Transparent Real Time Monitoring for Multi-tenant J2EE Applications

Octavian Morariu*, Cristina Morariu, Theodor Borangiu***

** University Politehnica of Bucharest, Dept. of Automation and Applied Informatics, Bucharest, Romania*
e-mail: { octavian.morariu, theodor.borangiu}@cimr.pub.ro
*** CloudTroopers Intl. , Cluj-Napoca, Romania*
e-mail: cristina@cloudtroopers.ro

Abstract: With the emergence of PaaS and SaaS cloud delivery methods, accurate monitoring and metering become an important challenge for the cloud providers, as they assure the input that drives the elasticity of the solution, by dynamically provisioning and de-provisioning resources, and at the same time enable the chargeback for the resources used. This paper presents an approach for real time monitoring of multi-tenant Java J2EE based applications. The main requirements of such a monitoring solution are: low monitoring performance overhead on the JVM, capability to monitor code execution by JVM thread based on the tenant running it and dynamic granularity adjustment at Java method level, enabling relevant reporting of code execution. The solution proposed in this paper combines three methodologies of JVM and J2EE monitoring to achieve a correlated view of the code execution. The first layer is at JVM Tooling Interface by using a native agent that captures the thread execution events and adds dynamic instrumentation in the code of the target application at method level. The second layer is at JAAS level in J2EE where the authentication operations are hooked and linked to the JVM threads. Finally the third layer is at EJB container level where the EJB 3.0 interceptors are used to monitor EJB calls. The real time data from these three layers is consolidated based on the tenant's identity and reported to a metering application. The experimental results are focused on the accuracy of the monitoring solution implemented on Apache Geronimo 3.0.0 application server, using a benchmark application. Finally the paper presents the monitoring performance overhead introduced by the solution proposed measured on SPECjvm2008 benchmark application, focusing on the granularity of monitored data.

Keywords: multitenancy, j2ee, monitoring, shared resources, isolation, ejb, JAAS, JVMTI, multi agent system, ESB, SOA.

1. INTRODUCTION

All cloud computing business models have one common characteristic that makes this new computing model so attractive for customers: the pay-as-you-go model (Armbrust et al., 2010). This is the key aspect because it enables flexibility for customers, allowing them to grow and shrink their IT architecture as dictated by the business needs. For cloud providers, implementing a low granularity, accurate pay-as-you-go model raises a serious challenge on the usage monitoring side (Patel et al., 2009). The monitoring capabilities are vital for this type of business model, as they provide the recorded metrics that are driving the elasticity of the solution and are used to generate the invoices for the customers. Commercial cloud providers today use various metrics for charge-back, starting from a "per user" fee in SaaS environments, CPU and storage usage in PaaS environments and direct resource allocation in IaaS environments (Buyya et al., 2009; Morariu et al., 2012). SaaS and PaaS cloud models assure superior resource utilization by implementing multi-tenancy at some level.

Multi-tenancy represents the operating mode of a software application in which multiple organizations or tenants are using the same application in a shared environment (Tharam et al., 2010). The tenants are isolated from each other at various levels, depending on the multi-tenancy model implemented by the cloud provider. Gardner Inc. (Yefim, 2012) provides a reference architecture for multi-tenancy in cloud computing, identifying seven models for multi-tenancy by dividing the vertical stack in four layers from bottom up: infrastructure layer, data platform, application platform and application logic. The multi-tenancy models identified are: (1) shared nothing, (2) shared hardware, (3) shared OS, (4) shared database, (5) shared container, (6) shared everything and (7) custom multi-tenancy.

1. In shared nothing model each tenant has its own complete stack (Abadi, 2009). The isolation between tenants is in this case complete as each application instance runs on separate hardware. The costs of such an operating model is high for the cloud providers as they still have to maintain the complete systems. However the benefit is on the release management side where the application provider has to support a single version of the application used by all customers. The resource utilization is not improved, being similar to a hosted environment model.

2. The shared hardware model relies on virtualization technologies to obtain better resource utilization (Bezemer et al., 2010). The isolation is realized in this model at the hypervisor level. The complete software stack in this case is

still separate for each tenant. Compared to the shared nothing model the costs are reduced by achieving better resource utilization.

3. The shared OS model implies using a single OS instance to host the application software stacks. From the application perspective, each tenant is still using a separate stack but the isolation is assured by the operating system which provides process execution and memory isolation. The resource utilization is comparable to the shared hardware model, but it has the advantage of eliminating the hypervisor induced performance overhead. Also, from licensing perspective it is cost effective to use a single operating system instance for all tenants.

4. The shared database model is the first model where the application software stack is shared between tenants. This model offers a reduced isolation between tenants compared to the previously mentioned models, but is more cost effective and enables an elastic design of the underlying resource models (Dean and Aulbach 2007).

5. In the shared container model the application platform layer is shared by leveraging a cloud-enabled container for applications and a shared database (Soltesz et al., 2006). This high level sharing implements isolation between tenants at the container level, each tenant using a different application instance deployed in the same container and the same shared data layer. This model provides the best resource utilization as the load can be globally scaled up and down at both middleware layer and database layer.

6. The shared everything model represents the delivery model where there is only one application instance for all the tenants. This is the most effective delivery method for the cloud providers, but is the most restrictive in regards of customizations for the customers.

7. Finally the custom multi-tenancy model is realized by using custom cloud enabled application logic to handle the multi-tenancy. In this model each tenant is using a shared application stack and the isolation is implemented in the application logic (Afkham et al., 2010). This model is very cost effective for the cloud providers as the resource utilization is very high and the maintenance costs are not impacted by the number of tenants.

Regardless of the implementation model chosen, multi-tenancy has a series of challenges that must be handled by the cloud providers. One of the most important challenges is application performance (Li et al., 2008). The application performance is a concern even in classic single tenant application deployments where it would affect only that single tenant; however in a multi-tenant scenario the shared resources model can amplify the effects by propagating the performance degradation to the other tenants depending on the isolation level (Wang et al., 2008). One common technique to assure even performance across all tenants in shared hardware model is to allocate a fixed and similar share of the physical resources to each tenant. This approach however has the disadvantage of limiting the overall physical resource utilization. Another important challenge for multitenant application deployments is related to data security (Takabi et al., 2010). The shared application stack reduces the data isolation and increases the risk of accidental data disclosure. This becomes even more important as tenants are usually competing in the same market area so confidential data disclosure can have a huge negative impact. Data encryption (Subashini and Kavitha, 2011) is a method to prevent such situations but it has a significant impact on the overall application performance.

This paper focuses on describing a mechanism for monitoring multi-tenant single instance applications based on JAAS. The main objective is to monitor CPU time per JVM thread and Heap Space by each tenant using the application. We consider the single instance JAAS-based separation for tenants. The solution proposed has a low overhead (<2%) demonstrated with SPECjvm2008 benchmark. The novel approach consists in the fact that the monitoring solution gathers data from multiple layers (JAAS, JVMTI, and EJB) and consolidates the data around tenants. The solution presented is transparent assuring that no code changes are required in the target application.

2. STATE OF THE ART IN M-T MONITORING

Currently around 30% of the enterprise applications in the world are running on the Java platform using technologies like Enterprise Java Bean (JSR220, 2006) (EJB), Servlets (JSR154, 2007), J2EE (JSR244, 2006), OSGi (OSGi 2009) and others. These technologies are expected to play an important part in the PaaS and SaaS developments in the years to come. However, Java technology itself predates multi-tenant cloud computing and so at this time there is no mechanism available out of the box for implementing such applications (Smith, 2011). Another problem observed in practice is that J2EE application servers usually run a single web application, even if these are designed to host multiple applications. The reason behind this is the lack of isolation between applications and this leads to poor resource utilization. There are three main approaches to multi-tenancy in Java described in the literature as illustrated in Fig. 1.

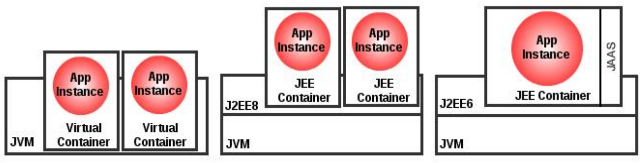

Fig. 1. Approaches to multi-tenancy with Java.

1. Waratek (Waratek Cloud VM) is providing a commercial implementation of the JVM that allows multi-tenancy by application isolation at JVM level. The Waratek Cloud VM introduces a feature called Virtual Containers (VC), which is a meta circular VM within the VM which shares the host VM environment (heap, classes, JIT) with other VCs. A VC is extremely lightweight adding a small overhead, allowing a single VM to host thousands of VCs. CPU priority, memory limits and bandwidth are isolated at VC level. Waratek Cloud VM aims at delivering Java-as-a-Service, allowing execution of existing Java/J2EE platform software as a multitenant

cloud service without code change. This is possible because the

VM claims to be binary compatible with existing applications and platforms. Essentially the isolation allows every .war/.ear application to run in a separate VC. By adding this VC layer at the JVM level some advanced functionality like VC mirroring for disaster-recovery, live snapshot and live migration can be implemented.

2. Java EE 6 Platform was released in December 2009 and has gained a lot of traction within the developers community mainly because of the POJO-based programming model, numerous extension points and Web Profile model. This is reflected also by the large number of application servers that support this technology (13 at this time). However Java EE 6 Platform was not designed with the new cloud introduced requirements in mind. In 2011 Oracle announced Java EE 7 which was planning to include new JSRs that reflect emerging needs in the community and to add support for use in cloud environments. The enhancements targeted directly at cloud deployments of Java EE are focused on PaaS enablement and multi-tenancy support. Despite the initial plans, in August 2012 the cloud related enhancements from Java EE 7 were deferred to the next version Java EE 8, planned for 2015. Even with Java EE 6, there are several cloud providers like IBM, Red Hat, CloudBees (CloudBees AnyCloud, 2012) and Oracle that support deployment of standard Java EE applications in the cloud. IBM Smart Cloud (IBM, 2012) offering allows monitoring of the deployed Java EE applications by providing IBM SmartCloud Application Performance Management tool. This tool is designed to intelligently manage traditional IT, virtualized, cloud and hybrid environments. IBM SmartCloud Application Performance Management can collect metrics like client volume, response time as well as transactions timings in the application server. The application monitoring tool offered by CloudBees is called New Relic and is available to all CloudBees subscribers for free. New Relic (New Relic Application Monitoring Solution, 2012) offers metrics like browser load time, application server response time and time taken by key transactions and various historical usage reports. While these monitoring solutions are enough for the scenario in which a single-tenant Java EE application is deployed in a PaaS environment, they fall short in a multi-tenant Java EE application scenario as it becomes impossible to distinguish between the tenants.

3. Finally, the third class of applications are using J2EE 6 technology and the isolation is enforced by using the Java Security mechanism. This class of applications is achieving multi-tenancy support by enhancing the Java Authentication and Authorization Service (JAAS)-based authorization mechanism to allow tenants access and customize their specific access control lists and privileges in an isolated way. Implementing multi-tenancy using JAAS involves providing authentication services by supporting multiple authentication sources which are tenant specific. Along authentication, the JAAS module handles authorization by maintaining a set of access control lists of users in the context of each tenant organization. A detailed study on how to implement this approach using IBM WebSphere is provided in an IBM devWorks article (Bo et al., 2009).

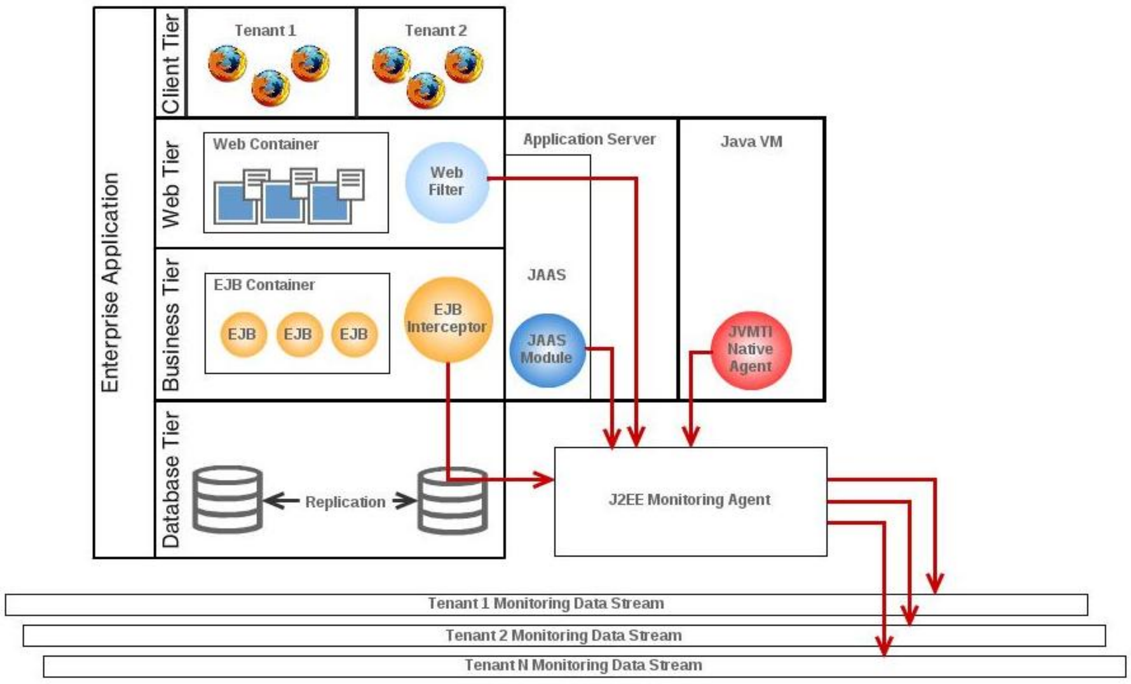

Fig. 2. Monitoring mechanism for a typical J2EE application.

3. A MECHANISM FOR TENANT MONITORING

The Java 2 Platform Enterprise Edition (J2EE) provides a standard for building enterprise multi-tier applications. The current business dynamics have created the need for better larger-scale solutions for information management. The J2EE specification comes as an answer for these challenges by offering a development platform that improves productivity, promotes standards for enterprise applications, and ensures portability.

The J2EE architecture offers component-based development of multi-tier enterprise applications. A J2EE application system typically includes client tier, middle tier and enterprise data tier. In the client tier, Web components such as Servlets and JavaServer Pages (JSPs) or standalone Java applications provide a user interface to the middle tier. In the middle tier, enterprise java beans (EJBs) and Web Services encapsulate reusable and distributable business logic for the application. These middle tier components are running in a J2EE Application Server, which provides the platform for these components to perform actions and store data. In the data tier, the application data is stored usually in a relational database. J2EE applications are comprised of components, containers, and services. Web components are Servlets and JSPs that provide responses to requests from a Web page. EJBs contain server-side business logic implementation for enterprise applications. Web and EJB component containers host services that support Web and EJB modules.

The monitoring mechanism proposed is illustrated in Fig. 2. There are four main components involved: a JAAS login module that links the current thread servicing a tenant with the thread identifier, a EJB interceptor that monitors EJB calls for each thread at the container layer and a JVMTI agent responsible with collecting thread CPU time and allocated heap objects. The information is aggregated and consolidated on tenants inside an external monitoring agent and then reported on real time tenant monitoring data streams. These data streams are consumed by an external reporting application described in (Morariu et al., 2012). The following sections detail the implementation of each module.

3.1 Web Filter

Web Filter component definition was introduced in Java Servlet specification version 2.3. The filter intercepts requests and responses and has full access to the information contained in these requests or responses. Filters are useful for many scenarios where common processing logic can be encapsulated. Historically filters have been used for access management (blocking requests based on user identity), logging and tracking users of a web application, data compression, localization, XSLT transformations of content, encryption, caching, triggering resource related events, mime-type processing and many others. The implementation of a Web Filter is governed by the following interfaces: Filter, FilterChain, and FilterConfig in the javax.servlet package. The actual filter is a implementation of the Filter interface. The filters are invoked in a chained fashion by the servlet container. The Filter interface declares the doFilter method, which contains the actual processing of the request/response objects.

In our implementation, the doFilter method sends a message to the Monitoring Agent containing the resource being requested by the user. The information sent by the Web Filter to the monitoring agent has the following structure:

Table 1. Web Filter message structure.

ThreadID	The ID of the thread in which the web filter is called
Resource URL	URL of the resource requested by the end user
Time	Time taken between request and response

The role of the Web Filter in the overall monitoring solution is to link the JVM thread with the resource URL and the HTTP session requested by the end user. In complex deployments with several nodes (with several JVM instances) that implement session replication, the HTTP session might be linked to several JVM threads in distributed nodes. Capturing the session information across all the nodes enables distributed resource monitoring per session and per tenant.

3.2 EJB Interceptor

The EJB Interceptor module consists in the implementation of a default external interceptor conforming to the EJB 3.0 standard. The interceptor implementation has the intercept(InvocationContext ctx) method with the following structure:

```
@AroundInvoke
public Object intercept(InvocationContext ctx) throws Exception {
    try{
      notifyMonitoringAgent(ctx,Constants.ENTRY);
      //do nothing
      return ctx.proceed();
    }finally{
      notifyMonitoringAgent(ctx,Constants.EXIT);
    }
  }
```

The interceptor is configured as Default interceptor, meaning that it will be called by the EJB container for every EJB method invocation.

The invocation context contains information about the target EJB and the target method in the context of which the interceptor was called. From the thread local data associated with the SecurityContext structure, the current JAAS principal that was authenticated by the EJB container is determined. The time taken to execute the EJB method is computed as the difference between the ENTRY and EXIT times using the System.getCurrentTimeMillis() JVM API. The information sent by the EJB interceptor to the monitoring agent has the following structure:

The registration of the EJB interceptor is done at the EJB container layer, during deployment of the target application, avoiding the need to directly instrument the target application.

Table 2. EJB Interceptor message structure.

ThreadID	The ID of the thread in which the interceptor is invoked
JAAS Principal	Principal object which identifies the user that called the EJB method
EJB Name	Name of the EJB being called
EJB Method	Name of the EJB method being called
Time	Time taken to execute the EJB method

3.3 JAAS Login Module

The Java Authentication and Authorization Service (JAAS) was introduced in Java 2 SDK as an extension and integrated into the Java 2 SDK 1.4. JAAS offers two important services for Java based applications.

- The first service is authentication of users, allowing determining what user is executing the Java class. This applies to all the layers of a Java application, like standalone application, applet, Java Bean, Servlets and so on.
- The second service is authorization of users. Authorization checks are enforced based on access control rights or permissions, whenever a user tries to perform an action.

From an architectural perspective JAAS is an implementation in Java of the standard Pluggable Authentication Module (PAM) framework. Historically Java has used code source based access controls that were based on where the code originated from and who signed the code. This approach was not enough to enforce access controls based on who runs the code.

JAAS is a framework that extends the standard Java security architecture by adding information about the user that executes the code.

JAAS, just as PAM, provides a pluggable architecture allowing applications to remain independent from underlying authentication technologies (LDAP, Kerberos, etc.). The authentication process begins when the client is instantiating a LoginContext object, which in turn references a Configuration to determine the LoginModules to be used in performing the authentication. The JAAS framework calls the login() method on each LoginModule registered. Once all the login modules have authenticated the user the commit() method is called and the Subject object containing all the Principals are returned to the caller. The integration diagram for the monitoring login module developed and standard JAAS implementation is presented in Fig. 3.

The JAAS module consists in an implementation provided for javax.security.auth.spi.LoginModule interface. This implementation always returns a positive authentication response to JAAS, as is designed to monitor the user activity rather than to authenticate the users. The login() and logout() methods are sending a data structure to the monitoring agent, as detailed in Table 3.

Table 3. JAAS Login Module message structure.

Thread ID	The ID of the thread in which the user is authenticating
JAAS Subject	Subject object which identifies the user that performs the action
Operation	Operation (Login, Logout)
Result	Result of the operation
Time	Timestamp when the operation was attempted

The monitoring Login Module captures each login and logout operation in the application server and notifies asynchronously the monitoring agent. This asynchronous approach is done to improve performance on the JAAS module and is implemented using an internal queue where the data structure instances are posted. A push thread is used to send the objects from this queue to the monitoring agent. By capturing the login() and logout() operations, the JAAS module provides the link between the thread ID and the Subject, allowing monitoring data to be consolidated for each tenant.

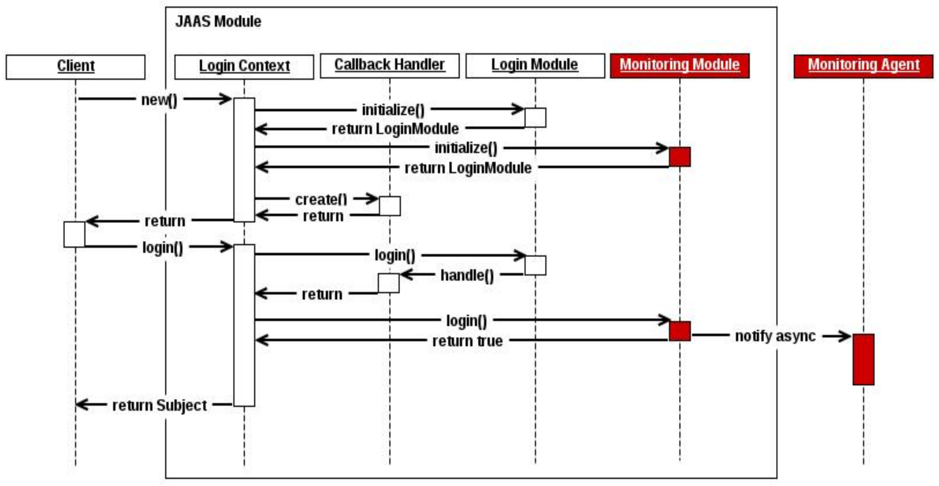

Fig. 3. JAAS Monitoring Module integration.

3.4 JVM TI Native Agent

The JVMTM Tool Interface (JVM TI) is the new generation of native programming interface that allows external tools to access the JVM. It provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). JVM TI is designed to support various tools that need access to JVM state for activities like profiling, debugging, monitoring and thread analysis. JVM TI replaces the Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI) available before 1.5 version. JVM TI is a bi-directional interface. A JVM TI agent can be notified by an event registration mechanism. JVM TI can also control the running application by calling specific functions, either in the context of an event or at give times. A JVM TI Agent runs in the same process as the JVM and the communication with the JVM is through a native interface which allows maximal control with minimal intrusion.

JVM TI Agents are native agents that are implementing using a language that supports C language calling conventions and C or C++ definitions. The function, event, data type, and constant definitions needed for using JVM TI are defined in the include file jvmti.h. On JVM start-up the agent library is loaded. The library must export a start-up function with the following prototype:

*JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options, void *reserved)*

This function is called by the VM when the agent library is loaded, but after all the other native libraries are loaded. Similarly the agent may export a shutdown function with the following prototype:

*JNIEXPORT void JNICALL Agent_OnUnload(JavaVM *vm)*

This function will be called by the VM when the library is about to be unloaded. Agents are notified by the JVM by using events. To handle events, the agent registers a callback function by calling SetEventCallbacks() API. For each event the corresponding callback function will be called by the JVM providing arguments that contain additional information about the event. The callback function is called synchronously by the JVM TI. The agent implemented for the monitoring solution registers callbacks for the events listed in Table 4.

The JVMTI Native monitoring agent collects monitoring data for each thread. The THREAD_START event handler creates a new ThreadData internal structure in the memory storage. This data structure contains initially the thread ID and the time-stamp of the thread creation. The THREAD_END event handler fills in the time-stamp of the thread exit and sends the ThreadData to the external monitoring agent via the Data Monitoring Bus.

The Data Monitoring Bus implements a simple queue mechanism in order to assure temporal decoupling between the Thread End callback function and the external monitoring agent. This queuing mechanism improves performance by allowing return of the control to the JVM without any external delays (Fig. 4).

Table 4. JVM TI Events and callbacks.

JVMTI Event	Callback Handler	Description
JVMTI_EVENT_THREAD_START	void JNICALL MONThreadStart(...)	This method is called by the JVM when a new thread is started in the application. The agent extracts the thread ID from the jthread structure and stores it in the internal memory store.
JVMTI_EVENT_THREAD_END	void JNICALL MONThreadEnd(...)	This method is called by the JVM when a thread dies. The agent extracts the thread ID from the jthread structure.

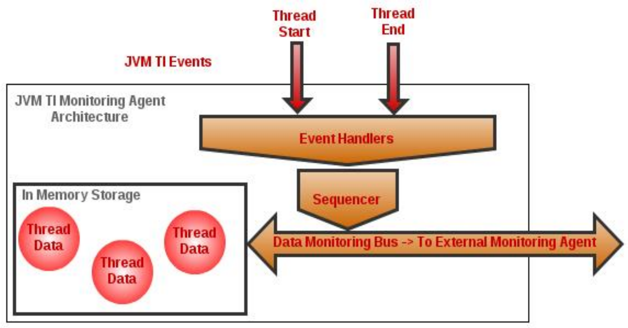

Fig. 4. JVM TI Monitoring Agent Architecture.

3.5 J2EE Monitoring Agent

The J2EE Monitoring Agent is an external application that collects the raw metrics provided by the JVM TI agent, the EJB Interceptor and the JAAS module and consolidates the information in tenant specific monitoring streams. The agent is implemented as a JADE agent (Bellifemine et al., 2001), implementing an incoming message queue and an internal cyclic behaviour. The incoming message queue accepts FIPA INFORM messages (O'Brien et al., 1998) and submission to this queue is asynchronous. The typical sequence of messages for a user resource request is illustrated in Fig. 5.

When a user requests a resource, or in other words the web browser invokes a URL, the first event detected by the JVMTI agent is a Thread Start event. The monitoring agent receives the event and stores in memory the thread ID of the newly created thread. The next event Resource Requested is generated by the Web Filter when the request is handled by the Web Container. As this event is generated from the same thread that was recorded previously, the monitoring agent associates the resource requested with the thread ID stored.

At this point the user will authenticate with JAAS, which generates a JAAS Login event. Now, the monitoring agent adds the user authentication information (obtained from the

JAAS Subject) to the thread servicing the request. The CPU time recorded by the JVMTI agent for this thread is summed and represents the Web Tier CPU usage.

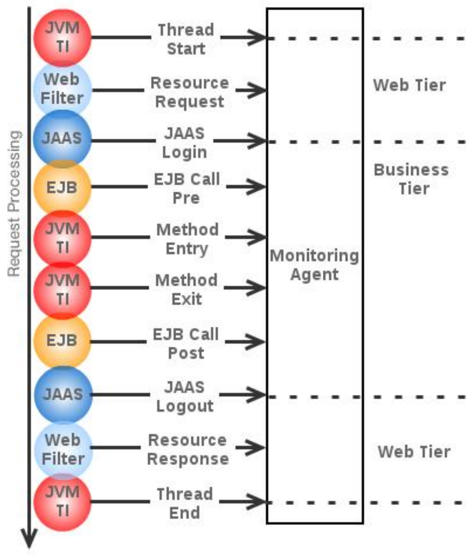

Fig. 5. Typical sequence of messages for a user resource request.

During the JAAS session one or more EJB calls are performed depending on the application design and the operation performed by the user. The EJB calls are monitored using EJB interceptors and generate EJB Call Pre and EJB Call Post events. In between these two events, several Method breakpoint events are generated by the JVMTI agent for the monitored methods. The JVM TI agent sends CPU Time usage information during the Method Entry and Method Exit events which are associated with the JAAS user owning that thread and summed up. After the EJB calls, the next event is JAAS Logout which represents the end of the JAAS session for the user. The request processing is ending with the Resource Response event generated by the Web Filter and with the Thread End event generated by the JVM TI agent, representing the exit of the worker thread. The monitoring agent consolidates the CPU time allocated to the JAAS user which corresponds to the thread ID against the organization of the JAAS user. The organization of the JAAS user represents the tenant.

This approach allows mapping of the CPU time consumed by the JAAS user to the tenant organization and enables low granularity reporting such as: CPU time in Web Tier / tenant, CPU time in Business Tier/ tenant for relevant (pre-configured) methods, total CPU time/ tenant, total CPU time/ resource requested, total CPU time/ EJB call.

4. JVMTI AGENT PERFORMANCE OVERHEAD

To evaluate the performance overhead of the JVM TI agent developed, the SPECjvm2008 benchmark was executed several times and results were consolidated (Table 5). SPECjvm2008 (Java Virtual Machine Benchmark) is a benchmark suite for measuring the performance of a Java Runtime Environment (JRE), containing several real life applications and benchmarks focusing on core java functionality. The suite focuses on the performance of the JRE executing a single application; it reflects the performance of the hardware processor and memory subsystem, but has low dependence on file I/O and includes no network I/O across machines. The SPECjvm2008 workload mimics a variety of common general purpose application computations. These characteristics reflect the intent that this benchmark will be applicable to measuring basic Java performance on a wide variety of both client and server systems (Shiv et al., 2009; Gu et al., 2009).

Table 5. SPECjvm2008 benchmark results and performance overhead of the agent.

Benchmark	Without agent Ops/m	With agent Ops/m	Overhead %
xml	408.17	395.41	**3.13%**
sunflow	78.78	75.48	**4.19%**
serial	161.81	161.41	**0.25%**
scimark.small	259.07	253.93	**1.98%**
scimark.large	43.85	41.11	**6.25%**
mpegaudio	121.78	121.41	**0.30%**
derby	255.67	241.83	**5.41%**
crypto	193.49	191.56	**1.00%**
compress	183.58	180.16	**1.86%**
compiler	338.56	327.58	**3.24%**
Overall	**146.49 Base ops/m**	**144.02 Base ops/m**	**1.68%**

The OS image used for the test is a virtual image running Oracle Linux 5 x64, configured with 4 CPU and 16GB RAM, running on top of IBM CloudBurst 2.1 System x. The underlying hardware is IBM HS22V Blade Server, equipped with two Intel Xeon 5600 processors and 74GB RAM DDR-3. The JVM used Sun Java HotSpot(TM) 64-Bit Server VM 20.6-b01 mixed mode.

It can be observed from the above results that the most significant performance overhead is obtained in the scimark.large tests, due to extensive usage of threads within the benchmark design. However, the overall results show a 1.68% performance overhead (Fig. 6).

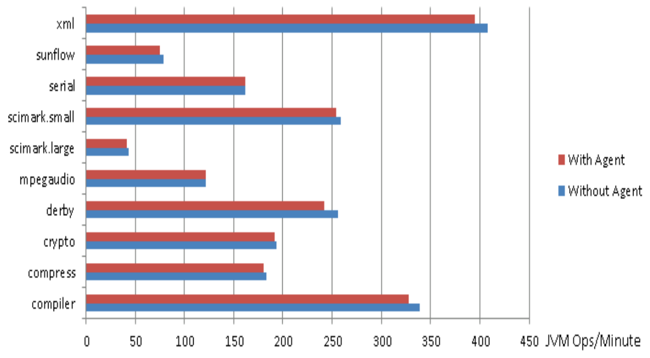

Fig. 6. SPECjvm2008 benchmark results.

5. BENCHMARK APPLICATION AND MONITORING DATA

The benchmark application used for testing the solution is the classic DayTrader application. DayTrader is a benchmark application designed to simulate an online stock trading system. The application was originally developed by IBM for WebSphere and was known as the Trade Performance Benchmark Sample. In 2005, IBM donated the DayTrader application to the Apache Geronimo community. The functionality implemented in the DayTrader application includes user authentication, portfolio management, lookup of stock quotes, buy or sell stock. Using a load generation tools like Apache JMeter, the workload provided by DayTrader can be used to evaluate the performance of Java Enterprise Edition (Java EE) application servers. Additionally the application is designed to offer a set of primitives for functional and performance testing of various Java EE components in the J2EE platform and as well some common design patterns. These characteristics make DayTrader the perfect benchmark application to evaluate the capabilities of the monitoring system described in this paper. The application configuration was modified to integrate with JAAS for user authentication and authorization similar to the concept presented by IBM [25]. The multi-tenant JAAS configuration is based on two LDAP authenticators as illustrated in Fig. 7.

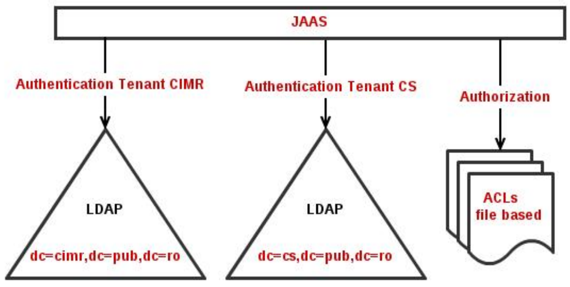

Fig. 7. Multitenant JAAS configuration.

The test environment was built using IBM CloudBurst 2.1 with the following architecture of workloads: 4 nodes running the DayTrader application configured in an Apache Geronimo 3.0.0 cluster configured with session replication. The application server runs embedded Tomcat 7.0.27.1.

The operating system is Solaris 10 x86_64 configured with 8GB RAM, a HTTP Load Balancer based on Apache 2.0 on CentOS Linux 5.5 x86_64 with 4GB RAM,with Round Robin balancing of requests, two LDAP servers running OpenLDAP 2.4.33 on CentOS Linux 5.5 x86_64 with 8GB RAM each (Fig. 8).

The client machine is based on Windows XP SP3 with 4GB RAM, running Apache Jmeter 2.8 software. The user base is divided among tenants, 10 users in CIMR tenant (a Research Lab within University Politehnica of Bucharest - UPB) and 5 users in CS Dept. - UPB tenant. The test scenario involves a repetitive behaviour for each user, which includes login to the application and invocation of the Trade Scenario URL (/daytrader/scenario). This URL generates a random action in the application at each invocation. Users are started with a 5 second delay, starting with the CIMR tenant and alternating after each user. The test involves three stages: the idle stage during start to second 45, the warm up stage during second 46 and second 100 during which all users become active, and the running stage with all users active starting with second 100 and finishing at second 300.

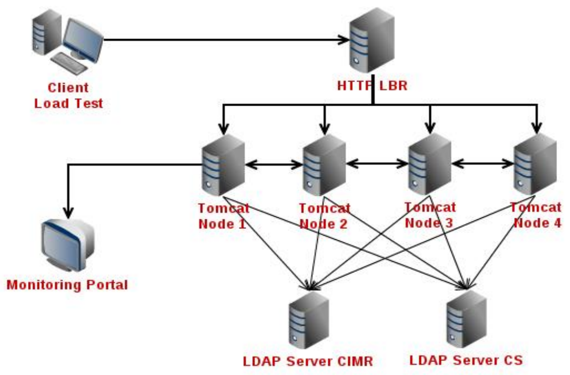

Fig. 8. Test Workload setup in IBM CloudBurst 2.1.

The monitoring data is collected from each of the four Geronimo nodes and averaged for each tenant, due to Round Robin load balancing mechanism. The monitoring streams for each tenant are logged in a CSV file and compared to the total CPU time and memory obtained from JConsole connected to each JVM.

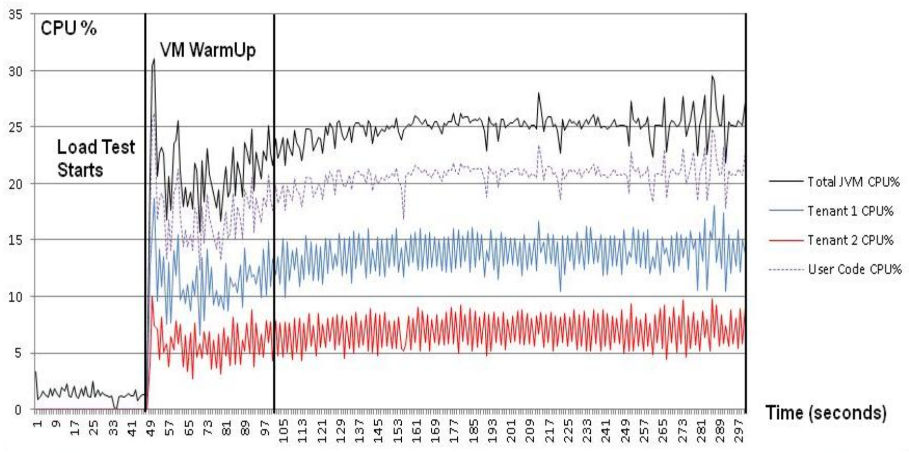

Fig. 9. CPU time monitoring per tenant.

In Fig. 9 the CPU time allocated to the threads summed up for each tenant is shown in blue and red. The dotted line shows the total CPU time for tenants, while the black line shows the total CPU time as indicated by JConsole. The spikes in the graph represent the request processing for each request created by JMeter. The amplitude depends on the operation performed in the DayTrade application when the Trade Scenario URL is invoked. The difference between the black line and the dotted line represents the CPU time of the application server threads that are not in the context of a JAAS user (application server overhead).

Fig. 10 illustrates the heap memory usage for each tenant, against the total heap usage of the JVM. One can see that the usage remains generally low and is directly proportional to the number of users for each tenant. These results are explained by the nature of the load test, which imply multiple user sessions that start and end relatively quick without allocating large objects in the heap. The large spikes seen in the idle stage and in the warm up stage are caused by the garbage collector. Unlike the CPU time which is accurately measured for each thread, the heap memory usage is derived from the EJB calls. Each EJB call has an estimated memory footprint for its execution.

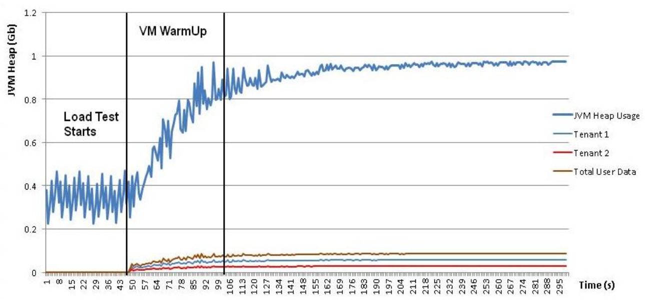

Fig. 10. Heap Memory monitoring per tenant.

This approach was chosen instead of an customized GC approach because of the low overhead on the JVM. A customized GC approach would offer higher accuracy but would require traversing the heap to tag each object accessible only from the given thread and compute the size. The results presented in Fig. 10 use a 4Mb / EJB call estimation for heap allocation in DayTrader application. This estimation was determined by source code analysis of the trade operation implementation considering the default dummy data provided by the application initial configuration.

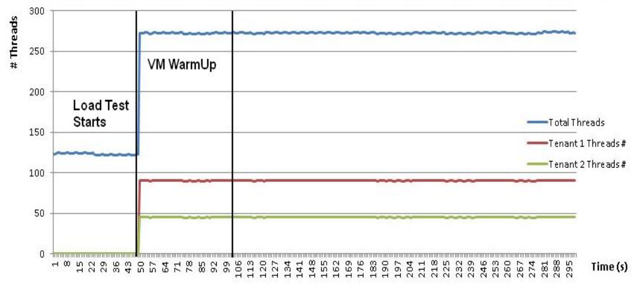

Fig. 11. JVM Threads per tenant.

Finally Fig. 11 shows the number of JVM threads per tenant during the test. As expected the number of threads is dependent on the number of users for each tenant. Also, the application server overhead from a thread perspective remains constant during the load test.

6. CONCLUSIONS

This paper presented the main approaches available today to implement multi-tenant Java applications discussing the particularities associated with real time monitoring of each. Focusing on the JAAS based multi-tenancy solution described by IBM as the most promising approach for adding multi-tenancy support for today's applications; a monitoring mechanism has been presented. This mechanism uses a series of probes at JVM, Servlet Container, EJB container and JAAS layers and allows gathering accurate usage data for each JVM thread. The JAAS custom module links the JVM threads with a user identity and with the tenant organization allowing consolidation of monitoring data per tenant. The JVMTI agent is able to gather CPU time for each JVM thread at a 1.68% performance overhead as indicated by the SpecJVM2008 benchmark.

Real time monitoring per tenant of JVM resources has two important benefits for the provider of multi-tenant enabled applications. First it allows a low granularity charge-back mechanism, enforcing the pay-per-use paradigm, where customers are not charged on a fixed (per user) subscription, but rather on the actual user activity recorded during a period of time. Application providers can define charge-back schemes based on CPU time per tenant and Memory time per tenant. Another benefit is the ability to scale the resources allocated to the application (number of nodes for example) based on real time data and user behaviour, rather than on an historic estimation based on number of active users. The monitoring mechanism presented is capable to distinguish between Web Tier load and Business Tier load, allowing independent scaling of these layers.

One important aspect of code execution monitoring in the context of multi-tenant applications running on J2EE platform is the ability of the solution to distinguish between tenant specific code execution that should be recorded, reported and charged-back, and common utility code executed in the context of a tenant. This common utility code execution should not be assigned to the tenant even if it is executed in the context (thread) of the tenant user. To accomplish this goal future research is focused on enhancing the JVM TI agent described in this paper, to allow definition of tenant code and common utility code distinction by external configuration while keeping the overall performance overhead as low as possible. A dynamic byte-code injection approach is considered in order to replace the event handler approach used currently for method execution monitoring.

REFERENCES

*** (2012). Move beyond monitoring to holistic management of application performance. *IBM SmartCloud Application Performance Management: Actionable insights to minimize issues*, Whitepaper IBM.

*** (2012). New Relic Application Monitoring Solution, http://newrelic.com/product/real-user-monitoring, Available Online.

Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull* 32.1, p. 3-12

Afkham, A. et al. (2010). Multi-tenant SOA middleware for cloud computing. *Proceedings of 3rd International Conference on Cloud Computing* (CLOUD'10).

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A. and M. Zaharia (2010). A view of cloud computing. *Communications of the ACM*, 53(4), p. 50-58.

Bellifemine, F. Poggi, A. and G. Rimassa (2001). JADE: a FIPA2000 compliant agent development environment, *Proceedings of the 5th International Conference on Autonomous Agents*, ACM.

Bezemer, C-P., et al. (2010). Enabling multi-tenancy: An industrial experience report. SM), *Proceedings of IEEE International Conference on Software Maintenance.*

Bo, G., Chang, J.G., Zhi, H.W., Wen, H.A. and W. Sun (2009). Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 4: Design patterns for sharing resources in single instance multi-tenant applications, *IBM developerWorks*.

Buyya, R., Yeo, C., Venugopal, S., Broberg, J. and I. Brandic (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25(6), p. 599-616.

CloudBees AnyCloud: Business Value, Architecture and Technology (2012). *Whitepaper CloudBees Inc*., Available Online.

Dean, J. and S. Aulbach (2007). Ruminations on multi-tenant databases. *BTW Proceedings* 103, p. 514-521.

Gu, X. et al. (2009). Virtual reuse distance analysis of SPECjvm2008 data locality. *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ACM.

JSR154 (September 2007). Java Specification Request 154: *Java Servlet 2.5 Specification*, Available Online.

JSR220 (May 2006). Java Specification Request 220: *Enterprise Java Beans 3.0*, Available Online.

JSR244 (May 2006). Java Specification Request 244: Java Platform, *Enterprise Edition 5* (Java EE Specification), Available Online.

Li, X.H., Liu, T., Li, Y. and Y. Chen (2008). SPIN: Service performance isolation infrastructure in multi-tenancy environment. In *Proc. Int. Conf. on Service-Oriented Computing* (ICSOC), vol. 5364 of Lecture Notes in Computer Science, Springer, p. 649-663.

Morariu, O. at al. (2012). Resource Monitoring in Cloud Platforms with Tivoli Service Automation Management. *Proceedings of INCOM'12, IFAC PapersOnLine*, Vol. 14. No. 1.

O'Brien, P.D. and R.C. Nicol (1998). FIPA - towards a standard for software agents. *BT Technology Journal 16.3*, p. 51-59.

OSGi (June 2009). OSGi Service Platform Core Specification, Available Online.

Patel, P., Ranabahu, A. and A. Sheth (2009). Service Level Agreement in cloud computing. *Cloud Workshops at OOPSLA*.

Shiv, K. et al. (2009). SPECjvm2008 performance characterization, *Computer Performance Evaluation and Benchmarking*, p.17-35.

Smith, D.M. (2011). Hype cycle for cloud computing. *G00214915 Gartner*.

Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A. and L. Peterson (2006). Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: *Proceedings of EuroSys'06 Conference*.

Subashini, S. and V. Kavitha (2011). A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications* 34.1, p. 1-11.

Takabi, H., Joshi, J.B.D. and G.-J. Ahn (2010). Security and privacy challenges in cloud computing environments, *Security & Privacy*, IEEE 8.6, p. 24-31.

Tharam, D., Wu, C. and E. Chang (2010). Cloud computing: Issues and challenges., *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications* (AINA).

Wang, Z. H. et al. (2008). A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. *Proceedings of IEEE International Conference on e-Business Engineering*.

Waratek Cloud VM, http://www.waratek.com/resources/whitepapers/technical-architecture-overview, *Technical Architecture Overview, White Paper*, Available Online.

Yefim, N. (2012). *Gartner Reference Model for Elasticity and Multi-tenancy*, Gartner Inc.