Memory Monitoring in a Multi-tenant OSGi Execution Environment

Koutheir Attouchi
Orange Labs.
Grenoble, France.
koutheir@gmail.com

Gaël Thomas
Inria & LIP6.
Paris, France.
gael.thomas@lip6.fr

André Bottaro
Orange Labs.
Grenoble, France.
andre.bottaro@orange.com

Gilles Muller
Inria & LIP6.
Paris, France.
gilles.muller@lip6.fr

ABSTRACT

Smart Home market players aim to deploy *component-based* and *service-oriented* applications from *untrusted* third party providers on a single OSGi execution environment. This creates the risk of *resource abuse* by buggy and malicious applications, which raises the need for resource monitoring mechanisms. Existing resource monitoring solutions either are too intrusive or fail to identify the relevant resource consumer in numerous multi-tenant situations. This paper proposes a system to monitor the *memory* consumed by each tenant, while allowing them to continue communicating directly to render services. We propose a solution based on a list of configurable *resource accounting rules* between tenants, which is far less intrusive than existing OSGi monitoring systems. We modified an experimental Java Virtual Machine in order to provide the memory monitoring features for the multi-tenant OSGi environment. Our evaluation of the memory monitoring mechanism on the DaCapo benchmarks shows an overhead below 46%.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.2.11 [**Computer Systems Organization**]: Performances of systems—*Measurement techniques, Reliability, and availability*

Keywords

memory monitoring; multitenancy; software platform; smart home; home gateway; OSGi technology

CBSE'14, June 30–July 4, 2014, Marcq-en-Baroeul, France.

ACM 978-1-4503-2577-6/14/06 ...$15.00.
http://dx.doi.org/10.1145/2602458.2602467.

1. INTRODUCTION

A new world of smart home services emerges thanks to the growing variety of sensors and actuators available. Many application domains are involved, e.g., security, energy efficiency, ambient assisted living, multimedia, communication. Consequently, different software editors, called here *tenants*, are developing applications that take advantage of the multitude of devices in the smart home. The smart home ecosystem, as conceived by the Home Gateway Initiative [6], is based on OSGi [14] and Java, in an effort to support the openness of a multi-tenant execution environment hosting component-based applications collaborating to render services. Figure 1 illustrates this component-based and service-oriented architecture [15]. Java and OSGi provide sharing and isolation mechanisms between OSGi components called *bundles*, which are also OSGi deployment units. Components interact via service method calls to each other exported interfaces. Different tenants deploy their components on the same execution environment. This sharing by untrusted competing tenants raises the need to "protect the box against badly written bundles" [16, 4]. Here, a *badly written* bundle is a component that consumes resources, such as CPU usage and memory and network, far above a normal expected level. Therefore, mechanisms that regulate resource consumption at component granularity are necessary, and in particular, *resource monitoring* mechanisms.

Resource monitoring provides essentially two information: (1) *counting*, i.e., how much of the resource is consumed, and (2) *accounting*, i.e., which entity should be charged for using that quantity of the resource. Existing work performed in component-based and service-oriented platforms [11, 3, 10, 8, 4] enable resource accounting at component level in only two cases: (1) when a component calls a service method implemented by a component belonging to the operator platform, or (2) when the opposite happens. Still, there is a third case that occurs when a component from a given tenant calls a service method implemented by a component belonging to another tenant. None of the cited existing work provide a correct way to account for resources consumed during the third case, because such an accounting requires information related to the *business logic* implemented in the components of the tenants. The first approach adopted to deal with the third case is avoiding it entirely, by forbidding interaction between tenants, which is too restrictive, as tenants need to communicate to render integrated services to the

end-user. The second solution requires explicit declaration of the component that is accounted for resources consumed during each interaction via a defined API [2]. However, this API cannot be granted to *untrusted* components, because a buggy or malicious component can use this API to declare resource consumption to the wrong entity, causing resource miscounting. A first challenge is implementing a component-level monitoring system that accounts for resources consumed during interactions between tenants.

A second challenge in designing a monitoring system in a multi-tenant environment is the expression of resource accounting rules during interactions between monitored tenants. Declarations of accounting rules should be *precise* and *expressive* enough for all cases, without requiring developers to write and maintain long rules lists.

A third challenge is implementing a *sufficiently accurate* monitoring system with a minimal performance overhead. Previous methods are not accurate enough, mainly because accounting is either (1) *always direct*, i.e., resources consumed during a method call are always accounted to the called entity, or (2) *always indirect*, i.e., resources consumed during a method call are always accounted to the caller entity. This inaccuracy is more *frequent* and more *critical* when untrusted tenants interact via method calls in multi-tenant environments, which are often service-oriented and component-based. For example, Miettinen et al. [11] perform *indirect* monitoring by attaching every thread to a component and accounting resources consumed by that thread to the attached component no matter what code the thread executes. This method produces inaccurate monitoring statistics in some cases, e.g., components managing a thread pool, and components that notify about events by calling event handlers.

In this paper we describe an *OSGi-aware memory monitoring system* that is mostly transparent to application developers, and that allows collaboration between distinct tenants sharing the same OSGi execution environment, therefore preserving the service-oriented nature of OSGi. The system monitors interactions between tenants and provides snapshots of memory usage statistics on demand.

The monitoring subsystem has predefined *implicit* resource accounting rules that describe correctly most interactions between tenants. We purposely focused on the types of interactions that are very common in component-based and service-oriented design, e.g., calls to service methods and event handlers. Because implicit rules cannot accurately account for all cases, the platform operator and each tenant provide configuration files loaded by the monitoring system at start-up, and containing *explicit* rules for resource accounting between tenants. The configuration files are written in a *Domain-Specific Language* (DSL) that we defined, which is tailored to service-oriented applications, as it allows, for example, to specify the *component* implementing the *interface* containing the method called during an interaction. Our prototype requires the resource accounting rules to remain constant during the lifetime of the JVM. Each configuration file lists (1) the components belonging to a specific tenant, and (2) the resource accounting rules, each of which describes which tenant is accounted for resources consumed during a specific interaction. At runtime, the monitoring system applies those rules to correctly account for memory used by components, e.g., local variables, loaded classes, and created objects. In most of the cases, component developers do not need to write accounting rules because implicit rules handle their interactions correctly. Components that need to write explicit accounting rules are mainly those that generate asynchronous activity, such as those that publish events. Examples include the OSGi framework, and components exposing data of home sensors in real time.

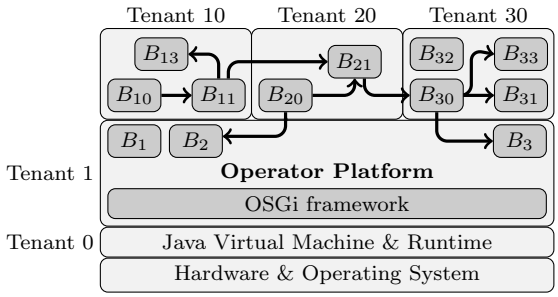

Figure 1: A multi-tenant OSGi execution environment. $B_1 ... B_{33}$ are OSGi components, i.e., bundles.

We implemented the monitoring system inside J3, a Java Virtual Machine based on VMKit [5], LLVM [9] and MMTk [1]. Even though we slightly change the native object structure, the changes are invisible to the Java code, which helps preserve the component model of OSGi. We validate our implementation of the monitoring system using a set of micro-tests that mimics common communication patterns in a component-based and service-oriented micro-structure composed of seven tenants. We evaluate the overhead generated by the system based on the real life applications of the DaCapo benchmark, in addition to highly focused micro-benchmarks. The performance overhead of monitoring is always below 46% for DaCapo applications, which is still reasonable for development and testing environment, and tolerable in low-pace production environments.

This paper is organized as follows. We begin by discussing the design of the monitoring mechanism in Section 2. In Section 3, we illustrate our prototype implementation of the design. Section 4 measures the overhead of monitoring and reveals the parts of the system that cause most of it. Previous work related to resource management are described in Section 5. Finally, Section 6 concludes the paper and discusses future directions.

2. DESIGN

In this section, we present the design of a memory monitoring subsystem that is intended to be a part of a more complete system for resource management inside a long running JVM running OSGi. The memory usage information reported by this subsystem would indicate memory leaks and too high memory usage patterns.

Motivating example 1: Service method.

In practice, a tenant would often require the caller of the service it provides to be accounted for resources consumed by that service. For example, a tenant providing a service `playRingTone()` would require the tenant calling that service to be accounted for memory consumed to play the ring tone. No rules should be needed in order to fulfill this accounting, i.e., it should be the default behavior of resource accounting.

Motivating example 2: Event handlers.

Less often, a tenant A provides an interface to notify observers about an event. Another tenant B subscribes to the event and expects to be called when the event occurs. When the event occurs, A calls B as contracted. In this case, implicit rules (see the step `3(d)` in Section 2.4) specify that A should be accounted for resources consumed during the call. However, this is unfair given the fact that A calls B only because B asked for that by subscribing to the event. In this case, A needs to write an accounting rule for the notification interface, accounting resources for the called entity (see Section 2.3).

2.1 Assumptions

We assume a number of preconditions on the system where our monitoring subsystem runs.

No need for isolation.

We do not suppose any form of isolation beyond what is provided by OSGi. Therefore, all tenants are able to communicate via service method calls. This enables different tenants to collaborate to create integrated services and user experience, even when the user is using sensors and actuators and applications from different manufacturers and editors.

Constant monitoring, infrequent reporting.

It is typical for a long-running system to have a *resource manager* subsystem that periodically requests memory usage statistics from a memory monitoring subsystem. In the smart home gateway, memory usage statistics would be requested once or twice a day in relatively stable configurations, and the period can be as frequent as every hour when hardware or software configurations change. The memory usage statistics enable detecting *abnormal* memory usage situations, e.g., memory leaks. When such abnormal activity is detected, the resource manager subsystem carries out actions to restore the system to a normal state by making memory available again, such as terminating some applications. The resource manager subsystem can be human driven or autonomous.

Our memory monitoring subsystem runs *continuously*, collecting *raw* memory usage data on running applications. This generates a *persistent* overhead that must be kept to a minimum. Furthermore, in order to report relevant memory statistics, raw data need to be *aggregated* and *filtered*, generating an overhead every time a memory monitoring report is requested. We target systems where memory monitoring reports will be requested *sparingly* in time, in order to check for abnormal resource usage. This is the case of the smart home gateway, where memory monitoring reports would be generated from once per hour to once per day. Therefore, we tolerate the aggregation and filtering overhead needed to generate memory monitoring reports, and we rather focus on the persistent overhead caused by continuous monitoring.

Constant resource accounting configuration.

In order to simplify our prototype and keep performance overhead acceptable, we require that resource accounting rules and tenants list remains *constant* during the JVM lifetime. This allows performing early calculations in order to accelerate inference of accounting rules.

2.2 Goals

We designed our monitoring subsystem in order to fulfill the following goals.

Detailed memory monitoring.

Our prototype monitors memory usage of every tenant in *call stack* space and *heap* space. Call stack space is where most methods variables and parameters are stored. The subsystem reports the number of bytes accounted to each tenant, in the call stacks of all running threads. Heap memory is used to store Java classes and objects, particularly their static fields and object fields. The subsystem reports the number of classes loaded by each tenant and the number of bytes used by those classes. It also reports the number of reachable objects that are accounted to each tenant, in addition to the number of bytes used by those objects.

Expressive resource accounting rules.

The tenants and the platform operator are required to provide accounting rules that describe how accounting should occur when two tenants interact. We designed a Domain-Specific Language (DSL) to enable expression of resource accounting rules.

Only specify special cases.

We want our subsystem to require the smallest configuration possible. The default configuration should work well for most of the cases, and developers and platform administrators should configure and maintain only the special cases of interactions. For this, we armed our subsystem with a list of implicit rules to handle most cases correctly. We also defined a resource accounting algorithm that decides which tenants to account for resources, so that:

- Explicit rules always override implicit rules.
- The order of processing rules is from the most specific, to the most generic.

2.3 Domain-specific language for resource accounting rules

We defined a DSL that enables developers and platform administrators to specify the rules to decide which tenant should be accounted for resources consumed during an interaction. Illustrated in Figure 2, the DSL allows rules of varying levels of precision, which allows factoring the rules, thus writing and maintaining less of them. It also allows correct handling of all possible cases of component interactions.

The DSL describes separately two aspects: a list of tenants, then a list of rules. Each element in the tenants list describes the identifier assigned to a tenant, and the names of components it deploys. Each element in the rules list describes a method call between two tenants, and which tenant should be accounted for resources consumed during that call. We reserve `0` as the tenant ID of the Java runtime and the JVM native code, and we reserve `1` as the tenant ID of the platform operator (see Figure 1). A rule starts with the caller tenant ID (an integer), which can be `*` to indicate that any caller tenant matches this rule. Then, the called site is specified, followed by the accounting decision. The called site is (1) a method of (2) an interface defined in (3) a tenant and implemented by (4) a component. The four components are optional, and omitting a component matches

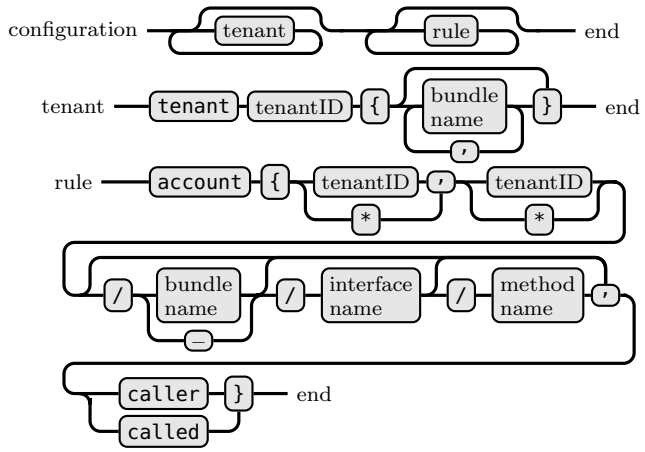

Figure 2: DSL of resource accounting configuration

```
tenant   1 { org.knopflerfish.framework }
tenant  20 { tests.A                    }
tenant 200 { j3mgr                      }

account { 0, 0/_/java.lang.Runnable/run, called }
account { *, 200/j3mgr/j3.J3Mgr,         caller }
account { *, 20/tests.A/tests.A.A,       caller }
```

Figure 3: Sample resource accounting configuration

all possible values, e.g., specifying "_" as the implementation component name matches any components implementing the specified interface or method. Finally, the accounting decision specifies which tenant is accounted for resources consumed during the call. Note that when the called tenant should be accounted for resources, it is the tenant holding the implementation of the called method that is accounted, not the tenant defining the interface.

Figure 3 shows an example of resource accounting configuration. It first associates components with tenants, then it declares rules, each of which specifies which tenant is accounted for resources consumed during a given method call. So, first it declares the platform operator as `tenant 1`, holding the main `framework` component of the Knopflerfish OSGi implementation. It declares two other tenants whose identifiers are 20 and 200, with one component for each tenant. Then it declares that a call from the Java runtime to the method `run` of the `java.lang.Runnable` interface implemented by any component will be accounted to the tenant of the called component, i.e., the tenant of implementation component. Next, a call from any tenant to a method of the interface `j3.J3Mgr` implemented in the component `j3mgr` of the tenant 200 is accounted to the tenant of the caller. Finally, a call from any tenant to a method of the interface `tests.A.A` implemented in the component `tests.A` of the tenant 20 is accounted to the tenant of the caller.

Back to the *Motivating example 2: Event handlers* previously described, all what the developer needs to write is the following accounting rule:

```
account(42, 42/_/notify.event/happened, called)
```

This rule specifies that a call from the tenant 42 (the tenant ID of A) to the method `happened()` of the interface `notify.event` defined in the tenant 42 (in A) and implemented by any tenant (the _) is accounted to the called tenant implementing the interface, i.e., the tenant subscribed to the event published by A.

2.4 Resource accounting algorithm

This section describes the resource accounting algorithm that decides which tenant is accounted for resources consumed during an interaction.

Algorithm.

The list of tenants described in Section 2.3 is stored in the following data structure associating each tenant ID with the list of components it is responsible of:

$$map = \{..., (\text{TenantID}_i \rightarrow \{..., \text{ComponentName}_j, ...\}), ...\}$$

The list of resource accounting rules is stored in the following data structure associating accounting rules with decisions:

$$\begin{aligned} map =& \{..., (\text{CallConfig}_i \rightarrow \text{caller}|\text{called}), ...\}, \text{where:} \\ \text{CallConfig}_i =& ([\text{CallerTenantID}], \text{CalledSite}), \text{where:} \\ \text{CalledSite} =& (\text{TenantID}, [\text{ComponentName}], \\ & [\text{InterfaceName}], [\text{MethodName}]) \\ & \text{where: } [x] \text{ means } x \text{ is optional} \end{aligned}$$

Given a call configuration λ_i (a.k.a. CallConfig_i in the map expression above) as an input, the algorithm proceeds as follows:

1. The accounting rules map is searched for an exact match for the key λ_i. If that is found, then its associated value indicates explicitly which tenant is accounted for the resource usage, and the process ends.

2. If λ_i is totally generic, i.e., if its method name and interface name and component name are all missing, then continue to 3, otherwise, continue to 4.

3. No rules are defined for this interaction. Apply the following *implicit rules*:

   (a) If the call is an internal operator platform call, then account resource usage to the caller, i.e., the platform operator.

   (b) If the platform operator is calling a tenant, then account resource usage to the called entity, i.e., the tenant.

   (c) If a tenant is calling the platform operator, then account resource usage to the caller, i.e., the tenant.

   (d) Otherwise, account resource usage to the caller.

   And the process ends here.

4. Make λ_i more generic, i.e., remove one non-missing piece on information from it, in the following order: method name, then interface name, then component name.

   Loop to 1.

The illustrated decision algorithm ensures that:

- Accounting rules order is unimportant, i.e., rules are always matched from the most specific, to the most generic.

- Implicit accounting rules account the platform operator only when the interaction is an internal operator platform call.

- Implicit accounting rules between two tenants accounts the caller.

3. IMPLEMENTATION

This section describes our implementation of the design we promote in the previous section. We divided the monitoring subsystem into three major components that run inside the JVM:

- *OSGi state tracker* that acts as a bridge between the JVM subsystems and the OSGi framework.

- *Accounting configuration manager* that parses accounting rules and performs accounting decisions.

- *Monitoring manager* that generates, on-demand, snapshots of detailed memory statistics.

The monitoring subsystem implementation is around 2000 lines of C++ code mostly inside the J3 JVM based on VMKit [5], LLVM [9] and MMTk [1]. The OSGi framework used is Knopflerfish 5.0.

3.1 OSGi state tracker

In resource accounting rules, each component is identified by its name. But, at runtime, each OSGi component instance is identified by a unique framework-assigned ID that is not reused even if that instance is uninstalled. The link between the component name and the component ID enables the monitoring subsystem to know which rule to apply when two components interact. The OSGi state tracker component makes the links between the static component information (e.g., component names) given in the resource accounting rules and the dynamic states of OSGi components at runtime (e.g., component ID, component class loaders). It tracks the OSGi states (e.g., `resolved`, `uninstalled`) of components installed in the OSGi framework running on top of the JVM, by listening to events of component state changes from the JVM and from the OSGi framework.

This component associates component identifiers with their respective component information. The component information includes the component name, and its current and previous class loaders. The association map is expressed as:

$$map = \{..., (\text{CompID}_i \rightarrow \text{CompInfo}_i), ...\}, \text{where:}$$
$$\text{CompInfo}_i = (\text{CompName}, \{..., \text{ClassLoader}_j, ...\})$$

In order to discover component state changes, this component places two hooks in the Knopflerfish 5.0 OSGi framework, which makes it dependent on that particular OSGi implementation. However, those hooks need no more than 10 lines of code inserted into the framework code. Therefore, the dependency is fairly limited.

This component encapsulates all the logic necessary to interact with the OSGi framework. Therefore, porting the monitoring subsystem to another OSGi framework implementation only requires porting this component. This makes the major part of the subsystem independent of any particular OSGi framework implementation.

3.2 Accounting configuration manager

When the JVM starts up, the accounting configuration manager component loads the resource accounting configuration, before loading the OSGi framework code. The configuration is stored in memory in the data structures described in Section 2.4, and it remains constant during the execution of the JVM. We implemented a parser inside this component, in order to load the configuration from any text file (e.g., disk files, names pipes, sockets) that conforms to the DSL described in Section 2.3.

At runtime, each time a component calls a method via direct invocation (i.e., `invoke*`) or object construction (i.e., `new`), this component decides which tenant should be accounted for resources consumed during that call. The decision is based on the algorithm described in Section 2.4, which takes into account the resource accounting configuration, default accounting rules, and runtime information. Some runtime information, e.g., states of components, is provided by the OSGi state tracker component previously described. The accounting decision concerns solely resources consumed in the thread running the called method, and it remains effective until another method is called in that thread.

For each Java thread, we set a thread-local variable γ_i holding the thread's *currently accounted tenant ID*. Initially, γ_i is set to `0`: the special tenant ID reserved to the Java runtime. Before every method call, the resource accounting algorithm decides which tenant is accounted for the resources consumed during the call. If the algorithm decides that the called tenant shall be accounted for resources, then γ_i is set to the tenant ID of the called method. Otherwise, γ_i remains unchanged. In all cases, the tenant identified by γ_i is accounted for resources consumed by the thread. Most changes needed to track method calls and to invoke the decision algorithm are implemented in the Just-In-Time (JIT) subsystem of the JVM.

The monitoring subsystem adds, in every Java object, a hidden field that holds the tenant ID accounted for the object. Every time the JVM creates a new object, i.e., executes a `new` instruction, it reads γ_i to determine which tenant is accounted for the newly created object. The object is consequently tagged with the tenant ID set in of γ_i.

This mode of operation regarding γ_i ensures that resources are accounted to the caller tenant, unless otherwise specified by an implicit or declared accounting rule. The notion of the caller tenant is transitive. Consider the example call sequence:

$$S_1 = ... \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M_4 \rightarrow ...$$

where M_i is a method belonging to a tenant T_i, and arrows indicate method calls. In each method call, before entering the called method M_i, the decision algorithm makes the decision D_i, which determines the new value of γ_i. An example of decisions follows:

$D_1 = ... \rightarrow M_1$: the **called** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_2 = M_1 \rightarrow M_2$: the **caller** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_3 = M_2 \rightarrow M_3$: the **caller** tenant is accounted. $\Rightarrow \gamma_i = T_1$
$D_4 = M_3 \rightarrow M_4$: the **called** tenant is accounted. $\Rightarrow \gamma_i = T_4$

This example implies that resources consumed during the execution of M_1, M_2 and M_3 are all accounted to T_1, whereas resources consumed in M_4 are accounted to T_4.

```
tenant 1  { org.knopflerfish.framework }
tenant 50 { tests.A }
tenant 60 { tests.B }
tenant 70 { tests.C }
tenant 80 { tests.D }
tenant 90 { tests.E }
```

Figure 4: Tenants declarations in functional tests.

It is worth noticing that γ_i is restored to its previous value in any scenario that makes the method return to one of its callers, e.g., when a method returns, or when an exception is caught outside the method where it is thrown. The previous values of γ_i are stored in the call stack, as hidden local variables.

This frequent execution of the decision algorithm in every method call is the primary source of the permanent overhead added to the normal execution of Java code.

3.3 Monitoring manager

The monitoring manager component generates, on-demand, snapshots of detailed memory statistics. In order to do so, this component triggers a special garbage collection cycle, during which it scans the object graph and call stacks of running threads, while accumulating statistical counters. This scan is performed by placing hooks in the garbage collector code which scans objects and threads. The counters accumulate the following information, grouped by tenants (see Section 2.2):

- Number of reachable objects and their size.
- Number of loaded classes and their size.
- Amount of used stack space.

It is worth noting that the modifications performed on the garbage collector do not depend on the algorithm used for collection, and do not modify the accessed objects and classes and threads. We only depend on the fact that the garbage collector can perform a collection cycle during which it scans the whole objects graph, and during which all threads are suspended. Otherwise, this component is independent of the garbage collector implementation.

4. EVALUATION

The benchmarks were executed on a computer running the 32-bits version of the Linux 3.12 kernel, on an Intel Xeon CPU running at 2.7 GHz, with 12 megabytes of cache, 12 gigabytes of RAM[1] and 1 terabyte of disk space.

4.1 Functional tests

In order to ensure that the monitoring subsystem works as intended, we performed functional unit tests. To verify correct accounting, we ask the monitoring system to output detailed accounting calculations.

4.1.1 Operator platform internal accounting

A part of the detailed accounting calculation is shown in Figure 5, consisting of a thread call stack prefixed with monitoring information, i.e., `T`: identifier of the tenant that is accounted for resources consumed during the method frame, and `Size`: the total size, in bytes, of the method frame. For example, in Figure 5, 1456 bytes of stack space are accounted to the Java runtime (tenant 0), and 1120 bytes of stack space are accounted to the operator platform (tenant 1). The package `java.lang` belongs to the Java runtime. The line `1` in Figure 4 states that the package `org.knopflerfish.framework` belongs to the operator platform.

```
T Size MethodFrameName
1 480  java.lang.VMObject.wait
1 128  java.lang.Object.wait
1 240  org.knopflerfish.framework.Queue.removeWait
1 272  org.knopflerfish.framework.StartLevelController.run
0 96   java.lang.Thread.run
0 304  java.lang.VMThread.run
0 1056 <native>
```

Legend:
`T`: Currently accounted tenant ID, i.e., γ_i.
`Size`: Stack size, in bytes, of the method frame.

Figure 5: Detailed memory accounting calculations on a call stack of a thread internal to the OSGi framework.

The choice of which tenant is accounted for a particular method frame is based on the algorithm described in Section 2.4. As specified in Section 3.2, γ_i denotes the *currently accounted tenant ID*. In fact, γ_i corresponds to `T` in Figures 5 and 6. Initially, $\gamma_i \leftarrow 0$: the tenant ID of the Java runtime. If the algorithm decides that the *called* tenant shall be accounted for resources, then γ_i is set to the tenant ID of the called method. Otherwise, γ_i remains unchanged. In all cases, the tenant identified by γ_i is accounted for resources consumed by the method frame.

In Figure 5, the `<native>` frame is the first code to run in the thread, consisting of operating system thread initialization routine and JVM routines that prepares for Java code execution. Initially, the native frame is accounted to the Java runtime, i.e., the tenant 0 ($\gamma_i \leftarrow 0$). The native code calls the `java.lang.VMThread.run()` method implemented in Java runtime, which makes it an internal Java runtime call. Step 3(a) in the algorithm described in Section 2.4 accounts the call to the caller, i.e., tenant 0 ($\gamma_i = 0$) which is the Java runtime. The same accounting goes for the next method frame.

Next, the Java runtime calls the method `run()` of the class `org.knopflerfish.framework.StartLevelController` defined in the operator platform (tenant 1), and implementing the interface `java.lang.Runnable` defined in the Java runtime (tenant 0). This call matches the predefined explicit rule: `account {0, 0/_/java.lang.Runnable/run, called}`. This is why the called, i.e., the operator platform (tenant 1) is accounted for the call, and $\gamma_i \leftarrow 1$. The next call is internal to the tenant 1, so the caller, i.e., tenant 1 ($\gamma_i = 1$) is accounted for it.

Later, the operator platform calls the Java runtime method `java.lang.Object.wait()`. The step 3(c) in the algorithm implies that the caller, i.e., tenant 1 ($\gamma_i = 1$), is accounted for the call. The rest of the calls are internal Java runtime calls, for which the caller tenant, i.e., the tenant 1 ($\gamma_i = 1$) is accounted for the resources consumed during the calls.

4.1.2 Implicit and explicit accounting

To test monitoring of other types of interactions between different tenants, we define five components `tests.A` through `tests.E`, each belonging to a tenant, as declared in Figure 4

[1]The kernel allowed addressing the whole RAM via Physical Address Extensions.

```
T  Size MethodFrameName
50 480  java.lang.VMObject.wait
50 64   java.lang.Object.wait
50 256  java.lang.VMThread.sleep
50 112  java.lang.Thread.sleep
50 224  tests.D.D.sleep
50 160  tests.D.D.handler
50 144  tests.B.C.someProcessing
50 256  tests.B.C.e
50 144  tests.A.Activator.heavyInitialization
50 128  tests.A.Activator.start
1  384  org.knopflerfish.framework.Bundle.start0
1  272  org.knopflerfish.framework.BundleThread.run
0  272  java.lang.VMThread.run
0  1056 <native>

Tenant Stack Objects Classes HeapStatic HeapVirtual
0      14816 7964    129     1600       167336
1      4176  35908   154     1597       1107636
50     1968  55525   16      313        666500
60     0     15      13      104        316
80     0     19      14      104        384
```

(a) Only implicit accounting used.

```
T  Size MethodFrameName
80 480  java.lang.VMObject.wait
80 64   java.lang.Object.wait
80 256  java.lang.VMThread.sleep
80 112  java.lang.Thread.sleep
80 224  tests.D.D.sleep
80 160  tests.D.D.handler
50 144  tests.B.C.someProcessing
50 256  tests.B.C.e
50 144  tests.A.Activator.heavyInitialization
50 128  tests.A.Activator.start
1  384  org.knopflerfish.framework.Bundle.start0
1  272  org.knopflerfish.framework.BundleThread.run
0  272  java.lang.VMThread.run
0  1056 <native>

Tenant Stack Objects Classes HeapStatic HeapVirtual
0      14816 7964    129     1600       167336
1      4176  35906   154     1597       1107592
50     672   5525    16      313        66496
60     0     15      13      104        316
80     1296  50019   15      168        600388
```

(b) Implicit and explicit accounting used.

Legend: `T`: Currently accounted tenant ID, i.e., γ_i. `Size`: Stack size, in bytes, of the method frame.

Figure 6: Detailed memory accounting calculations on a thread call stack.

and illustrated in Figure 7. Then we examine a thread that executes methods from these different components. We execute the thread without defining explicit rules (except a few predefined explicit rules, such as the one shown in the previous subsection), and we show the accounting results in Figure 6a, then we execute it after adding one explicit rule, and we show the accounting results in Figure 6b.

The Figure 6 describes the call stack of a thread that is created by the operator platform in order to start and stop components. A generic description of the thread activity goes as follows. The thread begins running operator platform code in method `org.knopflerfish.framework.BundleThread.run()`. Later, it starts the component `tests.A` by calling the *event handler* `tests.A.Activator.start()` that indirectly calls the method `e()` of the class `tests.B.C` which implements the interface `tests.C.C`, as shown in Figure 7. `tests.B.C` performs some processing before notifying the component `tests.D` of an event by calling the event handler method `handler()` of the class `tests.D.D` which implements the interface `tests.E.E`. Next, `handler()` calls a Java runtime service to sleep.

In Figure 6a, and starting from the first frame executed by the thread, i.e., `<native>`, the three following frames are accounted as previously described for Figure 5. Then, the operator platform (tenant 1) calls method `start()` of the class `tests.A.Activator` (tenant 50) which implements the operator platform interface `org.osgi.framework.BundleActivator`. Step 3(b) of the algorithm described in Section 2.4 states that the called tenant, i.e., tenant 50, is accounted for resources consumed in the method frame, which sets γ_i to 50. In the next five call frames (up to `tests.D.D.sleep()`), step 3(d) of the algorithm states that the caller tenant, i.e., tenant 50 ($\gamma_i = 50$), is accounted for resources consumed in method frames. Then, step 3(c) of the algorithm accounts consumed resources to the caller, i.e., tenant 50 ($\gamma_i = 50$). The remaining method frames are internal Java runtime calls, so step 3(a) of the algorithm accounts consumed resources to the caller, i.e., tenant 50 ($\gamma_i = 50$).

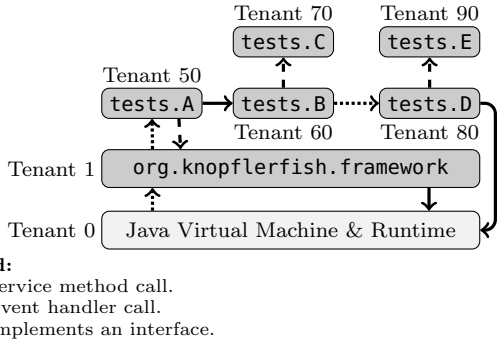

Figure 7: Functional tests components.

The problem observed in Figure 6a is that tenant 50 is held accounted for resources consumed during execution of the event handler method `handler()` and all the methods the latter executes, even though `handler()` is being executed only because the component `tests.D` requested it. Due to this issue, the monitoring system inaccurately accounts to tenant 50 the following amounts of memory: 1296 bytes of stack space, and 50 000 objects totaling 600 004 bytes of heap space, even though that memory was consumed by the event handler. To correct this inaccuracy, we add the following explicit rule: `account {*, 90/_/tests.E.E, called}`, which indicates that `tests.E.E` is an event handling interface (for any caller tenant, for any implementation component, and for any method in the interface).

Figure 6b shows the accounting results after adding the explicit rule and restarting the JVM. Adding the explicit rule effectively solves the problem by accounting the resource consumed by `tests.D.D.handler()` to the called tenant, i.e., tenant 80, which accordingly sets γ_i to 80. This boost in accuracy is also visible in the final statistics, as the monitoring system now accounts tenant 80 for the stack and heap memory that was wrongly accounted to tenant 50.

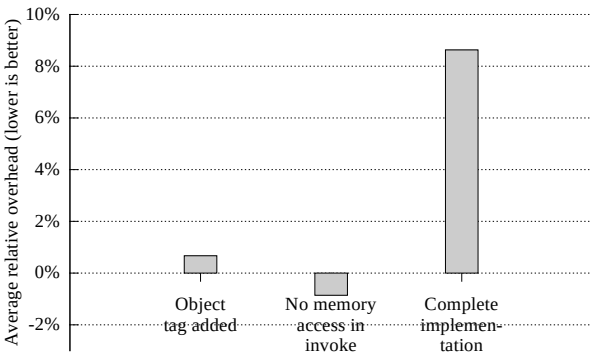

Figure 8: Execution overhead of the method call micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the "Zero implementation". Overhead is an average of 10 runs.

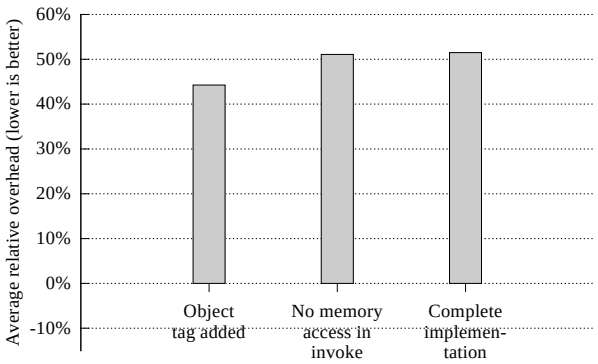

Figure 9: Execution overhead of the small objects micro-benchmark when run on partial implementations of the monitoring subsystem, compared to the "Zero implementation". Overhead is an average of 10 runs.

4.2 Performance micro-benchmarks

In order to show the details of the performance overhead of monitoring, we made four versions of the monitoring subsystem implementation, as follows:

1. *Zero implementation*, i.e., a JVM without monitoring.
2. *Object tag added*, i.e., the only thing modified in the JVM is adding 4 bytes to every Java object to hold the ID of the tenant accounted for the object.
3. *No memory access around* `invoke`, i.e., the whole monitoring subsystem is implemented in the JVM, except that we do not generate code to save and restore the thread's currently accounted tenant ID (6 `store` and 2 `load` instructions, see γ_i in Section 3.2) around the `invoke` bytecode instruction.
4. *Complete implementation* of the monitoring subsystem, i.e., the same as the previous version, in addition to 6 `store` and 2 `load` instructions generated around every `invoke` bytecode instruction.

We also performed some micro-benchmarks that test very specific aspects of execution.

4.2.1 Method call micro-benchmark

This benchmark performs numerous calls to the same Java method. The method itself does nothing special. We are only interested in the overhead of calling a method, with and without the monitoring subsystem.

Figure 8 illustrates the results of this benchmark. A comparison of the results of running this micro-benchmark on the "No memory access around `invoke`" version and the "Complete implementation" version reveals that method invocation performance decreases by 9%. We did not observe a significant additional loss in performance with other versions, i.e., less than 1%. Therefore, method invocation performance is only affected by the additional memory access performed by the monitoring subsystem around every `invoke` instruction.

4.2.2 Object creation micro-benchmark

This benchmark performs numerous creations of objects of the class `java.lang.Integer`. The objects themselves are relatively small. We are only interested in the overhead of creating an object, with and without the monitoring system.

Figure 9 illustrates the results of this benchmark. A comparison of the results of running this micro-benchmark on the "Zero implementation" version and the "Object tag added" version reveals that object creation performance decreases by 44%. We did not observe a significant additional loss in performance with other versions, i.e., less than 7%. Therefore, object creation performance is mostly affected by the addition of 4 bytes to every Java object.

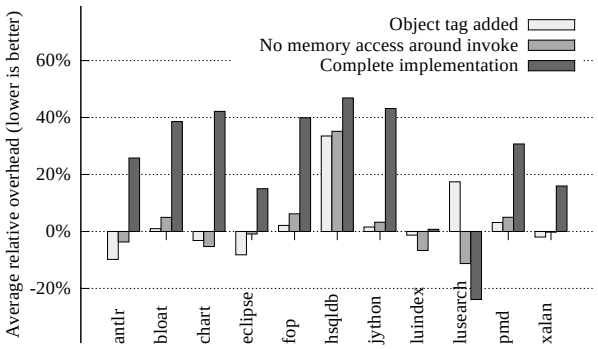

Figure 10: Execution overhead of DaCapo 2006 benchmark applications when run on partial implementations of the monitoring subsystem, compared to the original JVM. Overhead is an average of 10 runs.

4.3 DaCapo benchmarks

In order to measure the overhead of the monitoring subsystem on real life Java applications, we performed the DaCapo 2006 benchmarks[2]. Benchmark results in Figure 10 show that the monitoring subsystem overhead is always below 46% for real life DaCapo applications.

It is worth noticing that the monitoring overhead in Figure 10 stays below 6% for all DaCapo applications (except `hsqldb`) when we run DaCapo benchmarks using the "No memory access around `invoke`" version. This suggests that

[2]www.dacapobench.org

accessing memory (`load`, `store`) in every `invoke` bytecode instruction is an expensive operation.

We also notice that the performance of the application `hsqldb` drops by 33% when we run it using the "Object tag added" version. This suggests that this application creates many objects and would suffer from changes to the object structure. The performance of this application drops by another 13% between the "Object tag added" version and the "Complete implementation". This further indicates that `hsqldb` performs much more objects creations than method invocations.

5. RELATED WORK

Thanks to the popularity of Java-based systems, many monitoring tools were developed for them, such as A-OSGi [3], JMX [12], JVM-TI [13], and the method described in [11]. All these solutions are designed to monitor low granularity Java elements, e.g., threads, classes, objects, methods. Taken as is, the information produced by these solutions is of limited interest in OSGi platforms. Thus, we need to raise the abstraction to, at least, the OSGi component, i.e., the OSGi bundle. On a higher level, information about OSGi applications (set of components) and application tenants is also useful in industrial OSGi platform. Furthermore, most of existing tools requires heavy instrumentation, such as the method presented in [7], and often cannot be dynamically activated. However, these tools and methods form a foundation for the techniques described in this paper.

The method presented in [11], and refined in A-OSGi [3] and also in our previous work on *adaptive monitoring* [10] address CPU usage monitoring of components. These solutions show that observing resource consumption at the component granularity can be performed without modifying components. The goal of our previous work was to provide support for runtime activation/deactivation of monitoring of component service bindings, without stopping components and losing states. However, in all the previous work, the authors admitted that, during a service method call between two components, correct resource accounting needed information related to business logic regarding the caller component and the service being called, which is not provided by OSGi and Java. Our current work addresses the remaining challenge.

A rather disruptive approach was presented in the previous work I-JVM [4], where component isolation was achieved while preserving the OSGi component interaction model, relying on direct method calls. Each component runs in a dedicated isolate concept[3], composed of a separate class loader and a private copy of some JVM entities, i.e., static variables, strings and `Java.lang.Class` objects. Isolates run in the same address space, and objects are passed by reference between isolates. Threads migrate between isolates when methods are called/returned and exceptions are thrown/caught. The migration is a context switch to execute in the scope of the target isolate, enabling memory isolation of isolates and direct memory and CPU accounting. This approach enables isolating components and monitoring them. However, it also avoided addressing the challenge of identifying which component should be accounted for the consumed resources.

The standardization work performed at the OSGi Alliance [2], backed by existing OSGi solutions created by Makewave[4] [8] and ProSyst[5], raises the abstraction to the OSGi component. Existing solutions enable the OSGi platform operator to monitor, on a component basis: CPU usage, memory usage, number of threads, number of sockets, and disk usage. However, the ProSyst solution requires the framework developer and every component developer to call an API to indicate which entity is responsible for the resource consumption before any method call. Therefore, access to this API must be granted only to trusted entities, e.g., the OSGi framework and the components provided by the OSGi platform operator. Third party service providers sharing the platform will have no access to the API. This approach is too intrusive and cannot work in an environment where untrusted tenants share the platform. Our paper brings here a first improvement, as it is far *less intrusive* and *does not assume trusted tenants*.

The platform operator uses the standard API to control resource accounting behavior. Five situations can happen:

1. A call between two components in the platform operator is accounted to the platform operator.
2. A call between two components belonging to a given tenant is accounted to the tenant.
3. A call from a component belonging to a given tenant to a service method implemented by the operator platform is accounted to the tenant.
4. A call from a component belonging to the platform operator to a service method implemented by a given tenant is accounted to the tenant.
5. A call from a component belonging to a given tenant to a service method implemented by a different tenant cannot be correctly accounted unless specific information related to business logic is provided.

Often in Smart Home use cases, cases (1) through (4) are adopted, and the last case is completely avoided by disallowing communication between distinct tenants. That is where our paper brings a second improvement by *accounting resources between distinct tenants without requiring any form of isolation* more than the standard OSGi deployment unit.

6. CONCLUSION

OSGi gets increasingly adopted in the smart home as a framework to host service-oriented applications delivered by multiple untrusted tenants. This raises the need for monitoring systems that can provide useful accurate information about OSGi components and applications.

In this paper, we present a monitoring system that monitors memory usage at the component granularity, without requiring isolation of distinct tenants. Our system is far less intrusive than existing systems and methods, and it does not assume trusted tenants. It is based on a list of accounting rules that enable correct resource accounting in all cases. The monitoring system is modular and mostly independent of the implementations of the OSGi framework and the garbage collector.

Based on DaCapo benchmarks, we showed that the overhead of our system was below 46% for real life Java applications. This overhead is acceptable in development and

[3]Not to be confused with Java Isolates (JSR 121)

[4]www.makewave.com

[5]www.prosyst.com

testing time, and it is tolerable in slow pace applications which are frequent in the Smart Home. Our thorough investigation of performance overhead showed the specific aspects of monitoring that caused the most overhead, and the types of applications that would suffer the most from memory monitoring.

We are still searching for ways to make the list of accounting rules *dynamic* while keeping the overhead acceptable. Making the explicit rules further more generic, such as using *regular expressions*, would allow factoring of many explicit rules, but can reduce performance of matching against rules, so we are still unsure if more expressiveness would be worth the performance loss. We also long for monitoring other relevant resources, e.g., CPU usage, disk usage, and network usage. The relevant information reported by the monitoring system can become an accurate input to an *autonomous resource manager*, enabling the latter to automatically detect resource-related threats, such as resource abuse.

References

[1] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[2] G. Bonnardel, A. Bottaro, S. Dimov, E. Grigorov, and A. Rinquin. OSGi RFC 200 Resource Management. Request For Comments, OSGi Alliance, December 2013.

[3] J. Ferreira, J. Leitão, and L. Rodrigues. A-OSGi: a framework to support the construction of autonomic OSGi-based applications. *Int. J. Autonomous and Adaptive Communications Systems*, 5(3):292–310, 2012.

[4] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frénot, and B. Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 544–553, 2009.

[5] N. Geoffray, G. Thomas, J.Lawall, G. Muller, and B. Folliot. VMKit: a substrate for managed runtime environments. In *Virtual Execution Environment Conference, VEE 2010*, Pittsburgh, USA, March 2010. ACM Press.

[6] Home Gateway initiative. Requirements for Software Modularity on the Home Gateway, version 1.0. Technical Report HGI-RD008-R3, Home Gateway initiative, June 2011.

[7] J. Hulaas and W. Binder. Program transformations for light-weight CPU accounting and control in the Java virtual machine. *Higher Order Symbol. Comput.*, 21: 119–146, June 2008. ISSN 1388-3690.

[8] C. Larsson and C. Gray. Challenges of resource management in an OSGi environment. In *OSGi Community Event 2011, Darmstadt, Germany*, September 2011.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[10] Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi. Adaptive monitoring of end-user OSGi-based home boxes. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012*, pages 157–166, 2012.

[11] T. Miettinen, D. Pakkala, and M. Hongisto. A method for the resource monitoring of osgi-based software components. In *Proceedings of the 2008 34th Euromicro Conference Software Engineering and Advanced Applications*, SEAA '08, pages 100–107, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Oracle. Java management extensions (JMX) technology, January 2012. URL `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`.

[13] Oracle. Java virtual machine tool interface (JVM TI), January 2012. URL `http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/`.

[14] OSGi Alliance. OSGi service platform core specification release 4 version 4.3. Technical report, OSGi Alliance, April 2011.

[15] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, Nov. 2007. doi: 10.1109/MC.2007.400.

[16] P. Parrend and S. Frénot. Classification of component vulnerabilities in Java service oriented programming (SOP) platforms. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 80–96, Berlin, Heidelberg, October 2008. Springer-Verlag.