. In the following, the paper briefly describes the JVMPI and JNI technologies, to provide the needed background for the full understanding of the MAPI design and implementation.

2.1. The JVMPI

JVMPI is an experimental API of the Java 2 platform, mainly designed to help developers in monitoring Java-based applications during debugging, without imposing any modification in the application code. JVMPI is an interface between the JVM and a dedicated profiler agent, often implemented as a platform-dependent native library for the sake of performance. In one direction, the JVM notifies several VM-internal events to the profiler agent; 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.

With a finer degree of detail, several JVM conditions trigger JVMPI events: Java thread state change (start, end, when blocking on a locked monitor); beginning/ending of invoked methods; class loading operations; object allocation/deallocation; beginning/ending of the JVM garbage collection. In addition, the profiler agent can use JVMPI to modify dynamically the behavior of monitored applications. Apart from notification enabling/disabling, the agent can intervene on the JVM by invoking a very small set of JVMPI methods: management actions are limited to suspend/resume Java threads and to enable/disable/force the immediate execution of the JVM garbage collector.

The SUN distribution provides a simple implementation of the profiler agent for both Solaris and Windows NT

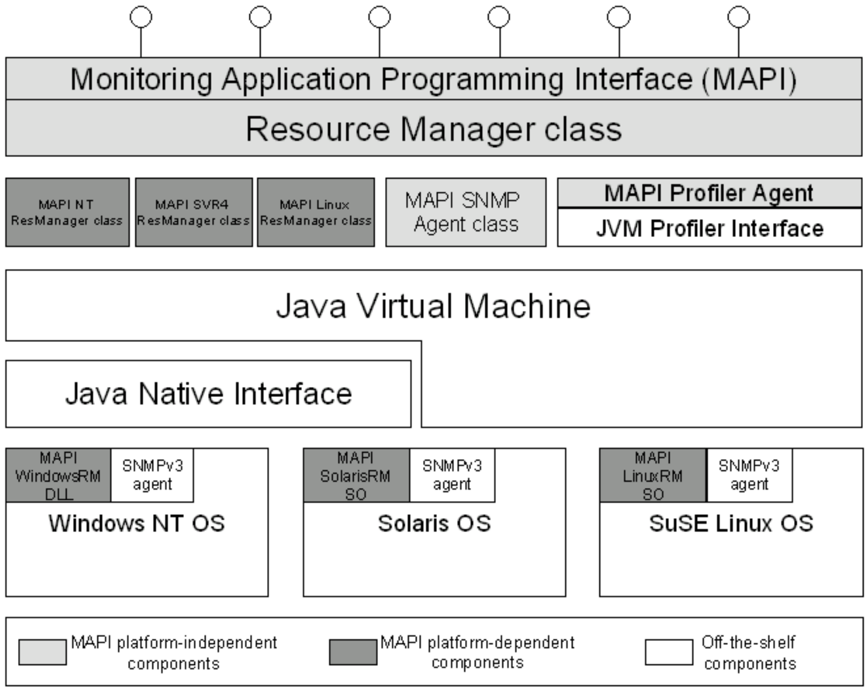


FIGURE 1. The architecture of the Java-based portable MAPI.

operating systems. This agent, called HPROF [5], collects general-purpose events and allows simple static configurations. It is not designed for on-line monitoring, but works mainly as an off-line post-mortem tool for debugging and performance analysis. In fact, it tends to collect a large volume of monitoring data that requires heavy filtering and processing to obtain significant and concise service indicators. For this reason, some researchers have implemented their *ad hoc* profiler processes to organize HPROF data in immediately readable graphic interfaces [8, 9].

2.2. The JNI

JNI permits Java threads to invoke *native methods*, i.e. platform-specific functions typically written in C/C++, usually available as Dynamic Link Libraries (DLL) in the Windows platform and shared object (SO) libraries under Solaris and Linux.

JNI is a two-way interface. In one direction, a Java program can invoke a native method, by declaring the method with the keyword `native` and with no body, and by binding to the native method library including the requested function. JNI specifies the details of method invocation: for instance, it rules the parameter marshalling/unmarshalling between the Java invoking thread and the invoked native method. In the other direction, from the native library towards the JVM, JNI allows native methods to interact with their invoking Java framework. JNI permits native code to callback the Java environment and the invoking Java object, to access and modify object values, to call methods and to raise Java exceptions.

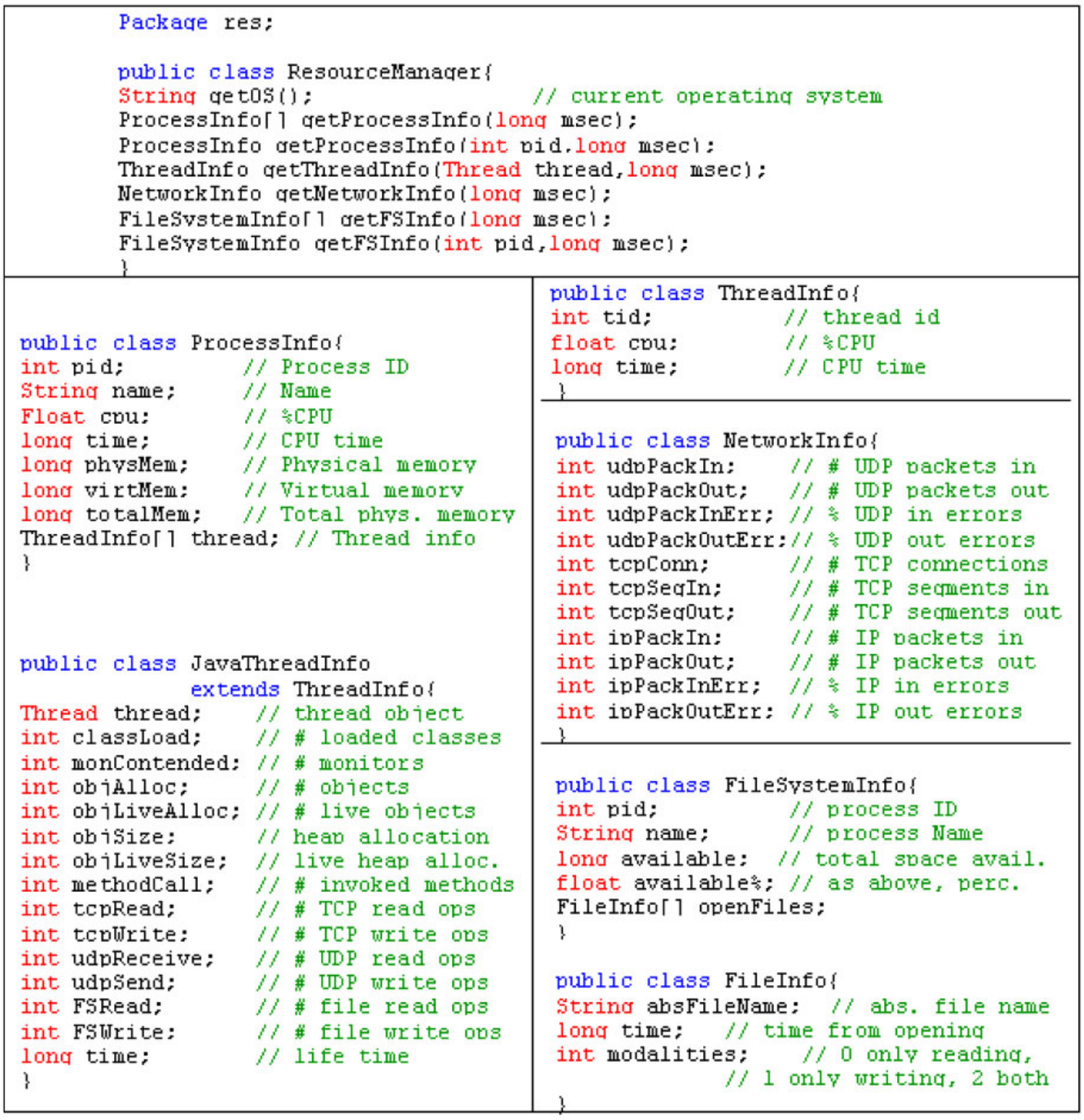
With regard to monitoring, MAPI exploits JNI to integrate with native monitoring libraries, in order to obtain the visibility of kernel and application indicators not accessible via JVMPI. For instance, MAPI collects information about process CPU usage by invoking the execution of C-based native libraries that extract the monitoring information differently depending on the monitored target, from either the Windows NT registry or the Solaris `/proc` directory.

3. THE MAPI COMPONENT FOR ON-LINE HETEROGENEOUS MONITORING

When dealing with the best-effort Internet, QoS control and adaptation require the capacity to monitor the level of quality offered by service components. Service components operate in an open and global distributed system, which is intrinsically heterogeneous, and often include legacy elements that cannot be modified/instrumented for monitoring purposes [2, 3, 10]. In addition, the adaptation of Web-service quality requires monitoring information to be available at runtime without imposing any service suspension.

This section presents the MAPI architecture and interface. MAPI permits the on-line monitoring of kernel/application resources independently of possible heterogeneity in their platform implementation. Figure 1 shows that the MAPI interface is implemented by the *ResourceManager* class, which integrates three different components: the *MAPI Profiler Agent*, the *MAPI SNMP Agent* and the *MAPI*ResManager* (*MAPI NT ResManager* for Windows, *MAPI SVR4 ResManager* for Solaris, etc.). MAPI is the core portable component of the distributed monitoring framework described in Section 5.

The *MAPI Profiler Agent* is able to gather application-level information about the Java environment of the



```
Package res;

public class ResourceManager{
String getOS();                    // current operating system
ProcessInfo[] getProcessInfo(long msec);
ProcessInfo getProcessInfo(int pid,long msec);
ThreadInfo getThreadInfo(Thread thread,long msec);
NetworkInfo getNetworkInfo(long msec);
FileSystemInfo[] getFSInfo(long msec);
FileSystemInfo getFSInfo(int pid,long msec);
}
```

```
public class ProcessInfo{
int pid;           // Process ID
String name;       // Name
Float cpu;         // %CPU
long time;         // CPU time
long physMem;      // Physical memory
long virtMem;      // Virtual memory
long totalMem;     // Total phys. memory
ThreadInfo[] thread; // Thread info
}
```

```
public class JavaThreadInfo
            extends ThreadInfo{
Thread thread;       // thread object
int classLoad;       // # loaded classes
int monContended:    // # monitors
int objAlloc;        // # objects
int objLiveAlloc;    // # live objects
int objSize;         // heap allocation
int objLiveSize;     // live heap alloc.
int methodCall;      // # invoked methods
int tcpRead;         // # TCP read ops
int tcpWrite;        // # TCP write ops
int udpReceive;      // # UDP read ops
int udpSend;         // # UDP write ops
int FSRead;          // # file read ops
int FSWrite;         // # file write ops
long time;           // life time
}
```

```
public class ThreadInfo{
int tid;           // thread id
float cpu;         // %CPU
long time;         // CPU time
}
```

```
public class NetworkInfo{
int udpPackIn;      // # UDP packets in
int udpPackOut;     // # UDP packets out
int udpPackInErr;   // % UDP in errors
int udpPackOutErr;// % UDP out errors
int tcpConn;        // # TCP connections
int tcpSegIn;       // # TCP segments in
int tcpSegOut;      // # TCP segments out
int ipPackIn;       // # IP packets in
int ipPackOut;      // # IP packets out
int ipPackInErr;    // % IP in errors
int ipPackOutErr;   // % IP out errors
}
```

```
public class FileSystemInfo{
int pid;           // process ID
String name;       // process Name
long available;   // total space avail.
float available%; // as above, perc.
FileInfo[] openFiles;
}
```

```
public class FileInfo{
String absFileName;  // abs. file name
long time;   // time from opening
int modalities;    // 0 only reading,
             // 1 only writing, 2 both
}
```

FIGURE 2. The `ResourceManager` interface.

monitored target. It not only collects JVMPI events but also filters and processes them on-line, to offer concise monitoring indicators during service execution. These JVMPI-based monitoring functions are immediately portable on any host that runs the JVM version 2.

According to the SNMP terminology, the *MAPI SNMP Agent* acts as an SNMP manager that interrogates the standard SNMP agent available on its local target to obtain kernel-level monitoring data. The *MAPI SNMP Agent* not only provides a uniform Java interface by wrapping possibly non-Java SNMP agents. It also implements several local optimizations of the SNMP protocol, as described in the following. In addition, it simplifies the configuration of the security parameters needed in SNMPv3, by integrating with the SOMA distributed security infrastructure [11].

To be portable even in the case when the monitored targets do not host the execution of suitable SNMP agents, *ResourceManager* exploits the *MAPI*ResManager* classes to integrate with platform-dependent monitoring functions via JNI. These functions are implemented as native libraries with uniform interfaces for different platforms (*MAPI WindowsRM* DLL on Microsoft Windows NT 4.0, *MAPI SolarisRM* SO on SUN Solaris 7 and *MAPI LinuxRM* SO on SuSE Linux 6.2). *ResourceManager* achieves portability by sensing dynamically the implementation platform of the current monitored target and by consequently loading at runtime the specific native library.

Figure 2 shows the MAPI set of methods that provide concise monitoring parameters to summarize the current state of the monitored target. MAPI has been designed and implemented to enable service administrators (or autonomous software-based service managers) to obtain kernel/application resource state for on-line service management and adaptation. In this scenario, the overhead is critical and monitoring results should be prompt and immediately available to managers (see Section 7). For this reason, MAPI can tune its intrusion to service-specific time constraints: all MAPI methods have a `msec` invocation parameter that indicates the time interval to update the statistics of collected JVMPI events, to interrogate SNMP agents and to invoke native monitoring libraries.

MAPI methods return either an object or an array of objects of the three classes `ProcessInfo`, `NetworkInfo` and `FileSystemInfo` described in Figure 2. The `ProcessInfo` object maintains all the data related to the current `pid` process. Monitored data include the CPU usage (percentage and total time) for any specified process, its allocated memory (physical and virtual) and miscellaneous information on its composing threads. In addition, in the case of JVM threads, MAPI maintains the reference to the Java thread object, its lifetime and the number of loaded classes, used monitors, allocated objects, invoked methods and network and file system operations. For non-Java threads, MAPI provides the thread identifier and the percentage/effective time of CPU usage.

The `NetworkInfo` class reports aggregated monitoring data about the usage of the communication infrastructure on the target host. Monitored data include the total number of sent/received UDP/IP packets, TCP connections and sent/received segments, the percentage of UDP/IP packets received with errors and the percentage of discarded UDP/IP output packets. These parameters are sufficient to give an overall evaluation of the host traffic conditions and to identify congestion situations.

Finally, the `FileSystemInfo` class maintains general information about the file system of the target (disk free space and its percentage on total size) and detailed data about currently opened files. In particular, for any active process and for any file opened in the current session, the class returns the opening time and its opening mode (read/write/both/locked).

4. THE MAPI IMPLEMENTATION

We have implemented MAPI to be the portable core component of an on-line distributed monitoring tool for the open Internet infrastructure. On the one hand, this imposes the need to achieve complete portability over the most diffused heterogeneous platforms, with the definite implementation constraint of not modifying the standard JVM. On the other hand, Internet openness and dynamicity call for the possibility of monitoring service components without requiring any intervention in either their source code or their executables. The respect of this further implementation constraint makes MAPI a completely original contribution in the Java-based monitoring arena, as detailed further in Section 8.

MAPI has required the design and implementation of several *ad hoc* modules: (1) the *MAPI Profiler Agent* for dynamically configurable on-line monitoring of the JVM state; (2) the *MAPI SNMP Agent* to obtain monitoring data from SNMP agents in execution on the targets; (3) the *MAPI*ResManager* and its native libraries (*MAPI Windows/Solaris/Linux RM DLL/SO*) for uniform data acquisition via heterogeneous platform-dependent mechanisms.

4.1. The *MAPI Profiler Agent*

The SUN JVMPI components significantly constrain the provision of monitoring information. Developers can only specify whether the JVMPI supported events should be notified to the profiler agent, and the specification is coarse-grained, with no possibility of fine selection and dynamic refinement. For instance, a profiler agent can only choose to enable/disable all events related to all Java classes (or objects/methods/monitors), but it can neither focus on the events generated by a specific class nor define user-/application-specific events.

The only way to obtain more fine-grained indicators is to implement *ad hoc* profiler agents, as our *MAPI Profiler Agent*, capable of filtering the events of interest and suitable for composing them in higher level indicators. In addition, the *MAPI Profiler Agent* gives the possibility of changing the set of notifiable events with no suspension of the monitoring execution, by implementing methods to enable/disable dynamically the event notification related to object allocation/deallocation, method invocation/exit and lock/unlock of Java monitors. Our profiler agent keeps and updates statistics of the monitored events, to provide immediately readable indicators without the need to maintain huge logs of monitoring data. For instance, it traces only the size of the total memory allocated to a Java thread and does not log the full data related to the execution of any system call for memory allocation.

Figure 3 sketches a piece of the *MAPI Profiler Agent* code. When a registered event occurs, JVMPI signals an event `ev` to the profiler that performs event-specific actions. In particular, the figure shows the initializations made when the class `SocketInputStream` is loaded. After initializing the internal `socketread` variable, the profiler can trace any invocation of the method `socketRead()` by incrementing the `stat->tcp_read` counter, which maintains the account for the TCP read operations of any Java thread in a specified time interval. These data represent a rough estimation of the incoming network traffic produced by Java service components. If there is the need for more precise information about the traffic due to specific Java threads, *ResourceManager* can command the profiler to examine dynamically the invocation parameters of the `socketRead()/socketWrite()` methods. This is possible via the JVMPI-based triggering of `JVMPI_EVENT_OBJECT_DUMP` of the required objects, at the maximum level of detail (`JVMPI_DUMP_LEVEL_2`). The *MAPI Profiler Agent*, of course, behaves differently at default to avoid the excessive overhead of the dynamic generation and processing of object dumps.

4.2. The *MAPI SNMP Agent*

The *MAPI SNMP Agent* refines and extends the SNMP component included in our MA-based MESIS framework for

```
JVMPI_Event *ev;                          // JVMPI event reference
jmethodID socketread = NULL;              // method reference

switch(ev->event_type)
{...
case JVMPI_EVENT_CLASS_LOAD:
  if(strcmp(ev->u.class_load.class_name,
            "java/net/SocketInputStream")==0)
    {
    JVMPI_Method *meth;
    for(meth=ev->u.class_load.methods; ...; meth++)
      if(strcmp(meth->method_name,"socketRead")==0)
        socketread=meth->method_id;
    }
  break;
case JVMPI_EVENT_METHOD_ENTRY2:
  stat = tab1.get(ev->env_id);
  if(ev->u.method.method_id==socketread)
    stat->tcp_read++;                         // update TCP statistics
...}
```

FIGURE 3. Monitoring of the invocation of the `socketRead()` method in *MAPI Profiler Agent*.

network and service management [10]. It acts as an SNMP manager that locally interrogates its co-located SNMP agent. The *MAPI SNMP Agent* is programmed to request monitoring information maintained not only in the standard MIB (monitoring data about network elements and protocols), but also, where supported, in some Host Resources Groups MIB extensions, called Storage, Running Software and Running Software Performance [12]. These groups provide the data about resource usage of processes currently in execution to obtain the MAPI `ProcessInfo` and `FileSystemInfo`, while `NetworkInfo` exploits the standard SNMP MIB.

The *MAPI SNMP Agent* significantly improves the efficiency of standard client/server SNMP operations, especially when dealing with the network transfer of large chunks of monitoring data. First, it transmits only the changed MAPI indicators to *ResourceManager*, which maintains old values for the non-received parameters. Most importantly, it locally interrogates its SNMP agent and pre-processes the obtained results to offer concise indicators to possibly remote managers, thus significantly reducing the generated network traffic. In fact, a single MIB variable is usually at a lower level than the MAPI indicators, and an aggregation of multiple variables is required. These aggregations are known as health functions [13]. For instance, the percentage of discarded IP output packets is obtained by combining five MIB variables:

$$\begin{aligned}\texttt{ipPackOutErr} = [(&\texttt{ipOutDiscards}\\ &+ \texttt{ipOutNoRoutes}\\ &+ \texttt{ipFragFails}) * 100]\\ &[\texttt{ipOutRequests}\\ &+ \texttt{ipForwDatagrams}]^{-1}\end{aligned}$$

where `ipOutDiscards`, `ipOutNoRoutes` and `ipFragFails` are the number of output IP datagrams discarded (for problems in buffer space, in routing and in fragmentation, respectively), while `ipOutRequests` and `ipForwDatagrams` are the total number of IP datagrams transmitted (for locally generated packets and forwarded ones, respectively) [7].

In addition, the *MAPI SNMP Agent* can perform all the operations needed for the support of mutual authentication in the case of interaction with SNMPv3 agents. It can obtain dynamically the needed security information from the public key infrastructure integrated with the SOMA programming framework [11, 14]. Finally, it can locally store configuration parameters specific for its SNMP agent (e.g., the supported MIBs), in order to automate the possibly complex phase of initialization of the MAPI tool.

4.3. The *MAPI*ResManager*

To monitor target hosts not supporting either the SNMP agent or the Host Resources MIB extensions, MAPI also integrates native monitoring mechanisms. MAPI native modules extract uniform data by exploiting heterogeneous monitoring mechanisms provided by the target operating systems. The *ResourceManager* class employs JNI to load the target-specific native library at runtime. We have currently implemented the native monitoring components for Windows NT (*MAPI WindowsRM* DLL), Solaris (*MAPI SolarisRM* SO) and Linux (*MAPI LinuxRM* SO), as depicted in Figure 1.

Figure 4 shows some lines of the *MAPI WindowsRM* DLL, which accesses kernel and application state indicators maintained in Microsoft system registry keys. In particular, the figure reports the polling of the registry to obtain updated information about the processes in execution. The system call `RegQueryValueEx(HKEY_PERFORMANCE_DATA)` returns a `perfdata` reference which is used as the base to access the monitoring information about the process with identifier `PID`.

For Solaris and Linux platforms, we have implemented native monitoring modules as dynamic SO libraries that mainly exploit the `/proc` feature. `/proc` is a virtual directory that exports kernel/application state indicators

```
RegQueryValueEx(HKEY_PERFORMANCE_DATA, "232", NULL, NULL, perfdata, &size);
                                    // "232" for process-related data
RegCloseKey(HKEY_PERFORMANCE_DATA);
pointer = (PBYTE)perfdata + perfdata->HeaderLength;
obj = (PPERF_OBJECT_TYPE)pointer;
pointer = (PBYTE)obj + obj->HeaderLength;
cnt = (PPERF_COUNTER_DEFINITION)pointer;
while (cnt->CounterNameTitleIndex != PID)
          { pointer = (PBYTE)cnt + cnt->ByteLength;
            cnt = (PPERF_COUNTER_DEFINITION)pointer;
          }
pointer = (PBYTE)obj + obj->DefinitionLength;
inst = (PPERF_INSTANCE_DEFINITION)pointer;
pointer = (PBYTE)inst + inst->ByteLength + cnt->CounterOffset;
value = *((jlong*)pointer);
```

FIGURE 4. Monitoring process information in *MAPI WindowsRM* DLL.

as a specific sub-tree of the file system. The *MAPI SolarisRM/LinuxRM* library polls monitoring information about currently executing processes by reading the corresponding files in the /proc directory. For instance, via the ioctl() call it obtains prpsinfo and prusage information, which maintain several data about the identity of a specified process and its CPU usage, respectively. Similarly, *SolarisRM/LinuxRM* native components extract the descriptors of the open files from the /proc/PID/fd virtual directory, where PID is the identifier of the monitored process, and combine them with the information from the system file table, as in the implementation of the Unix fuser utility. Aggregated information about network usage is obtained by invoking the standard netstat system call [15].

5. MAPI-BASED DISTRIBUTED MONITORING

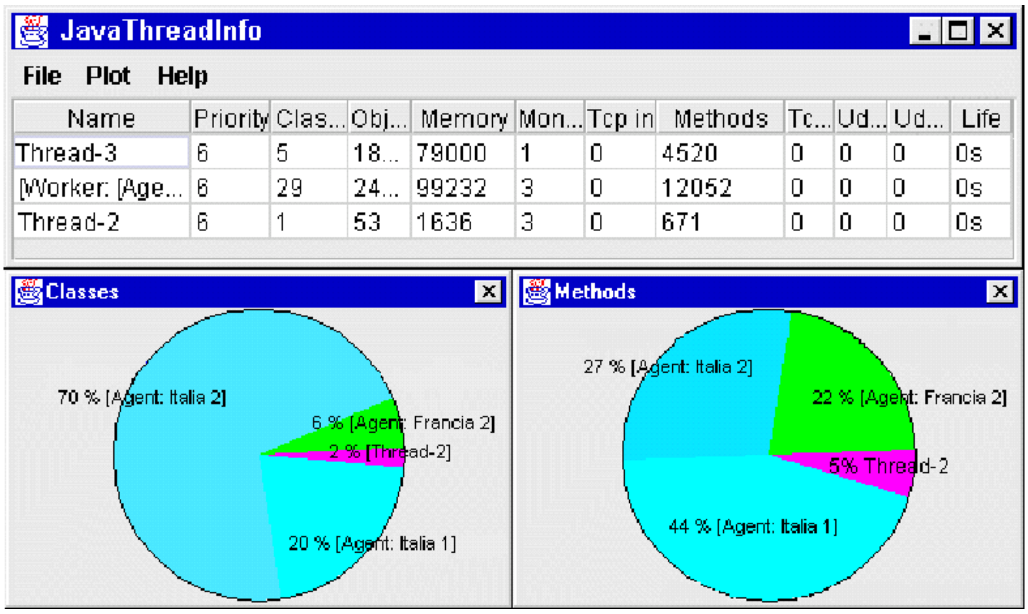
We have designed and implemented a distributed monitoring (DM) tool consisting of mobile agents that use MAPI to monitor each target node. We have adopted the SOMA mobile agent technology to improve the performance in collecting/processing distributed monitoring information and to facilitate the coordination of autonomous monitoring agents. SOMA provides a distributed infrastructure for the design, implementation and support of Web services. It includes middleware facilities for agent mobility, communication, security and interoperability, built on top of the standard portable JVM. Further details about SOMA are presented in [11, 16] and are beyond the scope of this paper, which focuses specifically on agents for MAPI-based distributed monitoring.

DM combines two types of monitoring agents: *Managers* and *Explorers*. Each *Explorer* agent is in charge of collecting monitoring data from one set of targets (i.e. *target domain*), usually belonging to the same network locality. *Manager* agents command *Explorers*, combine their monitoring results and are in charge of giving a global view of monitored domains to system administrators. In addition, *Managers* can delegate some management operations to *Explorers* that can autonomously control and handle assigned resources according to the enforced management policies, in order to achieve decentralization and scalability [10, 17, 18, 19].

DM permits several organizations, with different hierarchical levels and numbers of *Managers/Explorers* per target domain. The default choice is one single *Manager* that autonomously interworks with its *Explorers* (one *Explorer* for each administered target domain). In particular, the *Manager* can either ask *Explorers* to gather specified monitoring data, with specific alert thresholds and refresh time intervals, or command management operations. In addition, a *Manager* can create new monitoring *Explorers* at runtime when in need of controlling new target domains.

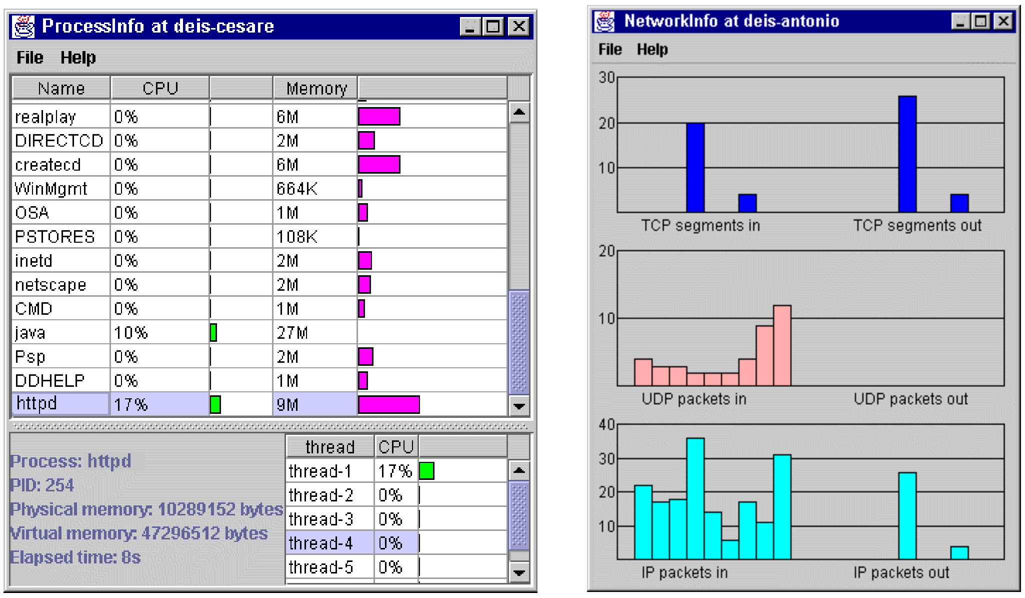
Explorer agents migrate to their target hosts to access locally MAPI functions and to collect both kernel and application state indicators. *Explorers* eventually control dynamic alert thresholds: when thresholds are exceeded, the *Explorer* either notifies its *Manager* or autonomously takes local corrective operations. *Explorer* local decisions significantly reduce the network overhead of distributed monitoring. On the one hand, *Explorers* can perform autonomous monitoring/management operations locally to the administered resources without the intervention of either *Managers* or system administrators. On the other hand, they can carry parts of state of their monitoring activities while migrating from one host to another in the target domain. That can permit monitoring/management global decisions based on the state of already visited system parts. SOMA facilitates multi-hop mobility patterns by defining agent itineraries, i.e. sequences of hosts for an agent to visit. Single-hop mobility behavior, suitable for monitoring and controlling resources at one host, is implemented by itineraries that include a single destination. In addition, the SOMA platform supports the caching of agent code on destination targets, thus significantly reducing the migration overhead in the case of usual enforcement of the same management policies [11].

For instance, one *Manager* agent can be in charge of controlling the storage devices in administered domains. It can delegate one *Explorer* for each domain to perform cache cleanup of browser directories when the available space becomes lower than a specified threshold. The threshold can state a per-host local constraint or an overall per-domain limit. In the latter case, the *Explorer* can keep partial results of space availability for the already visited domain targets



Name	Priority	Clas...	Obj...	Memory	Mon...	Tcp in	Methods	Tc...	Ud...	Ud...	Life
Thread-3	6	5	18...	79000	1	0	4520	0	0	0	0s
[Worker: [Age...	6	29	24...	99232	3	0	12052	0	0	0	0s
Thread-2	6	1	53	1636	3	0	671	0	0	0	0s

FIGURE 5. *Manager* GUIs with `JavaThreadInfo` for a target domain.



Name	CPU	Memory
realplay	0%	6M
DIRECTCD	0%	2M
createcd	0%	6M
WinMgmt	0%	664K
OSA	0%	1M
PSTORES	0%	108K
inetd	0%	2M
netscape	0%	2M
CMD	0%	1M
java	10%	27M
Psp	0%	2M
DDHELP	0%	1M
httpd	17%	9M

Process: httpd
PID: 254
Physical memory: 10289152 bytes
Virtual memory: 47296512 bytes
Elapsed time: 8s

thread	CPU
thread-1	17%
thread-2	0%
thread-3	0%
thread-4	0%
thread-5	0%

FIGURE 6. *Manager* GUIs with `ProcessInfo` and `NetworkInfo` of two Windows NT targets.

and can suspend migrations when the partial results already satisfy the required threshold.

Explorers can invoke the *MAPI Profiler Agent* functions to control and manage local Java threads. In particular, they can modify the priority of running threads and can force thread suspension/termination. These functions help in controlling the execution of Java applications and to make possible the limitation of their resource consumption at runtime, as described in the following section. Let us observe that the specification and implementation of the JVM itself limits strictly the control actions feasible with the *MAPI Profiler Agent*. Additional control functions are recognized as being crucial, especially in the emerging market of Java-enabled embedded devices, and have recently stimulated SUN research work to provide separated execution environments for different Java thread groups (*task isolation*), to permit independent control [20].

The DM overhead can be tuned at runtime by modifying different monitoring parameters. It is possible to change dynamically both the *Explorers* visit time interval to target domains and the refresh time interval for MAPI modules. In addition, the *Manager* can command *Explorers* to invoke the *MAPI Profiler Agent* methods to enable/disable the notification of specific kinds of events, thus adapting dynamically the collection of monitoring data to the enforced management policy. Section 7 presents the experimental evaluation of the MAPI overhead.

Figures 5 and 6 show some graphic user interfaces (GUIs) that the *Manager* offers administrators to show service/system conditions. In particular, Figure 5 reports the monitoring data about four Java threads in execution on three hosts of its target domain A. Apart from thread names and execution hosts, the GUIs report monitoring information about thread priority, allocated memory/objects, loaded classes, contended monitors, invoked methods, in/out TCP/UDP packets and lifetime.

Figure 6 visualizes some MAPI indicators about running processes and network traffic on two Windows NT monitored targets. For any process, the GUIs show the process name and identifier, the current CPU usage percentage, the physical/virtual allocated memory, the composing threads and the detailed CPU usage. Network traffic information includes the total number of sent/received TCP segments and UDP/IP packets. Authorized system/service administrators can choose between different forms of visualization of monitoring data, such as the pie charts and histograms shown in the figures.

6. MONITORING AND CONTROL OF MOBILE AGENT RESOURCE CONSUMPTION

The SOMA platform itself uses MAPI to monitor and limit mobile agent operations at runtime and, consequently, to control the consumption of distributed resources in SOMA-based Web services. The ultimate goal is to charge the responsible principals for the actions of their mobile agents and to prevent denial-of-service attacks by either malicious or badly programmed agents.

SOMA administrators can define agent permissions and duties, which DM controls and enforces dynamically. Agent permissions specify the actions agents are authorized to perform on a set of resources, even depending on runtime conditions. Agent duties specify the actions agents must perform on a set of target resources when a specified condition takes place [19].

As an example of agent permission specification, one administrator can control the disk resources consumed in a target domain with a distributed authorization policy that limits the total number of bytes that a specified SOMA agent can write during one day over the target hosts in the domain. This limit can also apply to groups of agents, e.g. the whole set of agents of the same responsible principal. At any request of file opening in writing mode by one of the specified agents, the *Manager* commands one *Explorer* to probe the target domain. This introduces delay in agent authorizations but permits global policies that depend on the current distributed state. Let us point out that, even in this simple example, the *Explorer* significantly benefits from the possibility of bringing its reached execution state with it while migrating in the target domain.

As an example of agent duty specification, one administrator can request an *Explorer* agent to lower autonomously the priority of a SOMA agent running on a host when the total CPU usage percentage on that host is higher than a specified threshold. Differently from the case of permissions, the *Manager* commands the *Explorer* to control the state of the CPU usage on the specified host periodically, depending on the interval time indicated in the duty specification. When the threshold is overcome, the *Explorer* itself acts on the specified agent by exploiting MAPI functions to modify dynamically the Java thread priority.

The conditions for SOMA agent permissions/duties can also be expressed as complex functions of MAPI monitoring indicators. The above examples show that it is often necessary not only to monitor the application state of SOMA agents and of related Java threads, but also to combine these data with kernel and application state of service components even external to the JVM.

7. MAPI PERFORMANCE

On-line monitoring tools should limit introduced overhead since they should operate during service provision. We have carefully followed the guidelines of overhead limitation, together with the feature of dynamic tuning, in the design and implementation of DM. To validate MAPI applicability, we have measured the overhead on different platforms, e.g. Intel Pentium III 600 MHz PCs with either Microsoft Windows NT 4.0 or SuSE Linux 6.2, and SUN Ultra 5 400 MHz workstations with Solaris 7. The hosts are interconnected via 10 Mb Ethernet Local Area Networks (LANs).

This section reports first the costs of local *MAPI Profiler Agent* and *MAPI*ResManager* modules. For the evaluation of their performance results, we have used a Java benchmark application that stresses CPU and memory usage by generating a fixed number of different threads and objects. In particular, we report the measurements for the case of 50 benchmark processes in execution, each one with an average number of five threads. We have measured the time delay due to the monitoring tool intrusion as the overhead percentage (*Overhead%*), i.e. the difference of T_{Mon} and T_{noMon} (normalized to T_{noMon}):

$$\mathit{Overhead\%} = \left(\frac{T_{\mathrm{Mon}} - T_{\mathrm{noMon}}}{T_{\mathrm{noMon}}}\right) \times 100$$

where T_{Mon} is the average completion time of the Java benchmark on unloaded targets with the MAPI tool in execution and T_{noMon} is the analogous time measured for the benchmark without MAPI. Let us note that unloaded targets represent the worst possible case for the *Overhead%* indicator. In fact, as soon as the load increases, T_{noMon} grows faster than T_{Mon}, and, consequently, the ratio of their difference tends to decrease. This consideration has been validated by measurements of *Overhead%* with several general-purpose benchmark tests running: the measurements have shown an average decrease of about 1.0–1.5% of the *Overhead%* indicator for average load conditions.

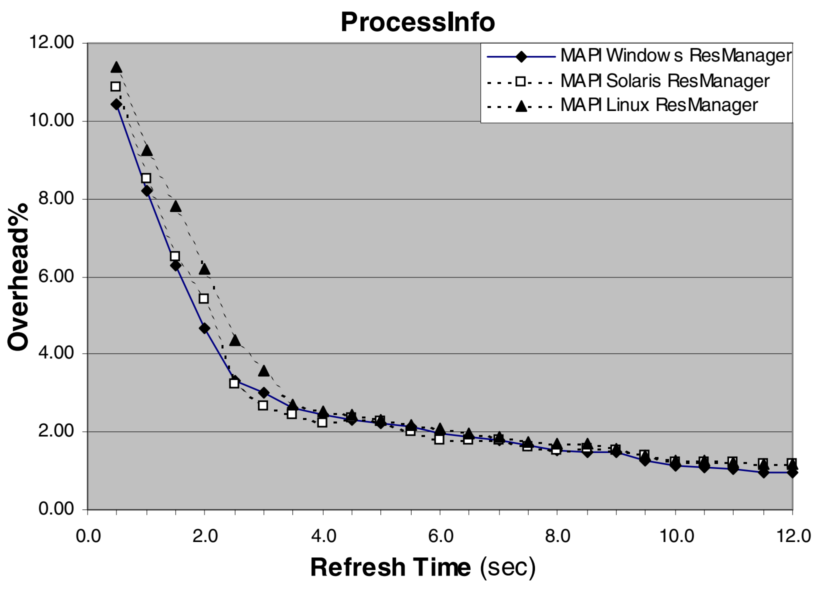


FIGURE 7. *Overhead%* of *MAPI*ResManagers* for process monitoring.

Figure 7 depicts the *Overhead%* introduced by the *MAPI*ResManager* to monitor the whole information contained in `ProcessInfo` for the three platforms. The graph reports *Overhead%* as a function of the refresh time interval, i.e. the interval between two successive invocations of the native monitoring modules via JNI. *Overhead%* exhibits a linear dependence with the reciprocal of the refresh time. We have obtained the same trends in intrusion measurements for native monitoring of `NetworkInfo` and `FileSystemInfo`.

In general, the whole set of tests shows that the *MAPI*ResManager Overhead%* is always lower than 3% when the refresh time interval is greater than 3 seconds. This is largely acceptable because this refresh time is enough for the time requirements of most Web-based services. Let us recall that native modules keep on collecting monitored events, and the refresh interval represents only the polling period between successive requests for native monitoring results.

We have also measured the overhead due to the *MAPI Profiler Agent*. In this case, the JVM notifies events continuously (and not only at specified polling intervals), and the refresh time represents the interval to process the collection of observed events to obtain the concise MAPI indicators. Figure 8 splits the *Overhead%* into the parts due to data access and to the JVMPI notification mechanisms, i.e. monitor, method and object tracing. The results obtained for the Solaris platform are very similar to those for Windows NT and Linux, with deviations of the maximum *Overhead%* of less than 3% from the maximum *Overhead%*.

Figure 8 also shows that the JVMPI notification is scarcely intrusive under different load conditions and is independent of refresh times. The refresh time seems to affect only the overhead for processing collected events and for reading MAPI indicators. Object tracing has been shown to be the most relevant factor in the *MAPI Profiler*

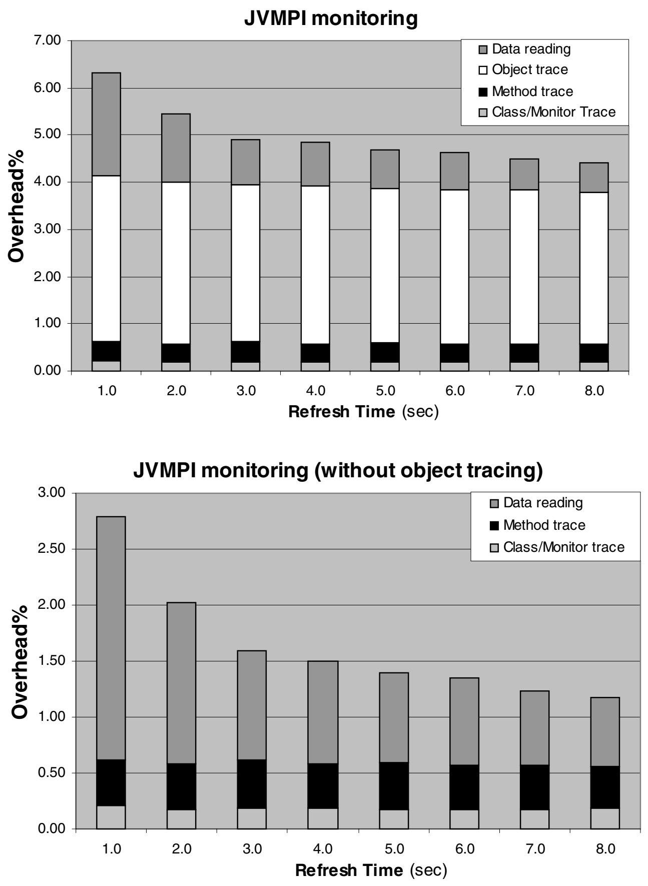


FIGURE 8. *Overhead%* of the *MAPI Profiler Agent*.

Agent intrusion because it requires the profiler to receive and collect a large amount of data. In any case, the total overhead is always below 2.0%, for refresh intervals greater than 2 seconds and with object tracing disabled.

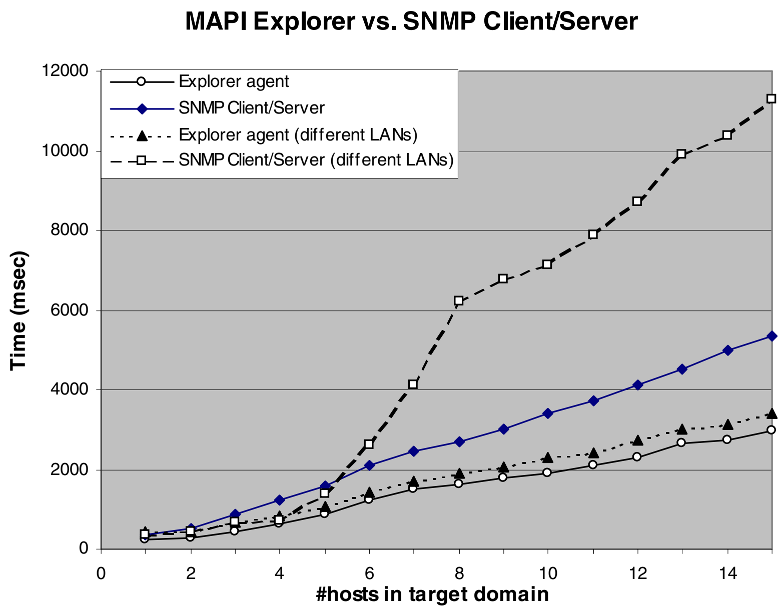


FIGURE 9. MAPI *Explorer* versus SNMP client/server to monitor one target domain.

Finally, we have also validated the applicability of our mobile-agent-based DM. The *Explorer* size varies from about 8 kB (at their first migration, without carrying any monitoring indicator) to 15 kB (at the end of exploration, including all the monitoring state of the target domain). Let us note that *Explorers* need to transfer their code only at the first migration because the SOMA platform at the destination host caches Java classes and triggers the code migration only when it is not available locally [11]. Figure 9 reports the time an *Explorer* takes to collect monitoring indicators by using *MAPI SNMP Agents*, depending on the number of hosts in the target domain. The average time has been measured over a large set of experiments, by assuming a 30% probability of finding the needed agent classes already cached at the destination targets.

The presence of *MAPI SNMP Agent* permits the collected monitoring data to be filtered and pre-processed. For several different management policies, we have experienced that the *MAPI SNMP Agent* operations significantly reduce the size of MAPI data collected by the *Explorer* with respect to the size of corresponding raw MIB values by a factor of three. This optimization is impossible in a traditional client/server approach where a centralized manager should remotely interrogate involved SNMP agents (SNMP client/server graph in the figure). The same experiments confirm that the MA technology is particularly suitable when the target domain includes different LANs, interconnected by low bandwidth links (in our measurements, two Ethernet LANs with $n/2$ hosts, connected via a 56 kb modem link). In this case, *Explorer* uses the slow link only once to migrate from one LAN to the other. The client/server solution, instead, should operate over the link at least for $n/2$ SNMP requests and $n/2$ SNMP replies, wherever the centralized manager is located.

All these results show that, when data filtering is possible and heterogeneity is present, the adoption of mobile agents in DM reduces both time performance and generated traffic. Our results confirm model estimations and first experimental measurements of research work on SNMP distributed monitoring based on mobile agents [13, 21]. In addition, the results point out that the overhead of *MAPI*ResManager* is significantly higher than that of *MAPI SNMP Agent*. This has brought us to prefer the latter module when it is available at the target host and supports the Host Resources MIB extensions.

8. RELATED WORK

Recent research activities have worked on the definition of methodologies, on the design of architectures and on the implementation of supports for local and distributed monitoring by following very different approaches [4, 7, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33].

Several distributed monitoring systems that are based on code instrumentation have achieved interesting results, especially in limiting local overhead and network traffic [23, 26, 28]. However, they require instrumenting either the source code or the binary of managed service components, and force one to recompile or at least restart the monitored services. In addition, the solutions that require source/binary rewriting tend to be language- and platform-specific. This is the main reason why instrumentation-based tools are not a viable solution for monitoring legacy heterogeneous systems and service components in the open and global Internet.

Other research efforts have specifically addressed on-line monitoring. They typically concentrate on tools that minimize monitoring intrusion, by exploiting *ad hoc* mechanisms available only for specific operating systems [30, 31]. These solutions also suffer from platform dependency and lack of portability.

In the area of network monitoring and management, many researchers have focused on standard protocols to exchange information about the state of network equipment. The widespread protocol is still SNMP, mainly because of its simplicity [25]. Other approaches start to be widely employed: some provide network traffic monitoring with the granularity of a whole network segment (Remote MONitoring, RMON [7]); others exploit platform-dependent libraries and commands (such as the UNIX `libpcap` library) to capture, filter and analyze network packets at general-purpose hosts [34, 35]. These tools, however, aim at the dynamic observation of network traffic and not at making available application-level monitoring data on the state of distributed service components.

The wide adoption of Java in Internet applications has changed the perspective in the monitoring area, by calling for Java also in the implementation of monitoring/management systems. The first activities simply addressed the enhancement of standard SNMP solutions with Web accessibility. Some other proposals started to exploit Java networking facilities and code mobility to provide an integrated middleware for distributed monitoring [36]. The SUN research work in the Java Management API and in the Java Dynamic Management Kit has recently produced the specification and the reference implementation of the Java Management Extensions (JMX) [32]. JMX represents a standard and state-of-the-art solution to instrument monitored resources and to develop management agents. JMX provides an interesting and rich architecture for distributed on-line monitoring, but imposes the instrumentation of the source code of managed resources in accordance with the JMX Instrumentation Level Specification. In addition, monitoring targets should be designed in Java (or enclosed by a Java wrapper). In one sentence, JMX suffers from the same limitations pointed out for instrumentation-based monitoring tools.

Due to the novelty of the technology, there are few examples of Java monitoring tools based on JVMPI. PerfAnal [9] exploits the SUN HPROF profiler agent to perform an off-line analysis of collected monitoring data and to obtain a user-level concise view for debugging Java applications. JProf [8] implements its own profiler agent and process. In addition, it provides a large set of functions to present the off-line data in user-level interoperable formats, such as tables and diagrams organized by using XML. None of the two implements an *ad hoc* profiler agent for on-line monitoring. In addition, to the best of our knowledge, there are no implemented Java-based tools that currently exploit JNI to integrate with native monitoring mechanisms.

The mobile agent technology has been considered extremely suitable for monitoring and managing distributed systems/services [37]. Several research experiences have demonstrated the mobile agent flexibility and effectiveness to decentralize and to automate the control of service components [13, 18]. However, DM is the first mobile-agent-based tool that provides distributed on-line monitoring of both kernel and application states without imposing any modification to the standard JVM and without requiring the instrumentation of the target source/binary code. Both the above requirements are crucial to the application of the tool in legacy systems and service components [33, 38, 39].

9. CONCLUSIONS AND FUTURE WORK

The control and management of Web services at runtime require the availability of portable on-line monitoring tools to ascertain the current state of distributed systems and services, without imposing any intervention on the implementation of the monitored targets. These monitoring tools should be core components of any integrated infrastructure for the support of differentiated levels of service quality over best-effort networks.

The MAPI solution addresses this scenario and its performance demonstrates the applicability of Java-based solutions for the on-line monitoring of most classes of Web services. By using MAPI, service developers can tune dynamically the level of detail of monitored events and their refresh time, to adapt the collection of monitoring data (and the corresponding monitoring overhead) to runtime conditions and service-specific constraints. In addition, the mobile agent technology has been shown to be extremely suitable for reducing both monitoring traffic and control latency, by permitting one to filter/process data and to perform management operations directly at the monitored targets.

Given the encouraging performance results achieved, we are currently working on extending the functions and the usability of MAPI-based distributed monitoring. We have extended the MAPI local component to support monitoring-based user accounting in a non-repudiable way, and we have experimented with it in the context of user/terminal mobility [40]. In addition, we are testing the first prototype of an enhanced version of the MAPI-based distributed tool that automatically organizes hierarchies of *Managers* depending on the topology of the administered systems and service components, thus improving the scalability of our solution when applied to large-scale deployment scenarios. Finally, we are working to increase the usability of the MAPI-based solution by integrating it with user-friendly GUIs for the definition of monitoring/management policies and for their automatic translation in the Ponder Policy Specification Language [19].

REFERENCES

[1] Lewis, T. (1998) Information appliances: gadget Netopia. *IEEE Computer*, **31**, 59–60.

[2] Chalmers, D. and Sloman, M. (1999) A survey of quality of service in mobile computing environments. *IEEE Commun. Surveys Tutorials*, **2**, 2–10.

[3] Xipeng, X. and Ni, L. M. (1999) Internet QoS: a big picture. *IEEE Network*, **13**, 8–18.

[4] Buyya, R. (2000) PARMON: a portable and scalable monitoring system for clusters. *Softw.—Pract. Exper.*, **30**, 723–739.

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

[6] Gordon, R. (1998) *Essential Java Native Interface*. Prentice-Hall, Upper Saddle River, NJ.

[7] Stallings, W. (1998) *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2* (3rd edn). Addison-Wesley, Reading, MA.

[8] Pennington, G. and Watson, R. (2000) *JProf—a JVMPI Based Profiler*. http://starship.python.net/crew/garyp/jProf.html.

[9] Meyers, N. (2001) *PerfAnal: a Performance Analysis Tool*. http://developer.java.sun.com/developer/technicalArticles/GUI/perfanal/index.html.

[10] Bellavista, P., Corradi, A. and Stefanelli, C. (2000) An integrated management environment for network resources and services. *IEEE J. Selected Areas Commun.* (Special Issue on Recent Advances in Network Management and Operations), **18**, 676–685.

[11] Bellavista, P., Corradi, A. and Stefanelli, C. (2000) Protection and interoperability for mobile agents: a secure and open programming environment. *IEICE Trans. Commun.* (IEICE/IEEE Special Issue on Autonomous Decentralized Systems), **E83-B**, 961–972.

[12] Waldbusser, S. and Grillo, P. (2000) *Host Resources MIB*, RFC2790. http://www.ietf.org/rfc/.

[13] Gavalas, D., Ghanbari, M., O'Mahony, M. and Greenwood, D. (2000) Enabling mobile agent technology for intelligent bulk management data filtering. In *Proc. of NOMS'00*, Honolulu, HI, April 11–13, pp. 865–876. IEEE Press, Piscataway, NJ.

[14] Entrust Technologies. *Entrust/PKI*. http://www.entrust.com/products/pki/.

[15] Nemeth, E., Snyder, G., Hein, T. R. and Seebass, S. (2000) *UNIX System Administration Handbook* (3rd edn). Prentice-Hall, Upper Saddle River, NJ.

[16] Bellavista, P., Corradi, A. and Stefanelli, C. (2000) A mobile agent infrastructure for terminal, user and resource mobility. In *Proc. of NOMS'00*, Honolulu, HI, April 11–13, pp. 877–890. IEEE Press, Piscataway, NJ.

[17] Goldszmidt, G. and Yemini, Y. (1995) Distributed management by delegation. In *Proc. of ICDCS'95*, L'Aquila, Italy. IEEE Computer Society, Los Alamitos, CA.

[18] Pagurek, B., Wang, Y. and White, T. (2000) Integration of mobile agents with SNMP: why and how. In *Proc. of NOMS'00*, Honolulu, HI, April 11–13, pp. 609–622. IEEE Press, Piscataway, NJ.

[19] Corradi, A., Dulay, N., Montanari, R. and Stefanelli, C. (2001) Policy-driven management of agent systems. In *Proc. of the Policy Workshop*, Bristol, January 12–14, pp. 214–229, Springer, Berlin.

[20] Java Community Process. *Application Isolation API Specification (Java Specification Request #121)*. http://www.jcp.org/jsr/detail/121.jsp.

[21] Baldi, M. and Picco, G. P. (1998) Evaluating the tradeoffs of mobile code design paradigms in network management applications. In *Proc. of ICSE'98*, pp. 146–155. IEEE Computer Society, Los Alamitos, CA.

[22] Al-Shaer, E., Abdel-Wahab, H. and Maly, K. (1999) HiFi: a new monitoring architecture for distributed systems management. In *Proc. of ICDCS'99*, pp. 171–178. IEEE Computer Society, Los Alamitos, CA.

[23] Bakic, A., Mutka, M. W. and Rover, D. T. (2000) BRISK: a portable and flexible distributed instrumentation system. *Softw.—Pract. Exper.*, **30**, 1353–1373.

[24] Corradi, A. and Stefanelli, C. (1997) HOLMES: a tool for monitoring heterogeneous architectures. In *Proc. of HPC'97*, pp. 486–491. IEEE Computer Society, Los Alamitos, CA.

[25] Jiao, J., Naqvi, S., Raz, D. and Sugla, B. (2000) Toward efficient monitoring. *IEEE J. Selected Areas Commun.*, **18**, 723–732.

[26] Lange, F., Kroeger, R. and Gergeleit, M. (1992) JEWEL: design and implementation of a distributed measurement system. *IEEE Trans. Parallel Distrib. Syst.*, **3**, 657–671.

[27] Liang, Z., Sun, Y. and Wang, C. (1999) ClusterProbe: an open, flexible and scalable cluster monitoring tool. In *Proc. of IWCC'99*, Melbourne, Australia, December 2–3, pp. 57–64. IEEE Computer Society, Los Alamitos, CA.

[28] Miller, B. P., *et al.* (1995) The Paradyn parallel performance measurement tools. *IEEE Computer*, **28**, 37–46.

[29] Russ, S. H., *et al.* (1999) Hector: an agent-based architecture for dynamic resource management. *IEEE Concurrency*, **7**, 47–55.

[30] Schroeder, B. A. (1995) On-line monitoring: a tutorial. *IEEE Computer*, **28**, 72–78.

[31] Weiming, G., Eisenhauer, G., Schwan, K. and Vetter, J. (1998) Falcon: on-line monitoring for steering parallel programs. *Concurr.—Pract. Exper.*, **10**, 699–736.

[32] SUN Microsystems. *Java Management Extensions (JMX)*. http://java.sun.com/products/JavaManagement/.

[33] Guiagoussou, M. H., Boutaba, R. and Kadoch, M. (2001) A Java API for advanced faults management. In *Proc. of IM'01*, Seattle, WA, May 14–18, pp. 657–665. IEEE Press, Piscataway, NJ.

[34] Deri, L. and Suin, S. (2000) Effective traffic measurement using Ntop. *IEEE Commun. Mag.*, **38**, 138–143.

[35] Kumar, A. *et al.* (2000) Nonintrusive TCP connection admission control for bandwidth management of an internet access link. *IEEE Commun. Mag.*, **38**, 160–167.

[36] Lee, J. (2000) Enabling network management using Java technologies. *IEEE Commun. Mag.*, **38**, 116–123.

[37] Fuggetta, A., Picco, G. P. and Vigna, G. (1998) Understanding code mobility. *IEEE Trans. Softw. Eng.*, **24**, 342–361.

[38] Czajkowski, G. and von Eicken, T. (1998) JRes: a resource accounting interface for Java. In *Proc. of OOPSLA'98*, Vancouver, Canada, October 18–22, pp. 21–35. ACM Press, Broadway, NY.

[39] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A. and Jeffers, R. (2000) Strong mobility and fine-grained resource control in NOMADS. In *Proc. of ASA/MA'00*, Zurich, Switzerland, September 13–15, pp. 2–15. Springer, Berlin.

[40] Bellavista, P., Corradi, A. and Vecchi, S. (2002) Mobile agent solutions for accounting management in mobile computing. In *Proc. of ISCC'02*, Taormina, Italy, July 1–4, pp. 753–760. IEEE Computer Society Press, Los Alamitos, CA.