Transparent Resource Management and Self-Adaptability Using Multitasking Virtual Machine RM API

Arkadiusz Janik
Institute of Computer Science, AGH
al. Mickiewicza 30
30-059 Krakow, Poland
arkadiusz.janik@agh.edu.pl

Krzysztof Zielinski
Institute of Computer Science, AGH
al. Mickiewicza 30
30-059 Krakow, Poland
kz@ics.agh.edu.pl

ABSTRACT

The Multitasking Virtual Machine has been provided with many useful features like Isolation API or Resource Consumption Management API. The latter one can be used to help in managing resources in Java applications. However, using RM API does not guarantee separation between a resource management activity and a business activity. In this paper we present the concept of The Transparent Resource Management (TRM) system. The system can be used to run Java applications with resource management policies added dynamically, as a separate aspect. Each application running in the TRM system can be started with a reusable graph of different *states*. Each state encapsulates currently used resource management policy. An active state can change automatically during runtime, depending on the state of an application, the state of the whole system, as well as on the utilization of different resources in different applications.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*frameworks*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Languages, Management

Keywords

Resource Management, Transparent Management, Java, Isolates, Multitasking Virtual Machine, Self-adaptability

1. INTRODUCTION

The new Sun's Java Multitasking Virtual Machine ([2], [5]) has been provided with many interesting features like the resource consumption management ([3], [4], [7]) and isolation of Java applications ([1], [8], [6]). Although the Resource Consumption Management API provides useful features for writing resource-aware applications in Java, an application's developer using RM API has to include parts of the code responsible for managing resources directly into the application's code. In other words, RM API provides features which can be used to build the highest-level of management policies. The main difference between original RM API and the proposed extension is the *transparency* of resource management in our solution. RM API can be used by developers to prepare their own resource management policies. Its weakest point is the fact that much of the source code has to be written every time a new application is prepared or policy is changed. The *transparency* of the proposed architecture means that the resource management activity of an application is separated from its business activity and specified on an abstract level. Resource consumption management is invisible for an application. Moreover, the system automatically chooses proper resource management policy from a set of predefined ones to fit an application's needs. The proposed architecture makes management simple, understandable and transparent both from the application's and developer's point of view. Different management policies can be saved for later use by different developers and by different applications. The resource management functionality can be added to the target application in different cycles of the application's life and in different ways. When considering different cycles, we can add functionality either during the deployment process or during the runtime. When considering different ways of adding RM, such possibilities as an aspect oriented programming ([9]) or wrappers seem to be useful. The fostered approach can be used to implement specific resource consumption management policy, to guarantee quality of service (QoS) and to add self-adaptability to Java applications ([11], [10]). In this article we present the overall, high-level architecture of The Transparent Resource Management (TRM) system. The RM API has been originally provided with Multitasking Virtual Machine and presented approach bases on features of MVM.

The remainder of this paper is organized as follows. Section 2 introduces the Resource Consumption Management API and Isolates API while Section 3 presents the architecture of The Transparent Resource Management system. Section 4 gives an overview of the concept of *states* used to define a policy in TRM. The main focus of Section 5 is the question how the Aspect Oriented Programming can be used to implement the proposed architecture whereas the results of experiments carried out with the prototype implementation of the TRM are the subject of Section 6. The conclusions and further research are presented in Section 7.

SEAMS'06, May 21–22, 2006, Shanghai, China.

2. OVERVIEW ON RM API AND ISOLATION API

Resource Consumption Management API (RM API [7], [3], [4]) is provided as an experimental part of MVM. It is a framework used to supplement Java with resource and consumption management as well as control and reservation features. Each resource in RM API is described as a set of *resource attributes*. Resources can be *reservable*, *unbounded*, *revokable* and *disposable*. *Granularity* and *measurement delay* of the resources are defined. In RM API resource policies are encapsulated in *resource domains*. All *isolates* ([8]) that are bound to the same resource domain share the same policy. An isolate can be bound to many domains as long as each domain is responsible for a different resource. The policy is implemented by *consume callbacks* and a *trigger* that specify when callbacks are called. Callbacks can act either as *constraints* (called before a resource *consumption* is done) or as *notifications* (called after a consumption). An important element of RM API is a *dispenser*, a bridge between resource abstraction and its implementation. The dispenser monitors the amount of resource available for resource domains as well as it decides how much of the resource can be granted when the consumption is done. In order to stress that there is only one dispenser per resource system we should describe it as *global*.

The most important concept introduced by The Application Isolation API ([8]) is an *isolate* - an abstraction of a single Java task running in a Virtual Machine. In Sun's Multitasking Virtual Machine various Java applications may be started in isolates using the same virtual machine.

3. TRANSPARENT RESOURCE MANAGEMENT ARCHITECTURE

The goal of the proposed system is to separate application's business logic from its resource management logic and to make resource management invisible from application's point of view. The system is built on top of the RM API. What differs the TRM architecture from RM API is that in the TRM resource management is hidden from the application, not included into application's source code and applied to it during deployment. The resource management policy is dynamic and the application's developer may specify it as a concern of the application. Dynamic means that the policy may change during the runtime, depending on the state of an application and the whole system. What is more, in the Transparent RM architecture various policies may be easily applied. In contrast with the Transparent RM architecture, the RM API can be used to build rather static policies, which are separated from the policies of different resources while in the TRM architecture policies of different resources may be dependent. As a result, the TRM can be used to built a self-adaptable applications in Java.

3.1 High-level architecture

The high-level architecture of the Transparent RM is presented in Figure 1. The architecture specifies different elements (rectangles) and different phases (arrows). More detailed description of elements of the system is presented below.

Policy definition is a high-level, user-readable description of desired application's behavior, resource management logic, quality of service etc. This definition may limit CPU usage or network bandwidth whereas another definition may stress that the number of concurrently running threads should not increase given threshold. A policy definition depends on a type of an application and/or specific requirements for its current deployment. The same application running on different hardware configurations may have completely different policy definitions.

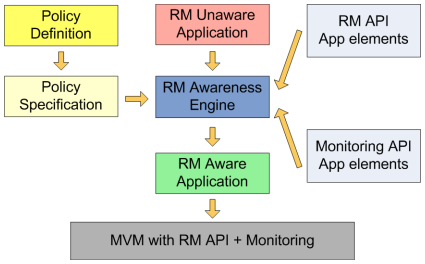

Figure 1: The architecture of The Transparent Resource Management system.

Policy specification is a formalized, more detailed presentation of policy definition. A specialized policy specification language can be used to prepare a specification. Exemplary policy specification requires that the application should not use more than 10% of CPU time nor to exceed 56kB of the network bandwidth. The limitations of different resources can be also composed to specify more complex policies like not to exceed 56kb of the network bandwidth as long as CPU load is less than 90%. If CPU load increases the higher network bandwidth is allowed (because of the switching off the CPU consuming socket compression module and sending raw data). The policies of different applications can be also merged as to treat previously seperate applications as a group. Exemplary policy can specify an upper limit of a number of threads that can be running concurrently in all started applications. The policy specification can be more or less detailed as if there was a conceptual line between the formal high-level specification and the informal low-level programming realization.

RM Unaware application is an input application which is not aware of resource consuming policies and resource management. A developer of an application can focus on business logic. Moreover, isolating business logic from RM logic makes reusability possible. Each time an application is deployed/started different resource policy and different QoS can be applied.

Transformation engine is the central point of the proposed system. The engine is responsible for transforming RM unaware application into the aware one. The transformation techniques are not presented here and will be discussed later in this paper (see Section 3.3). The transformation engine has four inputs:

- RM unaware application - an input application to be transformed;
- policy specification - policy to be applied to the application (presented above);
- **RM API App elements** - the elements of the Resource Consumption Management API to be added to the application; a transformation engine uses these

elements to add (to the input application) an entity responsible for communication with RM API; entity knows RM API, understands it and can use it as to enable desired application's behavior;

- **Monitoring API App elements** - the elements of the Monitoring API to be added to the application; the transformation engine uses these elements to add (to the input application) a entity responsible for the communication with monitoring part of the system; entity uses Monitoring API to provide information about the state of the whole system, as well as feedback on the application's behavior; the information from the Monitoring entity can be used by the RM entity as to modify the application's behavior and to guarantee RM policy or QoS;

RM Aware application is the only output from the transformation engine. The application has been modified in such a way that it can use RM API and Monitoring API to apply given RM policy. More detailed description of the application after transformation is presented in Section 3.2.

3.2 Architecture of RM Aware application

An abstract view on the RM aware application (which is the result of a transformation process) is presented in Figure 2. The RM aware application is built on top of the input, the RM unaware application. Two entities have been added to the original application. There exist three types of connections between elements in the system:

- connection between entities and external environment - each entity can exchange information with the external parts of the system such as entities belonging to other applications or specialized modules common for all of the elements in the system. E.g. monitoring entity can communicate with monitoring module which gathers and analyzes information about the state of the whole system (see Figure 3). Alternatively, a monitoring entity can communicate directly with its opposite number in a different application.

- connection between entities and RM unaware application - connection that can be used to get information about resource utilization and/or the application's state.

- connection between different entities within a single application - this type of a connection can be used for an effective exchange of information about different aspects of the same application. As a result, decisions made by entities are more precise.

As for entities there are a few ways of applying them. Aspect oriented programming can be used to make communication between entities and original application possible.

Figure 3 illustrates four exemplary applications in TRM system. Each application has two entities built in it. Each entity communicates with a proper module (common for the whole system). The communication is bidirectional which means that modules use entities to gather information and to perform adequate actions (e.g. modify the application's behavior, definitions of RM API triggers and consume-actions etc.).

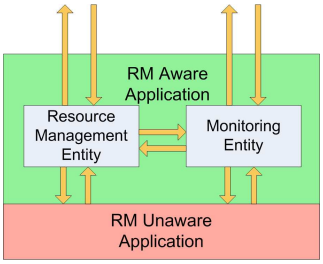

Figure 2: The architecture of the RM aware application. The RM unaware application is instrumented with entities to communicate or juxtapose application with RM Module and Monitoring Module.

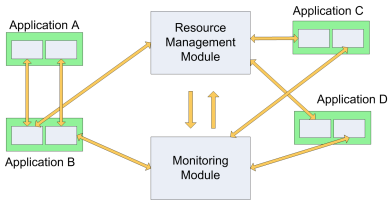

Figure 3: The management and monitoring modules in the TRM system.

Entities of different applications can communicate with each other. This type of communication may be necessary when the strict cooperation between two applications is needed.

3.3 Transformation Engine

Transformation engine is the central point of the TRM system, responsible for transforming the RM unaware application into the RM aware one. The RM API is just a framework which does not imply any resource management policies. One can use RM API to write its own resources and resource management strategies. The desired result can be achieved by defining proper resources, resource domains and limitations (via consume-action callbacks and triggers). The obvious solution of the transformation process is to implement different resource policies by providing a specialized API (built on top of RM API) with a set of ready-to-use triggers and actions. The simplest way of connecting the RM unaware application to the RM API is to use a specialized manager (isolate or program) to create and start an input application. The manager is responsible for recognition (and optionally creation) of proper resource domains, binding the original application to them and applying resource management policies (by creation of proper triggers, actions etc.). The application's source code is not modified. However, some of the resource management limitations may result in an improper behavior of the application. Let's imagine an application bound to the resource domain with limits on number of started threads. If maximum number of threads is present and the application wants to start another thread, resource exception is thrown and the application starts behaving in an incorrect way. The conclusion is that some code modifications might need to be done. The scheme of the transformation process is presented in Figure 4. Re-

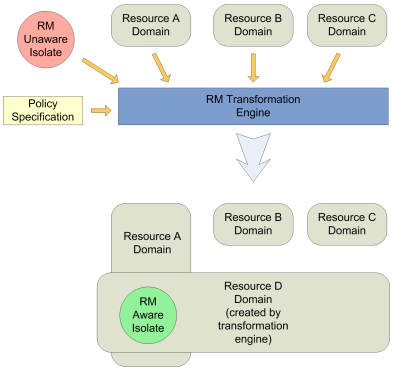

Figure 4: The process of transforming the RM unaware isolate into the RM aware one.

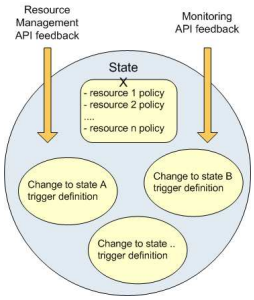

Figure 5: The elements of a *state* in the TRM.

source Domain D is created by the transformation engine to meet conditions specified in the policy specification. The rationale is that for some resources the application may be bound to the same resource domains as applications which are already running in the TRM (to share the same management policy). For other resources the application has to be bound for its own resource domains (in case of per-application policy).

As it was mentioned before, the RM aware application has resource management and monitoring entities built in them. Entities can be added either using threads "injected" into the original application or started in separated isolates. The specialized monitoring/managing modules can be used to change resource management policies dynamically (e.g. if a network policy in the application A depends on CPU usage policy in the application B).

The presence of entities in running application means that each application provides a well defined interface which can be used by other applications or by the TRM system to collect information about the application's behavior and resource utilization. Collected information may then be analyzed to check whether the conditions specified by resource policies of all applications are met. Proper activities can then be performed to guarantee that resource policy constraints are not violated. In particular, triggers and consume-action callbacks can be modified. Moreover, the architecture can be also extended to make changes of the application's behavior possible. E.g. if the CPU load increases, the application may change network traffic compression algorithm to the simplest one (less CPU), some of the application's features can be switched off, and some rescue features can be enabled. As one can see the TRM can be extended to introduce reflection and self-adaption to RM aware applications.

4. REALIZING POLICY IN TRM

The policy definition of the TRM system is realized by introducing the concept of *states*. A state defines current resource management policy and/or application's behavior. Each isolate running in the system can be in a one of a predefined set of possible states. The state being currently used is called an *active* one. Each state definition includes a few elements (see Figure 5).

State defines which policy (if any) is currently used for different resources in the system (for given isolate). State may define conditions upper which another state is *activated*. It means that an active state can change, depending on the current system image, including resource consumption information (from RM API) and feedback from a monitoring system (Monitoring API). The application's behavior can be easily changed, transparently to the application's business logic. As a result, the application is run with a graph of states applied to it (see Figure 6).

Each isolate has an *initial state* defined, the state an isolate is *bound to* after creation, and an *active* state, the one that the application is currently bound to. The graph of states can be either provided directly by a system designer, or generated automatically by transformation engine for a particular policy specification (see Section 3.1). Transitions between states are performed when conditions defined in the *active* state's definition (see Figure 5) are met. The state which should be *activated* is chosen by means of the feedback information from the Resource Management Module and the Monitoring Module. The state transition model is used because the resource policy may change during the runtime, depending on a dynamic criteria. E.g. the current CPU usage limits for the application may increase if the current system time is between 8pm and 6am (at night) or decreased if another, high-priority application has been started on the same node. The current network bandwidth limit can be increased if the CPU load is high (a socket compression is then switched off to reduce CPU load) or decreased (if a socket compression is switched on again after CPU load has been reduced). The resource limits can also be changed depending on the privileges of the user which has logged into the application etc.

The Figure 7 shows two layers of a policy definition. The policy defined on the Application Layer as a graph of states is translated to the policy on the Resource Domain Layer. This layer utilizes the RM API. It means that each state in the graph of states (on the Application Layer) has its own set of resource domains defining proper resource policies (on Resource Domains Layer).

5. RM API VS ASPECT ORIENTED PROGRAMMING

In Section 3.2 we have presented different techniques which can be used to add the resource management and monitoring modules into the RM unaware application. Aspect Oriented Programming methods can be used to support this operation.

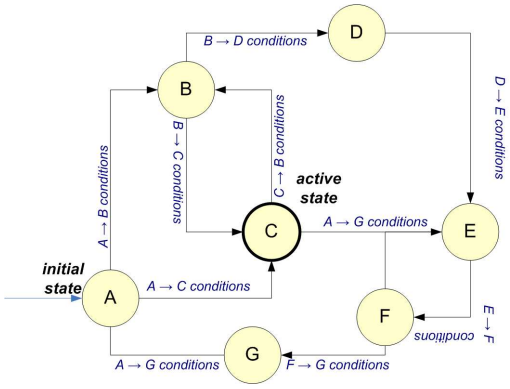

Figure 6: An example graph of states applied to an isolate running in the TRM. The state to be *activated* is chosen by means of the feedback information from the Resource Management Module and the Monitoring Module.

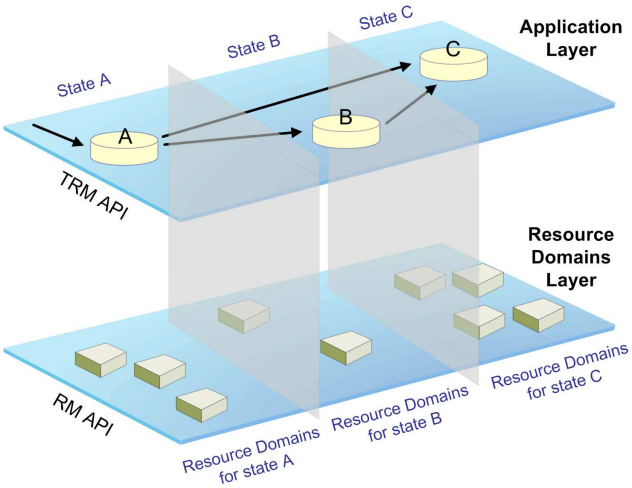

Figure 7: The two layers of a policy definition in the TRM system.

Most applications using RM API does not see dispensers directly [3]. Dispensers are visible if the application defines its own new resources and wants them to be managed via RM API. Usually, only JRE will create and register dispensers.

In this term, user's application is unaware of resources management.

However, resource domain's policy has to be specified as a set of pre-consume and post-consume actions or included into trigger's code. It means that an application's developer is involved into resource management stuff and has to consider it during code development.

As it is presented before, a specialized manager can be used to start an application's isolate, and bind it to proper resource domains (either the exiting or the new ones). The manager can also create and start resource managing and monitoring entities and the code responsible for changing the currently active state. AOP can be used to let the

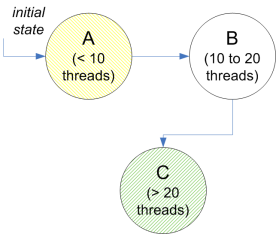

Figure 8: The graph of states applied to the example application in the TRM.

RM unaware applications behave properly when resource exceptions are thrown. AOP can also define jointpoints between the application and entities. AOP may also be used to enable self-adaptation of running application via changing parts of code whenever application's state is changed.

6. EXPERIMENTAL RESULTS

In order to illustrate the ideas of the presented architecture and to check how aspect oriented programming may support managing resources and self-adaptability, we have carried out some experiments. The following configuration was used: Solaris 9 for SPARC, Sun's Multitasking Virtual Machine 1.5.0 03-mvm-preview (build 1.5.0 03-mvm-preview-b7) with RM API in `experimenatlstuff.rm.*` package and JBoss Aspect Oriented Programming framework v. 1.3.4 [9].

6.1 CPU usage shaping

To present the concept of self-adaptability and graph of states we have written a simple Java application (*resource unaware*) which creates and starts 40 threads. All threads are identical and each of them performs dummy calculation. Because the calculation is time-consuming, single thread consumes all available CPU time (see Figure 9 "TRM disabled"). It means that this application is resource unaware. The TRM architecture can be used to make the application *resource aware* by applying a given resource policy to it.

The *RM Awareness Engine* (running in an isolate) gets the name of the application to be started (fully qualified class name), and the path to the file with the policy specification. The specification is a description of the application's states and rules of triggering them.

As presented in Figure 8, to make the example understandable, we specified only three different states in our policy (state A, B and C). Each state limits a single resource - a CPU usage. It means that the maximum CPU usage for the application being in the state A is different than for the application being in the state B etc. The state A is set as an *initial*. Triggering rules are based on a number of running threads. If a number of threads exceeds 10, the state B is *activated* and stays *active* as long as 20th thread is started (when state C is *activated*). Figure 9 presents the timepoints when the state of the application is changed as two vertical, dotted lines. As presented in Figure 8 three different policies were applied. Although the application running without the TRM consumes 100% CPU, when started in the TRM does not consume more than the limits specified for an *active* state. Different resource policies may be easy applied by changing the XML file with a policy specification.

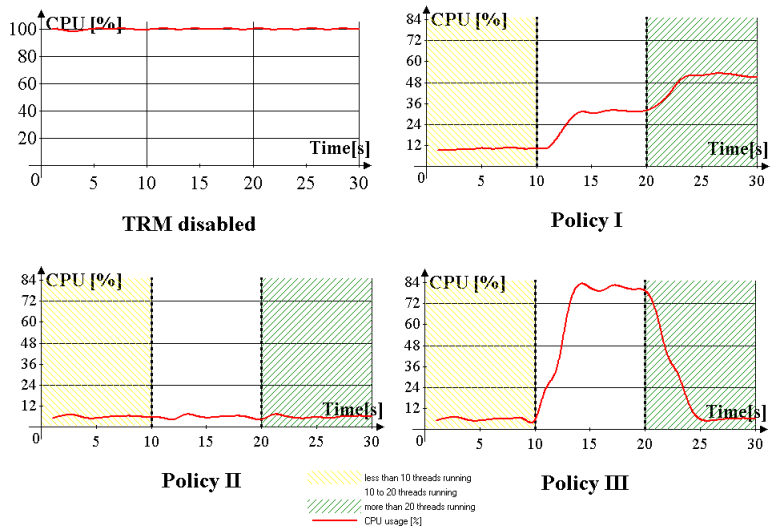

Figure 9: The comparison between CPU usage in function of time, for a demo application running without and with the TRM system. For the TRM three different policies are applied. The maximum CPU usage for the application in state *A*, *B*, *C* is: 10%, 30%, 50% for Policy I, 5% for Policy II nd 5%, 80%, 5% for Policy III.

The TRM engine creates an isolate for a resource unaware application. A policy specification is read from an XML file and the engine creates all needed resource domains, callback actions and triggers. An application's isolate is bound to resource domains defined for an *initial* state. Then, the isolate is started. The TRM engine gathers information from a monitoring module and from a resource management module to check whether an *active* state should not be changed (in our example a monitoring module is not used because information about a number of running threads is obtained from RM API interface). When the *active* state is changed, the application's isolate is rebound to different resource domains (optionally, resource domain stays the same while triggers and consume actions are changed).

The Figure 9 shows that the TRM system successfully monitors conditions of the changing of the states and limits the CPU usage. The CPU usage was measured with `ps` tool during the `java` whole process. The isolate with the transformation engine was started in the same MVM as was the application's isolate. It explains the fluctuations of the function, as well as the CPU usage 0.5%-2% higher than it is specified in the policy. Some CPU time is used by TRM system's isolate and measured with `ps` tool.

In the example above, the TRM uses two global resources, provided with RM API:
`experimentalstuff.rm.resource.ThreadNumDispenser` and `experimentalstuff.rm.resource.CpuTimeDispenser`. The `experimentalstuff.rm.resource.cputime.period` property was set to `100000000` (nanoseconds) to decrease the maximum possible interval between "now" and when the CPU time measurement value returned was taken. The lower value is set, the more precise (and expensive) CPU time accounting is done.

The `CpuTime` in the RM API is an absolute CPU time consumed by the isolates bound to a given resource domain. The resource is unbounded, non-revokable and non-disposable. In order to express it in percentages we have added a consume action (with trigger *Yes*). A callback connected with the action calculates the current CPU usage for a given resource domain and (if necessary) uses the `IsolateOperations` class to suspend all the isolates bound to the resource domain. The period of time that isolates are suspended for, is calculated using the equation 1.

$$ST = \frac{PU}{MU} - WST \tag{1}$$

where ST is Suspend Time we look for, WST is Whole System Time since resource domain was created, MU is Maximum Usage in percentages and PU is Proposed Usage passed as the `preConsume` method's parameter.

6.2 Limiting number of threads in RM unaware application

The second experiment has been carried out to present how Aspect Oriented Programming can be used to fix the behavior of the RM unaware application run with the TRM system. The application simply creates 10 threads, starts them and waits until they finish their jobs. Code inside thread's `run` method is extremely simple: just a loop from 1 to 10 with 1000ms sleep between iterations.

As presented in Figure 10 the TRM creates 3 isolates bound to a resource domain with the maximum number of running threads set to 5. The `ThreadNumDispenser` provided in RM API is used. Each isolate tries to start 10 threads.

Because of the resource domain's limitations when the application is started an `ResourceException` exception is thrown, informing that limit on the number of threads was exceeded. The application terminates.

There are different ways to solve this problem, including several modifications in the trigger's code. We have chosen an AOP solution. The TRM system provides the resource

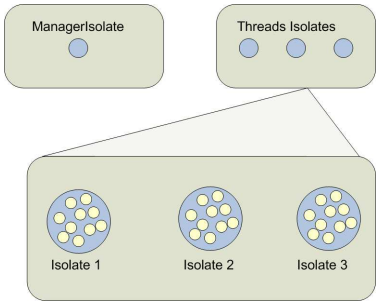

Figure 10: The deployment of the test.

Table 1: The execution time of the example three isolates in function of maximum number of threads available in a resource domain.

Number of threads	Execution time
no limit	12
30	12
20	22
10	42
8	42
5	72

management code as the crosscutting concern of the original source code. The exception is thrown in the `start()` method of the `DummyThread` class. It means that there may exist more than maximum allowed number of threads in a single resource domain but only limited number of them can be started simultaneously. To avoid the exception presented, we can add `try` and `catch` clause to catch the `ResourceException` and to try to start a thread after a while (another thread may finish its job in the meantime). A needed snippet of code can be provided outside business logic as an aspect. The interceptor may use a loop in an attempt to start a thread more than once. The AOP framework injects needed code into the original application.

The table 6.2 summarizes execution time of three isolates in the function of available threads. If no thread number limit is introduced, threads are created/finished almost in the same time, and each of them lasts for 10 seconds. If limits are introduced some threads have to wait until other finish their jobs. To understand why for no limited number of threads execution time is 12s instead of 10s, consider overhead due to thread creation/start and due to intercepting thread's `start()` method by JBoss AOP library. Let's take e.g. execution time for 20 available threads. The execution time is 10 seconds longer than for non-limited resources. The additional 10 seconds is needed because third isolate has to wait until threads from isolate 1 and 2 finish their jobs. Similar analysis can be provided for other values.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented the high-level architecture of the Transparent Resource Management system. We showed that the RM API can be used to build additional layer which helps to isolate the resource consumption management logic from the business logic. In the proposed architecture the RM unaware application is being transformed into the RM aware one, considering given resources policy. Different policies can be defined and stored for the re-use with different applications. The policies in the TRM system are dynamic, they use information about different resources in different applications. We presented the concept of the RM and monitoring modules. The modules are used to collect information about the state of the whole system. The information is analyzed to check whether resource policies constraints are not violated. Specialized entities are used as a bridge between the system and an application. We have shown that the Aspect Oriented Programming can be used to support the process of the transforming of the RM unaware application into the aware one. The next goal of our research is to investigate how the Aspect Oriented Programming can be used to change the behavior of the application dynamically to let the TRM system change the implementation of some application's method or even the whole application's object(s) during runtime. Another goal is to use JMX technology to implement the TRM system on the Java Virtual Machine (JVM) instead of Multitasking Virtual Machine (MVM).

8. REFERENCES

[1] Czajkowski, G.: Application Isolation in the Java Virtual Machine. ACM OOPSLA (2000), Minneapolis, MN.

[2] Czajkowski, G., Daynes, L.: Multitasking without Compromise: A Virtual Machine Evolution. ACM OOPSLA (2001), Tampa, FL.

[3] Czajkowski, G., Hahn, S., Skinner, G. and Soper, P., Bryce C. A Resource Management Interface for the Java Plaftorm. Sun Microsystems Laboratories Technical Raport, SMLI TR-2003-124, May 2003.

[4] Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., Bryce, C., Resource Management for Clusters of Virtual Machines.

[5] Heiss, J.: The Multi-Tasking Virtual Machine: Building a Highly Scalable JVM. Java developers forum. March 2005.

[6] Czajkowski, G., Daynes, L., Wolczko, Mario., Automated and portable Native Code Isolation. April 2001.

[7] Java Community Process. JSR-284: Resource Consumption Management API, `http://www.jcp.org/en/jsr/detail?id=284`

[8] Java Community Process. JSR-121: The Application Isolation API Specification, Java Community Process, `http://www.jcp.org/en/jsr/detail?id=121`

[9] JBoss Aspect-Oriented Programming home page. `http://www.jboss.org/products/aop`

[10] Sullivan, G., Aspect-Oriented Programming using Reflection and Metaobject Protocols. Arifical Intelligence Laboratory. Massachusetts Institute of Technology. April, 2001.

[11] Sadjadi, S., McKinley, P., Stirewalt, R., Cheng, B., TRAP: Transparent Reflective Aspect Programming. Proceedings of the International Symposium on Distributed Objects and Applications. 2004.